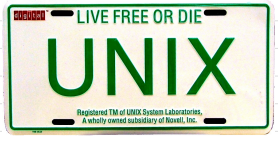
The Design and Implementation of the ULIX Operating System

Hans-Georg Eßer
Felix C. Freiling



ULIX

Informatik 1
Universität
Erlangen-
Nürnberg

The Design and Implementation of the

Operating System

Hans-Georg Eßer, Felix C. Freiling

Foreword

Welcome to the ULIX book. If you want to discover how precisely a modern operating system performs its multitasking magic, you've probably found the right textbook. On the next 600 pages you can read about the implementation of ULIX, a Unix-like system, and while you read the chapters which are organized along similar topics as those of most operating system textbooks, you will see the *full* implementation. We left out nothing, the whole code is there (actually: the book *is* the code, but more about that later), and you can read it step by step with each implementation part shown where it makes sense—which is very different from just providing a collection of source code files.

If you're already familiar with other operating system texts, you will notice that this book tends to be more practical. We often discuss only one solution to a problem (where other books mention a variety and sometimes give a qualified comparison of several techniques). We do not completely ignore alternative approaches, but since we give helpful explanations of all the code parts, we tend to write more about our solutions than about the ones which we did not choose for the ULIX kernel.

Writing this book was a fun experience, because for us, the authors, it meant learning a lot as we went along with the implementation and documentation task. While previous exposure to theoretical books on operating system principles was certainly helpful for deciding *what* to implement, it did not help a lot with the question of *how* to do it. Many technical references and some online tutorials for OS development beginners were useful (and much needed) in order to overcome the numerous challenges which the ULIX development encompassed. We hope that you will find reading our book as pleasant and instructive as we found writing it, and we encourage you to try your own experiments with kernel development: The book provides several exercises in which you can modify or add to the ULIX code.

Our overall hope is that you will find this book helpful and that it takes you on a journey that intensifies your knowledge of OS internals and the necessary hardware-related details. After reading, let us know if we've met your expectations (→ `feedback@ulixos.org`).

Hans-Georg Eßer
Felix C. Freiling

Erlangen, September 2015

This page intentionally left blank

Table of Contents

List of Figures

List of Tables

1 Introduction

Operating systems are an important part of computing. They mediate between the complex intricacies of modern hardware and the abstract needs of users and applications. Also, operating systems are one of the oldest research areas in computer science. Moreover, the topic is one of the core subjects in academic computer science curricula. So there are many books and other teaching resources available.

Because of the wealth of material and the practical appeal of the topic, operating systems are a fun subject to teach which can also make it an enjoyable course for students. As regular instructors of two of these courses we have learned that one of the key aspects of a good operating systems course is to discuss and analyze real operating systems, i. e., operating systems that work in practice and can actually be used by people. To this end, open source systems like FreeBSD and Linux have established themselves as good objects of study because they have commercial value and their source code can be accessed and scrutinized by lecturers and students in course.

FreeBSD, Linux

Unfortunately, it is hard to use the source code of such "real" systems directly and extensively in class. One main reason is the complexity of the code. Operating systems naturally have to deal with details of the underlying hardware. If a system is designed to be portable, the source code must cater for multiple computer architectures, which makes it even more complex. For example, at the time of writing this book, the current Linux version 3.14.5[1] had 29 subfolders in the `linux-3.14.5/arch/` directory which contained the specific code files for those architectures.

```
$ cd ~/Downloads/linux-3.14.5/arch/
$ find . -type d -maxdepth 1 -mindepth 1 | sed -e 's/\.\///' | column -c160
alpha   blackfin   ia64        mips       s390      um
```

[1] `https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.14.5.tar.gz`

```
arc     c6x         m32r        mn10300    score     unicore32
arm     cris        m68k        openrisc   sh        x86
arm64   frv         metag       parisc     sparc     xtensa
avr32   hexagon     microblaze  powerpc    tile
```

While not all platforms are supported equally well, in same cases there is a tremendous amount of platform-specific code as the following examples demonstrate which show the lines of code of source files in the `x86`, `ia64`, `alpha` and `powerpc` subdirectories:

```
$ for arch in x86 ia64 alpha powerpc; do printf "%8s" $arch:; \
> find $arch/ -type f | xargs cat | wc -l; done
     x86:  318881
    ia64:  115179
   alpha:   62383
 powerpc:  439961
```

Furthermore, modern operating systems usually offer an increasing number of features which must all be expressed by code. Another reason for the complexity of operating systems code is that practical operating systems must be extremely efficient. Every instruction cycle and memory cell used by the operating system cannot be used by the applications and their users. That is why operating systems code is often highly optimized. There are many other reasons that prevent instructors from showing real source code in class.

In our view the most important reason for not using real code directly in class is that operating systems have not been written with a human reader in mind, especially non-expert readers like students of operating systems classes who want to learn the basics of the area. Even when this is not the case and an operating system was developed for teaching purposes, such as the Minix operating system, it is often a tough task to browse through the complete source code that is split into several separate files and is ordered according to constraints of the programming language.

Minix

That is precisely where our approach differs: You can read this book from front to back and discover the theory and the implementation of the shown principles in the source code of the Ulix operating system that serves as an example.

1.1 Literate Programming

When we write software, do we really think about a human reader? The harder we try, the more we feel constrained by the programming tools available. When writing a complex piece of code, wouldn't you sometimes like to include a figure into the code to explain the complex interactions of variables? Or when you implement an algorithm from a textbook, wouldn't you like to give an automatic reference to this book in the source code? Or when one part of the code is similar to another part because you used the same idea, wouldn't you like to use automatic cross-referencing between these two parts? And aren't you bored of writing all these comment signs (like `//` in C or Java) all the time (or worse,

aligning the stars when using /* and */)? If you have ever felt such a desire, you are ready for literate programming.

Literate programming is a programming technique originally developed by Donald E. Knuth to write the TEX typesetting system. The source code of TEX appeared 1986 as a book called "TEX – The Program" [Knu86]. Reading that book is an entirely different experience from reading "normal" source code. It contains the *entire* source code, not just important excerpts, and it is *real* code that was compiled into the original version of the typesetting program.

TEX

With literate programming there is a conceptual change in the programming approach: Here, documentation comes first. A literate program is basically a text describing the program, and the actual code is inserted in the documentation, thus reversing the normal ordering. So instead of classically documenting code as in the following example:

```
// This function takes an argument, squares it, and adds the constant 42.
int square_and_add (x) {
  const int add = 42;     // declare the constant
  x = x*x;                // square x
  return x + add;         // calculate sum, return result
}
```

a literate programming version of the same code might look like this:

> We will write a function `square_and_add` which takes an argument `x`, squares it and adds some constant. Squaring a variable is just a multiplication of the value with itself:
>
> ⟨*square* `x`⟩ ≡
> ```
> x = x*x;
> ```
>
> With this knowledge we can implement the function:
>
> ⟨*function implementation*⟩ ≡
> ```
> int square_and_add (x) {
>   ⟨declare constant add⟩
>   ⟨square x⟩
>   return x + add;
> }
> ```
>
> Now, what's left is to decide on the constant `add`—we pick 42:
>
> ⟨*declare constant* `add`⟩ ≡
> ```
> const int add = 42;
> ```
>
> This is an implementation of the function $f\colon x \mapsto x^2 + 42$.

Note how the function definition uses two code chunks: one that was presented *before* the function and another one which came *after* the function. Literate programming makes the developer independent of any specific ordering of code fragments which the language may

demand, e. g. the declaration of variables before their first use. Also, both top-down and bottom-up styles of programming are possible and can be combined.

Following the idea of literate programming, this book is a book for students. Its source code can be used to generate executable code (for the Ulix operating system) *and* a LaTeX file that can be typeset to an introductory text on operating systems (this book).

Ulix

When you look at the source file `ulix-book.nw` from which this book was created, you will see how code chunks are declared: The chunk name is always put between `<<` and `>>=`, then follows the code which is terminated with a `@` character on a single line. When referencing a code chunk the same syntax (without the equals sign) is used. So for example, in order to declare the ⟨*function implementation*⟩ chunk from above, the following lines are needed:

```
<<function implementation>>=
int square_and_add (x) {
  <<declare constant add>>
  <<square x>>
  return x + add;
}
@
```

Chunks may also be continued, i. e., some pages (or even chapters) after the initial definition of a code chunk it is "defined again." That does not replace the original definition, but add to it. In the source file this is done by simply writing

```
<<function implementation>>=
...
@
```

again (with additional code lines inside), but in the PDF version the representation of a continued code chunk looks slightly different: Instead of

⟨*function implementation*⟩ ≡

it will look like this:

⟨*function implementation*⟩ +≡

with a +≡ sign that indicates the *continuation*—similarly to C's `+=` operator that also has an "add to" meaning (e. g. `x+=5;` means: add 5 to the value of `x`).

1.2 The Ulix Operating System

Throughout this book we present the whole source code of the Ulix operating system. That encompasses the kernel but also the user mode library which application developers must use in order to interface with kernel functions. The book does not show source code for (all) the applications.

1.2.1 The Name of the Game

The name Ulix is intentionally similar to the name Unix. This is meant to imply that the code is influenced a lot by things we have seen in Unix/Linux style operating systems. The focus on Unix is solely based in the past experiences of the authors, it is in no way intended to imply that Unix is better or worse than other operating systems.

The choice of the letter "l" in Ulix is supposed to concisely express that the system is meant for *learning* and teaching operating systems. Besides, the name Ulix is one of the few four-letter abbreviations that do not seem to have been chosen for other software systems yet. Furthermore, Ulix is probably the first operating system that is written as a literate program, and so Ulix can also stand for "literate Unix".

1.2.2 Design Principles of Ulix

The following principles have determined the design of Ulix:

- Ulix is for *learning and teaching* principles of operating systems in a course. Ulix should never be a practical system in the sense that it can be used to run real applications.
- Nevertheless, Ulix should be a *real* operating system, i. e., the code should be executable on some well-defined computer architecture. If necessary, it should be possible to port Ulix to other platforms, but portability is not a core requirement of Ulix.
- The design and implementation of Ulix should be governed by the principle of *simplicity*, avoiding optimizations, focusing on understandable and correct code.
- It should be possible to use the source code *directly* in class. The source code should be written with the human reader in mind.

Given the above design principles, one point should be clear—but is important enough to mention it anyway: While trying to be *real*, Ulix is not *practical*, i. e., we disclaim any fitness for practical use. On the one hand, the code is not guaranteed to be free of programming errors. On the other hand, the performance of Ulix is such that it will not achieve any required quality of service in practice. Ulix is *purely* for learning. The path chosen is the one of simplicity. So when you have read this book, you will have a fairly good idea of how Ulix works, but only a faint idea of how operating systems work in general. Therefore, this book is not (and never will be) a replacement for the available excellent general textbooks on operating systems.

1.2.3 Ulix Features

Some of the words in the following feature list may not make sense to you right now, but if you already have some knowledge of operating system principles, this gives you an idea of what kind of content to expect in the book. If you are familiar with some Unix system as a user or administrator, at least most of the application programs mentioned in the following paragraph should be well-known.

Ulix is a classical Unix-like operating system which features processes with separate address spaces, threads, paging, a virtual filesystem (currently supporting floppies and hard disks, with Minix as the primary filesystem) and synchronization via kernel mutexes and semaphores. It has a preemptive scheduler (implementing a simple Round Robin strategy) and allows for the integration of new interrupt and system call handlers at runtime. Ulix supports up to ten text mode terminals. It works on 32-bit Intel-compatible CPUs and provides system calls (via the classical `int 0x80`) and corresponding user mode library functions which are compatible to other Unix systems, e. g. `fork`, `execv`, `exit`, `waitpid`, `signal`, and `kill` for process control, some of the `pthread_*` and `pthread_mutex_*` functions for thread control [IEE95], `open`, `read`, `write`, `lseek` and `close` for file access, `brk` for dynamic memory (heap) management etc. There are also a few user mode programs (`cat`, `chgrp`, `chown`, `chmod`, `clear`, `cp`, `df`, `diff`, `free`, `grep`, `hexdump`, `kill`, `ls`, `man`, `mkdir`, `ps`, `readelf`, `rm`, `rmdir`, `stat`, `sync`, `touch`, `vi` and `wc`), and some more commands are implemented as shell built-ins of the Ulix shell `sh`, e. g. `cd` and `pwd`.

A login mechanism asks for user name and password and checks these against entries in `/etc/passwd` (where the passwords are stored in plaintext and world-readable since hashing and encryption are not available in Ulix), and authenticated users can only access files for which they have the required access permissions. Also, signaling other processes via the `kill` function or program requires the user to own the targeted process (or have administrator privileges).

Redirection of standard input, standard output and standard error are supported via closing one of the file descriptors 0, 1 or 2 and opening a new file (which will reuse that descriptor). The shell understands the `<`, `>` and `2>` syntax for starting programs with redirections, and it can also use the feature to execute shell script files (via `sh <script.sh`).

Ulix uses a buffer cache for disk read and write operations so that repeated access to the same data on disk is faster than first access. A `swapper` process runs in the background and checks whether physical memory fills up too much: if so, it will pick parts of memory and write them to disk in order to increase the available memory. If such a "paged-out" memory area is accessed later, it will be "paged in" again before the program that wants to use it can continue.

The system can be run inside a virtual machine using `qemu` and it writes a kernel log to a (virtual) serial port; you can save this output in a file for later analysis. The simple shell can launch user mode programs (which need to be compiled outside Ulix). Such programs are stored using the ELF executable format [TIS95]. Ulix expects to have 64 MByte of RAM and maps that physical RAM to a fixed address space region for easy access to the physical memory. POSIX threads are somewhat limited in that each thread has a fixed-size stack (whereas the primary or only thread has a stack which automatically increases its size as needed).

1.3 Tools

The original goal was to write a book like "TEX – The Program" [Knu86] with pretty-printed sourcecode and extensive automatic cross-references and indexes (especially the

famous mini indexes on right-hand pages). This can be achieved using the Knuth/Levy CWEB documentation tool [KL03, KL01] that also has hypertext extensions. Mini indexes can be generated by an extension called CTWILL [Knu93] that is the program used to generate the source of "TEX – The Program". However, the CWEB family of tools is restricted to TEX as typesetting language which we wanted to avoid in favour of LATEX. While there exists an experimental adaption of CWEB for LATEX by Joachim Schrod [Sch13], CWEB is also restricted to the C programming language, so more than one language (e. g. C code, assembler code and configuration files) cannot be handled.

For this book we decided to use Norman Ramsey's `noweb` tool [Ram94] which combines a simple syntax with language independency: it uses LATEX (or alternatively HTML) as input and output format for the documentation. There are also several extensions which add pretty-printing and indexing.

`noweb`
LATEX

The main source for the book is one huge LATEX file that serves as input to the `noweb` tool chain. The program `noweave` converts it into a classical LATEX file that can then be processed as usual; for this book we chose the XELATEX variant [XeL14] of LATEX because it handles UTF-8-encoded input files and provides better options for font selection. Together with standard LATEX tools (`bibtex` and `makeindex`) it produces the PDF file that you're currently reading in the PDF viewer or as a printed copy.

`noweave`
XELATEX

The other part of the `noweb` tool chain generates the ULIX source code files: `notangle` extracts code chunks and saves them in source files which will then be compiled or assembled with the GNU C compiler `gcc` and the `nasm` assembler.

`notangle`
`gcc`, `nasm`

Figure 1.1 shows a simplified view of the build process for both the book as well as the kernel, the user mode library and a sample application. We will give a detailed description in Chapter 18.

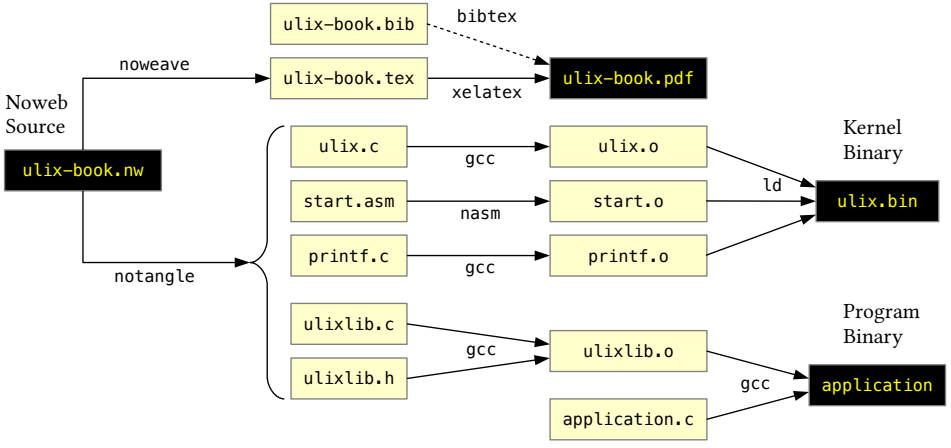


Figure 1.1: Using the Noweb tool chain it is possible to generate this book and also the ULIX kernel binary and other system files.

1.4 Copying

Free Software

Ulix is Free Software, following the definition of this term that was coined by the Free Software Foundation (FSF). You can copy and browse the code and compile it into any form you like. If you find bugs in the source code, please drop us a message so that we can fix the bug in the next iteration of the software. Similar to TEX, Ulix is meant to eventually become a stable platform that does not evolve anymore, i. e., we will eventually stop issuing new releases. That's why we retain the entire copyright that must be mentioned in all files associated with the system. If you think that Ulix should be fundamentally changed, become more efficient etc., you can take the code but should call the resulting system something other than "Ulix".

[24] ⟨*copyright notice* 24⟩≡ (44a 48)

```
/*
Copyright (c) 2008–2015 Felix Freiling, University of Erlangen–Nürnberg, Germany
Copyright (c) 2011–2015 Hans–Georg Eßer, University of Erlangen–Nürnberg, Germany

This program is free software: you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this
program.  If not, see <http://www.gnu.org/licenses/>.                          */
```

You can find version 3 of the GPL at the end of the book.

1.5 Notation

In this section we describe the various conventions used throughout the book.

1.5.1 Numbers and Units

This book makes heavy use of hexadecimal numbers and (less often) octal and binary numbers. We expect readers to be familiar with these positional numeral systems with bases 16, 8 and 2.

hexadecimal

- Hexadecimal numbers such as `0xAB12CD34` will appear with a monospaced font and a `0x` prefix (since that is the notation also used in C). Sometimes we will insert a dot in the middle (`0xAB12.CD34`) to improve readability.

octal

- Octal numbers such as 56701_o also use the monospaced font but have a small $_o$ index at the end to indicate that the octal system is used. C uses a different notation: every

number which starts with a zero is considered to be an octal number which sometimes causes problems for readers who are not familiar with this convention.

- We write binary numbers like $\texttt{100101}_b$ in a similar fashion, but with a $_b$ index at the end. Standard C has no way to express binary numbers, but the GNU C compiler allows the non-standard syntax `0b100101` (with a `0b` prefix, similar to the `0x` prefix for hexadecimal numbers). binary

We often discuss kilobytes, megabytes (or rarely gigabytes) when talking about memory or disk areas and filesizes. When we do so, we use the historical meanings as displayed in Table 1.1. Note that most newer books use the reformed interpretation according to which kilo-, mega- and gigabytes refer to powers of 1000 of bytes, which is more consistent with other uses of "kilo" and "mega" like, for example, kilometers and megatonnes. The new terms for the old units are "kibibyte", "mebibyte", "gibibyte" etc. [Int00]. kilobyte megabyte

However, having a special unit name for 1000 bytes or a million bytes is quite useless since these sizes have no meaning: 1000 is not a power of 2 (but $1000 = 2^3 \times 5^3$), and everything in the hardware is organized by powers of 2. That is why we have decided to stick with the classical meanings even though they are considered wrong by today's standards. If you're already used to the new notation, please replace any occurrence of "kilobyte" with "kibibyte", "megabyte" with "mebibyte" etc.

Note also that the often-used term "1.44 megabyte floppy disk" makes no sense at all, because such a floppy stores 2880 sectors each of which is 512 bytes large. Thus, the disk size is 1440 KByte which are 1.40625 MByte. When using the new terms the size could be expressed as 1474.56 kB or 1.47456 MB, and none of those numbers should lead to "1.44"—you can only arrive there by mixing the old and the new terminology and claiming that a megabyte was 1000×1024 bytes.

Unit	**Abbreviation**	**Size in bytes**	**hexadecimal**
Kilobyte ("Kibibyte")	KByte	$2^{10} = 1024^1 = 1024$	`0x00000400`
Megabyte ("Mebibyte")	MByte	$2^{20} = 1024^2 = 1\,048\,576$	`0x00100000`
Gigabyte ("Gibibyte")	GByte	$2^{30} = 1024^3 = 1\,073\,741\,824$	`0x40000000`
"New Kilobyte"	kB	$1\,000^1 = 1000$	`0x000003E8`
"New Megabyte"	MB	$1\,000^2 = 1\,000\,000$	`0x000F4240`
"New Gigabyte"	GB	$1\,000^3 = 1\,000\,000\,000$	`0x3B9ACA00`

Table 1.1: In this book we use the classical meanings of "kilobyte", "megabyte" and "gigabyte".

For comparison, we have added the new interpretations to the table; especially when looking at the hexadecimal representations you can see that these units do not occur in practical settings.

1.5.2 Identifiers

Names of variables, functions, constants and other code elements that appear in a regular paragraph also use the monospaced font, and we sometimes add round brackets to a function name when we want to emphasize that it is the name of a function. So you may find references to "the `read_file` function" or simply to "`read_file()`". When a variable or function is declared or defined in a code chunk, there will often be a suffix which indicates the page number where the definition can be found, e. g. for the `readblock_hd`$_{531b}$ function.

Some functions appear to exist twice: once inside the kernel, and once in the user mode library which applications must link to access those kernel functions. An example is the `open` function which opens a file. Surely, the kernel needs a way to open files, and applications also have that need. While both functions will never appear in the same context, they do appear in this same book which contains cross references to the place where a function was defined. Thus, using the same name would cause some confusion, so we use slightly different names in the kernel by adding a `u_` prefix to the name. The kernel uses the `u_open`$_{412c}$ function, and the user mode library provides an `open`$_{429b}$ function.

u_ prefix

Constants (defined via the pre-processor's `#define` macro) use all-upper-case names and underscores as word separators, for example `MEM_SIZE`$_{111c}$.

1.5.3 Margin Notes

You will have noticed already that the page margin sometimes contains a few words, for example, in the above paragraphs, there's a "`u_` prefix" margin note. These notes are helpful in two ways: when you thumb through the pages of the book, you can quickly identify key words, and when you have looked up a word in the index and gone to the right page, a margin note leads you to the right line.

The margin also contains code chunk numbers in `[]` brackets (see the following section).

1.5.4 Code Chunks

The most important bits of information within each code chunk are the chunk name and the actual code, but there is more, and knowing what the additional elements of a code chunk mean will help you navigate the code more easily, for example when you find a function call to a function that you have forgotten (or that may be defined only later in the text).

Figure 1.2 shows two example code chunks that demonstrate all the elements you will find in the chunks.

- First of all, each code chunk has a name. When a chunk is defined for the first time and you're on page 123 of the book, the chunk will begin with a line like

  ⟨*chunk name* 123⟩ ≡

  which confirms that this chunk indeed starts on page 123. If more than one chunk starts on this page, a lower case letter is appended to the page number so that you can distinguish between them, e. g. 123a, 123b and 123c for three code chunks on

page 123. If the chunk definition started earlier (and this is a continuation) then the original chunk number will appear next to the chunk name.

- Regardless of whether the current chunk is an initial definition or a continuation, it always has an individual chunk number. This number is displayed in the margin inside [] brackets, e. g. [122c] in the figure.
- The top line of the chunk contains up to three further chunk numbers on the right-hand side. If you find a chunk number in round brackets (such as (62a) in the figure), then it is a reference to the first place where this chunk is used. The next two chunk numbers are prefixed or suffixed with a left or right facing triangle. If a number with a left triangle exists (in the example: ◁ 122b), this points to the previous continuation, and a number with a right triangle (in the figure: 124c ▷) leads you to the next continuation. Using these forward and backward pointers you can find all locations where this chunk is defined (and thus read the whole source code that the chunk is made of).

  When one or more of these three numbers do not exist, that has the obvious meaning: A missing (...) reference means that the chunk is used nowhere. That should only happen for "root chunks": those are the chunks which `notangle` extracts to create the

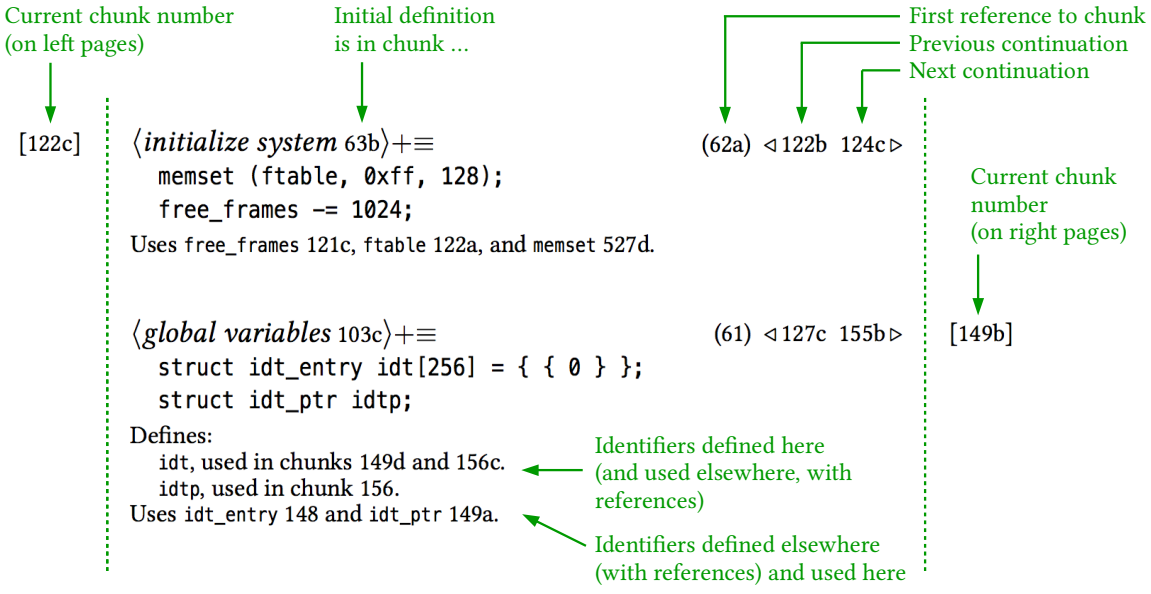


Figure 1.2: Code chunks come with links to other parts of the source code and let you navigate through the sources: you can quickly find out where a function or variable was defined, and below the defining chunk you see all the places where it is used.

compiler/assembler source files. A missing forward or backward pointer simply indicates that you have reached the last or first part of the chunk definition, respectively.

- Under some chunks you find a line that starts with "Used:" and lists all (known) identifiers and the chunk numbers where they were defined. Both examples in Figure 1.2 contain such a line.
- Finally, a code chunk may define one or more identifiers. In that case, you will see a block starting with "Defines:" and a separate line for each defined identifier, giving its name and all the locations in the book where it is used. In the example figure only the second code chunk has such a block. The "Used:" and "Defines:" blocks let you track the usage of functions, variables and constants throughout the whole code which is often more helpful than searching for an identifier in the code files.
- There is one case that we treat in a special way: The C language demands that functions are declared before their first use. Since we do not want to consider the ordering of function implementations it is often necessary to insert a function prototype which will appear early in the C file, whereas the implementation occurs at a later position. So there will be prototype code of the form

  prototype vs. implementation

  ```
  int function_name (arguments);     // prototype
  ```

  and implementation code that looks like this:

  ```
  int function_name (arguments) {    // implementation
    ...
    return ret_val;
  }
  ```

  Technically (from a literate programming point of view), only one of those code blocks *defines* the function, and the other one *uses* it. In the book we will often, but not always, present both parts on the same page though they will appear in different places in the generated C file. We have decided to make the implementation chunk the *defining* chunk (since that is what you will want to look up when you see usage of a function). We have also removed the "Used:" line for the prototype chunk since it will refer to the chunk which immediately follows.

Even if you have a printed copy of this book, it is helpful to use the PDF version as well, since all chunk numbers are links to those chunks, and most identifier names are clickable, too, leading to the place where a variable, constant or function is defined.

1.5.5 Coding Standard

The code uses two spaces for indentation, and the curly brackets `{` and `}` which declare the beginning and the end of a block are always written in this form:

```
int sum_of_first_ten () {
  int j, sum;
  for (j = 1;  j < 11;  j++) {
```

```
    sum += j;
  }
  return sum;
}
```

We often use the // one-line comments which originally were not part of the C standard, but are supported by modern C compilers. So instead of classical C comments such as // vs. /* */

```
int tmp = array[i].mem;   /* the memory address */
```

you will more often see this form:

```
int tmp = array[i].mem;   // the memory address
```

In the book, the comparison operators <= and >= are displayed as ≤ and ≥, but this is the only kind of pretty-printing that we have applied to the source code chunks, except for a little color highlighting of comments, brackets and the exclamation mark and slanted printing of the #define and #include pre-processor commands: ≤, ≥

⟨*example for syntax highlighting* 29⟩≡ [29]

```
#include "ulixlib.h"           // use the library
#define TRUE 1                 // define a symbolic constant
void example_function (int param) {
  int vector[10];
  for (i = 0;  i ≤ 10; /* inline comment */  i++) {
    if ( !param ) { vector[i] = vector[i] * 2; }
  }
}
```

In functions which have no return value (typed as void) we omit the return statement because the function automatically returns when it reaches the end of the function's body; the C standard allows this practice, and it saves a line in each such function.

1.5.6 "Going where?"

At the beginning of some sections you will find a short text in two-column layout using a different font, such as the following:

This is an example for a "Going where?" paragraph. Right now it has no useful contents, so let's just note that you will soon see the first lines of real code. **Going where?**

Sometimes it is easy to get lost and wonder: why have we been dealing with this or that function? In such cases, the "Going where" paragraph gives you some orientation. How do things fit together? Where are we in the larger picture?

1.6 Creating an Operating System From Scratch

Writing an operating system is very different from developing an application. That is because, when you start out, there is literally nothing. For example, there are no libraries that would provide standard functions such as `printf`. Also there are no rules dictating how the operating system is going to handle things—it is up to the developers to decide what those rules should be.

First of all, some elementary questions need to be answered:

- What kind of hardware will the OS work on?
- What programming language(s) should be used for development?
- What kind of applications will the OS be able to host?

If you expect to create an OS using Java that will be compatible with Windows, Linux and OS X and will also sport a high performance 3D engine so that the latest console games run on it, then this textbook will be pretty disappointing.

1.6.1 Selection of Target Hardware

The computing world is diverse, allowing for all sorts of hardware architectures. CPUs can have very different features—if you have attended a course on computer architectures, you will have noted things like RISC and CISC CPUs with very small and simple or huge and complicated instruction sets. In this book we will focus on the 32-bit Intel architecture, for the simple reason that most people have quick access to an Intel-compatible machine or can at least run an Intel-based operating system in an emulator. With their 32-bit architecture, the CPUs can access 4 GByte of physical memory which is enough for most purposes. During the last years 32-bit CPUs have become a bit outdated, as the latest processors have a 64 bit wide address bus and provide internal registers with the same size.

Intel hardware has some legacy problems because even the latest (64-bit) Intel chips are compatible with old systems from last century's 80s. We will see this when we discuss the management of memory.

1.6.2 Language of Choice

C language

Operating systems are very close to the actual hardware. In fact you won't see any other class of "programs" which get any closer to the hardware (except for a computer's firmware), because there's always the OS as a natural barrier between hard- and software. We're in the area of systems programming, and this is where "old school" languages still dominate. So with most operating systems you'll see lots of C code. For those who have never heard of C (without a ++ or # postfix): C is a procedural language that was created in 1969–1973 by Dennis Ritchie[2] and it's a predecessor to C++, Java and C#. It does not know objects.[3] Ulix was (mostly) written in C.

[2] Ritchie reviewed the early history of C in an article [Rit93].

[3] For those readers who are unfamiliar with C, we have included a short introduction to C which requires C++ or Java knowledge, see appendix A on page 635.

The principle of simplicity demands that we use "clear C", i. e., we discipline ourselves to non-optimized and clear code that can also be understood by people familiar with Java. We also restrict inclusion of library header files (we want to be self contained).

Assembly language

Even closer to the hardware is assembler code, and for that reason all the early operating systems were programmed in assembler. Assembler code is what a C compiler will generate when you provide it with some C source code. Today it is no longer necessary to write complete operating systems in assembler (some people still do this, e. g. the BareMetal [Sey13] developers), but you'll still need some assembler code from time to time, because some parts of the OS need to access CPU registers or execute special machine instructions which are not available in the C language.[4]

For the Intel processor platform, two "dialects" exist, the Intel and the AT&T one. The GNU C compiler supports both but defaults to the AT&T variant. We have decided to use the Intel syntax, because it is closer to C's syntax: For example, you can load the *EAX* register with the value 0 via the command `mov eax, 0` (in Intel syntax). So the target of the `mov` command comes first which resembles the C command `eax = 0`. In AT&T syntax, the operands are reversed, with the target coming last and extra syntactical elements being needed (`mov $0, %eax`).

1.6.3 Applications

An operating system X will run applications which have been developed for X (let's call them "X applications"), and it will be either impossible or very hard to run Y applications for any Y which is not X. Every OS creates its own software universe, and if you want to run a program from a parallel universe, you'll need some sort of emulation—which is not a topic of this book.

Most applications require libraries which are typically considered part of the operating system. Even for something as simple as printing "Hello world", you need a library that contains the code which is necessary to make the OS print something (in a text console, a window or perhaps on a printer).

In principle it would be possible to port many of the existing Unix (text-mode) applications to Ulix, however most programs make excessive use of the available libraries, so those would have to be ported first. This version of Ulix comes with a limited set of tools, including a very limited version of the `vi` editor. So do not expect to replace your current Linux installation with a Ulix system.

1.7 What's in the Book?

Reading this book, you will see introductory descriptions of several theoretical concepts, and at the same time you'll see the complete source code necessary to implement these concepts.

What's in the book is in principle what's required for a typical operating system—but note that for all theoretical problems there are lots of possible solutions, and Ulix pro-

[4] Appendix B on page 647 provides an introduction to the x86 Assembly language.

vides *one* solution to most problems. In many chapters we will begin with a theory section which gives broader information about the topic. We might present several different approaches but show only the implementation of one of them: the one we picked for Ulix. This will typically be the most simple solution, because we did not want to make the code too bloated.

Let us first look at some of the tasks an operating system needs to perform. For every modern system, the process (or more precisely: several processes) is a central idea. Many processes may coexist on a machine, and the CPU shall execute them in parallel. We need ways to spawn new processes and terminate processes which have completed their tasks.

The OS kernel will need internal `fork`, `exec` and `exit` functions for process creation and termination, but those will not be available to running programs (which are not running inside the kernel)—this problem is typically solved via system calls. So we need a system call interface and system calls `fork`, `exec` and `exit` which allow access to the kernel functions.

Starting a new process often requires loading the program code from a disk, and once the program is running, it will perform I/O operations: typically with the disk, the keyboard and the video display. Keyboard input arrives via keyboard interrupts, so our operating system must deal with interrupts. These are also needed for talking to the disk, and since we do not want to perform raw sector read/write operations to access data, we need a logical filesystem (which will be the Minix filesystem, version 2). Both the OS itself and processes will `open`, `read` and `write` files, and in order to let processes perform these actions, we need more system calls that—again—allow access to these functions.

Since each process has its separate memory, we need kernel routines for memory management. Ulix uses paging and creates address spaces for all processes (which are basically page tables with some additional administrative data). For each newly created process the system must reserve some memory; at least it will have to create a new address space. Later (at process termination) this memory must be set free. We must define a memory layout that describes what parts of (virtual) memory belong to the kernel and what parts belong to processes. A process may also ask for more memory while it is running, this requires another system call (`brk`) which can be implemented by allocating a new page for the process.

Unix systems use a signaling system that allows both processes to send signals to other processes as well as the kernel itself to signal a process. Processes may register signal handlers in order to avoid the default actions for receiving a signal (typically: ignore or abort). Again we need more system calls for registering handlers (`signal`) and sending signals (`kill`). The scheduler must check whether a process has pending signals when it activates it; in that case it must call the signal handlers (or perform the default action).

If we combine the ideas about processes which we have presented so far and add some further process properties, we arrive at the mind map shown in Figure 1.3: it is rather complex, and we have only looked at the system from the process perspective.

Thus, we will cover the following topics:

- **Memory Management Theory** (Chapter 3) gives an introduction to the possible ways in which the system's main memory can be shared between several processes. We start with some simple models and quickly turn to paging, today's standard method.

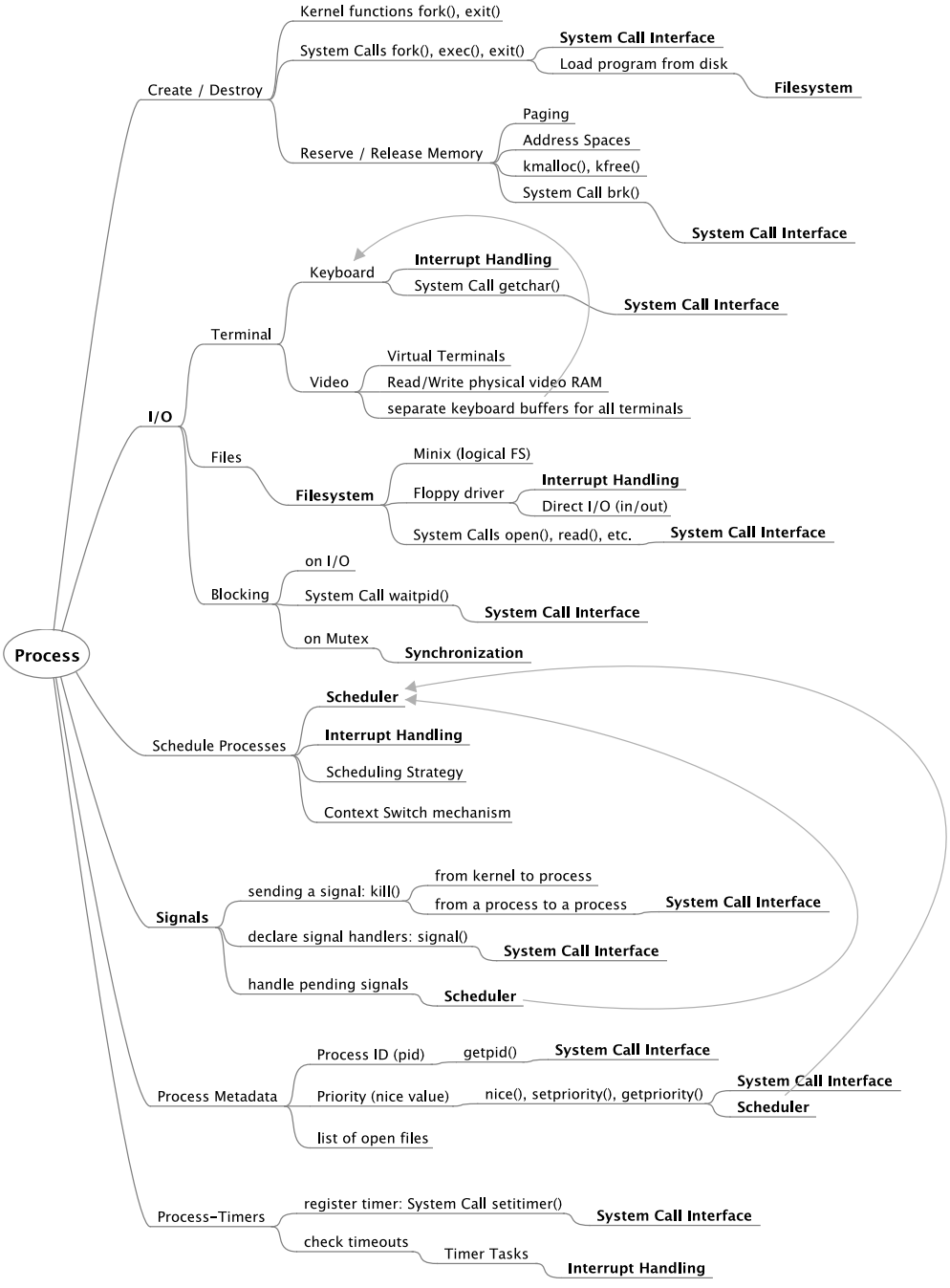


Figure 1.3: Mind map for process functionality.

- **Boot Process and Memory Management in ULIX** (Chapter 4): After the theoretical introduction you're prepared to look at the first steps of the system initialization. We need to load the kernel, but where should we place it in RAM? The code in this chapter enables the paging mechanism and builds the foundation for process-related code which will allow to switch between several address spaces.
- **Interrupts and Faults** (Chapter 5) trigger the execution of handler functions which we must provide—they should do something useful about the event that started them, for example read data from a device that signaled the completion of some activity.
- **Processes and Threads** (Chapters 6 and 7) are all about running programs which most people consider the primary task of an operating system. We present code for classical Unix processes (handled by the `fork`, `exec`, `exit` functions) and an approximation of POSIX threads.
- **Scheduling** (Chapter 8) provides ULIX with its multitasking feature. You will learn about some approaches towards scheduling and then see the implementation of a round-robin scheduler. This is both about deciding when to switch to a different process (and which one) and about the switching itself: What do we have to do to temporarily halt one process so that another one can run?
- **Handling Page Faults** (Chapter 9) is a continuation of both the chapters on memory management and fault handling; a page fault is a fault that we can often recover from without terminating the causing process. In this chapter we also present our routines for paging out and back in: When memory becomes scarce, we write parts of a process' memory to a disk file in order to free space—when we need it again, we bring it back.
- **Talking to the Hardware** (Chapter 10) deals with support for some standard devices, including the screen, the keyboard and the on-board clock's timer.
- **Synchronization** (Chapter 11) is necessary because we allow multitasking. We show the implementation of kernel semaphores and mutexes which are standard primitives that help protect data against data-corrupting simultaneous access. We also explain how interrupt handlers and system call handlers (called by processes) which access shared data can be synchronized.
- **Filesystems** (Chapter 12): Here we deal with the logical aspects of accessing files on media. We present the ULIX Virtual Filesystem and our implementation of the Minix filesystem. Actually talking to the disk drive controllers is left to the next chapter.
- **Accessing Hard Disk and Floppy Disk Drives** (Chapter 13) requires some understanding of the protocols that these controllers "speak". This chapter is very technical, but you do not have to deal with all the details in order to see how disk access works and data can be transferred between a disk and memory.
- **Signals** (Chapter 14) are a classical Unix mechanism which allows a simple kind of messaging: processes can send signals to other processes which makes them either terminate or call a registered signal handler. One use case is killing a process.

- **Users and Groups** (Chapter 15) let several users share one system but keep each user's data private. Every file belongs to one specific user (its owner), and each process is associated with one user (its creator), as well. We show how Ulix implements the standard user/group mechanisms of Unix systems.
- The **Small Standard Library** (Chapter 16) provides often-used but less interesting functions such as `printf`$_{601a}$ and `memcpy`$_{596c}$. This is not really what an operating system is concerned with, but without output and string management functions we could not do a lot.
- **Debugging Help** (Chapter 17) contains the code of the kernel shell and its internal commands. That shell is only available for debugging purposes.
- **Build Process** (Chapter 18): In this chapter we discuss how you can extract the source code from this book (assuming you have its Noweb source file `ulix-book.nw` and then build the system. Read it if you want to *modify* the system. If you only want to *run* Ulix, there are easier ways to get started.
- Finally, for those new to C and/or Assembler, there are introductions to C (Appendix A) and to the Intel x86 Assembly Language (Appendix B).

We will give a more detailed description of some of these topics in Chapter 2.

If you copy all the code from the book into appropriate files (or download the version we provide on the website) you can compile it into an operating system that will actually boot on the `qemu` PC emulator.

We'll set the OS name and version now:

⟨ *macro definitions* 35a ⟩≡ (44a) 46d▷ [35a]

```
#define UNAME "Ulix-i386 0.13"
#define BUILDDATE "Mon Nov  2 17:33:51 CET 2015"
```

Defines:
`BUILDDATE`, used in chunks 337c and 605a.
`UNAME`, used in chunks 337c, 605a, 609, and 610a.

⟨ *version information* 35b ⟩≡ (44a) [35b]

```
/*
v0.01  2011/06     first version: boots, enables interrupts, keyboard handler,
                   protected mode; most code taken from kernel tutorials
v0.02  2011/07/31  paging for the kernel (not yet for user space)
v0.03  2011/08/12  paging with Higher Half Kernel / GDT trick (preparation for
                   user space)
v0.04  2011/08/17  dynamic memory allocation: request frame, request new page
                   (with update of page table; creation of new page table if
                   last used one is full)
v0.05  2012/10/02  serial hard disk and external storage server (for use with
                   qemu & co.)
v0.07  2013/04/05  Scheduling and fork / exec / exit / waitpid are working.
v0.08  2013/07/13  Minix Filesystem support (replaces "simplefs"). Can read,
                   write, create files.
                   Kernel uses floppy (FDC controller) instead of serial disk
                   Terminal support (up to ten terminals with shells)
```

```
v0.09  2013/11/02  execv (with ELF loader) works; moved internal shell
                   commands to ELF binaries in the bin/ directory
v0.10  2014/01/07  Mounting works, VFS functions (u_open, u_read etc.) work
                   across several mounted volumes
                   Enter kernel shell: Shift+Esc, return to user mode: exit
v0.11  2014/04/23  Filesystem code is complete, buffer cache also used for
                   writing. Kernel level threads with pthread_* and pthread_
                   mutex_* functions.
v0.12  2014/08/19  Code is complete for the first public release.
v0.13  2015/09/02  Modified version (error fixes).
*/
```

Welcome to Ulix 0.13!

1.8 Helpful Previous Knowledge

This book is targeted towards both undergraduate and post-graduate students and instructors who consider using a Unix-like system in a course on the design and implementation of operating systems. At the minimum, readers should be familiar with the following topics:

- Software development with a C-like language, e. g. C++, C# or Java
- Data structures and algorithms
- Basic understanding of *some* assembler language (the Ulix sources contain a few lines of 32-bit Intel assembler code)

Experience with the following topics is not required but considered helpful:

- Standard Unix library functions such as `open`, `read`, `write`, `fork`, `exec` (as they are sometimes taught in a system programming course)
- Unix command line (shell), such as `bash` or `ksh`
- LaTeX document preparation system (required for most of the exercises in the book)
- Software development with C
- 32-bit Intel Assembler language
- Build process in a Unix environment, using makefiles and command line tools for compiling, assembling and linking

1.9 Online Resources

The Ulix website `http://www.ulixos.org/` has a download area with files which are needed for working on the exercises in this book (Figure 1.4).

This is the

First Edition (09/2015)

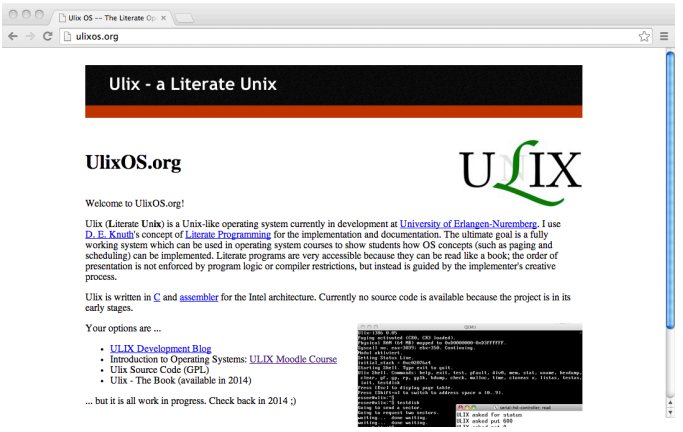


Figure 1.4: Download resources and information about the project are available on the `http://www.ulixos.org` website.

of the book, so at the time of writing the website contained only the resources for this edition, but later editions may use different versions of the download files; so you should make sure that you access the right files.

The two main files you find in the download area are the following:

- Development system: We provide a VirtualBox appliance which you can import in the VirtualBox virtualization software. It contains a Debian Linux 6.0.1 installation with a simple desktop (Xfce) and all the development tools that you need for compiling the current Ulix version (see Figure 1.5). The sources are included as well, and also a few feature-reduced versions of the Ulix source code which you can extend when you work on the exercises. This virtual machine is the recommended development environment.
- The Noweb source file `ulix-book.nw` from which you can generate this book and the Ulix kernel. We recommend this download if all you want to do is regenerate the PDF file of the book. For compiling the Ulix kernel you should use the development system described above, because the source code depends on the right compiler version being installed.

For a detailed description of how to get started, see Section 18.3 which describes how you can set up a development environment. Note that you need no such installation if you simply want to read the book. However, if you want to work on the exercises which you can find in some chapters, you need the tools and sources.

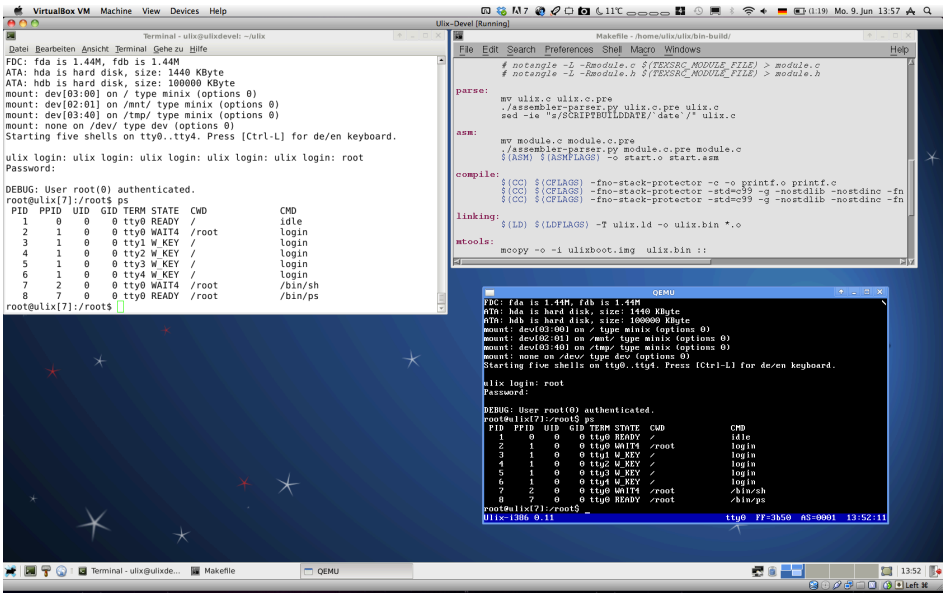


Figure 1.5: The website provides a virtual machine image (a VirtualBox appliance) that can be used for compiling ULIX and working on the exercises.

1.10 Further Reading

There are several educational operating systems which have been used in courses or as the basis for textbooks with a similar purpose as this one. They are all united by the idea that students should take a look at real code in order to fully understand what tasks an operating system has to fulfill and how this can be done in practice. In addition, for some "real" operating systems documentation of design principles and implementation details is available that is also helpful in an educational setting. In Appendix C we give several examples for both categories.

In our view, the two most positive examples of operating system exposition for students are the well-known Minix operating system and the less well-known Xinu operating system:

Minix

- Minix was originally written by Andrew Tanenbaum [Tan87] to serve as a minimal working example for teaching his operating systems course at VU University Amsterdam (*Vrije Universiteit Amsterdam*). In its most advanced form (Minix 3), the system is still well-structured, simple and very well documented [TW06]. However, it has also evolved into a commercially relevant system with all advantages and disadvantages. Although the Minix book [TW06] contains the entire source code of the operating system, it lives a separate life being relegated to an appendix that essentially fills the second half of the book.

Ulix uses version 2 of the Minix filesystem as its native filesystem because Linux supports this filesystem very well. However, no code was borrowed from Minix except for some structure declarations.

- The Xinu system was written by Douglas Comer [Com84] for the DEC LSI 11/2 microcomputer (a successor of the famous PDP 11 minicomputer for which UNIX was initially written). This system was later ported to the IBM PC's Intel 8088 processor [CF88], the Apple Macintosh's Motorola 68000 CPU [CM89] and, quite recently, to the Linksys E2100L Wireless Router [Com11] which uses a MIPS processor. There are even further ports of the Xinu code (e. g. to 32-bit Intel machines), but those have not led to new editions of the Xinu book. While not written in literate programming style, the documentation and presentation of code portions somewhat resemble literate programming. Xinu

1.11 About the Authors

Felix C. Freiling is a professor of computer science at Friedrich Alexander University Erlangen-Nürnberg and heads the Chair for IT Security Infrastructures of the Computer Science department (`http://www1.cs.fau.de/`). Before coming to Erlangen, he was a professor at RWTH Aachen and University of Mannheim. He started the Ulix project during his time in Mannheim where he gave lectures on operating system principles.

Hans-Georg Eßer is a PhD student at the same chair; he studied computer science and mathematics at RWTH Aachen. He is the editor-in-chief of EasyLinux magazine (`http://www.easylinux.de/`) and has been teaching operating systems at several universities of applied sciences since 2006. The completion of this book (and the Ulix implementation) was the major part of his PhD thesis.

2
Layout of the Kernel Code

You will soon see the first lines of code of the Ulix operating system. The presentation of code roughly follows the order in which it was developed and in which you could also work if you wanted to write your own kernel.

When the system runs through its initialization steps, some tasks need to be handled before others, for example we need to get the memory access right before everything else, and we need to establish methods for handling interrupts and faults before we can start talking to hardware components. Here's an overview:

Memory. This is the first thing the OS will face: The boot loader will load the kernel into RAM where the kernel will start executing. But where precisely is that? And how do we have to compile the kernel so that it will work properly in the RAM areas we load it to? (Think of function calls and references to data addresses: We must know at compile time where the kernel's code and data will be located.)

There are different modes of memory usage, and we need our OS to use Virtual Memory so that we can protect the kernel from the processes and the processes from each other.

virtual memory

We need to check in which mode the CPU runs when the kernel is loaded, and then we have to create a transition to the mode we want to use: The latter one is called Paging, and it provides all the protection mechanisms we need while using the memory in a flexible way.

Paging

So, the first implementation chapter deals with loading the kernel and setting up the memory in such a way that the next initialization steps of the OS can work properly. Then we switch to paging which is a needed preparation for introducing processes.

Interrupts and Faults. Before we can start using the hardware, we need to deal with interrupts: several devices use interrupts to tell us that they have completed some activity, and there are components like the clock chip which regularly generates an interrupt.

Closely related to interrupts are faults: they occur when the operating system (or a process) tries to do something that's impossible, e. g. access a memory address that does not exist or for which access is forbidden. We need to treat these, too, because if we don't, then any fault will just make the whole system halt.

The main difference between interrupts and faults is that faults occur as a direct consequence of some specific instruction that our code executes. In that sense they are synchronous. Interrupts on the other hand occur without any connection to the currently executing instruction, since they are not triggered (immediately) by our code but by some device. That is why they are called asynchronous.

synchronous

asynchronous

Handling an interrupt and handling a fault are very similar tasks, so we deal with both in the same chapter. We present a framework which lets us supply handlers as we need them, so for example the concrete interrupt handler for the IDE disk controller will be shown later, but it will use the code that's presented here.

Basic Hardware Support. Once we've established the general interrupt handling mechanism, we can start setting up the hardware. We begin with the clock chip (which generates timer interrupts) and the keyboard (which generates an interrupt each time you press a key). We'll look at further hardware components in later chapters, for example the code for floppy and hard disks will follow after the filesystem chapter.

Processes. Here we introduce one of the most important data structures of the operating system, the process control block (PCB). We also have to revisit the virtual memory code and introduce a mechanism to switch between different page tables—every process will use its own one, since every process has its own virtual memory.

process control block

The following chapter discusses the necessary changes for supporting threads which share an address space if they belong to the same process.

In the chapters that deal with processes, threads and the scheduler you will also see how the system can make a context switch which happens when one process is interrupted and another one continues executing.

context switch

System Calls. The operating system does not only guarantee that the memory areas of the kernel and all the processes are protected against one another, it also disables direct access to the hardware. So if a process wants to open and read a file, it needs a mechanism for calling a kernel function which does just that. The standard mechanism for calling OS functions is a system call. The system call interface somewhat resembles the interrupt handling interface which we've seen earlier. We only explain how the general mechanism works—concrete system calls will be implemented where we need them, for example in the filesystem chapter.

Filesystem. The ULIX filesystem code provides us with a virtual filesystem. With it, we can use several kinds of drives (we'll show the code for floppy disks, hard disks and a virtual device we've named the "serial hard disk"), and we can support several logical filesystem formats: ULIX has a driver for the Minix filesystem which was introduced for the Minix operating system. Linux can use Minix media as well which makes it easier to prepare the disk images we use with ULIX. Code for other logical filesystems (e. g. FAT or NTFS) could easily be added.

Minix filesystem

We will now present the overall layout of the ULIX kernel code. Basically every kernel has to do some essential setup (such as initializing RAM and other components of the machine), then activate interrupts and the process system, create a first (`init`) process and transfer control to that process (which will start further processes). ULIX is no different, but where it differs from other operating systems is that we're going to store (almost) everything in one single C source file, `ulix.c`. If you take a look at other systems you will find a huge collection of source (`*.c`) and header (`*.h`) files and several makefiles which turn the source files into object files (`*.o`) independently before linking them all together into a single binary. We'll also link the kernel binary because we have some assembler code in the file `start.asm` that is translated with `nasm`, but all the rest goes into `ulix.c`. The literate programming approach allows us to combine everything without losing the overview: the pages you look at right now structure the code well enough.

Since C requires functions to be defined before they are called and user-defined types to be declared before they are used in function prototype definitions, we need to make sure that all symbols are known at the right time.

- We start with two code chunks ⟨*constants* 112a⟩ and ⟨*macro definitions* 35a⟩ where we put all pre-compiler (`#define`) statements. — constants and macros
- Then follow ⟨*public elementary type definitions* 45e⟩ (mostly stuff like `typedef unsigned int uint32_t`) and ⟨*type definitions* 91⟩ (for structures). The distinction is necessary since structure definitions use elementary types. — types
- Next come the ⟨*function prototypes* 45a⟩ and the ⟨*global variables* 92b⟩: the latter ones often receive their initial values at the point of declaration. — prototypes and variables
- Finally, the real code starts: with all kernel functions in the chunk named ⟨*function implementations* 100b⟩, and at the end of the file you will find the ⟨*kernel main* 44b⟩ function (`main`$_{44b}$`()`). — implementation

Most of these code chunks "exist" twice, for example ⟨*constants* 112a⟩ and ⟨*public constants* 46a⟩. The code chunks with a *public* prefix will later be included in the user mode library's header file (`ulixlib.h`) or the implementation file (`ulixlib.c`), and this trick allows us to automatically keep data structures and constants synchronized between kernel and user land, and it saves us from having duplicate code chunks for standard functions like `memcpy`$_{596c}$ which are used both in kernel and user level land. — public chunks

So this is the basic structure, with most of the kernel code collected in just one C file, `ulix.c`:

[44a] ⟨*ulix.c* 44a⟩≡
⟨*copyright notice* 24⟩
⟨*version information* 35b⟩
⟨*constants* 112a⟩ ⟨*public constants* 46a⟩
⟨*macro definitions* 35a⟩ ⟨*public macro definitions* 596a⟩
⟨*public elementary type definitions* 45e⟩
⟨*type definitions* 91⟩ ⟨*public type definitions* 142a⟩
⟨*function prototypes* 45a⟩ ⟨*public function prototypes* 454b⟩
⟨*global variables* 92b⟩
⟨*function implementations* 100b⟩ ⟨*public function implementations* 455a⟩
⟨*kernel main* 44b⟩

2.1 The `main()` Function

The `main`$_{44b}$`()` function of the Ulix kernel brings the system up and starts the first (user mode) process, an `init` program which launches several copies of a `login`$_{584c}$ program that in turn start simple shells. This happens on a number of virtual terminals (text consoles), allowing tests with several logged-in users.

[44b] ⟨*kernel main* 44b⟩≡ (44a)

```
void main () {
  ⟨initialize kernel global variables 184d⟩
  ⟨setup serial port 345a⟩ // for debugging
  ⟨setup memory 97⟩
  ⟨setup video 337c⟩
  ⟨setup keyboard 318e⟩
  ⟨initialize system 45b⟩
  ⟨initialize syscalls 173d⟩
  ⟨initialize filesystem 45c⟩
  ⟨initialize swap 293b⟩
  initialize_module (); // external code
  ⟨start init process 45d⟩
}
```

Defines:
`main`, used in chunks 94, 214, 225, 229b, 235f, 247, 248, 311b, 513e, 535, and 623a.
Uses `initialize_module` 45a.

Before all other hardware we initialize the serial port since we use it for debugging purposes; the function `debug_printf`$_{601d}$ sends information to the first serial port, and when we run Ulix in the `qemu` emulator, we can grab that output and display it elsewhere—even before it is possible to write to the (virtual) screen. Note that you will not find code lines with `debug_printf`$_{601d}$ statements in the PDF version of this book, but if you download the sources, you can see them directly in the Noweb file (and also in the generated source code). In order to enable this debugging, the `DEBUG` macro must be defined via `#define DEBUG` (see Chapter 16.3.1).

Setting up the memory is a complex task which we describe in detail in Chapter 4.4. Next we can start accessing the memory buffer of the graphics card to display messages

on the screen, we will define the ⟨*setup video* 337c⟩ chunk in Chapter 10.2.

Since ULIX can be (and has been) used for Bachelor's or Master's thesis projects, we provide a mechanism to integrate the OS code with a module that is implemented by the student; during kernel initialization we will call the

⟨*function prototypes* 45a⟩≡ (44a) 109b▷ [45a]
```
extern void initialize_module ();
```
Defines:
`initialize_module`, used in chunk 44b.

function which may perform further initialization tasks.

⟨*initialize system* 45b⟩≡ (44b) 111b▷ [45b]
```
⟨install the interrupt descriptor table 146d⟩
⟨install the fault handlers 148b⟩
⟨install the interrupt handlers 139b⟩
⟨install the timer 339a⟩
⟨enable interrupts 47b⟩
```

For initializing the filesystem, we first detect the hardware and then print the (currently static) mount table:

⟨*initialize filesystem* 45c⟩≡ (44b) [45c]
```
⟨setup serial hard disk 345d⟩
fdc_init ();  ata_init ();     // register floppy and hard disks
print_mount_table ();
```
Uses `ata_init` 534b, `fdc_init` 552c, and `print_mount_table` 406.

When everything is prepared we can finally enter user mode: We enable the interrupts, load the `init` program and start it as the first process.

⟨*start* `init` *process* 45d⟩≡ (44b) [45d]
```
⟨enable interrupts 47b⟩
printf ("Starting five shells on tty0..tty4. Press [Ctrl-L] for de/en keyboard.\n");
start_program_from_disk ("/init");   // load flat binary of init
// never reach this line!
```
Uses `printf` 601a and `start_program_from_disk` 189.

With the first process being active, initialization of the system is finally complete.

2.2 Type Definitions

Before we introduce the first data structures, we define some elementary data types which make our code more readable. For example, the C language has no specific "byte" and "boolean" data types. Instead, an `unsigned char` is used whenever bytes or booleans are needed. We also define a "word" type.

⟨*public elementary type definitions* 45e⟩≡ (44a 48a) 46b▷ [45e]
```
typedef unsigned char           byte;
typedef unsigned char        boolean;
typedef unsigned short          word;
```

Along with the `boolean` datatype we also provide constants for the two standard values 1 and 0 (note that C regards any non-zero integer value as true). `NULL`$_{46a}$ is a null pointer.

[46a] ⟨*public constants* 46a⟩≡ (44a 48a) 111c▷

```
#define true  1
#define false 0
#define NULL ((void*) 0)
```

Defines:
`NULL`, used in chunks 120, 121, 146a, 164a, 258b, 367b, 369c, 463a, 467–70, 475a, 484e, and 607b.

Sometimes we create data structures with differently-sized components, and in those cases we want to show clearly how "big" each element is. So we also define `uint*_t` types which are standard in many systems, e. g. in the Linux kernel:

[46b] ⟨*public elementary type definitions* 45e⟩+≡ (44a 48a) ◁45e 46c▷

```
typedef unsigned char         uint8_t;
typedef unsigned short       uint16_t;
typedef unsigned int         uint32_t;
typedef unsigned long long   uint64_t;

typedef int                    size_t;
typedef unsigned int             uint;   // short names for "unsigned int",
typedef unsigned long           ulong;   // "unsigned long" and
typedef unsigned long long  ulonglong;   // "unsigned long long" (64 bit)
```

Defines:
`size_t`, used in chunks 420c, 429b, 594, and 596.
`ulong`, used in chunks 109, 340e, and 341.
`ulonglong`, used in chunk 534.

Memory addresses in our code are always 32 bits wide since Ulix is a 32-bit operating system. We intruduce an address type:

[46c] ⟨*public elementary type definitions* 45e⟩+≡ (44a 48a) ◁46b 158b▷

```
typedef unsigned int       memaddress;
```

Defines:
`memaddress`, used in chunks 100, 103, 105b, 108, 111b, 113b, 115d, 151c, 161, 166a, 169, 170, 172a, 173a, 175, 192b, 197, 211–13, 228b, 231, 232, 234b, 255a, 257–59, 279c, 289–91, 515a, 567c, 568b, and 604a.

2.3 Assembler Code

As we will occasionally have to use assembler statements and the standard command in the GNU C compiler `gcc` is `__asm__`, we define a shorthand:

[46d] ⟨*macro definitions* 35a⟩+≡ (44a) ◁35a 101a▷

```
#define asm __asm__
```

We have created an assembler pre-processor which replaces code that has the form of the left side with code that looks like the right side:

```
asm {
  starta: mov eax, 0x1001   // comment
  mov ebx, 'A'              // more comment
  int 0x80
}
```

```
asm (".intel_syntax noprefix; \
  starta: mov eax, 0x1001; \
  mov ebx, 'A'; \
  int 0x80; \
  .att_syntax; ");
```

This allows usage of the Intel assembler syntax (without changing the normal compilation process which uses AT&T syntax), it also enables us to add comments in the code, and the new syntax is closer to C.

The pre-processor also understands `asm volatile`. What it cannot cope with is variable / register usage; thus, occasionally there will be appearances of the less readable standard assembler syntax.

Note that it does not change the number or position of code lines. The source code for the assembler parser is shown on page 624 ff.

2.3.1 Turning Interrupts On and Off

We will often have to disable and re-enable interrupts. The assembler instructions are `cli` (clear interrupt flag; disables the interrupts) and `sti` (set interrupt flag; enables them). Instead of writing `asm("cli")` or `asm("sti")` (which would force you to remember which of the commands turns the interrupts on or off) we provide code chunks for them:

⟨ *disable interrupts* 47a⟩≡ (282c 352 353 357b 383a 390 533b) [47a]

```
asm ("cli");    // clear interrupt flag
```

⟨ *enable interrupts* 47b⟩≡ (45 151c 282c 290b 324b 352 353 357 383a 384b 390 533b 610a) [47b]

```
asm ("sti");    // set interrupt flag
```

2.4 The User Mode Library

Besides the kernel we also need a user mode library which provides some standard features for user mode applications.

The library code consists of two files: `ulixlib.c` contains the implementations of the library functions, whereas `ulixlib.h` provides declarations which have to be included both in `ulixlib.c` and any program that wants to use the library functions. In this section we introduce the code chunks that the library files are made of.

The header file shares some constants and type declarations as well as some generic function prototypes with the kernel:

[48a] ⟨*ulixlib.h* 48a⟩≡
```
// ulixlib.h
// To compile a Ulix program, include "ulixlib.h"
```
⟨*copyright notice* 24⟩
⟨*public constants* 46a⟩ ⟨*ulixlib constants* 207b⟩
⟨*public macro definitions* 596a⟩ ⟨*ulixlib macro definitions* 432a⟩
⟨*public elementary type definitions* 45e⟩
⟨*public type definitions* 142a⟩ ⟨*ulixlib type definitions* 584d⟩
⟨*public function prototypes* 454b⟩ ⟨*ulixlib function prototypes* 174c⟩

2.4.1 Functions of the Library

Some functions are needed both in the kernel and in the user mode library. We put their implementations into the ⟨*public function implementations* 455a⟩ chunk (which is included both by the kernel and the library C files) so that we need not present the code for `memcpy`$_{596c}$, `strncpy`$_{594b}$ and other standard functions twice.

[48b] ⟨*ulixlib.c* 48b⟩≡
```
// ulixlib.c
// To compile a Ulix program, include "ulixlib.h"
```
⟨*copyright notice* 24⟩
```
#include "ulixlib.h"
```
⟨*public function implementations* 455a⟩
⟨*ulixlib function implementations* 174d⟩

So, whenever you see a code chunk that starts with "⟨*public*", you know that it contains declarations or code which will appear both in the kernel and in the user mode library.

2.5 Next Steps

In the following chapter we introduce the theory of memory management—that's a requirement for doing any further steps, since the next Ulix code chunks show you how we load the kernel from the boot manager. But in order to do that, we need to first think about how we want to use the physical memory. Thus, Chapter 3 gives you all the needed theory, and then the following Chapter 4 explains the boot process and the Ulix approach towards memory usage.

3
Managing Memory

Processing power (that is, the computing cycles of the CPU) and memory are the two most important resources for any machine. In the next chapter we will describe how to boot the Ulix operating system, and that procedure will include copying the kernel into the computer's main memory. We need to know where to put it and how to continue using memory.

resources

Doing that properly requires some understanding of the general concepts of memory management and also of the concrete mechanisms provided by the target CPU which, in our case, is the Intel i386. In this chapter we start with an overview of the memory management theory, and the following chapter will present the implementation details of the management solution which we have chosen.

Memory management is all about the question: "How can we make the best use of the available physical memory?" When a computer only needs to execute one single program, there is not much to do. But the invention of multi-tasking led to the task of providing several processes with sufficient memory to store their code and data. The extra requirement of memory protection (processes shall not access the memory areas of other processes or of the operating system) further complicates the task.

There are also many similarities between memory management and filesystem management (which we discuss in detail in Chapter 12): In both cases, a fixed-size resource (the physical memory or the collection of sectors on a disk) needs to be shared.

Note that from a process' point of view, memory is a direct resource and is needed for running the process: If the process' program code is not available in memory (at least partially), it cannot be executed, because the CPU can only execute instructions that are located in RAM. On the other hand, disk space is not something that a process will (directly) need: while access to certain files may be necessary for running a program, it is not needed permanently. But if we look at disk space from a file's point of view, we could

say that a file (in order to exist and allow access to it) needs disk space in a similar way as a process needs memory to run. From that point of view, we can compare a process which has no memory with a disk file that was moved to tertiary storage (e. g., a magnetic tape that is part of a tape collection).

These are some of the concepts that appear in both memory management and filesystems:

Partitioning of resources: On a system that will handle several processes in parallel, memory must somehow be partitioned so that each process can use a fraction of the RAM. Individual cells of memory are exclusive: They can hold precisely one byte of information, and it has to belong to a specific process (or the operating system itself) at any given time. It may not be necessary that a process has memory throughout his whole lifetime (for we will see that concepts such as swapping and paging allow data to be stored on the disk for a while), but at least in those moments when a process is actually excecuted by the CPU, it will have to be given some memory. It may be useful to limit the maximum RAM that a process can access at any given time.

Similarly, if a system allows several files (and possibly directories) to be created, accessed and modified on the disk, disk space has to be partitioned in a way that the disk can hold all these files and present simple means to look up files on the disk and access them. The smallest unit of storage would in theory also be a byte, and such a byte (now meaning the fixed location on disk) can only belong to one file (or to the filesystem metadata) at any given time. As in the memory situation, it may be useful to define a maximum filesize so that no file uses too much of this resource, though most filesystems that implement such limits do this on a per-user basis and not on per-file basis—limiting disk usage per user (or per user group) is called a *quota system*.

Access control: Access to memory locations should always be exclusive to one process (or the operating system), otherwise a process could read or even modify the process memory of a different process which is not advisable, because it would be a source of instability or security problems.

Access to files is also often handled in a way that makes it exclusive to a file owner, typically the file creator (or perhaps some other users, depending of the access concepts a specific filesystem may have). And from the view of processes it may be necessary to restrict file access to only one process (even in a situation when several processes belong to the same user who is also the owner of the file), so that no errors can result from parallel access to a file.

Free space management: Memory and disk usage must be handled dynamically, because processes newly appear and are removed from the system all the time, their needs for memory may change during the process runtime, and also files can be created and deleted as well as grown while the system is active.

In many memory management schemes there will be a list of free memory locations. We will see that is not useful to grant memory access byte-wise, memory will often be partitioned into equal-sized smallest chunks of memory that can be assigned to a process or removed from it. If we call these smalles chunks (say, of size 1 KByte) memory frames, then there will be need of a "free frame list" that knows which frames are currently unused.

In the same way disks aren't typically accessed byte-, but block-wise, a block being a fixed size segment of the disk space. Note also that read and write operations on the raw disk device always transfer a whole block of data and not a single byte (which is why they are called *block devices* as opposed to *character devices*. We will need a "free block list" in order to know which blocks are still available for file storage and which are not.

Methods for administering such free frame lists and free block lists will be similar.

3.1 Contiguous Allocation

In this section we present the most simple methods to distribute memory among processes and disk space among files. Contiguity means that a process gets to use a contiguous (connected) area of memory, there are no "holes" in it which would be memory areas assigned to a different process or not assigned at all. If memory did have such holes, a process would have to keep track of which memory regions it can use and which not.

3.1.1 Fixed Equal Size Partitioning

Consider a computer that has 1 GByte of RAM. If we divide this memory into 1-MByte-sized partitions, then we get 1024 such partitions, some of which will have to be reserved to the operating system itself (see Figure 3.1). Assuming that 1000 "unused" partitions will remain, such a system would allow for up to 1000 processes to be started and held in memory in parallel. Each of the processes will then have its own 1 MByte memory partition, meaning it can use up to this 1 MByte for storing its own program code, stack and data.

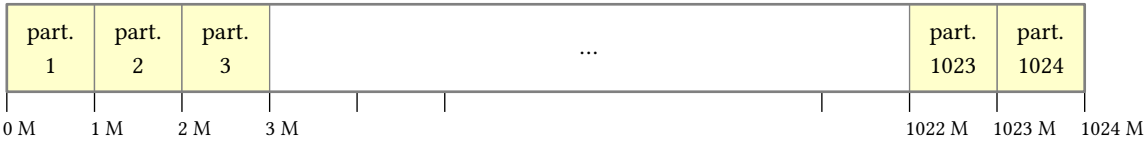


Figure 3.1: Fixed equal size partitioning is the simplest but also the least flexible approach to partitioning memory or disk space.

Obviously this method is not very flexible and it limits the possible usage of the system in two ways:

- No more than about 1000 processes can be run in parallel. If there was need for, say, running 2000 or more processes at the same time, then the whole system would have to be reconfigured (with smaller, but more memory partitions) and completely rebooted.
- No more than 1 MByte of RAM can be given to a single process. If a program required more than that, say 2 MByte or more memory, again the system would have to be reconfigured.
- It would be completely impossible to change the system parameters in either direction, i. e., allow for more than 1000 processes *and* some of them using more than 1 MByte RAM.

Now, in the same way consider a harddisk of size 1 GByte and a similar partitioning scheme that would allow 1024 (minus a few) files of up to 1 MByte size to be written to this disk. The same problems as in the memory example would occur: There would be a filesize limit as well as a limit on the number of files, and changing the filesystem structure in order to either allow more or larger files would require the disk to be newly formatted, and a change would only increase one of these numbers while reducing the other.

This simple approach is called "fixed equal size partitioning" in both the memory and the harddisk case, and besides the limitations already discussed it leads to a problem called *internal fragmentation*: While the RAM is fully split into partitions, i. e., there remains no unpartitioned and possibly unusable memory (that would be external fragmentation), a lot of memory will go unused, e. g. when a process runs that needs only a few kilobytes of RAM but still gets the whole 1 MByte. There is no way for other processes to claim some of this unused memory because the fixed partitioning forbids this (see Section 3.1.6).

internal fragmentation

It is an example of contiguous allocation methods: Contiguity means that all the parts of a process' memory (or of a file on disk) are stored in consecutive frames/blocks, and also in order. So no jumps to other memory locations or disk blocks are necessary when reading the whole file (or the process' whole memory) from the first to the last byte. The opposite of this in non-contiguous allocation, and we will get to that approach in Section 3.2.

Note that we use the term "disk partition" in a non-standard way; we do not mean the logical partitions into which a disk is separated on today's standard computers in order to create several logical volumes (in Windows language: *drives*) each of which is formatted with its own filesystem. For simplicity we assume that a harddisk contains exactly one filesystem and that this filesystem uses all of the disk, as it is the case on floppy disks and (some) USB sticks. See section 3.1.4 for a few words about the classical understanding of "disk partition".

3.1.2 Fixed Variable Size Partitioning

The partitioning scheme that gives all partitions equal size causes the two limitations in file size and file number. A little more flexibility is introduced when we dispense with the equality condition: That leads us to a new method of creating fixed partitions, but of varying sizes (see Figure 3.2).

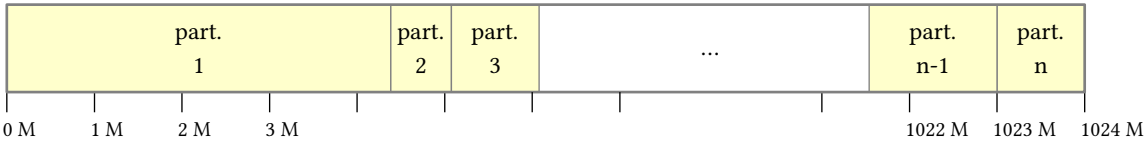


Figure 3.2: Fixed variable size partitioning gets rid of the file size and file number limitations, but still the partitioning parameters cannot change once the system uses the partitions.

It is just a small alteration, but it already improves the situation a lot: In the memory case, if a process is associated with a memory partition and it wants to extend its memory usage beyond the current partition's limits, it can be relocated to a different (larger) partition, and on the other hand many more processes can use the system if there is a good mix of processes with small and large memory demands. Note that this partitioning scheme is still fixed: At system boot-time, the memory partitions are created and cannot be modified until the next booting (and a modification is likely to require a recompilation of the operating system kernel).

In the same way a filesystem with fixed partitions will profit from this modification by allowing both more and (some) larger files. The strictness of the partitioning applies here as well: Once the disk has been formatted, the partition (i. e.: maximum file) sizes can only be changed by reformatting the whole disk.

3.1.3 Dynamic Partitioning

A lot more freedom in memory or disk space allocation is possible if the partitioning becomes fully dynamic: This means that no partitioning occurs at system initialization or while formatting the disk, but instead partitions are created as need for them occurs.

This has the effect that the operating system must carry out a lot more administrative work. For example, keeping an overview of free areas of memory becomes more complicated, because whatever data structures are used for the memory or disk allocation, they are now dynamic.

3.1.3.1 Free Frame Lists

The simplest approach is to keep a list of free locations. This list will be called a *free frame list* in memory management or a *free block list* in filesystems. Typically there is a smallest possible fragment that can be allocated, called a frame or block, and free space

managements only deals with these frames/blocks. The smaller the frames or blocks are, the more of them exist and have to be handled by the free frame list.

One approach is to have a linked list that contains descriptions of free areas, e. g. a start address and a length for each one. In the list each entry points to the next entry when working with pointers. In order to find a free area of a given size an algorithm will walk through this list and stop when it finds an area of sufficient size. For this purpose it may be necessary to scan the whole list if (in the worst case) the only fitting area is at the end of this list. When a number of previously free frames is allocated to a process (or blocks to a file), the list has to be modified,

- either by removing the entry if the whole lot of contiguous blocks are allocated,
- or by modifying the entry if only a few of the blocks are allocated, and they are located at the beginning or end of the area described by this entry,
- or by splitting the entry in two parts, if (for whatever reason) a section taken from the middle is allocated, leaving free areas in front of and behind them.

If the used space is later released, it must be added to the list again, possibly creating a list entry that has to be merged with entries describing directly neighboring areas.

bitmap

Another possibility is to work with bitmaps: For each frame/block a bit in this bitmap defines whether it is free (0) or in use (1). Here no complex list administration (with the mentioned splitting and mergers of list entries) is required, however allocation and release of blocks lead to modification of several bits in the bitmap, and looking up free space of a given size means finding a number of consecutive 0-bits in the bitmap.

Note that it does not matter at all whether we think of memory frames or disk blocks, the concepts are identical. Differences will however appear when thinking of storage of these lists or bitmaps: In the memory case it is obvious that the list must also lie in memory for quick access. In the filesystem case it might make sense to store the list in memory (and not on disk as well) in order to speed up the lookup of free areas—but depending on the size of the free block list, it may be too large to keep all of it in memory.

3.1.3.2 Allocation

When working with dynamic allocation of free areas, there will typically be a choice among several free areas which are of sufficient size, and the procedure for choosing one of them will have consequences both on performance and on *external fragmentation* (an increasing number of small unallocated areas): If the decision algorithm is very complex, allocation will always take a lot of time; if it is simple, there will be many small unpartitioned (not allocated) areas which are too small to be useful anymore, so this external fragmentation will lead to memory or the disk filling up more quickly than necessary.

external fragmentation

On the following pages we will present five simple approaches to allocation called first-fit, next-fit, best-fit, worst-fit and quick-fit; and after that a more advanced concept called the *Buddy System* will be introduced.

First-fit This strategy picks the first free area of sufficient size. It has the advantage of being fast, because once an acceptable area has been found, the system looks no fur-

ther. On the negative side, first-fit leads to a lot of fragmentation and continuously reduces big areas, so that processes which start later and need a big area cannot run.

Next-fit This is a variant of first-fit. The difference is that after every allocation the system keeps in mind the position of this allocated area. The search algorithm then continues immediately behind this area. That way all of memory is being used, whereas first-fit might use only areas in the low-address range if demands can be fulfilled there.

Best-fit The idea behind best-fit is to allocate space in the smallest possible area where the process fits. Assuming that what is left after the allocation is likely to become unusable (because it is too small), this approach tries to minimize the waste of space.

Worst-fit Exactly the opposite of best-fit, worst-fit searches for the biggest free area and allocates space within it. From the perspective of the remaining space, this maximizes the size of the new free area that is left after allocation, hoping that it will still be large enough to allow for further allocations.

Quick-fit is a combination of a first-fit approach with fixed partitioning. A part of the available memory is partitioned into areas of some varying sizes which are often requested. (What is often requested must be known from experiences with memory allocations.) So there will be lists of free partitions of some standard sizes, say, 1 KByte, 2 KByte, ..., 16 KByte. If a memory request of one of those sizes occurs, the system will first try to satisfy it with one of the partitions in the corresponding list. Only if that fails, memory will be allocated from the unpartitioned space using first-fit.

3.1.3.3 Buddy System

The *Buddy System* [Kno65] assumes that we start with a free memory area whose size in bytes is a power of 2. The system reacts to memory requests of arbitrary sizes by repeatedly dividing a free chunk of memory in two halves until a chunk becomes available which is just large enough to satisfy the request. An example illustrates this system better than the description: Let's assume that 1 MByte of memory (1024 KByte) is available at the start. This memory chunk is not partitioned. If a request for 90 KByte arrives, the Buddy System takes the following steps:

- Since the 1024 KByte chunk is too large, it is split into two 512 KByte chunks.
- 512 KByte are still too large, so the first of these chunks is split into two 256 KByte chunks.
- Again, 256 KByte are too large, so another split takes place, turning the first of the 256 KByte chunks into two chunks of size 128 KByte.
- The first of the 128 KByte chunks is chosen since it can satisfy the request. It is marked as used.

<table>
<tr><td colspan="4">1024 KB</td></tr>
<tr><td colspan="2">512 KB</td><td colspan="2">512 KB</td></tr>
<tr><td colspan="2">256 KB</td><td colspan="2">256 KB</td></tr>
<tr><td>128 KB</td><td>128 KB</td><td colspan="2">256 KB</td></tr>
</table>

Figure 3.3: The Buddy System repeatedly splits available space in halves. The chunks with bold face descriptors are chosen for splitting; the green and bold chunk satisfies the request of 90 KByte.

Figure 3.3 shows how the Buddy System partitions the memory step by step when trying to satisfy this request. Afterwards, there are three free chunks left, their sizes are 128 KByte, 256 KByte and 512 KByte. If another request for 90 KByte arrives, the free 128 KByte chunk is chosen; in case of a 40 KByte request, the 128 KByte chunk would be split again. The way in which memory is partitioned can also be represented by a tree; Figure 3.4 shows the tree which corresponds to the situation described above.

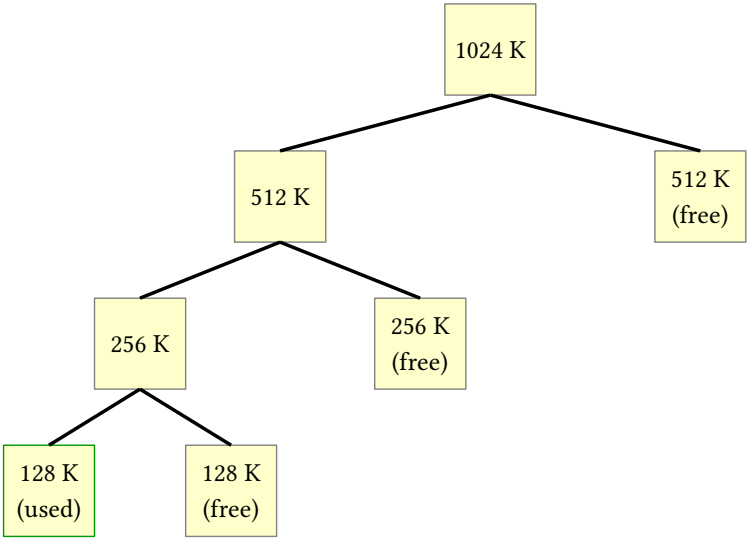


Figure 3.4: The tree representation of memory that was partitioned by the Buddy System shows available memory chunks in the tree's leaves.

When a used chunk of memory is returned and its equal-sized direct neighbor (in the tree) is a free leaf, these two leaves are joined to form a new leaf of twice the size; this process continues, possibly all the way up to the root of the tree. The tree view makes it easier to see which chunks can be joined and which cannot. Figure 3.5 shows an example in which joining is impossible: the second and third 128 KByte blocks are not direct neighbors, they would have to be joined with their left and right neighbors first.

<table>
<tr><td>128 KB</td><td>128 KB</td><td>128 KB</td><td>128 KB</td><td>512 KB</td></tr>
<tr><td>128 KB</td><td colspan="2">256 KB</td><td>128 KB</td><td>512 KB</td></tr>
</table>

Figure 3.5: This is an impossible join operation; a tree representation would show that the two 256 KByte blocks marked bold are not direct neighbors.

3.1.4 A Few Words on Disk Partitions

As mentioned above, we have not been talking about hard disk partitions in the sense of creating several logical volumes on a disk for use by various operating systems (e. g. a Windows and a Linux partition) or for structuring the disk so that different data can be stored on different partitions (e. g. "drives" `C:` and `D:` for Windows or partitions `/`, `/home` and `/usr` for Linux)—now we do, because this kind of partitioning is another example for contiguous allocation with flexible size. Most disks have a partition table as created by Windows, Linux, DOS and other operating systems when initializing a hard disk. (The BSD operating systems use a different method to partition disks, calling the partitions *slices* and the partition table *disklabel*.)

A classical partition table puts no limits on the sizes of individual partitions, but allows only up to four (*primary*) partitions for whose administrative data it reserves space in the first blocks of the disk. There, you basically find the start address and the length of each partition. If more than four partitions are needed, one of the four must be set up as an *extended partition* that holds an additional partition table and the *logical partitions* that reside inside the extended one.

If we ignore logical partitions, we see that this is a simple implementation of dynamical contiguous allocation with flexible size; partitions can be created and deleted, each partition has to be contiguous, and in principle the partitioning also suffers from external fragmentation: If you start with a 40 GByte disk that is partitioned into four 10 GByte partitions and you resize each of them to 9 GByte, you end up with four unused 1 GByte areas that cannot be used. If there was no "four partitions" limitation, the four free areas could be made into four separate 1 GByte partitions, but never into one 4 GByte one, since these four areas are not contiguous.

In order to change the size of a formatted partition (i. e., one with a valid filesystem on it), always two steps are necessary, with their order depending on whether the partition is to be extended or shrunk: The logical filesystem must be resized and the partition itself must be resized.

- When extending a partition, the operation on the partition (and partition table) comes first. Only when this is completed, can the filesystem size be increased as well so that it grows into the newly available space.
- When shrinking a partition, the filesystem has to be modified first, because for example files residing in the parts of the partition that is to be removed must be relocated to a different area on the partition first. Only then can the partition itself be resized (making the removed parts unaccessible to the filesystem).

Note that modifying a filesystem size requires more than (possibly moving files from an area that is to be removed and) changing the information about the partition size in the partition's metadata, for example on a Unix filesystem the free block bitmap has to be grown or shrunk as well in order to correspond to the changed number of blocks.

3.1.5 Segmentation

Segmentation is a memory management technique that requires support by the processor. In general, an address consists of a segment number and an offset (an address that is relative to the physical start of that segment). Segments may (but need not) have a size—if they have, the CPU can check whether access to an address exceeds the boundary of that segment.

base address

The CPU must know the start addresses (often called *base addresses*) and sizes of the segments. That can be achieved by filling a segment table or by loading special CPU registers. Table 3.1 shows an example with three segments on a 64 MByte machine. Figure 3.6 shows the CPU-internal implementation of address calculation for a machine which is not aware of segment limits.

No.	start address	size	absolute range
1	`0x00000000`	`0x100000` (1 MByte)	`0x00000000` – `0x000FFFFF`
2	`0x00800000`	`0x400000` (4 MByte)	`0x00800000` – `0x00BFFFFF`
3	`0x03F00000`	`0x100000` (1 MByte)	`0x03F00000` – `0x03FFFFFF`

Table 3.1: Example of a segment table with three segments.

In the first and the third segment, relative addresses in the `0x00000` – `0xFFFFF` range are valid (since those segments are 1 MByte large), and in the second segment, relative addresses from the `0x000000` – `0x3FFFFF` range can be used (4 MByte).

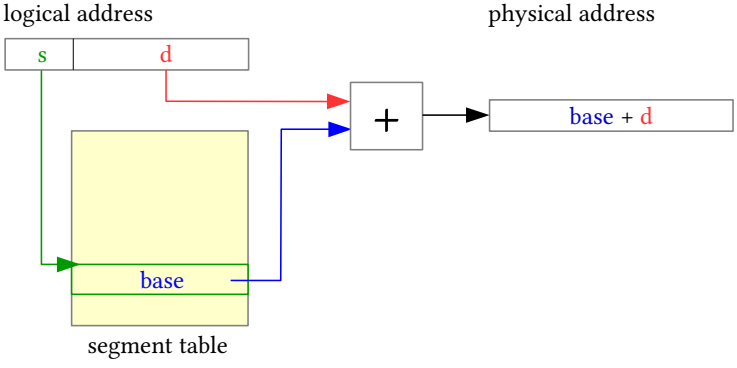


Figure 3.6: Calculating the physical address from a segment number and a logical address is simple: The CPU just adds the segment base address and the logical address.

A complete address is a (segment, address) tuple, for example, (2, `0xABCD`) would refer to the relative address `0xABCD` in segment 2, and its physical address can be calculated by adding `0xABCD` to the segment's start address: `0xABCD` + `0x800000` = `0x80ABCD`.

If limits are not checked, then there is the special case of overflow (when the sum of relative address and base address exceed the maximum addressable value). Consider for example the case of a CPU with a 32-bit address bus (which allows addresses ranging from `0x0` to `0xFFFFFFFF`. If the base address is set to `0xE0000000` and the relative address `0x40000000` is used, the processor will calculate the sum `0x120000000`—which is not a 32-bit value. Overflow occurs, and the 33rd bit is dropped, the resulting absolute 32-bit address is `0x20000000`. You will see an application of this behavior in the next chapter.

Figure 3.7 shows the additional integration of a limit check.

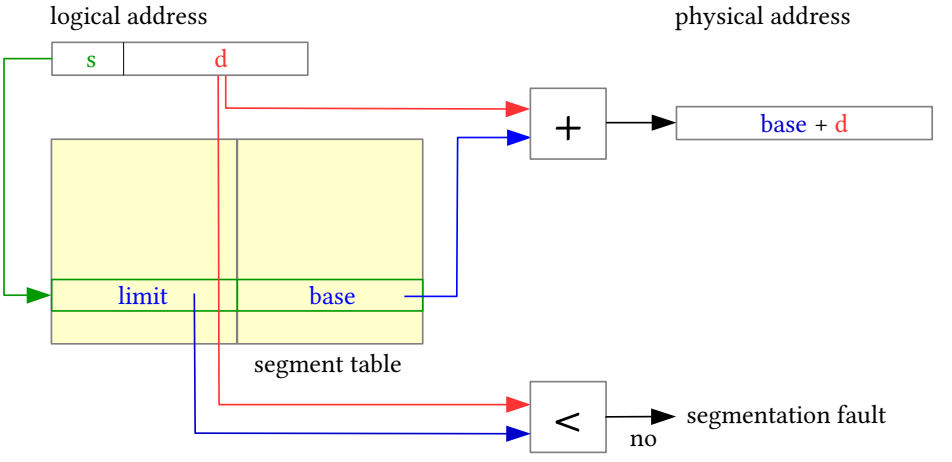


Figure 3.7: An additional limit value guarantees that segment borders are never overstepped.

We will discuss two concrete segmentation mechanisms in the next chapter: The Intel 8086 processor used a segmentation technique that is still available in today's Intel-compatible processors for compatibility reasons (starting with the 80286, it has been called *Real Mode*; see Section 4.3.1), and the Intel 80286 and 80386 processors introduced two (similar) improved segmentation models that were and are used in *Protected Mode* (see Section 4.3.3).

3.1.6 Fragmentation

When memory is allocated dynamically, that can lead to a waste effect called *fragmentation* that we already mentioned. It comes in two varieties:

Internal Fragmentation means that a memory area was allocated that is larger than the actually needed area. For example, in the Buddy System example, the request for 90 KByte was served with a 128 KByte chunk of memory. If we assume that exactly 90 KByte are needed, then an additional 38 KByte were allocated which will not be used by the requester, and they also cannot be used for anything else since they are marked as used (from the allocator's point of view).

External Fragmentation is the effect that can be best seen in the general dynamic allocation systems (e. g., Best Fit or Quick Fit): After a longer sequence of allocating memory chunks and returning them there will be small memory areas which are marked as free, but are too small to be useful. In that case it is possible that the total free memory size is quite large, but even a modest memory request will fail because there is no sufficiently large contiguous chunk of memory.

To summarize the two types, internal fragmentation refers to unused memory that is part of allocated chunks, whereas external fragmentation represents free, non-allocated memory that is too small to satisfy a typical request (see Figure 3.8).

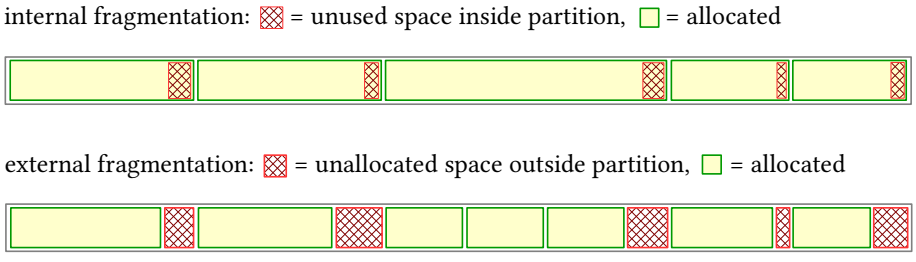


Figure 3.8: Internal vs. external fragmentation.

compaction

External fragmentation can be cured via a process that is called *compaction* (also: *defragmentation*) where the memory management system detects the fragments and reorders the allocated memory chunks so that those fragments disappear; after compaction all (or most) allocated chunks will be located right behind one another, and all free areas will have moved to the end of the physical memory (Figure 3.9). But this approach is costly (because it requires intensive memory copy operations) and it is only applicable if the processes which use the memory can cope with relocation of their memory.

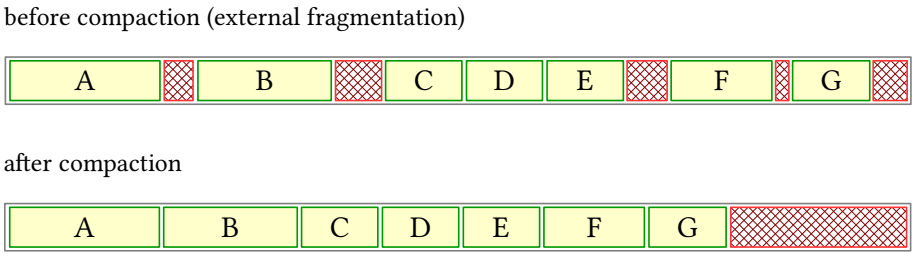


Figure 3.9: Compaction cures external fragmentation but is costly and needs to be repeated whenever the fragmentation level is high again.

3.2 Non-Contiguous Allocation

So far we have seen several examples for contiguously assigning memory or disk space to processes or files. That allows for a very simple handling of accesses, because only an absolute start address and the length of a (memory or disk) partition must be known.

However, since this leads to strong limitations in usability, all modern operating systems use a more flexible approach both for process memory and files, assigning memory frames and harddisk blocks non-contiguously.

Non-contiguous allocation makes things more complicated, and for process memory it is worse than for files:

- If a file consists of several, non-contiguous blocks which are spread all over the disk, there has to be a list of blocks that tells the operating system where to find the data. When trying to read a specific byte from the file, the address within the file has to be translated into a disk block and a relative address inside that block. It also means that reading a file from beginning to end can no longer be achieved by reading several disk blocks in their natural order, but the operating system has to jump from one location to another all the time, so several disk head movements are involved which slows the access.
- With memory things become even more complicated: Here also some kind of table is needed that will be used for address translation, but memory access is different from file access. A program performs a lot of memory access operations: every (absolute) jump to another instruction in the program code, every direct data access (where the content of some memory address is loaded into a CPU register) and similarly each indirect memory access (e. g. `mov [edx], eax`) works with an absolute address.

A process could be told the absolute address ranges of the memory locations it was given access to, imagine it has a list like this:

1. region 1: `0x10000–0x10FFF` (4 KByte)
2. region 2: `0x14000–0x15FFF` (8 KByte)

3. region 3: `0x20000–0x2FFFF` (64 KByte)

Assume further that the program code is 6 KByte long and the rest of the memory will be used for data (we ignore the stack in this simple example). Then the program code will have to be split between the first and second region, and the data between the second and third one. (For this example we also ignore that it might be a better approach to store all of the program code in the second region and use the first and third one for data, especially since the program might not know the precise number and sizes of partitions before it actually gets them.)

If there is a jump instruction in the program code that leads from the front part of the code to the rear part, it will cross region borders. Also when the programs needs to access its data it has to be aware whether the currently needed data reside in the second or in the third region.

address relocation

All these problems can be solved with a method called *address relocation*. When using this system, at compile time a list of address references will be generated. It lists all references to data or instruction addresses that are used within the program code. At load time the system must know the maximum memory demand of the program, assign memory regions and then use the relocation list to adjust the memory references to the concrete region locations.

While this means some overhead during compile and load time, it works quite well, but only for static addresses. If the program dynamically "acquires" memory using some function such as `malloc()`, then this function will also have to be informed about the memory regions and return proper addresses.

swapping

Another problem with this approach occurs when the operating system allows a process to be swapped out to disk (i. e., all of its memory is written to the disk) and swapped in again later: When swapping it back in, the process may be given different regions than before, and then all address references have to be relocated again, this time not only considering the addresses that were stored in the relocation table, but also the dynamically assigned addresses.

The relocation approach makes it hard to protect one process' memory against accesses by another process, because the address calculation in the relocation step only guarantees that static address references are fixed at program start; the program would however be free to access any part of the memory unless the operating system somehow checked each memory access against the region list for this process. The simple start and length registers from the contiguous case would no longer be sufficient, because there are possibly a lot of regions if a process uses much memory.

Note that for files a similar scheme can be adopted, and it causes much fewer problems: typical operations on a file are seek, read and write operations, and they will require translation of linear file positions (thinking of a file's bytes as numbered from 0 to $n-1$ for a file size of n) to absolute disk addresses inside the disk partitions. This translation occurs with every single access to the file. It could be avoided by also using some kind of address relocation as in the memory case, but the gained performance would not be worth the extra effort, because address translation is fast in comparison to disk access.

3.2.1 Virtual Memory

A *virtual memory* is an abstraction of physical memory. Roughly speaking, a virtual memory is an array of memory cells. Usually the size of the virtual memory corresponds to the maximum addressable space allowed by the hardware. A computer may handle multiple such address spaces (and therefore virtual memories) at the same time. They are used to encapsulate effects of programs on memory. Briefly spoken, every program has its own virtual memory and no program can (easily) access the virtual memory of another program.

physical vs. virtual

Physical memory is *physical*, i. e., it consists of hardware circuits that must be produced, bought and installed on the mainboard of the computer. Virtual memory can be created and destroyed on demand. This is its main distinguishing feature from physical memory. Virtual memory is *virtual*, i. e., it is a construct which exists only in software. A computer can have much more virtual memory than physical memory. In such cases, a mechanism "multiplexes" the available physical memory resources to possibly multiple virtual memories.

In this section we will have a look at how virtual memory can be implemented. We will look at the idea of address translation in Section 3.2.2 and sketch the requirements for virtual memory from a user's point of view in Section 3.2.3. We will go through the three historic stages of virtual memory development. In essence, these stages reflect the increasing hardware support for virtual memory in computer architecture. You have already seen the first technique (segmentation, in Section 3.1.5) which provides modified addresses by adding a (segment-dependent) base address to logical addresses. Early approaches basically organize physical memory in a slightly more convenient way, but the transparency of this mechanism is naturally limited. Here, we focus on virtual memory implemented with the help of an external memory management unit (MMU). This approach is the most common one and is also the one chosen in the implementation of Ulix which is described in Section 3.2.5.

3.2.2 Address Translation

address space

The term *address space* refers to a space of addressable units in a computer. Every computer based on the Von-Neumann architecture (named after John von Neumann) has at least one (physical) address space. Its size depends on the size of the address bus. If the computer has 32 bits on the address bus, the hardware can address 2^{32} distinct units of memory. If one such unit is a byte, the architecture supports an address space of 4 GByte.

Having 32 bits on the address bus, addresses are 32-bit values between `0x0000.0000` and `0xFFFF.FFFF`. In physical memory, not all addresses may be backed by real memory cicuits on the mainboard. (Access to such an address usually causes a specific type of interrupt on the CPU or returns an undefined value when it is accessed.) If we view the physical address space of a computer it therefore may have "holes" (see left side of Figure 3.10).

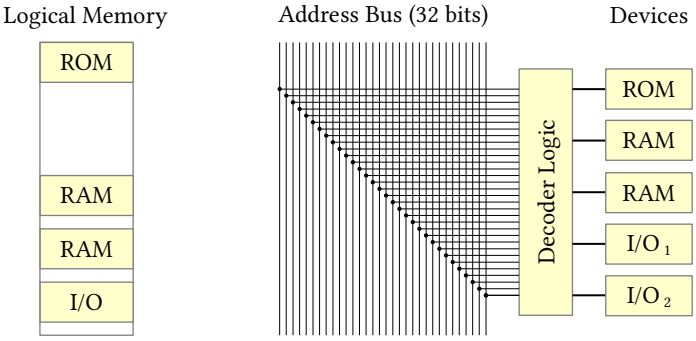


Figure 3.10: Logical view of physical address space (left) and address translation via address decoder logic (right).

address decoder logic

But how does the machine "know" where memory circuits are and where not? The hardware internally stores a mapping of parts of the address space to memory circuits in the form of an *address decoder logic*. This logic is a simple boolean circuit that translates an address on the address bus into one-out-of-n bits. This bit is used to select the particular memory circuit on the mainboard via its *chip select* pin. Only a circuit with an enabled chip select pin will load or store data which travels over the data bus. For example, if the address `0x0000.0000` is put on the address bus, the logic enables (only) the memory chip that is responsible for serving that address (see right side of Figure 3.10).

Not only memory chips can be activated through such a logic. Also external devices can be mapped into the physical address space. Through this mechanism, they can provide their programming registers just like normal memory cells which can be read and written by the CPU using normal load and store commands. This is the basis for *memory-mapped I/O* (which we do not discuss in this book, except for the video adapter's screen buffer).

memory mapped I/O

The effect of such an address decoder logic is that the mapping of memory chips to physical addresses is literally *hardwired* into the system. This mapping cannot easily be changed. Therefore, programming physical memory directly makes it necessary to know the precise whereabouts of the structure of physical memory. This is only advisable where the physical address space is rather small and well-structured. In the old days with less memory, operating systems like MS-DOS could afford to work directly on physical memory: Their programming manuals contained detailed accounts of where RAM and ROM were placed in the physical address space. With today's 32 or 64 Bit desktop systems this is not advisable anymore. It is far better to use a homogeneous *virtual* address space which is independent of the precise placing of memory chips in the physical address space. This can be achieved through an *address translation* step performed before the address decoder logic kicks in.

Address translation needs some form of hardware support. The idea is depicted in Figure 3.11: The *virtual address* put on the address bus is taken and translated by the hardware using some translation table to a physical address which is fed into the decoder logic to se-

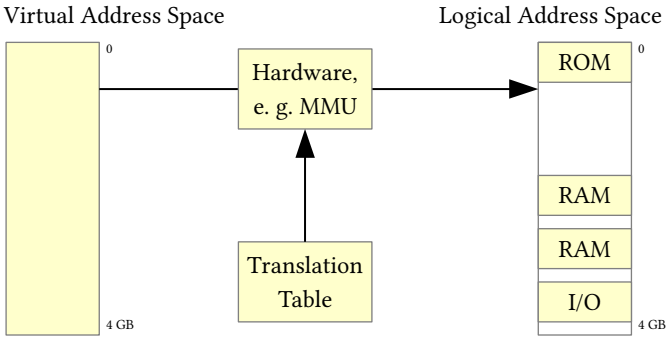


Figure 3.11: Address translation for virtual memory.

lect the right memory chip. This translation requires additional hardware/software effort. But from general experience this pays off quickly given the simplicity and homogeneity of the virtual address space. Program execution can now be performed entirely in the virtual space. An additional advantage is that the address translation is flexible: It can be redefined in *software*.

3.2.3 Virtual Memory Requirements

A *virtual memory* is a homogeneous sequence of memory cells together with their content. It can be regarded as a "well-behaved" address space with its content. The homogeneity is what makes the address space nice: All memory cells are considered to return well-defined values. So in contrast to physical memory there are no "undefined" regions of storage in a virtual memory.

3.2.3.1 Types of Data

A virtual memory completely defines the memory context of a running application. This means that it has to provide all necessary data for executing the program. Three types of data are commonly distinguished:

1. Program code (also called *text*). This refers to all instructions to be executed by the CPU. — text
2. Data. This refers to the contents of all variables used by the program.
3. Stack. This refers to data used to manage subroutine calls. — stack

Usually these different types of data are collected and stored in different regions of the virtual memory.

The data region is further separated into two areas. The first area is for *static data*. Static data are variables and data structures which are known at compile time of a program and exist throughout the execution of the program. Examples of static data are global variables. The second area is for *dynamic data* which is usually called the *heap*. The heap holds — static data — heap

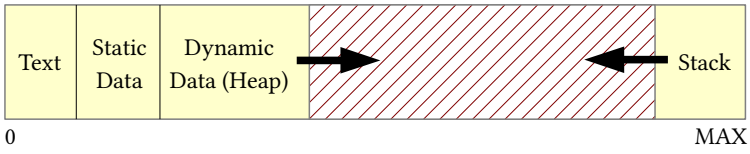


Figure 3.12: Organization of the virtual address space.

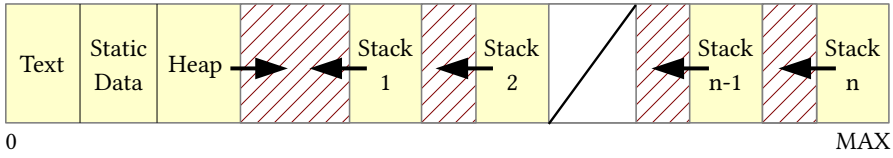


Figure 3.13: Organization of the virtual address space with multiple stacks.

variables which are dynamically allocated at runtime by the program (e. g., using `malloc` in C or `new` in C++ or Java). Data on the heap usually depends on program parameters which are only known at runtime.

For completeness we note that a certain form of dynamic data is also stored on the stack. Compilers often generate code that stores local variables of subroutines on the stack. This is especially noteworthy for recursive functions. Also, parameters are often passed to subroutines via the stack.

3.2.3.2 Address Space Organization

From a user's point of view we would like to organize virtual memory in a clear and tidy way. In practice, text, stack and data areas are located in virtual memory in a fixed order (see Figure 3.12). Starting at address 0 we find the text area with all program code followed by static data. These areas are both fixed in size throughout the lifetime of the program. The remaining part of virtual memory is divided up between the more dynamic parts: heap and stack. The heap is usually placed right behind static data at the "lower" end of the free space. The stack is located on the opposite side. Note that while the heap grows in an intuitive way towards rising addresses, the stack grows rather unintuitively into the direction of falling addresses. In this way, the free space between heap and stack is utilized in the most effective way since any memory cell can either be used by the stack or by the heap. Imagine the alternative where the stack would have been placed "half way" up the virtual memory just to allow it to grow in the direction of rising addresses. In such a case, the free memory cells could only be used by either stack *or* heap.

As we see later in Chapter 7, it may be necessary to provide multiple stacks in virtual memory (for the same program). In such a case we try to utilize the virtual memory as efficiently as possible by dividing up the remaining space into equal parts for each stack. This maximizes the distance between each stack.

Looking at Figure 3.12 and especially Figure 3.13 immediately shows a problem which arises with this memory layout: Dynamic data areas can grow to such an extent that they collide with others. In normal circumstances (i. e., one heap and one stack) this is not a problem because the free space between them is very large. As an example, consider the classic 4 GByte of virtual memory, a 20 MByte program (text and data) and initially empty stack and heap. The gap which opens up between them has a size of 4076 MByte. This is quite some memory to allocate in heap and stack. Of course, the probability that heap and stack collide multiplies with the number of stacks. If a collision is not avoided it usually causes strange and hard to track down runtime errors. As we will see later, it is possible to effectively protect from such collisions with hardware support.

3.2.3.3 Amount of Useable Virtual Memory

Without any additional help, the amount of effectively useable virtual memory cannot be larger than the amount of physical memory installed in the computer. Fortunately, most programs do not really use a lot of the available virtual memory so that you don't always have to equip your system with a full (e. g., 4 GByte) main memory. However, using some tricks it is possible to "simulate" more physical memory using secondary storage. The details of this mechanism will become clear later when we discuss page-based virtual memory. The main idea however is to add special information to the translation table and use main memory as a *cache* for secondary storage. If a part of virtual memory is not in the cache, program execution is interrupted, the missing data is brought into the cache, and the program resumes operation thereafter. Note that if something is brought into main memory in this process, other information may have to be written out of the cache, i. e., from main memory to secondary storage. This performance overhead is the price you have to pay. The advantage of this scheme is that secondary storage hardware is much cheaper than main memory chips. In well-designed systems it is possible to simulate a substantial amount of main memory using secondary storage without much performance overhead.

cache

3.2.3.4 Protection of Code, Data and Stack

We often want to make sure that certain memory areas are only used in the specific way they were intended to. For example, the program code of a process should not be modified, but only executed (which requires only read access). On the other hand, data areas must be changeable, but we don't want a process to treat data as code and start executing it. Some malware works by storing binary code in the stack of a process and then jumping to that code; if the system does not accept a jump to a stack address, this type of attack is impossible. We would also like to differentiate between what we will call *user mode* (a process executing its own code) and *kernel mode* (the process executing a service function inside the kernel) and grant access to specific addresses only when the system is in kernel mode.

user mode

kernel mode

Thus, it makes sense to have access attributes which allow reading, writing and executing and which may also depend on the current (user/kernel) mode. When we set up the memory for a new process we should be able to tell the system which memory areas can be accessed in which ways.

3.2.3.5 Summary of Requirements

To summarize, here are the main requirements we have for virtual memory from a user's perspective:

- Virtual memory should provide a homogeneous address space.
- The size of virtual memory should be independent of the size of physical memory in the system.
- Virtual memory should be able to protect different types of data from certain forms of access (e. g., text from being written).
- Collisions of heap and stack should be detected and avoided whenever possible.

If the system provides multiple virtual memories (one for each program), then we have the additional requirements:

- Virtual memories should be protected from one another, i. e., a program running in one address space should not be able to access the other address space and vice versa.
- The physical resources of the system should be distributed in a fair manner between the existing virtual memories.
- Physical memory should be used efficiently. Especially any type of fragmentation should be avoided.

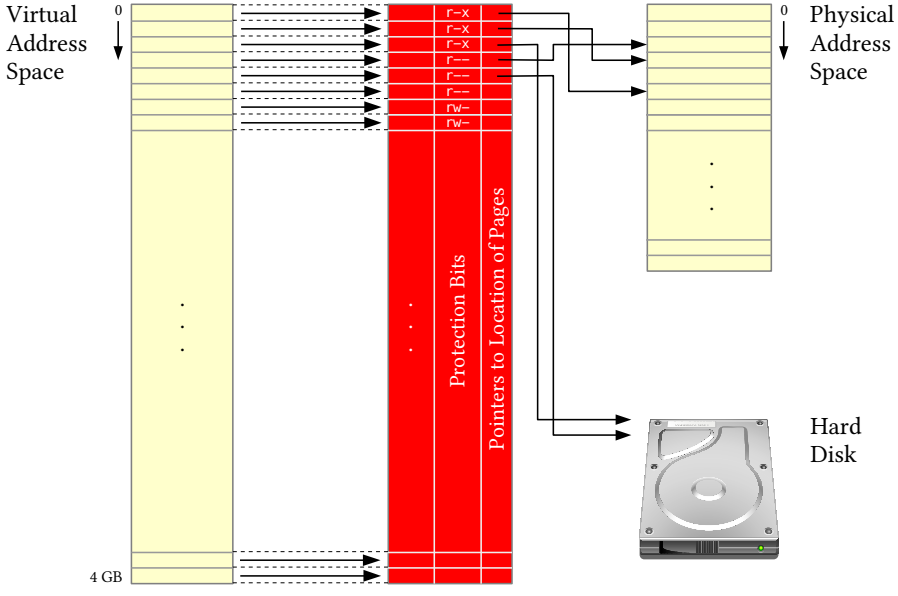


Figure 3.14: Schematic view of a refined translation table for virtual memory.

As a glimpse on to the implementation of the above requirements we return to the idea of a translation table which was previously discussed in Section 3.2.2, this time with some more details. Figure 3.14 depicts the idea of a translation table with the additional information necessary to implement all above requirements. The table not only holds information about the physical address which belongs to the virtual address. It also contains flags that indicate access restrictions (like read, write, execute) as well as pointers to secondary storage should this memory location be stored there. Note that the figure only gives a schematic view which is very simplistic, even impossible to realize. After all, the translation table must somehow fit into (physical) main memory to be useable. If we need one (physical) memory cell (at least) to store the translation information for every (virtual) memory cell, we could never simulate more virtual memory than we have physical memory available. The main challenge therefore lies in implementing this concept in a way such that the usage of memory as well as the translation time is minimized.

3.2.4 Page-based Virtual Memory

All modern operating systems use a *virtual* memory management mechanism called *paging*. The idea behind paging is to give each process a virtual memory space that is addressed contiguously and linearly (starting with an address 0 and ending with an address $size - 1$) and that is partitioned into a set of *memory pages*. All pages have the same size, say, 1 KByte, and they are mapped to *page frames* which are equally sized chunks of the real memory. With the help of a page table each access to a virtual address is translated to a real address by first calculating to which page the address belongs, then looking up the corresponding page frame via the page table and finally locating the relative position within that page frame (see Figure 3.15)—the technical details of this approach are what we discuss in the rest of this chapter.

paging

This approach makes compiling an application very easy: All references to addresses, be they jump instructions or data accesses, can be stored with absolute addresses inside the program, and no relocation takes place when loading the program. Instead each memory address will be translated using the page table.

When, for whatever reasons, locations of page frames have to be changed, it only takes a correction of the page table to make sure that the program continues to be runnable. This scheme also allows for individual pages to be removed from memory altogether and stored on the hard disk for later retrieval—this is called paging as well, and it is not to be confused with swapping a process' memory (meaning: writing all of it to the disk). A process that has some of its pages paged to disk can still be run, whereas a process that was swapped to disk must first be swapped back in before it can resume action. However, the disk space reserved for paging is often called swap space for historical reasons. E. g. the Linux operating system calls paging partitions or files swap partitions and swap files, but it does not implement swapping; it pages.

Although page frames and pages have the same size, they are totally different concepts. A page is a logical unit of virtual memory. Any virtual address resides in some page. A frame is a concrete area of physical memory waiting to hold some page. A large part of

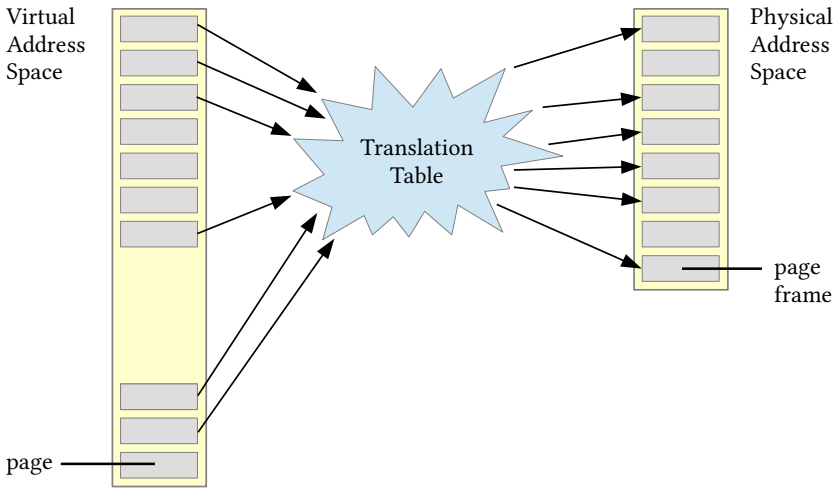


Figure 3.15: Pages and page frames.

physical memory will consist of frames, i. e., will be devoted to storing pages. But not *all* physical memory is part of some frame.

3.2.4.1 Hardware Support

In contrast to segment-based virtual memory, page-based virtual memory needs no hardware support on the CPU, i. e., no special registers. This means that this type of virtual memory can (at least in principle) be implemented with any CPU on the market. In a sense, the hardware support is "outsourced" to a dedicated device called the *memory management unit* (MMU). The MMU can be thought of as a hardware address translator that sits on the address bus and divides it into two parts. One part between the CPU and the MMU is considered the "virtual address" part of the address bus, the other (between MMU and main memory) is the "physical address" part. When the CPU issues a virtual address onto the address bus, the MMU transparently translates it into a physical address on the other side, i. e., it changes the value of the bits as the address passes through the MMU from one to the other side.

MMU

To tell the truth, the MMU doesn't change *all* the address bus bits, but only the higher order bits. The k lower order bits remain unchanged. The value d represented by the k lower order bits is called the *offset* of the address (see Figure 3.16). The idea of this separation is the following: The higher order bits of the virtual address implicitly refer to the *page* that the virtual address is located in. The k lower order bits then are interpreted as the *offset* of the address within the page, i. e., the distance from the beginning of the page to the address.

offset

page number

The interpretation of the address in page number and offset has several consequences. The main one is that the size of a page must be a power of 2. If the k lower order bits

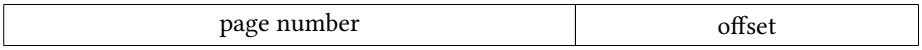


Figure 3.16: Structure of virtual address.

represent the offset within a page, the number of addresses in a page is exactly 2^k. For a value of $k = 10$, a page would contain exactly $2^{10} = 1024$ Bytes. The value of $k = 10$ is a typical value in practice where there are 32 bits on the address bus. This case is depicted in Figure 3.17. It shows that the k lower order bits (those with index 0 to 9) define the offset d and the remaining $32 - 10 = 22$ bits (with indexes 10 to 31) define the page number p.

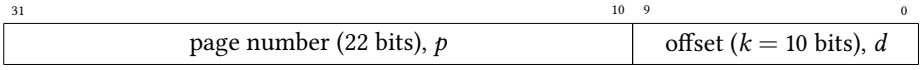


Figure 3.17: An address consists of a page number and an offset. This example uses $k = 10$.

3.2.4.2 Page Descriptors and Address Translation

The central data structure to manage pages in virtual memory is the *page descriptor*. There is exactly one page descriptor per page in virtual memory. All page descriptors are held within the operating system in a big internal table called the *page table*. The page descriptor contains all information necessary to locate the contents of the (virtual) page in physical memory, therefore knowledge of the starting address of the page table is the key to performing address translation. So to enable address translation, the page table register PTR of the MMU is pointed to that starting address (see Figure 3.18). Assuming that there is just one big table, the MMU can now directly locate the page descriptor of page p by doing a small address calculation.

page descriptor

page table

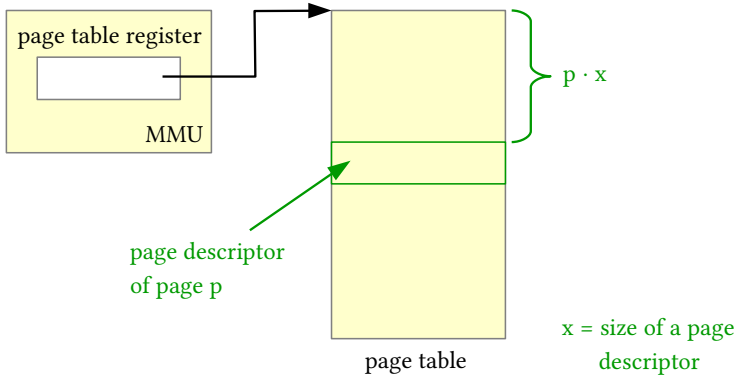


Figure 3.18: Address translation using page descriptors.

Given the size of a page descriptor to be x bytes, then the page descriptor of page p has the address:

$$\text{PTR} + p \cdot x$$

As mentioned above, a page descriptor is a data structure that holds all necessary information to manage the associated virtual page. Here is an overview over the types of information that can be stored in a page descriptor of page p:

- Since the page descriptor is used to perform address translation, it must contain a pointer to the *physical* page frame of page p. The MMU adds the offset of the virtual address to this pointer to yield the actual physical address of the virtual address in question.
- Since page frames act as a *cache* for the contents of pages, certain management bits must be present to handle cache contents. Recall that main memory is regarded as a cache for pages stored on secondary storage (see Section 3.2.3.3). The first such bit is the *presence bit* (P bit). The P bit indicates whether or not the page contents are present in main memory.

presence bit

reference bit

- The next management bit is the *reference bit* (R bit). Roughly speaking, it indicates whether or not the page descriptor was referenced within some period of time. The R bit is set by the MMU with every access to the page descriptor in main memory. Technically speaking, the reference bit is not actually an essential management bit of the cache, but rather a bit which is used to optimize the cache performance. This will be discussed later in Section 9.3.

dirty bit

cache coherence

- A vital cache management bit is the *dirty bit* (D bit), sometimes called *written bit*. It is set by the MMU whenever the contents of page p are written. The D bit is important since it solves the *cache coherence problem*: Contents of the cache (i. e., main memory) and secondary storage can diverge if main memory is written and secondary storage not (due to performance reasons). The D bit indicates exactly when this divergence exists. Pages which have diverged in main memory eventually have to be made coherent with secondary storage again.

protection bits

- The page descriptor also contains *protection bits* used to manage the type of access allowed to this page.
- The page descriptor usually also contains several multi-purpose bits which can be used by the operating system for different means.

Ignoring protection and multi-purpose bits, this is what a page descriptor could look like in code:

[72] ⟨*tentative declaration of page descriptor* 72⟩≡

```
typedef struct page_desc_struct {
  void *frame_addr;               // address of page frame for this page
  unsigned int present    : 1;    // presence bit
  unsigned int referenced : 1;    // reference bit
  unsigned int written    : 1;    // dirty bit
} page_desc;
```

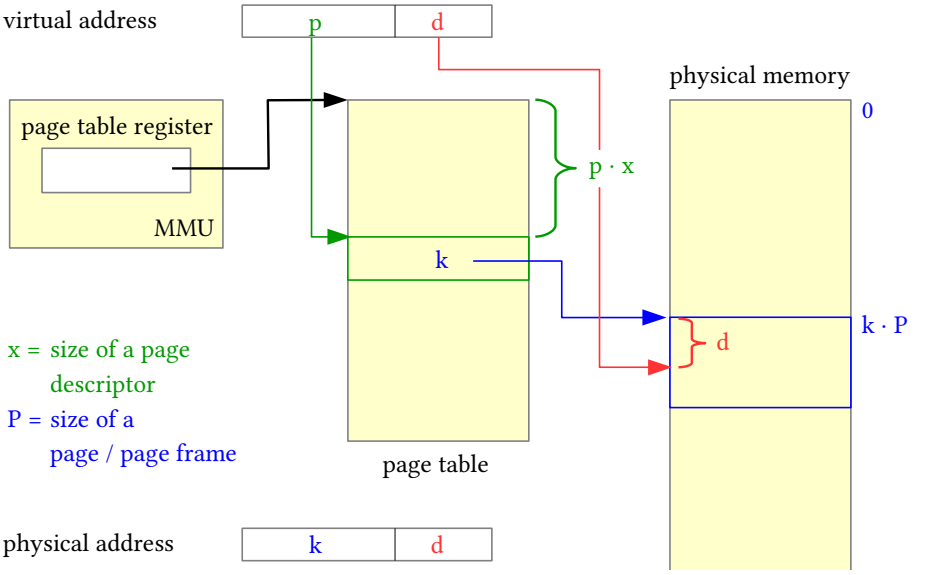


Figure 3.19: Successful page translation.

To summarize, we now recall how a successful page translation finally happens (see Figure 3.19):

1. The CPU accesses a virtual address v on the address bus. The virtual address consists of a page number p and an offset d in the page.
2. The MMU uses the page number p and the base address of the page table (stored in PTR) to locate the page descriptor of the page.
3. Using the page frame number k stored in the page descriptor, the MMU adds the offset d to form the final physical address in main memory.

What can potentially go wrong during page translation? The simplest error condition is that the virtual memory location doesn't (yet) exist in physical memory. This means that the page table doesn't know where it should point to. This information is encoded in a special *null page descriptor*. If the MMU tries to translate an address and finds such a null page descriptor in the page table it signals an interrupt to the CPU which must be handled immediately.

null page descriptor

The next error condition concerns the protection bits. The MMU checks whether the current access context is allowed by the protection bits. For example, if the CPU wants to write something to a virtual address and the protection bits don't allow write access, then again the MMU raises an interrupt with the CPU.

The final error condition which we discuss here refers to the fact that main memory is just a cache for secondary storage: a *cache miss* may happen. What is a cache miss in this context? It means that the page accessed by the current instruction exists but it currently isn't in main memory (the cache). This is indicated by the presence bit (P bit) in

cache miss

the page descriptor. If the P bit is not set, an interrupt is raised by the CPU. In effect the interrupt handler must try and load the page contents back in to main memory so that the application that wished to access the page contents can continue to operate. More details on how this works will follow later.

3.2.4.3 Page Descriptor Trees

In contrast to the naive address translation described at the end of Section 3.2.3 where we had one entry in the translation table per virtual address, the idea of pages reduces the size of the page table dramatically. The larger the page size, the smaller the page table because translation information and protection bits etc. are stored per page. However, page tables still have considerable size. The problem is partly a result of the memory layout sketched in Section 3.2.3.2 because code, data and heap reside on one end of virtual memory and the stack on the other end. This means the page table must *always* cover *all* pages in virtual memory. As an example, imagine you need eight bytes for each page descriptor (which is not much), a 12-bit offset for pages (giving a page size of 4 KByte, rather large) and a

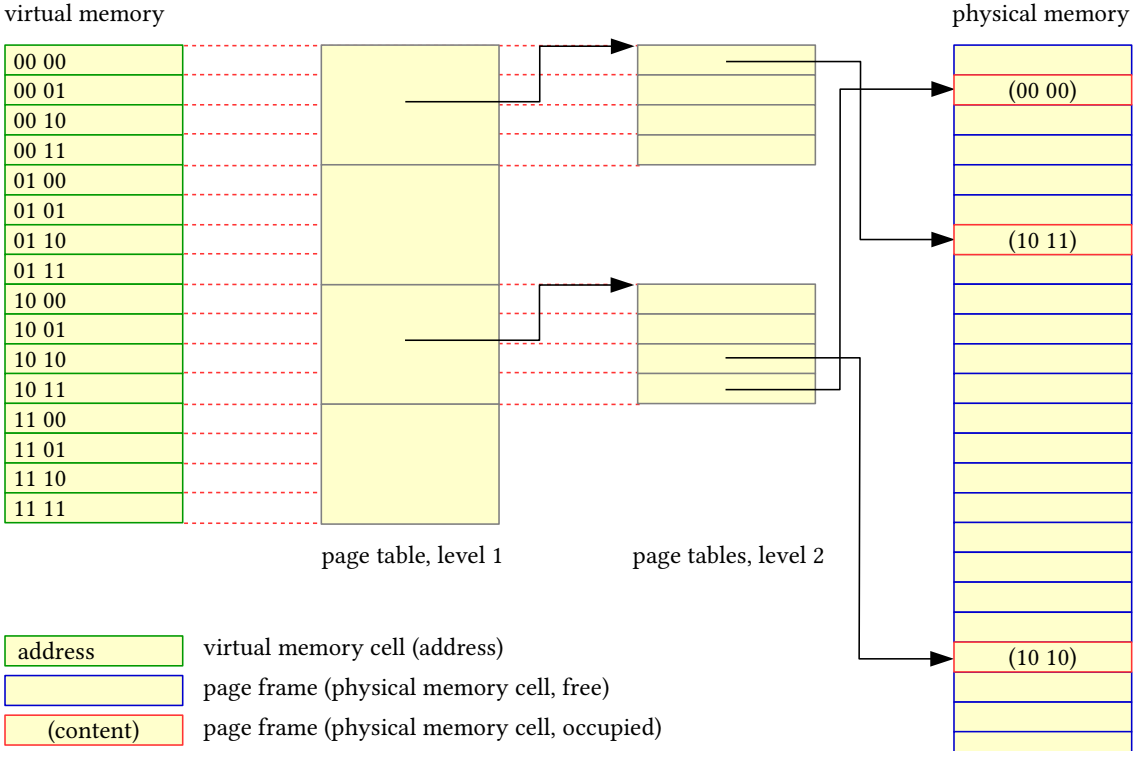


Figure 3.20: Example of a multi-level page table that maps individual virtual addresses to physical addresses (page size: 1 byte).

32-bit address bus. In total you have 2^{20} entries in the page table, each uses eight bytes, yielding a page table size of 8 MByte. Small computer systems with only 16 or 64 MByte of physical memory could only hold one or two page tables (at most) since additionally the *contents* of pages also must be stored in main memory.

Given the fact that most programs have a large void space in virtual memory between heap and stack, there is much potential to save storage space here. Recall the example we discussed in Section 3.2.3.2 where a program of 20 MByte had a gap of almost 4 GByte in virtual memory. If a page had the size of 1 KByte, then this program would need "only" 2000 entries (out of 2^{20}), meaning that more than 99.5 % of the page table is not used.

The common solution employed in operating systems is to have *hierarchic page tables* or *page descriptor trees*. A single page table is regarded as a special case of a hierarchic page table. Each entry in the page table is either a page descriptor or a pointer to another page table.

page descriptor trees

The idea is to start with a small page table (i. e., a page table with a small number of entries). Each such entry is responsible for covering a relatively large part of the virtual address space. Each entry can be refined by another page table in a similar way. For example in Figure 3.20, virtual memory has 16 addresses (numbered $\texttt{0000}_b$ to $\texttt{1111}_b$). The first level page table has four entries. Each entry is responsible for handling a quarter of the entire virtual address space, i. e., four addresses. An entry at the first level then points to a second level page table with again four entries but which deals with the details of address translation. In this example you can already see that a full (single level) page table would have needed 16 page descriptors. In the hierarchic version we need only three small tables with four entries each, i. e., twelve page descriptors altogether.

A real-world example from the Intel i386 architecture shows the effect even more dramatically: The Intel processor uses 4-KByte-sized pages, thus a virtual address is split into a 20-bit page number and a 12-page offset. Using a hierarchical page table, it is possible to split the 20 page number bits in two halves.

Following Intel terminology, the first ten bits are used as index into the *page directory*, and the second ten bits reference a *page table*.

page directory
page table

The first ten bits in this example can be called upper page number, the last ten bits lower page number (Figure 3.21). The effect on the page table size is a reduction to roughly the square root, i. e., from 2^{20} to 2^{10} entries. (If the size of one entry were one byte, the reduced table would have exactly square root size.)

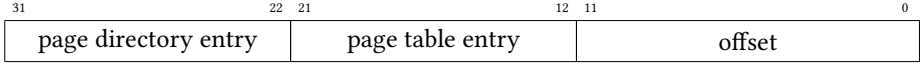


Figure 3.21: On the Intel i386 architecture, paging uses a (per-process) page directory whose entries point to page tables. This constitutes a hierarchical page table with 20 bits for the page number and twelve bits for the offset.

Notice however that while going from a million entries to 1000, this introduces 1000 secondary page tables. If a process actually used this much RAM, all the secondary page tables would be filled, and no space would be saved. (Actually, in that case you get an

increased amount of space used for tables, because the primary table counts extra.) But normal processes will not have such enormous memory demands, and this saves a lot of space because secondary page tables can be created on demand—as long as a process uses only a few kilo- or megabytes of RAM, only the first few secondary page tables need exist.

3.2.4.4 Page Descriptors and Page Table Descriptors

In general, a hierarchic page table is a tree of descriptors. Descriptors can be of two forms:

page descriptor
- *Page descriptors* (PD) are the "leaves" of the tree. They are page descriptors in their original sense, including information about the location of the page frame and protection bits.

page table descriptor
- *Page table descriptors* (PTD) are the "inner nodes" of the tree. They basically are pointers to descriptors (either page descriptors or page table descriptors).

The resulting tree-like structure is visualized in Figure 3.22.

Since we know what is part of a page descriptor already (see Section 3.2.4.2), what is part of a page table descriptor? Generally we can find these entries:

- A flag indicating the *type* of the descriptor, i. e., is it a page descriptor or a page table descriptor. In fact, also every page descriptor needs this field.
- The address of the page table which this page table descriptor points to.

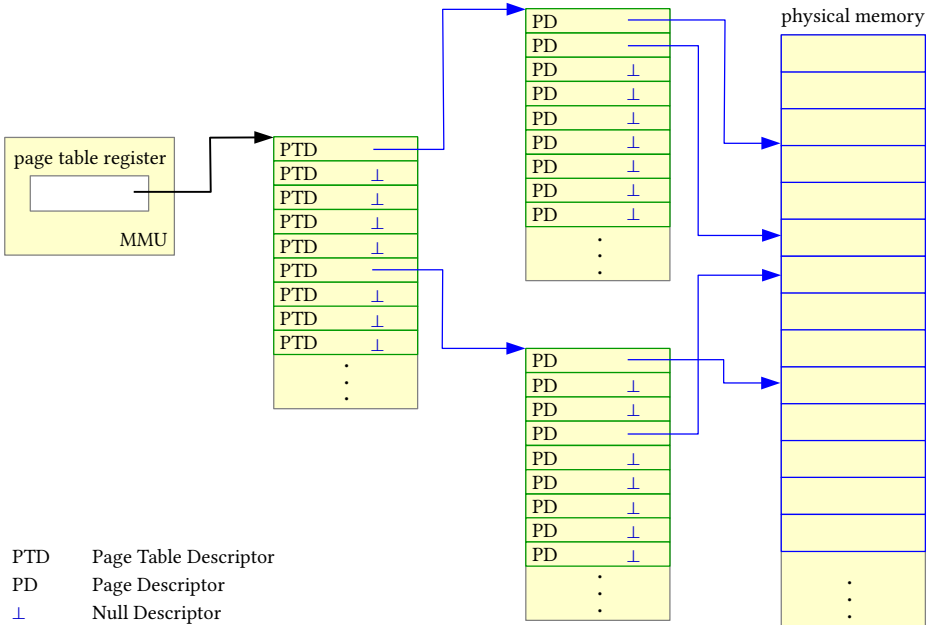


Figure 3.22: Tree structure of descriptors.

- In case it is not clear from the hardware architecture, a page table descriptor may also store the *size* of the page table. This is analogous to the size of a segment in segment-based virtual memory.
- A *presence bit* (P bit) which indicates whether the page table pointed to by the descriptor is in main memory or not. This indicates that also page tables can themselves be paged out into secondary storage. We will get back to the problems this may cause later in Section 3.2.5.
- Multi purpose bits, just like in a page descriptor.

3.2.4.5 Structure of a Virtual Address

In a hierarchic page table, the virtual address is used in a special way to traverse the tree of descriptors. If there are L levels in the page descriptor tree, the bits of the page address p within a virtual address are subivided into L parts $p_1, p_2, \ldots, p_L$ such that

$$p = p_1 \oplus p_2 \oplus \ldots \oplus p_L$$

(where $\oplus$ denotes the join operation of strings). Mathematically, this corresponds to

$$p = p_1 \times 2^{e_1} + p_2 \times 2^{e_2} + \ldots + p_L \times 2^{e_L}$$

for some decreasing sequence of exponents $\{e_1, e_2, \ldots, e_L\}$ with $e_L = 0$. The address translation starts with the leftmost (highest order) bits. Briefly spoken, the first bits (in p_1) are an index into the first level page table, the next bits (in p_2) an index into the second level page table and so on. Therefore the number of bits per level determines the size of the page table at that level.

As an example, consider the division of p into parts in Figure 3.23. The first seven bits (for p_1) allow a first level table size of $2^7 = 128$ entries. In a 32 bit system, each entry covers an area of 32 MByte. The second level's seven bits for p_2 handle again 128 entries, each of them now covering 256 KByte. Finally, the third level's seven bits (for p_3) are an index into a page table with 128 entries, each entry finally covering a full page of 2 KByte (eleven bits remain for the offset). As in this example, it is common to have two or three distinct levels in the page descriptor tree of a virtual address space.

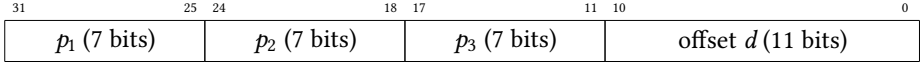


Figure 3.23: Example structure of a multi-level virtual address; $p = p_1 \oplus p_2 \oplus p_3 = p_1 \times 2^{14} + p_2 \times 2^7 + p_3$.

3.2.4.6 Multi-Level Address Translation

We now discuss several examples of how address translation works using hierarchic page tables. The first example is depicted in Figure 3.24. It shows a two-level descriptor tree. How does the MMU perform address translation here? It starts with the "root" page table,

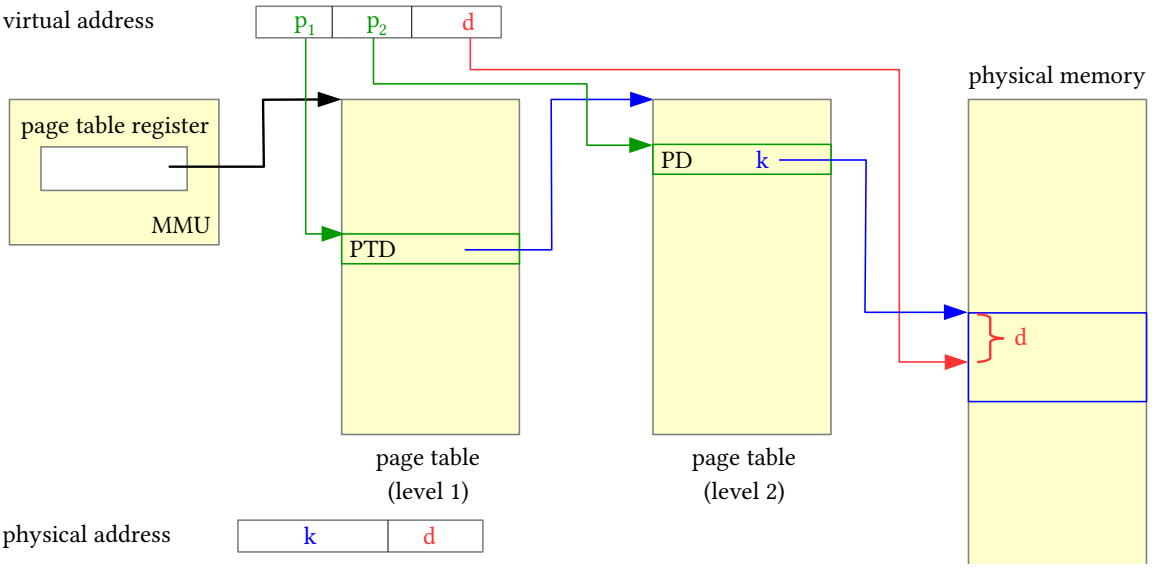


Figure 3.24: Address translation using a two-level page descriptor tree.

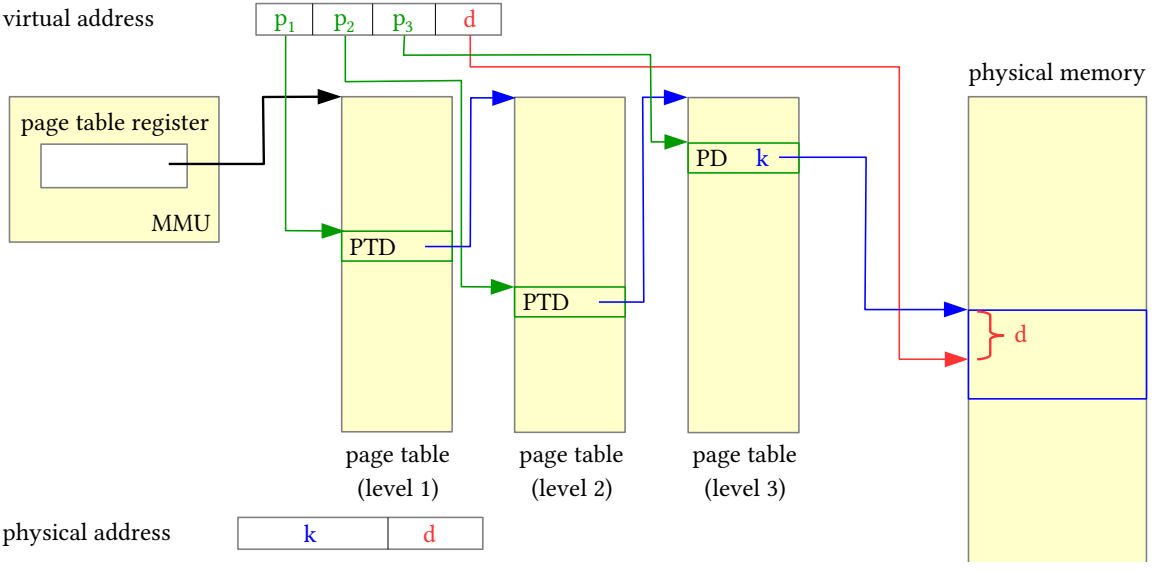


Figure 3.25: Address translation using a three-level page descriptor tree.

pointed to by the MMU register PTR. This is the first level page table. The MMU takes the first part p_1 of the virtual address as an index into this table where it finds a page table descriptor. This points to the relevant second level page table. Within this page table, the second part p_2 of the virtual address is used as an index. Since we are at the highest level of the tree, the descriptor at index p_2 in the second level page table is a real page descriptor allowing to perform the address translation into a physical address.

The example above can easily be extended to three-level page descriptor trees. As an example consider Figure 3.25. In contrast to the first example, the level two page table does not point to the page frame but to a level three page table. The virtual address has a third part p_3 which is used as an index into this table where we find the page descriptor finally pointing to the page frame.

We can save a little storage space in the descriptor if its type is clear from the context. For example, the Intel i386 CPU assumes a two-level descriptor tree. Any descriptor found at level 1 is automatically a page table descriptor. All other descriptors (i. e., those at level 2) must be page descriptors.

3.2.4.7 Discussion

The advantage of hierarchic page tables is their potential to save significantly on main memory. Unused parts of virtual address spaces can be "removed" from the page table using *null page descriptors*. A null descriptor is a special descriptor indicating that the virtual memory at this location is void or unused. Usually it is encoded by a special flag or value in a field of the descriptor (either page descriptor or page table descriptor). By placing a null descriptor into the descriptor tree, all pages below this descriptor are effectively removed from virtual memory. The lower the level of the null descriptor, the larger the part of virtual memory which is mapped out.

To see how effectively null descriptors can be used, we reconsider the example we introduced in Section 3.2.3.2 and revisited in Section 3.2.4.3: the classic organization of virtual memory with a 20 MByte program, an empty heap and an empty stack. Recall that a single page table would need roughly 8 MByte of main memory. Using hierarchic page tables and null descriptors we can reduce the amount of necessary storage to 5 KByte, as is shown in Figure 3.26. Here the example is simplified to the situation where the program has no code and data pages to show the effect more clearly. Remember that on each level of the descriptor tree we had 128 entries per table, each entry requiring eight bytes. This means each table has a size of 1 KByte. By placing null descriptors in all places which point to empty virtual memory, we end up with page tables only for those parts of the system which really exist. Since the assumed hardware of this example places page descriptors only on level 3, we need to extend the descriptor tree up to level 3 for the two page descriptors necessary to point to heap and stack. When you count the number of page tables, you end with 5. Hence we need only 5 KByte instead of 8 MByte.

The disadvantage of a multi-level page tables is that address translation takes slightly longer. This is because the MMU has to traverse the tree (i. e., follow the pointers) when doing the address translation. A multi-level page table needs one main memory lookup per level, which takes longer than a single memory lookup if there were only a single

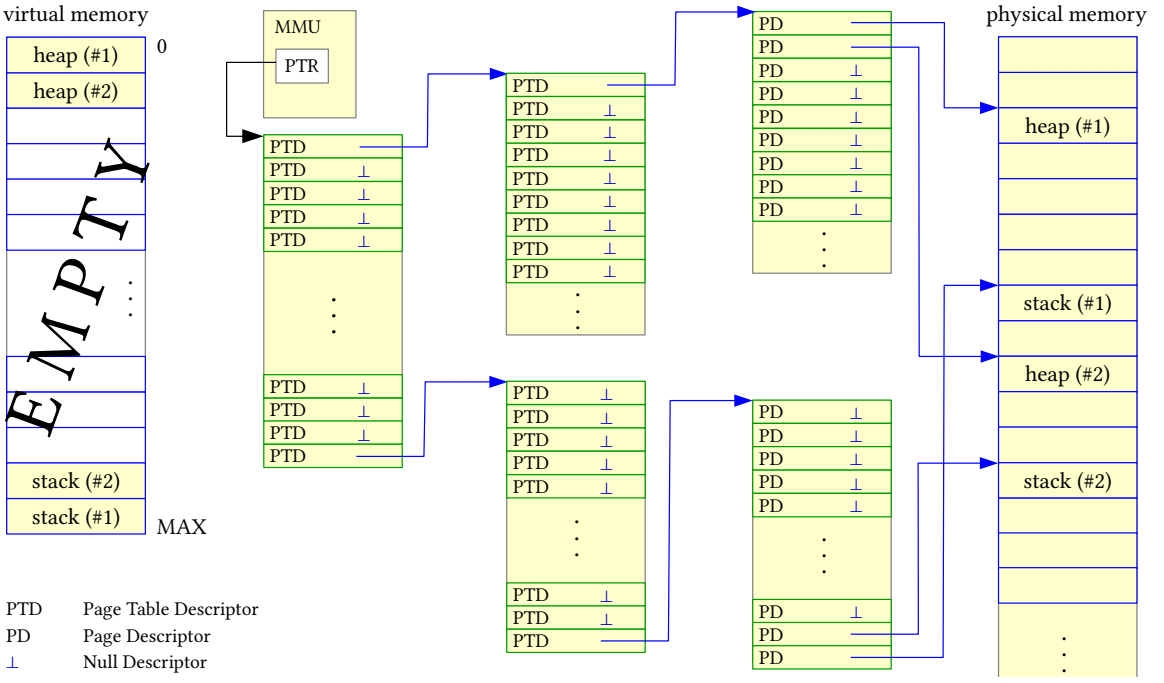


Figure 3.26: Saving main memory using null descriptors in hierarchic page tables.

page table. The memory savings however usually outweigh the performance drawback. Furthermore, performance can still be in the range of a single memory lookup (or less) by using caching, as we explain in the following section.

3.2.4.8 Translation Look-aside Buffers and the Locality Principle

Each memory access requires address translation which needs yet (at least) one other memory access for reading the page table, so it makes sense to use some kind of caching mechanism for the page to frame translation because most programs will not randomly access memory but instead access addresses which are close to one another. Think of loops reading all the elements of an array: they will be stored consecutively. So after one access to a memory frame it is likely that further accesses to the same frame will occur soon after the first one. This is called the *locality principle*. Lookups of the same frame would mean translating the page number to a page frame number again and again—in order to speed up this process many memory management units contain a *translation look-aside buffer* (TLB). That is a special type of memory called *associative memory* which can store page/page frame pairs and allows lookup in constant time: In order to find the page frame for a given page (assuming it is stored in the buffer) there is no need to loop over the entries in the buffer, but the buffer will return the frame number immediately if it contains the page number. If it does not, the result is an error, and the normal lookup process will start. But

locality principle

TLB

finding a frame via the TLB is orders of magnitude faster than going through the regular tables, and this holds even more if split page tables are used.

The size of the TLB is typically very small, because those kinds of chips are limited in their size but the locality principle will guarantee that for "well-behaving programs" (i. e., those that respect this principle) it will be sufficient to dramatically speed up the address translation.

Since the TLB is part of the memory management unit, it will be used automatically by the CPU; no specific programming is necessary to activate or use it.

Note that the (page $\mapsto$ page frame) mapping exists for every single process in the system: Since each process has its own virtual memory space, it makes no sense to combine their page tables in some kind of system-wide table. This has consequences for the TLB as well: If it, as described so far, only stores page and frame numbers, then every context switch to another process will *invalidate* all its entries. So if the scheduler switches processes very often, this will limit the use of the TLB. Alternatively the TLB could be constructed in a way that maps (process ID, page number) pairs to frames: That would keep all entries valid across context switches, but with different processes always accessing different page frames it would only work well in a setup with either very few processes or with a sufficiently increased TLB size.

invalidation of TLB

3.2.4.9 Digression: Indirection in Filesystems

Somewhat similar to the way in which a page table holds information about the page frames currently used by a process, filesystems keep records of disk areas used by a file. A thing that is shared by both methods is the use of equal-sized partitions of the medium—in the case of hard disks they are called blocks or clusters and typically have the size of a few kilobytes, e. g., 1, 2, 4 or 8 KByte.

For each file the operating system has to keep a list of blocks that the file's data occupy. With very large files this list also becomes very large, because a file of size 1 MByte uses 1024 blocks, if the block size is 1 KByte.

Storing the block list in the overall data structure that the operating systems keeps for administering the filesystem is not very efficient, because in order to allow for huge files, each such entry would have to reserve space for a possibly very long list—even for those files that only use a few blocks. Thus many filesystems store the block list in special data blocks. This approach is called *single indirection*: From wherever the information about a certain file is stored, entries do not point directly to a data block, but to an indirection block that contains further pointers to several other data blocks. These entries can be block numbers, since by multiplying the block number with the block size the absolute disk address can be calculated. If it takes two bytes to store a block number and a data block has a size of 4 KByte, then 2048 block addresses can be stored in one indirection block. When the first indirection block is fully used, a second one can be introduced in order to allow for even bigger files.

indirection

Typically the administration data will not only contain pointers to indirection blocks but also a few direct pointers (to data blocks) so that in the case of small files it is possible to find all data blocks without going through indirection blocks. Only when the number

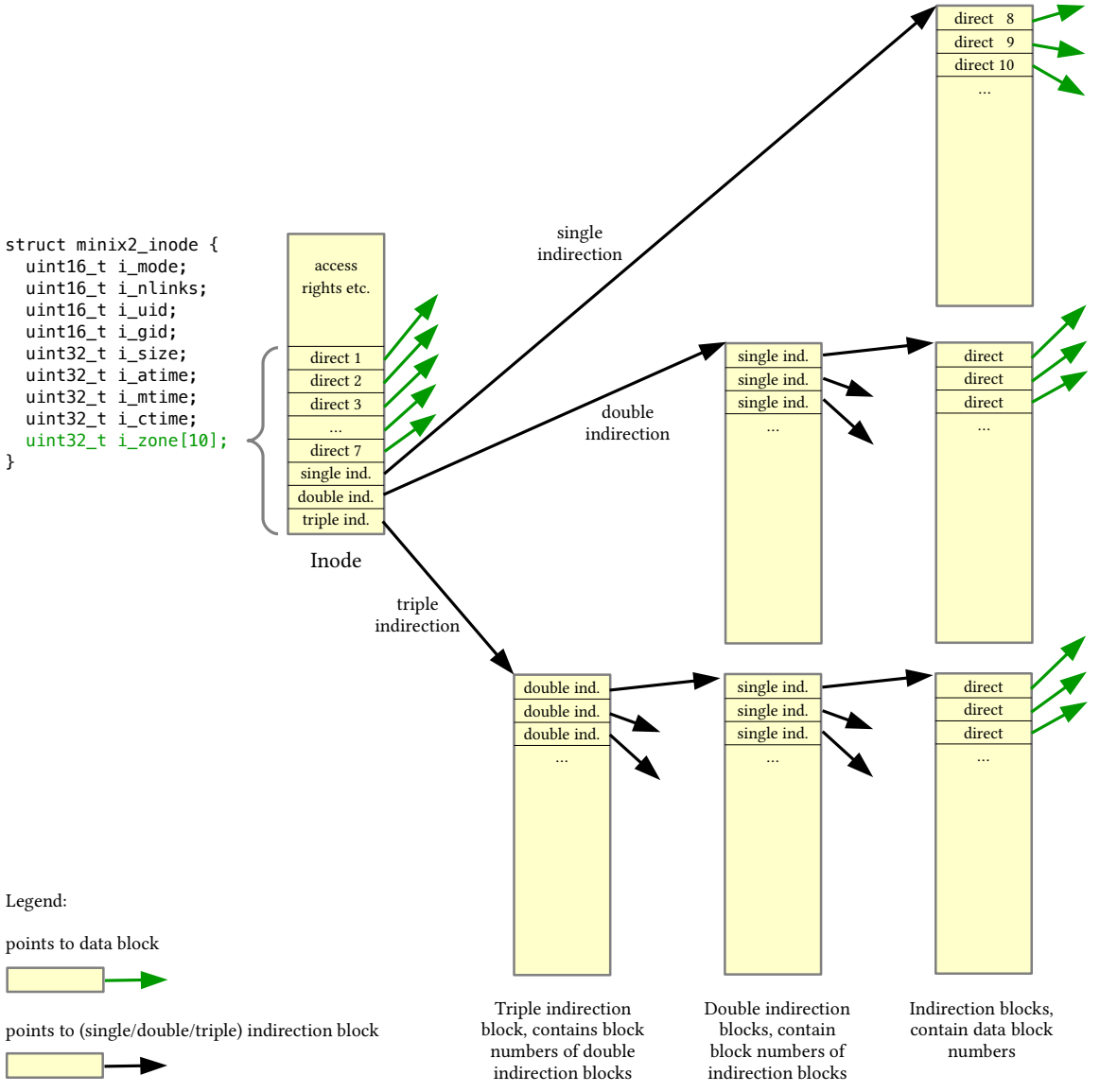


Figure 3.27: Multiple indirection in Unix filesystems: A Minix inode stores seven direct block numbers and three block numbers for single, double and triple indirection blocks. Triple indirection is not implemented in ULIX.

of data blocks exceeds the number of directly stored block addresses, a first indirection block will be used.

With single indirection the maximum size of files grows a lot; however it is still limited: If there are 20 pointers to indirection blocks and such a block stores 2048 block numbers (as above), then this allows for 40 K data blocks or file sizes of up to 40 K $\times$ 4 KByte = 160 MByte. By adding more and more indirect pointers in order to allow for yet bigger files, the administrative data for a single file grows equally; so a second level of indirection is introduced to keep the file entries small. With *double indirection* there are pointers that point to indirection blocks which link to further indirection blocks. Those then finally point to address blocks. What we said about the number of block addresses remains valid in the case of double indirection, but now one double indirection pointer allows to address $2048 \times 2048 = 2048^2$ (or roughly four million) data blocks.

If this is still not good enough, *triple indirection* or even higher levels of indirection can be introduced: With each additional indirection step the maximum file size grows by the same factor (2048 in the example). But notice that it makes no point to use, say, ten or eleven layers of indirection just to be prepared for any possible future demands on file sizes: Indirection leads to extra accesses; in order to read a specific block from the disk whose block number is only available through a long indirection, several blocks have to be read from the disk. If the block is on a triple indirection path, it actually takes at least five read operations to retrieve the data: The first one is for looking up the address of the first level indirection block in the file's administrative data. The second to fourth are for reading the indirection blocks, and the fifth one is the data block itself.

Whatever level of indirection is used, there are typically also indirection entries of all lower levels: In the same way that it makes sense to keep a few direct block number entries to speed up access to very small files, it is useful to have one (or a few) single indirections for those medium size files that do not require double indirection, and so on.

Figure 3.27 shows an example for multiple indirection in a Unix type filesystem. What is called *inode* in the image is a special administrative entry for a file that holds most of this file's attributes including direct and indirect block numbers as well as things such as owner, owning group, access permissions, but not a filename. We will come to this later when we discuss examples of real filesystems in Chapter 12.

inode

3.2.4.10 Further Reading

A comparison of three paging architectures (x86, PowerPC and MIPS) by Bruce Jacob and Trevor Mudge is available online [JM98].

3.2.5 Page-based Virtual Memory in Ulix

Virtual memory in Ulix is designed along the following principles:

- Every process will have its own virtual address space, i. e., an own page table (tree). Address spaces of processes are protected from one another, i. e., it is not possible to access address space *a* from a process that uses address space *b* and vice versa.

- Pages are stored in a set of page frames. Pages can be locked in physical memory. Locked pages cannot be paged out.
- Page replacement is done on a global basis, i. e., page replacement algorithms treat all frames in the same way irrespective of what pages reside in the frames (unless, of course, they are locked).
- The kernel has its own virtual address space. The virtual address space of every process is accessible from the virtual address space of the kernel.

In the following chapter you will see the data structures that ULIX uses to implement paging on the Intel i386 architecture, but we will only set up an initial page table that we need for completing the boot process. It will get more interesting in Chapter 6.1 when we introduce address spaces for processes.

4

Boot Process and Memory Management in Ulix

Our goal is to have an operating system which will be able to boot from some media. Obviously we cannot just compile a standard binary executable file for some platform (e. g. the one we use for development), but instead will need a file format that a boot loader can load and execute on a real machine (or inside some PC emulator or virtualization software).

Writing a boot loader is not a hard task as long as we create the boot disk in such a way that the kernel binary is stored contiguously on the disk and we know its first sector number on the disk: the BIOS provides functions for reading sectors from disk. The older Linux boot loader LILO [AC00] worked this way: after rebuilding the Linux kernel and writing it to the boot disk, the boot loader had to be reinstalled because the sectors containing the kernel were hard-coded into the boot loader. LILO's successor GRUB [Fre05] uses a more complex approach: it contains drivers for several filesystems and can find its configuration data and the kernel without knowing the sector numbers; it just looks up the relevant data in the disk's directories.

GRUB
boot loader

We have decided against implementing our own boot loader because this tool is not part of the kernel. As capable tools such as GRUB already exist, it makes no sense to reinvent the wheel.

4.1 GRUB Loads the Ulix Kernel

We will use a FAT-formatted GRUB boot floppy disk (`ulix-fd0.img`) onto which we copy the Ulix kernel binary `ulix.bin`, and we configure the boot loader by placing the following file `MENU.LST` in the `BOOT/GRUB` subdirectory of the same disk:

[86] ⟨*MENU.LST* 86⟩≡
```
timeout 5

title   ULIX-i386  (c) 2008-2014 Felix Freiling & Hans-Georg Esser
        root (fd0)
        kernel /ulix.bin
```

When we power on the emulated (or real) PC, the BIOS will search for bootable media, and it will find an acceptable boot sector on the floppy disk. It loads the GRUB boot loader into memory and executes it. GRUB understands several filesystems, including Minix and FAT, and it will recognize the FAT disk and locate the above configuration file. Its only entry tells it to load the kernel binary.

But to what memory location does GRUB copy the kernel, and where will it start executing it? The kernel binary will be an ELF file (Executable and Linking Format, see Section 6.8) that contains a description of what data to copy from the ELF file to which memory locations.

ELF

Multiboot header

GRUB supports kernel images which have a *Multiboot header*. The Multiboot specification [OFBI09] states:

> "An OS image must contain an additional header called Multiboot header, besides the headers of the format used by the OS image. The Multiboot header must be contained completely within the first 8192 bytes of the OS image and must be longword (32-bit) aligned. In general, it should come as early as possible and may be embedded in the beginning of the text segment after the real executable header."

In order to put a proper Multiboot header into our kernel binary, we will need to write some assembler code. The `nasm` assembler offers commands such as `db` (data byte), `dw` (data word) and `dd` (data double word) for writing eight bits, 16 bits or 32 bits of data into the file (see Appendix B.3), and the `equ` statement lets us define symbolic constants, similar to C's `#define` pre-processor statement. The full header is only twelve bytes long, its contents are shown in Table 4.1.

Bytes	Content	Fixed Values
00–03	magic string	`0x1badb002`
04–07	flags	
08–11	checksum	

Table 4.1: Contents of the Multiboot header.

What are the proper values for the *flags* and *checksum* fields? According to the Multiboot specification, we need to set the bits 0 and 1 of the *flags* entry:

0: this is the "page alignment" flag, it guarantees that the kernel will be loaded to a physical address which is a multiple of 4 096, the default page size on the Intel architecture.

1: the "memory information" flag provides the loaded operating system with data about the available memory. This data will be accessible via the "Multiboot information structure", and the boot manager must place the address of this structure in the *EBX* register. Ulix does not currently use these data.

The following code shows a slightly modified version of the Multiboot header definition which we took from Bran's Kernel Development Tutorial [Fri05].

⟨*start.asm* 87⟩≡ 93 ▷ [87]

```
[section .setup]
[bits 32]
align 4
mboot:
    MB_HEADER_MAGIC   equ 0x1BADB002
    ; Header flags: page align (bit 0), memory info (bit 1)
    MB_HEADER_FLAGS   equ 11b   ; Bits: 1, 0
    MB_CHECKSUM       equ -(MB_HEADER_MAGIC + MB_HEADER_FLAGS)

    ; GRUB Multiboot header, boot signature
    dd MB_HEADER_MAGIC   ; 00..03: magic string
    dd MB_HEADER_FLAGS   ; 04..07: flags
    dd MB_CHECKSUM       ; 08..11: checksum
```

Defines:
`mboot`, used in chunk 94.

(For an explanation of the assembler commands `equ` (which is similar to C's `#define`) and `dd` (which writes data right into the assembled object file) see Section B.3 of the appendix on x86 assembly language.)

When we look at the compiled kernel, we will find the Multiboot header. Note that 32-bit numbers are stored in *little-endian*: for example, the magic string `0x1badb002` shows up as `02 b0 ad 1b`. little-endian

```
$ hexdump -C ulix.bin
[...]
00001000  02 b0 ad 1b 03 00 00 00  fb 4f 52 e4 17 00 12 00  |.........OR.....|
[...]
```

4.2 The Ulix Memory Layout

While it's too early to discuss the memory management implementation in detail, we need to decide now what the general memory layout is going to be: when we load the kernel it will end up somewhere in memory, so we must say where that is going to be.

Ulix will implement virtual memory (with paging), and every process on the system will have its own address space which is basically a mapping of virtual addresses to physical memory.

We split the available 4 GByte of virtual memory which are available on a 32-bit machine into 3 GByte for user space (addresses `0x00000000` – `0xBFFFFFFF`) and 1 GByte for kernel User / Kernel Space

space (addresses `0xC0000000` – `0xFFFFFFFF`). Every process will see the same upper 1 GByte of kernel space (see Figure 4.1).

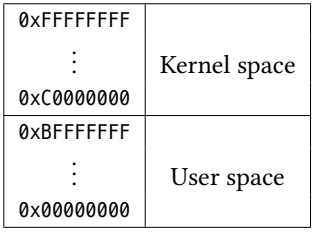


Figure 4.1: This is a simplified view of the Ulix memory layout.

This memory layout is similar to the one used by 32-bit-Linux which also puts the kernel in the upper quarter of the virtual memory.

With this in mind we will compile the Ulix kernel such that it uses addresses above `0xc0000000`.

4.3 From Real Mode to Protected Mode

Real Mode

When the computer powers up, it runs in *Real Mode*, a legacy mode of operation which is compatible to earlier Intel CPU generations—all the way back to the Intel 8086 processor from 1978.

Protected Mode

When using a simple boot loader the operating system has to do the switch from Real Mode to *Protected Mode* manually, however GRUB activates Protected Mode for us, so all we have to do in the early initialization is to set up segmentation properly.

4.3.1 Segmentation in Real Mode

In Real Mode, the CPU uses 16 bit wide registers to address the memory, and via a built-in method called segmentation a 20 bit wide address space can be used. That is achieved via an odd technique: several segment registers can be used to point to a different memory range. Segments are always 2^{16} bytes = 64 KByte large.

Physical Address

Let's see an example for this: Assume we have a machine with 512 KByte of RAM, it has *physical addresses* `0x00000` to `0x7FFFF`. In principle, the 16-bit registers of the 8086 CPU could only access the first 64 KByte (with addresses `0x0000` to `0xFFFF`) of that memory—the first segment. In order to access the second segment (addresses `0x10000` to `0x1FFFF`), we use a segment register, e. g. `DS` (data segment). The segment registers are also 16 bits wide, but we intend to store a 20-bit wide address inside them. Since 20 bits cannot fit inside a 16-bit register, we discard the four lowest bits, assuming they are always 0.

Segment Register

Here are the binary representations of `0x10000` and `0x1FFFF`:

$$\texttt{0x10000} = 10000000000000000_b$$
$$\texttt{0x1FFFF} = 11111111111111111_b$$

The 20-bit wide represention of `0x10000` is $\texttt{00010000000000000000}_b$, and when we remove the lowest four zero bits, we get $\texttt{0001000000000000}_b$ = `0x10000 >> 4` = `0x1000`.

We can now access addresses `0x10000` to `0x1FFFF` by setting `DS` to `0x1000` and actually addressing `0x0` to `0xFFFF`, because the CPU will automatically left-shift `DS` by four bits and add the resulting value to the addresses we supply.

Segmented addresses are always written in the form `seg:addr` and called *logical addresses*, for the example above, `0x10000` = `1000:0000` and `0x1FFFF` = `1000:FFFF`. Logical Address

We could partition the 512 KByte = 8×64 KByte RAM of our example machine into eight segments with addresses `0000:x`, `1000:x`, ..., `7000:x`. The maximum amount of memory that the 8086 CPU can use is 1 MByte, and for a machine with that much RAM we would add the segments with addresses `8000:x`, `9000:x`, ..., `F000:x`.

However, it is not required that a segment starts at a multiple of 64 KByte; instead a segment register may hold any 16-bit value (which will be left-shifted into a 20-bit address with four trailing zeroes), thus the start address of a segment is some multiple of $2^4 = 16$.

Now, for our operating system we do not want to use Real Mode, since it offers no memory protection and 1 MByte of memory is not much. The alternative is Protected Mode, and it also uses segmentation, but in a more complex way. The 80386 processor's segmentation mode is similar to segmentation on the 80286 CPU with the difference that the 80386 supports a 32-bit address space instead of a 24-bit address space, as well as paging.

4.3.2 Privilege Levels in Protected Mode

When we run the PC in Protected Mode there are four different "privilege levels" 0–3 in which the CPU can operate. Protected Mode segments allow us to declare the rights needed to access a segment, for example, if we are currently running in privilege level 3 and try to read a memory address in a segment which only allows access from level 0, our attempt will fail.

Later, when we talk about paging, we will give a more detailed description of the privilege levels; for now it is sufficient to note that Ulix will use two of these levels: level 0 for the kernel and level 3 for user mode applications (processes). We will use the following phrases as synonyms:

- "The system runs in privilege level 0", Level 0 (Kernel)
- "the system is in kernel mode", and
- "the system runs in ring 0".

Similarly for level 3, we treat

- "the system runs in privilege level 3", Level 3 (User)
- "the system is in user mode", and
- "the system runs in ring 3"

as equivalent statements.

4.3.3 Segmentation in Protected Mode

Segment Table

The more modern implementation of segmentation does not store addresses in the segment registers, instead it works with a *segment table* and lets the segment registers point to entries in that table. A table entry does not only specify where a segment starts but also what length it has. If the length is not set to the maximal value (`0xFFFFFFFF`), it is possible to generate a forbidden access (to an address outside the segment) which the CPU will block, generating a fault.

When we start the OS initialization we must provide such a segment table, since it is not possible to use Protected Mode without one: Volume 3 of the Intel (R) 64 and IA-32 Architectures Software Developer's Manual [Int11, p. 3-1] states:

> "When operating in protected mode, some form of segmentation must be used. There is no mode bit to disable segmentation."

Sample assembler code for setting up segmentation can be found in the same document on p. 419 (p. 9-23).

After GRUB turns over control to our kernel, a segment table is in use (after all, we're already running in Protected Mode), however we will discard that table and provide our own one.

GDT

The data structure we have to create is called the "global descriptor table" (GDT) and consists of several *segment descriptors*.

Segment Descriptor

Each segment descriptor is eight bytes long, and besides other values it contains a 32-bit *base* address and a 20-bit *limit* which is left-shifted by 12 bits to form a 32-bit limit. During the shift, 1-bits are inserted on the right, thus for example `0xFFFFF` (20 1-bits) becomes `0xFFFFFFFF` (32 1-bits) during the shift.

Both values are spread in a weird pattern across the descriptor:

- base (bits 0..23): in bytes 2, 3, 4 of the descriptor
- base (bits 24..31): in byte 7 of the descriptor
- limit (bits 0..15): in bytes 0, 1 of the descriptor
- limit (bits 16..19): lower four bits of byte 6 of the descriptor

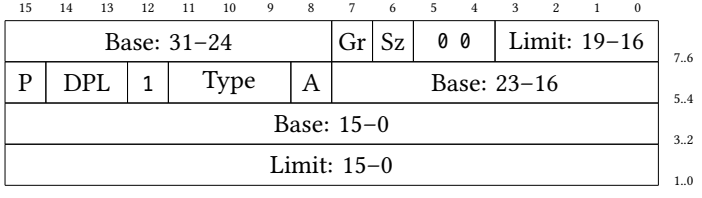


Figure 4.2: In a segment descriptor the base and limit values are spread across the eight bytes in a weird pattern.

A descriptor contains more than just the address range (see Figure 4.2): The upper four bits of byte 6 contain two flags (granularity and size) and two 0 bits;

- we must set the granularity bit to 1: this causes the left-shift for the limit value; if the bit was 0, we could declare the limit in bytes instead of multiples of 4 KByte.
- the size bit must also be set to 1, declaring this descriptor to be a 32-bit protected mode descriptor; otherwise it would be a 16-bit descriptor which is something we don't need (it exists for backwards compatibility with the 80286).

Thus, the upper four bits of byte 6 are always 1100_b.

We have now described everything except byte 5 of the descriptor which defines its type. Its bits have the following functions:

7: present bit, must be set to 1

6/5: descriptor privilege level (DPL) , must be set to 00 for ring 0 (kernel mode) or 11 (=3) for ring 3 (user mode) DPL

4: reserved, must contain 1

3: executable bit, we will set this to 1 in our code segment descriptor and to 0 in our data segment descriptor

2: direction bit / conforming bit:

- for the data segment, 0 means that the segment grows upwards;
- for the code segment, 0 means that the code in this segment can only be executed if the CPU operates in the ring that is declared in bits 6/5 (privilege level).

1: readable bit / writable bit: we always set these to 1; for a code segment it means that we can also read from this segment, and for a data segment it means we can also write to it.

0: accessed bit: we set this to 0; the CPU flips it to 1 when this segment is accessed.

The corresponding C datatype for a GDT entry is the following:

⟨*type definitions* 91⟩≡ (44a) 92a▷ [91]

```
struct gdt_entry {
  unsigned int limit_low   : 16;
  unsigned int base_low    : 16;
  unsigned int base_middle :  8;
  unsigned int access      :  8;
  unsigned int limit_high  :  4;
  unsigned int flags       :  4;
  unsigned int base_high   :  8;
};
```

Defines:
`gdt_entry`, used in chunk 110a.

When we tell the processor to use our table we cannot directly point it to the beginning of the GDT (e. g. via $\texttt{gdt}_{92b}\texttt{[0]}$), instead we need an extra data structure which just contains the size and the start address of the table:

[92a] ⟨*type definitions* 91⟩+≡ (44a) ◁91 100a▷

```
struct gdt_ptr {
    unsigned int limit       : 16;
    unsigned int base        : 32;
} __attribute__((packed));
```

Defines:
`gdt_ptr`, used in chunk 92b.

With `__attribute__((packed))` we force the compiler to store the data precisely in this way, otherwise optimizations could change the order.

[92b] ⟨*global variables* 92b⟩≡ (44a) 105a▷

```
struct gdt_entry gdt[6];
struct gdt_ptr gp;
```

Defines:
`gdt`, used in chunks 109c, 110a, and 196a.
`gp`, used in chunks 110 and 608.

Note that $\texttt{gdt}_{92b}$ allows us to store six segment descriptors; this is the number of descriptors we will use in Ulix—in general, many more descriptors (up to 8192) can be used.

Since we want to create segment descriptors for kernel mode (ring 0) and we need one code and one data selector, the type bytes will be

- $\texttt{10011010}_b$ for the code segment
  (present; ring 0; fixed-1; executable; exact privilege level; allow reading; not accessed)
- $\texttt{10010010}_b$ for the data segment
  (present; ring 0; fixed-1; not executable; grow upwards; allow writing; not accessed).

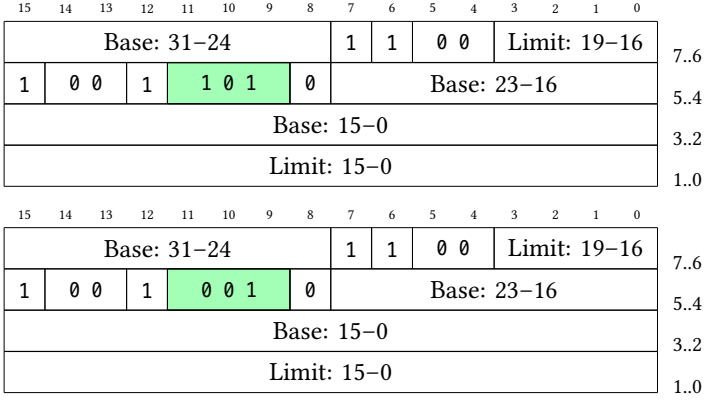


Figure 4.3: Our descriptors for the code segment (top) and the data segment (bottom) only differ in the Type fields.

4.3.4 Preparations for Paging

Without delving into the details of paging, we need to prepare our kernel for its usage right now: Later, when we turn on virtual memory, we want the kernel to use virtual addresses `0xc0000000` – `0xffffffff` and user mode programs to use virtual addresses `0x00000000` – `0xbfffffff`. This means that we have to compile and link the kernel in such a way that all absolute addresses (of functions and data) lie in the range above `0xc0000000`. But that's a large number, and we do not expect to have physical memory with such high addresses. By setting the base address in our descriptors to `0x40000000`, we can load the kernel to low addresses.

Imagine for example that the kernel calls a function which has the entry address `0xc0001234`. With the way we will set up the segments the base address `0x40000000` will be added, resulting in

```
    0xc0001234
+   0x40000000
=  0x100001234
```

which is no longer a 32 bit address; the leading 1 will be lost in this addition, resulting in the address `0x00001234`—where we'll physically put the code of this function.

This is called the "higher half trick" [Son07, Rob01] (even though we're reserving the upper quarter and not the upper half of the virtual memory for the kernel). Higher Half Trick

⟨ *start.asm* 87 ⟩+≡ ◁87 94▷ [93]

```
[section .setup]

trickgdt: dw gdt_end - gdt_data - 1        ; GDT size
          dd gdt_data                      ; linear address of GDT

gdt_data: dd 0, 0    ; selector 0x00: empty entry

          ; code selector 0x08:
          ; base 0x40000000, limit 0xFFFFF, type 10011010, flags 1100
          db 0x0F, 0xFF, 0, 0, 0, 10011010b, 11001111b, 0x40

          ; data selector 0x10:
          ; base 0x40000000, limit 0xFFFFF, type 10010010, flags 1100
          db 0x0F, 0xFF, 0, 0, 0, 10010010b, 11001111b, 0x40
gdt_end:
```

Defines:
`trickgdt`, used in chunk 94.

Now, in order to actually do the memory initialization, we need to load the global descriptor table and activate it. Again, we use code from Bran's Kernel Development Tutorial [Fri05]:

[94] ⟨*start.asm* 87⟩+≡ ◁93 95a▷

```
global start
[section .setup]
start:    ; BEGIN higher half trick
          lgdt [trickgdt]
          mov ax, 0x10
          mov ds, ax
          mov es, ax
          mov fs, ax
          mov gs, ax
          mov ss, ax
          jmp 0x08:higherhalf   ; far jump to the higher half kernel

[section .text]
higherhalf:    ; END higher half trick
          mov esp, _sys_stack   ; set new stack
          push esp              ; save ESP
          push ebx              ; address of mboot structure (from GRUB)

          extern main           ; C function main() in ulix.c
          call main
          jmp $                 ; infinite loop
```

Defines:
`start`, used in chunks 95b and 620b.
Uses `_sys_stack` 95a, `main` 44b, `mboot` 87, and `trickgdt` 93.

This code does the following things:

`lgdt`

- It loads our descriptor table via the `lgdt` instruction,
- it sets the segmentation registers *DS*, *ES*, *FS*, *GS* and *SS* to `0x10` (thus making them point to our data segment descriptor which has index `0x10`),
- since it cannot directly write a value to the *CS* register, it makes a far jump. The instruction `jmp 0x08:higherhalf`$_{94}$ jumps to the address `higherhalf`$_{94}$ in the segment specified by the segment descriptor with index `0x08`—and this automatically sets *CS* properly.
- Then it defines the stack we will use during system initialization (by loading *ESP*) and finally calls the `main`$_{44b}$`()` function from our C file `ulix.c`.

Note that most parts of the code "live" in the `.setup` section of the code, whereas the last lines (starting with the `higherhalf`$_{94}$ label) live in the `.text` section.

We will also need an additional stack, and we reserve its place in the same assembler file. The `resb` instruction does just that: it reserves a certain number of bytes, in our case 32×1024 for 32 KByte of stack memory. Since a stack grows from higher to lower addresses (downwards), the label `_sys_stack`$_{95a}$ follows after the reserved bytes:

⟨*start.asm* 87⟩+≡ ◁94 110b▷ [95a]

```
global stack_first_address
global stack_last_address

[section .bss]
stack_first_address:
    resb 32*1024        ; reserve 32 KByte for the stack
stack_last_address:
_sys_stack:
```

Defines:
_sys_stack, used in chunk 94.
stack_first_address, used in chunk 604.
stack_last_address, used in chunk 604.

We use separate sections because we will tell the linker `ld` to use different addresses for those sections. This can be achieved with the following linker configuration file `ulix.ld`. (This linker file is a modified version of the one provided in Bran's Kernel Development Tutorial [Fri05]; we changed the output format to `elf32-i386` and the start address to 0 and introduced an offset of `0xc0000000` for the main parts of the kernel.)

⟨*ulix.ld* 95b⟩≡ [95b]

```
OUTPUT_FORMAT("elf32-i386")
ENTRY(start)
phys = 0x00100000;
virt = 0xC0000000;
SECTIONS {
  . = phys;

  .setup : AT(phys)       { *(.setup) }

  . += virt;

  .text : AT(code - virt) { code = .;
                            *(.text)
                            *(.rodata*)
                            . = ALIGN(4096); }

  .data : AT(data - virt) { data = .;
                            *(.data)
                            . = ALIGN(4096); }

  .bss  : AT(bss - virt)  { bss = .;
                            *(COMMON*)
                            *(.bss*)
                            . = ALIGN(4096); }
  end = .; }
```

Uses start 94.

The file mainly accomplishes two things:

- it tells the linker `ld` to use addresses in the address range starting with `0x100000` (= 1 MByte mark) for everything that we have declared as part of the `.setup` section, and it will also cause the boot manager to load the whole kernel to `0x100000`.

  The `.setup` region contains the code that runs before paging is enabled. It prepares the segment table and switches it on.
- the line `. += virt;` lets the linker use modified addresses for all the other sections: wherever an absolute address occurs in a CPU instruction, it will get `0xC0000000` added to its original value. (Technically, it adds `0xC0000000` to the current output location counter "`.`": if the last linked instruction ended on position `0x105555`, linking would continue with address `0xc0105556`.)

  If this line was not followed by

  ```
  .text : AT(code - virt) { code = .;
  ```

  it would have the effect to generate code and data which would be loaded at addresses beyond `0xC0000000`, but the `AT` statement says that it shall be placed in a different location: `code` is set to `.` which is the current location, and `AT` calculates `code - virt` which is just the next address behind the `.setup` section. The consequence is that the `text` section will be loaded within the second MByte of physical RAM, and—without enabling paging—it will not be executable in that place because all addresses in the code will have an additional offset of `0xC0000000`. This must later be corrected before jumping into that section. Our segment table does just that.
- `.text` is the code section, it will contain everything that gets executed (except for the parts in `.setup`).
- `.data` and `.bss` contain program data structures which can be read and written, but not executed. The difference between the two is that variables in `data` have an explicit non-zero initialization in the code, whereas the ones in `bss` do not—the linker initially sets them up with zeroes.

This means: for code in the `.setup` section, physical addresses are identical with the addresses used in that part of the binary. That does not hold for the rest of the kernel, where all addresses are increased by `0xC0000000`—which would normally render that part of the code useless, but our segmentation trick with a base address of `0x40000000` makes it just right.

The `.ALIGN` statements force the linker to align each section to the start of a page (or page frame) of size 4096. You can find more information about the linker in the `ld` manual [CT04].

When we inspect the linked kernel binary `ulix.bin` with the `objdump` tool, we see what happens:

```
$ objdump -h ulix.bin
ulix.bin:     file format elf32-i386
Sections:
```

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .setup        00000049  00100000  00100000  00001000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         00012fa0  c0100060  00100060  00001060  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data         00002000  c0113000  00113000  00014000  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00189000  c0115000  00115000  00016000  2**12
                  ALLOC
[...]
```

The columns `VMA` and `LMA` display the first virtual memory address and the load memory address of each section. The first one shows what memory location the code was prepared for, and the second one shows the absolute load address, i. e., where the boot loader will store the section in RAM.

As we've already described, our code starts executing in the `.setup` section, where it sets up the GDT and enables it; then it continues execution in the `.text` section.

4.4 Virtual Memory for the Kernel

Setting up the memory consists of creating an initial page table for the kernel: This is a two-step procedure, we start with an *identity mapping*: that is a page table which maps virtual addresses 1:1 to physical addresses. The identity mapping lets us smoothly enable paging; when the CPU fetches the instruction which follows immediately after the enabling instruction, that memory access uses the memory management unit (MMU) and the translation information in the page table, whereas all earlier instructions accessed the memory (almost) directly.

identity mapping

⟨*setup memory* 97⟩≡ (44b) [97]
⟨*setup identity mapping for kernel* 108⟩
⟨*enable paging for the kernel* 109a⟩
⟨*install flat gdt* 110a⟩

We will discuss the details in this section.

Since we implement Ulix-i386 for Intel chips, we need to have a look at the Intel architecture which uses a two-layer design (see Figure 4.4).

When paging is active, the CPU register *CR3* (control register 3) points to a page directory which is a collection of 1024 page directory entries. Each of those entries is four bytes large, so the whole page directory has a size of 4 KByte (one page).

CR3 (control register 3)

Each page directory entry points to a page table (its address is given via bits 31..12).

Page tables have the same size as page directories (4 KByte), and they also hold 1024 entries, the page table entries. Such an entry points to a frame (again using bits 31..12).

We will look at these data structures in detail in the following sections. While—as an OS designer—you are free to implement many things in any way you can conceive, the Intel processor expects page directories and page tables to have a well-defined form that cannot be changed.

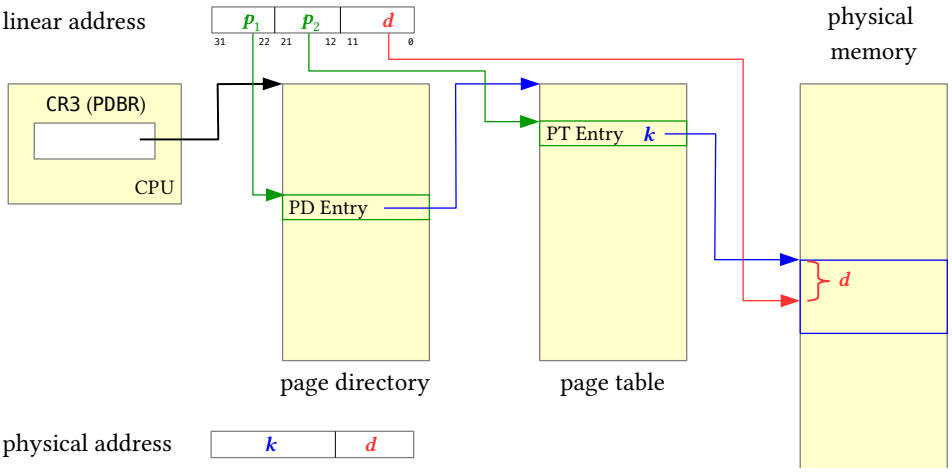


Figure 4.4: The Intel x86 Architecture uses a two-layered page table; the first layer is called "page directory", the second one "page table".

4.4.1 Page Descriptors and Page Table Descriptors in ULix

We now define the structures `page_desc`$_{100a}$ and `page_table_desc`$_{102}$ for page (table) descriptors. They use the layout that is required by the Intel CPU. Recall that it expects that the page table tree has exactly two levels. Descriptors at level 2 are either null descriptors or page descriptors. Descriptors at level 1 are either null descriptors or page table descriptors.

Intel uses a different vocabulary: In the Intel terminology,

page table entry
page directory entry

- a page descriptor is a *page table entry* (*PTE*; within a page table), and
- a page table descriptor is a *page directory entry* (*PDE*; in a page directory).

Each entry (for both page tables and page directories) is four bytes long; a page table or page directory contains 1024 such entries, filling exactly one page (of size 4 KByte).

A virtual address is always split in three parts: the page table number (bits 31–22), the page number (21–12) and the offset (11–0), see Figure 4.5.

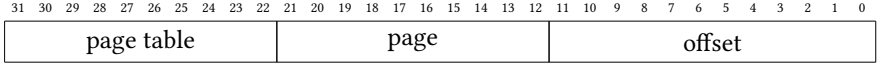


Figure 4.5: Virtual addresses consist of three parts: a 10 bit index into the page directory, 10 further bits as entry into a page table and 12 bits as offset.

Page table number and page number are Intel's specific terms for the general concept of split page table numbers where each portion corresponds to some level of page tables.

The system must associate a page directory with each process, the start address of the page directory must be stored in the process descriptor base register (*PDBR*), the upper 20 bits of control register 3 (*CR3*). PDBR (in CR3)

20 bits are sufficient to store an address because pages are page-size-aligned, i. e., they all start at addresses with zeroes in the lower 12 bits. (A page has size 4 KByte = 2^{12} bytes, which is why the offset length is 12.)

4.4.1.1 Page Table Entries

The upper 20 bits of a page descriptor contain the upper bits of a frame address (in RAM); the remaining bits of that address are zeroes for the same alignment reason as already described above: In RAM, no frame starts at an "odd" address which is not a multiple of the page size. Thus the frame address can easily be extracted from the page descriptor by setting the lower 12 bits to zero, if we treat the whole page descriptor as a 32-bit integer (which can be achieved with a cast operation in C, see also Exercise 13 on page 126):

⟨*get frame address from page descriptor's integer representation* 99⟩≡ [99]

```
frame_address = page_descriptor & 0xFFFFF000;
// F (hex) = 1111 (bin);  0 (hex) = 0000 (bin)
```

The remaining bits in the page descriptor are either unused and can be used by the operating system for its own purposes (bits 9–11) or store attributes of this page descriptor (see Figure 4.6):

- Bits 8 and 7 must always be 0.
- Bit 6 holds the *Dirty* (D) flag.
- Bit 5 holds the *Accessed* (A) flag. It is automatically set by the MMU when this page is accessed.
- Bit 4 is called *Page Cache Disabled* (PCD) – if set, data from this page must not be cached.
- Bit 3 is called *Page Write Transparent* (PWT), we will ignore this one and always set it to 0.
- Bit 2 holds the *User Accessible* (U) flag.
- Bit 1 holds the *Writeable* (W) flag.
- Bit 0 holds the *Present* (P) flag.

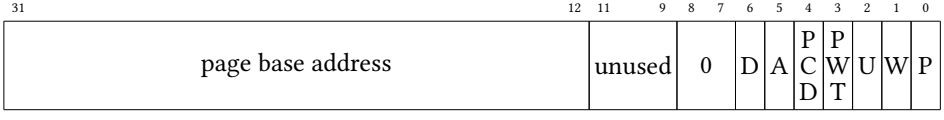


Figure 4.6: A page descriptor stores the upper 20 bits of the frame address and administrative data.

We can now describe the page descriptor in C:

[100a] ⟨*type definitions* 91⟩+≡ (44a) ◁92a 101b▷

```
typedef struct {
  unsigned int present         :  1;  //  0
  unsigned int writeable       :  1;  //  1
  unsigned int user_accessible :  1;  //  2
  unsigned int pwt             :  1;  //  3
  unsigned int pcd             :  1;  //  4
  unsigned int accessed        :  1;  //  5
  unsigned int dirty           :  1;  //  6
  unsigned int zeroes          :  2;  //  8.. 7
  unsigned int unused_bits     :  3;  // 11.. 9
  unsigned int frame_addr      : 20;  // 31..12
} page_desc;
```

Defines:
page_desc, used in chunks 72, 100, 101b, and 295b.

We repeat the code for calculating the physical address from this page descriptor, but now in a proper function that we can use later:

[100b] ⟨*function implementations* 100b⟩≡ (44a) 100c▷

```
memaddress page_desc_2_frame_address (page_desc pd) {
  // pointer magic/cast: a page descriptor is not really an unsigned
  // int, but we want to treat it as one
  memaddress address = *(memaddress*)(&pd);
  return address & 0xFFFFF000;   // set lowest 12 bits to zero
}
```

Uses memaddress 46c and page_desc 100a.

(This uses the casting trick we mentioned above: First we take the address of the page descriptor `pd` with the `&` operator, then we cast this pointer to a page descriptor to a pointer to a 32-bit integer with `(memaddress`$_{46c}$`*)`, and last we access the value with the `*` operator. Note that it is not possible to directly cast the page descriptor to an integer with a command like `address = (memaddress)pd;`—that is forbidden in the C language.)

The following function fills a page descriptor with values; its address must be provided (the space must be reserved by the caller). We provide the descriptor's address as the first argument via a pointer; the other arguments are the present, writeable, user accessible and dirty bits and—most important—the physical frame address:

[100c] ⟨*function implementations* 100b⟩+≡ (44a) ◁100b 103a▷

```
page_desc *fill_page_desc (page_desc *pd, unsigned int present,
                           unsigned int writeable, unsigned int user_accessible,
                           unsigned int dirty, memaddress frame_addr) {

  memset (pd, 0, sizeof (page_desc));      // first fill the four bytes with zeros

  pd->present         = present;           // then enter the argument values in
  pd->writeable       = writeable;         // the proper struct members
  pd->user_accessible = user_accessible;
```

```
    pd->dirty           = dirty;
    pd->frame_addr      = frame_addr >> 12;  // right shift, discard lower 12 bits
    return pd;
  };
```

Defines:
fill_page_desc, used in chunks 101a, 111b, and 123b.
Uses memaddress 46c, memset 596c, and page_desc 100a.

We have used the right shift operator >> in this function which takes the 32 bits from the memaddress variable frame_addr, right shifts them and fills the hole on the left side with zeroes. The function assumes that we always call it with a proper, page-size-aligned frame address frame_addr since its lower twelve bits will be discarded (they should have been zero to start with).

We provide two macros for quickly calling $\texttt{fill\_page\_desc}_{100c}$ with standard values, both for kernel memory and user mode memory:

⟨ *macro definitions* 35a⟩+≡ (44a) ◁46d 103c▷ [101a]

```
  #define KMAP(pd,frame) fill_page_desc (pd, true, true, false, false, frame)
  #define UMAP(pd,frame) fill_page_desc (pd, true, true, true,  false, frame)
```

Defines:
KMAP, used in chunks 106a, 111b, 115c, 121b, and 165b.
UMAP, used in chunk 165b.
Uses fill_page_desc 100c.

Both macros set the present and writeable bits to 1, they set the dirty bit to 0, and they supply the frame address; the difference is that $\texttt{KMAP}_{101a}$ sets the user accessible bit to 0 (making the page inaccessible to user mode code), whereas $\texttt{UMAP}_{101a}$ sets it to 1.

A page table contains 1024 page descriptors:

⟨ *type definitions* 91⟩+≡ (44a) ◁100a 102▷ [101b]

```
  typedef struct {
     page_desc pds[1024];
  } page_table;
```

Defines:
page_table, used in chunks 105a, 111a, 115a, 116e, 121–23, 165b, 166a, 169a, 171c, 211, 296, 297, 307a, and 308c.
Uses page_desc 100a.

We make this a struct so that we can easily create a pointer to such a page table.

4.4.1.2 Page Directory Entries

A page table descriptor or page directory entry looks similar to a page table entry: it has the same size and shares many common fields with the other one.

The upper 20 bits contain—again—the upper bits of a frame address (in RAM), the same alignment argument allows us to leave out the lowest 12 bits of the address.

The remaining bits in the page descriptor are either unused and can be used by the operating system for its own purposes (bits 9–11) or store attributes of this page descriptor:

- Bits 8 and 7 must always be 0. (Actually if bit 7 is set, this declares that the page described by this entry is a 4 MByte, not 4 KByte, page. We will not discuss 4 MByte pages.)

- Bit 6 is undocumented, we will always set it to 0.

The remaining fields are identical to those of a page table entry:

- Bit 5 holds the *Accessed* (A) flag.
- Bit 4 is called *Page Cache Disabled* (PCD) – if set, data from all pages belonging to this page table must not be cached.
- Bit 3 is called *Page Write Transparent* (PWT), we will ignore this one and always set it to 0.
- Bits 2, 1 and 0 hold the *User Accessible* (U), *Writeable* (W) and *Present* (P) flags.

The PCD, U, W and P flags enforce these properties for all pages that belong to this page table. Figure 4.7 shows the layout of the descriptor which is almost identical to that of the page descriptor, except for the missing Dirty flag.

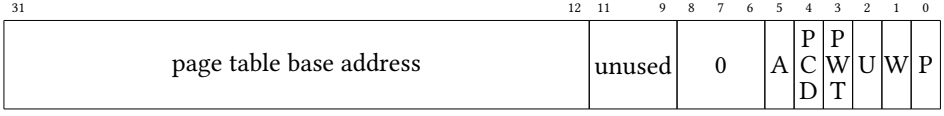


Figure 4.7: A page table descriptor stores the upper 20 bits of the physical address of the page table it points to and administrative data.

The C structure which describes page table descriptors only differs from the page descriptor structure `page_desc`$_{100a}$ by replacing the `dirty` field with an `undocumented` field:

[102] ⟨*type definitions* 91⟩+≡ (44a) ◁101b 103d▷

```
typedef struct {
  unsigned int present         :  1;  //  0
  unsigned int writeable       :  1;  //  1
  unsigned int user_accessible :  1;  //  2
  unsigned int pwt             :  1;  //  3
  unsigned int pcd             :  1;  //  4
  unsigned int accessed        :  1;  //  5
  unsigned int undocumented    :  1;  //  6
  unsigned int zeroes          :  2;  //  8.. 7
  unsigned int unused_bits     :  3;  // 11.. 9
  unsigned int frame_addr      : 20;  // 31..12
} page_table_desc;
```

Defines:
`page_table_desc`, used in chunk 103.

For extracting the frame address from a page table descriptor we rewrite the function `page_desc_2_frame_address`$_{100b}$ by simply using the new `page_table_desc`$_{102}$ structure instead of `page_desc`$_{100a}$:

⟨*function implementations* 100b⟩+≡ (44a) ◁100c 103b▷ [103a]
```
memaddress page_table_desc_2_frame_address (page_table_desc ptd) {
  memaddress address = *(memaddress*)(&ptd);
  return address & 0xFFFFF000;
}
```
Uses memaddress 46c and page_table_desc 102.

and we also duplicate $\texttt{fill\_page\_desc}_{100c}()$ as $\texttt{fill\_page\_table\_desc}_{103b}()$. Note that the function has one argument less since the dirty attribute does not exist in page table descriptors:

⟨*function implementations* 100b⟩+≡ (44a) ◁103a 109c▷ [103b]
```
page_table_desc *fill_page_table_desc (page_table_desc *ptd, unsigned int present,
                          unsigned int writeable, unsigned int user_accessible,
                          memaddress frame_addr) {

  memset (ptd, 0, sizeof (page_table_desc));  // fill the four bytes with zeros

  ptd->present         = present;             // then enter the argument values
  ptd->writeable       = writeable;
  ptd->user_accessible = user_accessible;
  ptd->frame_addr      = frame_addr >> 12;    // right shift, 12 bits
  return ptd;
};
```
Defines:
fill_page_table_desc, used in chunks 103c, 105b, 108, and 116b.
Uses memaddress 46c, memset 596c, and page_table_desc 102.

Just as we did for the page tables, we provide macros $\texttt{KMAPD}_{103c}$ and $\texttt{UMAPD}_{103c}$ which let us call $\texttt{fill\_page\_table\_desc}_{103b}$ with standard values:

⟨*macro definitions* 35a⟩+≡ (44a) ◁101a 113a▷ [103c]
```
#define UMAPD(ptd, frame) fill_page_table_desc (ptd, true, true, true,  frame)
#define KMAPD(ptd, frame) fill_page_table_desc (ptd, true, true, false, frame)
```
Defines:
KMAPD, used in chunks 105b, 108, 111b, 115d, 122, and 211a.
UMAPD, used in chunk 166a.
Uses fill_page_table_desc 103b.

In most cases we will use the $\texttt{UMAP}_{101a}$, $\texttt{KMAP}_{101a}$, $\texttt{UMAPD}_{103c}$ and $\texttt{KMAPD}_{103c}$ macros for modifying descriptors.

A page directory contains 1024 page table descriptors:

⟨*type definitions* 91⟩+≡ (44a) ◁102 137a▷ [103d]
```
typedef struct {
   page_table_desc ptds[1024];
} page_directory;
```
Defines:
page_directory, used in chunks 105a, 122c, 164a, 165b, 167c, 169a, 171c, 211, 296, 297, 307a, and 308c.
Uses page_table_desc 102.

4.4.2 Identity Mapping the Kernel Memory

We will now use a trick that allows a smooth transition from non-paging mode to paging mode: *identity mapping* creates a page directory for the kernel that maps the first virtual addresses to identical hardware addresses. When we later switch on paging, nothing changes for the kernel, because the MMU will be set up to use a page table that translates virtual addresses to the same addresses we've used before. This will also demonstrate why we've set up the segment tables with `0x40000000` offsets earlier.

4.4.2.1 First Attempt at a Kernel Layout

We now present a first and intuitive approach to placing the kernel in memory—both real memory and virtual memory; we will soon see that this approach is not the best possible choice and use a different layout.

Here are some general considerations that will lead us in the following steps:

- When the machine starts it must load the kernel into RAM. At that time paging is not yet enabled, so when the computer begins executing our kernel it uses physical memory addresses.
- Since we cannot know how much physical memory will be installed in a machine, it makes sense to place the kernel in some area with low memory addresses, such as the first megabyte of RAM.
- At some point in time during initialization of the operating system we will enable paging. However, code execution must logically continue at the next instruction and must not become confused by the fact that addresses are now translated by the MMU.

The easiest thing to do is compiling and linking the kernel with addresses starting at 0x0. If we assume that the kernel (and its stack) fit in 1 MByte of memory, we can reserve this physical memory area (`0x0000.0000`–`0x000F.FFFF`). The instruction that is going to enable paging will be sitting somewhere in this area, so we have to make sure that after paging is turned on, the instruction pointer will point to the instruction that follows immediately.

We need to identify the virtual addresses `0x0000.0000` to `0x000F.FFFF` with the same physical addresses. This amounts to 256 page table entries; they all fit in one page table ($\texttt{kernel\_pt}_{105a}$), and the page directory ($\texttt{kernel\_pd}_{105a}$) will have exactly one non-null entry pointing to that one page table.

This way, when the kernel has enabled paging, the first megabyte of virtual addresses will be in use (and reserved for the kernel), whereas the rest will have null pointers in the page directory and the page tables. So when we later talk about processes and threads, we can create processes which use virtual memory addresses starting at (virtual) address `0x00100000` (just after the first MByte), and those processes will have the first 2^{20} addresses unmapped. That way, when the process makes a syscall, we can modify the page tables so that the kernel's address space is added—then all addresses (`0x00000000` – `0x000FFFFF`: kernel; `0x00100000` and above: process) will be available so that data can be copied between process and kernel memory (see Figure 4.8).

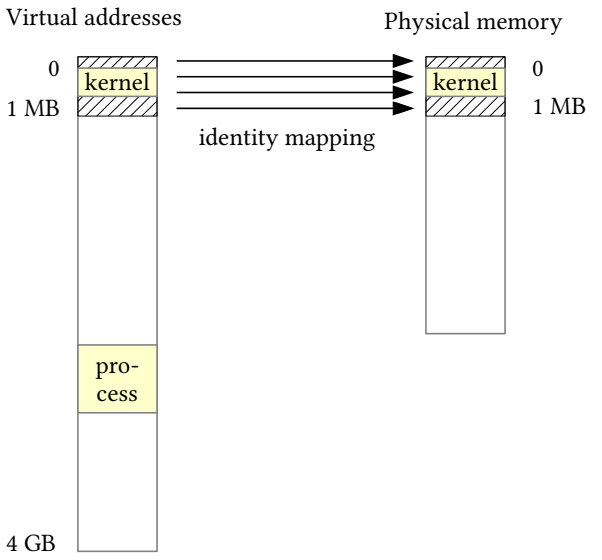


Figure 4.8: Identity mapping, first attempt: We identify the first MByte of virtual memory with the first MByte of physical RAM.

Let's start by declaring the necessary global variables:

⟨*global variables* 92b⟩+≡ (44a) ◁92b 106b▷ [105a]

```
page_directory kernel_pd  __attribute__ ((aligned (4096)));
page_table     kernel_pt  __attribute__ ((aligned (4096)));

// prefer to work with pointers
page_directory *current_pd = &kernel_pd;
page_table     *current_pt = &kernel_pt;
```

Defines:
`current_pd`, used in chunks 105b, 108, 109a, 111b, 115, 116, 121–23, 170c, 279c, 603, and 611b.
`current_pt`, used in chunks 106a, 108, and 603.
`kernel_pd`, used in chunks 105b, 106c, 108, 162e, 164b, and 604b.
`kernel_pt`, used in chunks 105b, 108, and 604b.
Uses `page_directory` 103d and `page_table` 101b.

We need to declare these with `__attribute__ ((aligned (4096)))` so that the C compiler aligns them properly in pages.

⟨*setup identity mapping for kernel 1st attempt* 105b⟩≡ [105b]

```
for (int i = 0;  i < 1024;  i++) {
  fill_page_table_desc (&current_pd->ptds[i], false, false, false, 0);
};

// make page table kernel_pt first entry of page directory kernel_pd
KMAPD ( &(kernel_pd.ptds[0]), (memaddress)(&kernel_pt) );
```

```
for (int i = 0;  i < 1024;  i++) {              // map 1024 pages (4 MB)
  ⟨identity map page i in kernel_pt 106a⟩
};
```

Uses current_pd 105a, fill_page_table_desc 103b, kernel_pd 105a, kernel_pt 105a, KMAPD 103c, and memaddress 46c.

In order to identity-map page i in $\texttt{kernel\_pt}_{105a}$ we need to fill the i^{th} entry. Page frame i starts at physical address $i \times$ PAGE_SIZE:

[106a] ⟨*identity map page i in* kernel_pt 106a⟩≡ (105b 108)

```
KMAP ( &(current_pt->pds[i]), i*4096 );
```

Uses current_pt 105a and KMAP 101a.

Finally, we have to enable paging in the CPU. That can be achieved by making some changes to the control registers *CR0* and *CR3* as follows:

- Control register 3 (*CR3*) must contain the address of the page directory ($\texttt{kernel\_pd}_{105a}$),
- CR0 (Control Register 0) in control register 0 (*CR0*) we must set the PG (paging) bit which is bit 31. Setting this single bit is done by calculating cr0 = cr0 | (1<<31).

[106b] ⟨*global variables* 92b⟩+≡ (44a) ◁105a 111a▷

```
char *kernel_pd_address;      // address of kernel page directory
```

Defines:
kernel_pd_address, used in chunks 106c and 109a.

[106c] ⟨*enable paging for the kernel 1st attempt* 106c⟩≡

```
kernel_pd_address = (char*)(&kernel_pd);
asm volatile ("mov %0, %%cr3" : : "r"(kernel_pd_address));     // write CR3
uint cr0; asm volatile ("mov %%cr0, %0" : "=r"(cr0) : );       // read  CR0
cr0 |= (1<<31);                // Enable paging by setting PG bit 31 of CR0
asm volatile ("mov %0, %%cr0" : : "r"(cr0) );                  // write CR0
```

Uses kernel_pd 105a and kernel_pd_address 106b.

We call this code block ⟨*enable paging for the kernel 1st attempt* 106c⟩ because we're not done yet; with the higher half trick, $\texttt{\&kernel\_pd}_{105a}$ will not be a physical address and won't fit the base values in our segment descriptors.

4.4.2.2 Second Attempt at a Kernel Layout

For several reasons (which we will not dig into), legacy properties of Intel machines suggest to keep the first megabyte of RAM unused. So we will physically store the kernel in the second MByte. However, once paging is turned on, the kernel's addresses shall be found in the last of the four gigabytes (starting at 0xc0000000), as we've shown at this chapter's beginning (see Figure 4.1 on page 88). We want processes to use the first three gigabytes and the kernel to reside in the last of the four available gigabytes of virtual memory. Figure 4.9 shows the necessary mapping of virtual addresses to physical addresses.

This is not an identity mapping, so when we start the system we run into a problem: We could link the kernel twice und also physically load it twice, into the physical ranges [1 MB, 2 MB [and [3 GB + 1 MB, 3 GB + 2 MB [—but that would require our physical memory to be big enough.

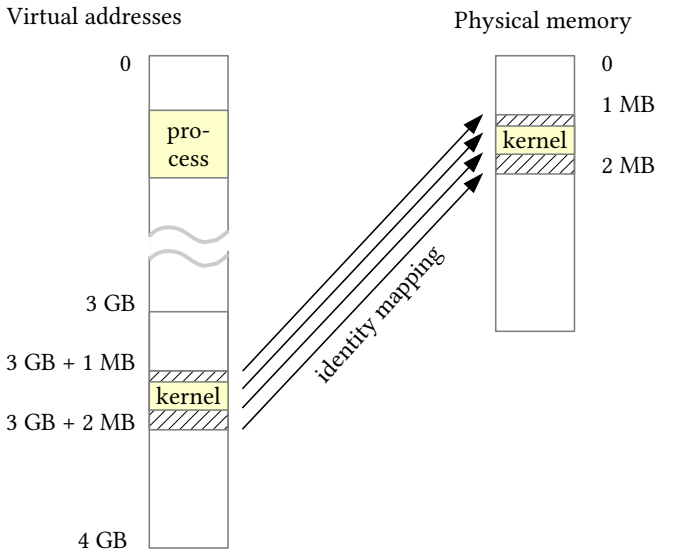


Figure 4.9: Second mapping attempt: Kernel starts at 3 GB + 1 MB (virtual) or 1 MB (physical).

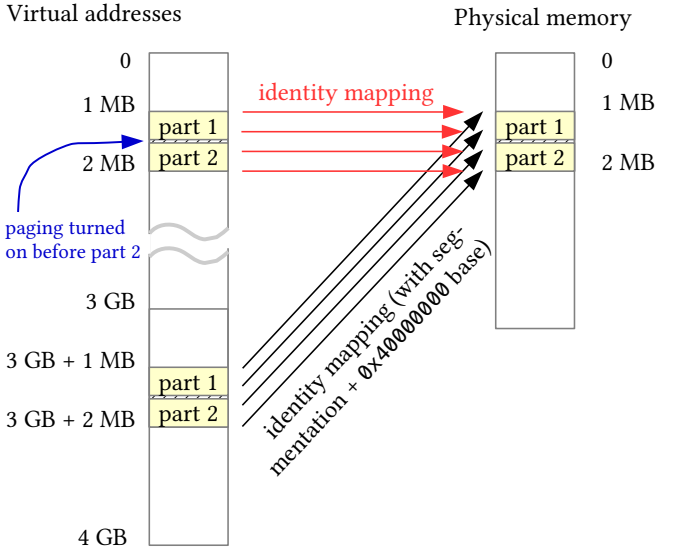


Figure 4.10: Final mapping: Kernel starts at 3 GB + 1 MB *and* 1 MB (virtual) or 1 MB (physical).

The solution is to work with a double mapping, as can be seen in Figure 4.10 (during initialization).

So we do the following:

1. Load the kernel to physical addresses [1 MB, 2 MB [(`0x100000` – `0x1FFFFF`).
2. Do the Higher Half Trick (see page 93): Enable the trick GDT with base address `0x40000000` and jump to the higher half.
3. Setup an identity mapping from [1 MB, 2 MB [(virtual) to [1 MB, 2 MB [(physical). Actually, we'll map the whole first 4 megabytes, since one page table describes 4 MByte of virtual memory.
4. Additionally set up a mapping from [3 GB + 1 MB, 3 GB + 2 MB [(`0xC0100000` – `0xC01FFFFF`; virtual) to [1 MB, 2 MB [(physical) – the corresponding page table has the same contents as the first one, it just gets pointed to from a different page directory entry. Again, we'll map a whole 4 MByte block, [3 GB, 3 GB + 4 MB [to [0, 4 MB [.
5. Activate paging.
6. Install a new "flat" GDT with base address `0x0`.
7. Get rid of the initial mapping for [0, 4 MB [(virtual).

After that the kernel sees virtual addresses starting at 3 GByte (`0xC0000000`) only, and things work perfectly with the linker configuration we've discussed on page 95.

Now we have to modify the setup of the identity mapping:

[108] ⟨*setup identity mapping for kernel* 108⟩≡ (97)

```
// file page directory with null entries
for (int i = 0;  i < 1024;  i++) {
  fill_page_table_desc (&(current_pd->ptds[i]), false, false, false, 0);
};

// make page table kernel_pt      the first entry of page directory kernel_pd
// maps: 0x00000000..0x003FFFFF -> 0x00000000..0x003FFFFF  (4 MB)
KMAPD ( &(current_pd->ptds[  0]), (memaddress)(current_pt)-0xC0000000 );

// make page table kernel_pt also the 768th entry of page directory kernel_pd
// maps: 0xC0000000..0xC03FFFFF -> 0x00000000..0x003FFFFF  (4 MB)
KMAPD ( &(current_pd->ptds[768]), (memaddress)(current_pt)-0xC0000000 );

// map 1023 pages (4 MB minus 1 page)
for (int i = 0;  i < 1023;  i++) {
  ⟨identity map page i in kernel_pt 106a⟩
};

kputs ("Kernel page directory set up.\n");
```

Uses `current_pd` 105a, `current_pt` 105a, `fill_page_table_desc` 103b, `kernel_pd` 105a, `kernel_pt` 105a, `KMAPD` 103c, `kputs` 335b, and `memaddress` 46c.

Note that we're leaving one entry free (mapping only 1023 pages)—we'll later need this one to create the next page table.

Then we can load the process descriptor base register (*PDBR*, part of *CR3*) and modify *CR0* so that the CPU switches to paging mode—the following code is almost the same as the one in ⟨*enable paging for the kernel 1st attempt* 106c⟩, but it calculates the physical address of the page directory by subtracting `0xC0000000`:

⟨*enable paging for the kernel* 109a⟩≡ (97) 162e▷ [109a]
```
kernel_pd_address = (char*)(current_pd) - 0xC0000000;
asm volatile ("mov %0, %%cr3" : : "r"(kernel_pd_address) ); // write CR3
uint cr0; asm volatile ("mov %%cr0, %0" : "=r"(cr0) : );    // read  CR0
cr0 |= (1<<31);            // Enable paging by setting PG bit 31 of CR0
asm volatile ("mov %0, %%cr0" : : "r"(cr0) );               // write CR0
```
Uses `current_pd` 105a and `kernel_pd_address` 106b.

After paging is enabled we first update the GDT (and make it flat), then we can get rid of the identity mapping which works with low addresses.

4.4.3 Installing the "Flat" GDT

One of the last steps is installing the "flat" GDT: it looks like our trick GDT, but uses a base address of `0x0` instead of `0x40000000`. We're already executing C code, so we'll provide a C function for loading the GDT[1]:

⟨*function prototypes* 45a⟩+≡ (44a) ◁45a 116d▷ [109b]
```
void fill_gdt_entry (int num, ulong base, ulong limit, byte access, byte gran);
extern void gdt_flush ();
```

⟨*function implementations* 100b⟩+≡ (44a) ◁103b 113b▷ [109c]
```
void fill_gdt_entry (int num, ulong base, ulong limit, byte access, byte gran) {
  // base address; split in three parts
  gdt[num].base_low    = (base & 0xFFFF);         // 16 bits
  gdt[num].base_middle = (base >> 16) & 0xFF;     //  8 bits
  gdt[num].base_high   = (base >> 24) & 0xFF;     //  8 bits

  // limit address; split in two parts
  gdt[num].limit_low   = (limit & 0xFFFF);        // 16 bits
  gdt[num].limit_high  = (limit >> 16) & 0x0F;    //  4 bits

  // granularity and access flags
  gdt[num].flags       = gran & 0xF;
  gdt[num].access      = access;
}
```
Defines:
`fill_gdt_entry`, used in chunks 110a, 194a, and 197a.
Uses `gdt` 92b and `ulong` 46b.

The following code shows how the first three GDT entries are created, however, we will later introduce three further entries which we need for user mode processes—you can ignore that for now, but that is the reason why we reserve space for six GDT entries

[1] The functions in this section are—again—based on code from Bran's Kernel Development Tutorial [Fri05].

(instead of three) and we include the code chunk ⟨*install GDTs for User Mode* 194a⟩ and function calls to `gdt_flush`$_{110b}$ and `tss_flush`$_{197c}$, the latter of which will be explained when we discuss processes.

[110a] ⟨*install flat gdt* 110a⟩≡ (97)

```
// We'll have six GDT entries; only three are defined now
gp.limit = (sizeof (struct gdt_entry) * 6) - 1;  // must be -1
gp.base = (int) &gdt;

fill_gdt_entry (0, 0, 0, 0, 0);    // null descriptor

// code segment: base = 0, limit = 0xFFFFF
fill_gdt_entry (1, 0, 0xFFFFF, 0b10011010, 0b1100);

// data segment: base = 0, limit = 0xFFFFF
fill_gdt_entry (2, 0, 0xFFFFF, 0b10010010, 0b1100);

⟨install GDTs for User Mode 194a⟩  // explained later
gdt_flush ();   // Notify the CPU of changes
tss_flush ();   // explained later
```

Uses `fill_gdt_entry` 109c, `gdt` 92b, `gdt_entry` 91, `gdt_flush` 110b, `gp` 92b, and `tss_flush` 197c.

(We've declared `gp`$_{92b}$ and `gdt`$_{92b}$ on page 92 after defining the C data structures for the GDT.) The function `gdt_flush`$_{110b}$ resides in the assembler file; it uses the `lgdt` instruction to load the new segment descriptors, sets all segment registers except *CS* to `0x10` and then makes a far jump for setting *CS* as well (to `0x08`).

`lgdt`

[110b] ⟨*start.asm* 87⟩+≡ ◁95a 144▷

```
[section .text]
            extern gp           ; defined in the C file
            global gdt_flush
gdt_flush: lgdt [gp]
            mov ax, 0x10
            mov ds, ax
            mov es, ax
            mov fs, ax
            mov gs, ax
            mov ss, ax
            jmp 0x08:flush2
flush2:     ret
```

Defines:
`gdt_flush`, used in chunks 110a, 111b, and 116b.
Uses `gp` 92b.

If you compare this code with the code for the higher half trick (see page 93), you will see that it basically does the same and just uses a different address in the `lgdt` instruction.

Effectively, the flat GDT sort of disables segmentation: the segmentation unit maps logical addresses to identical linear addresses (which are then translated into physical addresses by the paging unit).

4.4.4 Accessing the Video RAM

We need to access the video adapter's text mode framebuffer which is mapped into the physical address space starting at `0xB8000` (and takes up 4 KByte of memory). So we provide a mapping for this memory as well:

⟨*global variables* 92b⟩+≡ (44a) ◁106b 112b▷ [111a]

```
page_table video_pt __attribute__ ((aligned (4096)));  // must be aligned!
```

Defines:
`video_pt`, used in chunk 111b.
Uses `page_table` 101b.

We create a new page table (initialized with null entries) and from there we only create a mapping of 4 KByte (starting at `0xB8000`).

⟨*initialize system* 45b⟩+≡ (44b) ◁45b 112d▷ [111b]

```
for (int i = 0;  i < 1024;  i++) {
  // null entries:
  fill_page_desc ( &(video_pt.pds[i]), false,false,false,false,0 );
};

KMAP ( &(video_pt.pds[0xB8]), 0xB8*4096 );   // one page of video RAM

// enter new table in page directory
KMAPD ( &(current_pd->ptds[0]), (memaddress) (&video_pt) - 0xC0000000 );

gdt_flush ();
```

Uses `current_pd` 105a, `fill_page_desc` 100c, `gdt_flush` 110b, `KMAP` 101a, `KMAPD` 103c, `memaddress` 46c, and `video_pt` 111a.

4.5 Physical Memory: Page Frames in ULIX

The physical memory consists of page frames, some of which are already in use. When we dynamically assign frames to pages (i. e., change some page table), we need to know which frames are free and which are in use. For that purpose we use a bitmap (that we will call the *frame table*) which holds the current usage state of every frame. Since we have just set up the initial memory usage, we know exactly what our memory looks like at this point in time, so now is a good time to create and initialize that bitmap. frame table

We assume that our system has 64 MByte of physical RAM. The size of the frame table depends on the size of the available physical memory which we define to contain $\texttt{MEM\_SIZE}_{111c}$ many addresses. Dividing this by the $\texttt{PAGE\_SIZE}_{112a}$ gives us the number of page frames. This of course assumes that $\texttt{MEM\_SIZE}_{111c}$ is larger than $\texttt{PAGE\_SIZE}_{112a}$ and both values are powers of two.

⟨*public constants* 46a⟩+≡ (44a 48a) ◁46a 180a▷ [111c]

```
#define MEM_SIZE 1024*1024*64       // 64 MByte
```

Defines:
`MEM_SIZE`, used in chunks 112a and 499a.

[112a] ⟨*constants* 112a⟩≡ (44a) 132▷
```
#define MAX_ADDRESS MEM_SIZE-1       // last valid physical address
#define PAGE_SIZE 4096               // Intel: 4K pages
#define NUMBER_OF_FRAMES (MEM_SIZE/PAGE_SIZE)
```
Defines:
NUMBER_OF_FRAMES, used in chunks 112, 115b, 118c, and 613c.
PAGE_SIZE, used in chunks 113b, 115c, 121b, 122a, 163–65, 167, 169b, 172a, 173a, 209b, 211b, 257, 261, 289c, 291, 293d, 294, and 298a.
Uses MEM_SIZE 111c.

The usage of main memory is directly reflected in the amount of frames which are not free. We will try to keep track of the number of free frames throughout the lifetime of the system in a global variable free_frames$_{112b}$.

[112b] ⟨*global variables* 92b⟩+≡ (44a) ◁111a 112c▷
```
unsigned int free_frames = NUMBER_OF_FRAMES;
```
Defines:
free_frames, used in chunks 112e, 119, 123c, 310a, 311b, 342b, 513e, 604b, and 613c.
Uses NUMBER_OF_FRAMES 112a.

So NUMBER_OF_FRAMES$_{112a}$ is the number of bits we need to store in the frame table. Since a byte holds eight bits, we need a structure that is NUMBER_OF_FRAMES$_{112a}$/8 bytes large:

[112c] ⟨*global variables* 92b⟩+≡ (44a) ◁112b 115a▷
```
char place_for_ftable[NUMBER_OF_FRAMES/8];
unsigned int *ftable = (unsigned int*)(&place_for_ftable);
```
Defines:
ftable, used in chunks 112–14 and 603.
place_for_ftable, used in chunk 603.
Uses NUMBER_OF_FRAMES 112a.

[112d] ⟨*initialize system* 45b⟩+≡ (44b) ◁111b 112e▷
```
memset (ftable, 0, NUMBER_OF_FRAMES/8);  // all frames are free
```
Uses ftable 112c, memset 596c, and NUMBER_OF_FRAMES 112a.

Now we need to tell the frame table that some of our frames are already in use: We have two mappings for the first 4 MByte of physical RAM (even though we don't use the first MByte at all). So we declare the first 4 MByte as used. 4 MByte contain 1024 pages, thus the first 1024 frames must be marked used. 1024/8 = 128; we set the first 128 bytes to `0xff` = 11111111_b. We also subtract the frames in these 4 MByte from free_frames$_{112b}$:

[112e] ⟨*initialize system* 45b⟩+≡ (44b) ◁112d 115b▷
```
memset (ftable, 0xff, 128);
free_frames -= 1024;
```
Uses free_frames 112b, ftable 112c, and memset 596c.

4.5.1 Bitwise Manipulation

We want to be able to set and clear single entries in our frame table, so we have to access single bits: read them, write them and test them.

We can think of a frame number as consisting of

- an upper part that is an index into the frame table (which is built from 32-bit `unsigned ints`). Every such `unsigned int` stores 32 bits.
- and a lower part that is an offset whose value can lie between 0 and 31, giving a precise position within one such indexed `unsigned int`.

So we get *frameno* = 32 × *index* + *offset*, like this:

$$frameno = \ldots i_4\, i_3\, i_2\, i_1\, i_0\, o_4\, o_3\, o_2\, o_1\, o_0$$

When we divide a frame number by 32, we find the `unsigned int` which stores the bit we're searching for. The modulo function gives us the offset:[2]

⟨ *macro definitions* 35a ⟩+≡ (44a) ◁103c 116a▷ [113a]

```
#define INDEX_FROM_BIT(b) (b/32)   // 32 bits in an unsigned int
#define OFFSET_FROM_BIT(b) (b%32)
```

Defines:
`INDEX_FROM_BIT`, used in chunks 113b and 114a.
`OFFSET_FROM_BIT`, used in chunks 113b and 114a.

The following two functions allow us to set or clear individual bits in the frame table:

⟨ *function implementations* 100b ⟩+≡ (44a) ◁109c 114a▷ [113b]

```
static void set_frame (memaddress frame_addr) {
  unsigned int frame = frame_addr / PAGE_SIZE;
  unsigned int index = INDEX_FROM_BIT (frame);
  unsigned int offset = OFFSET_FROM_BIT (frame);
  ftable[index] |= (1 << offset);
}

static void clear_frame (memaddress frame_addr) {
  unsigned int frame = frame_addr / PAGE_SIZE;
  unsigned int index = INDEX_FROM_BIT (frame);
  unsigned int offset = OFFSET_FROM_BIT (frame);
  ftable[index] &= ~(1 << offset);
}
```

Defines:
`clear_frame`, used in chunk 119b.
`set_frame`, used in chunk 119a.
Uses `ftable` 112c, `INDEX_FROM_BIT` 113a, `memaddress` 46c, `OFFSET_FROM_BIT` 113a, and `PAGE_SIZE` 112a.

Note how individual bits are set or cleared:

- `|=` and `&=` work in a similar way as `+=` for addition, however they perform "bitwise or" and "bitwise and", respectively. So `x|=y` is short for `x=x|y` and `x&=y` is short for `x=x&y`.
- In the $\texttt{set\_frame}_{113b}$ function, `1 << offset` uses left shift to create a value whose offset's bit is set (and all others are not), e. g. `1 << 3` is `00000000000000000000000000001000` (in binary notation).

[2] The macros $\texttt{INDEX\_FROM\_BIT}_{113a}$ and $\texttt{OFFSET\_FROM\_BIT}_{113a}$ and the functions $\texttt{set\_frame}_{113b}$, $\texttt{clear\_frame}_{113b}$, and $\texttt{test\_frame}_{114a}$ have been taken from `http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html`, they were slightly modified and adapted to Ulix.

- Next the corresponding `unsigned int` is "bitwise-or"ed with this value. That means: all bits which were already 1, remain 1; and the `offset`'s bit is being set (whatever its value was before).
- In a similar way the `clear_frame`$_{113b}$ function can clear a bit. It also starts with a shift operation, but the result goes through bitwise negation (~) which flips all bits. For example, `~(1 << 3)` is $11111111111111111111111111110111_b$. So there is exactly one 0 bit in there with all other bits being 1.
- Then the corresponding `unsigned int` is 'bitwise-and"ed with this value. That means: all bits which were already 0, remain 0; and the `offset`'s bit is being cleared (whatever its value was before).

What remains is a function that can test a bit. It returns `true` (1) or `false` (0):

[114a] ⟨*function implementations* 100b⟩+≡ (44a) ◁113b 116e▷

```
static boolean test_frame (unsigned int frame) {
  // returns true if frame is in use (false if frame is free)
  unsigned int index = INDEX_FROM_BIT (frame);
  unsigned int offset = OFFSET_FROM_BIT (frame);
  return ((ftable[index] & (1 << offset)) >> offset);
}
```

Defines:
`test_frame`, used in chunks 114b, 118c, 119b, and 613c.
Uses `ftable` 112c, `INDEX_FROM_BIT` 113a, and `OFFSET_FROM_BIT` 113a.

A result of 0 means that a frame is available, whereas 1 means that the frame is already in use—which corresponds to the way we have already initialized a part of the frame table.

The function uses left and right shifts in order to always return either 0 or 1. If you never do any comparisons with 1, but only call the function in `if` statements such as

[114b] ⟨*example call of test_frame* 114b⟩≡

```
if ( test_frame (frameno) ) {
  // result non-0 (true); frame is not available
} else {
  // result 0 (false);    frame is available
}
```

then you can skip the right shift at the end of the line and make the calculation a bit faster.

4.5.2 Direct Access to the Physical RAM

So far we haven't encountered any conceptual problems, but consider this: The information stored in the page directories, page tables and in the frame table refers to physical memory. But the kernel has activated paging, and even though it runs with the most privileges any code on the machine can get, it cannot directly access physical memory. Yet it has to modify or create new page tables and it has to update the frame table. So the kernel needs to have an understanding of what is going on in the physical memory, without accessing it.

If physical memory is very small in comparison to the virtual address space, it is possible to permanently map all of the RAM into some area of the kernel's virtual address space. For our code we assume that the machine has only 64 MByte of RAM—compared to the 4 GByte address space that is not much. We can spare 64 MByte of the virtual kernel memory and sponsor a mapping to this physical RAM. We will put this in the area `0xD0000000` ...`0xD3FFFFFF`, so that any physical address x can be accessed via the virtual address x + `0xD0000000`. However, there's a cost: the corresponding page table entries will require some room: 64 MByte = 16384 pages, so we will need 16384 page table entries each of which uses 4 bytes. Thus, it requires 16 pages (64 KByte) to store the extra page tables. Where can we put these tables? To simplify things we'll declare yet more static variables which hold our 16 tables:

⟨*global variables* 92b⟩+≡ (44a) ◁112c 138a▷ [115a]
```
page_table kernel_pt_ram[16] __attribute__ ((aligned (4096)));
```
Defines:
kernel_pt_ram, used in chunk 115.
Uses page_table 101b.

and we initialize them with references to all of our RAM.

⟨*initialize system* 45b⟩+≡ (44b) ◁112e 115d▷ [115b]
```
for (uint fid = 0;  fid < NUMBER_OF_FRAMES;  fid++) {
  ⟨map page starting at 0xD0000000 + PAGE_SIZE*fid to frame fid 115c⟩
}
```
Uses NUMBER_OF_FRAMES 112a.

The code for this mapping is not too complicated, either:

⟨*map page starting at* `0xD0000000 + PAGE_SIZE*fid` *to frame* `fid` 115c⟩≡ (115b) [115c]
```
KMAP ( &(kernel_pt_ram[fid/1024].pds[fid%1024]), fid*PAGE_SIZE );
```
Uses kernel_pt_ram 115a, KMAP 101a, and PAGE_SIZE 112a.

(Note that instead of `&(kernel_pt_ram[fid/1024].pds[fid%1024])` we could have used `&(kernel_pt_ram[0].pds[fid])` which would access out of bound indices of `kernel_pt_ram[0].pds`, but since these arrays are arranged one after the other without other data in between, it would work as well.)

To finalize this, we have to enter the 16 new page tables in 16 page directory entries. Note that we need the physical addresses of the page tables, not the virtual ones. While `&(kernel_pt_ram`$_{115a}$`[i])` delivers the virtual address just fine, it does not help to write it into the page directory. Subtracting `0xC0000000` does the job: we know that we loaded the kernel at `0x100000` with addresses starting at `0xC0100000`, so we just need to subtract that artificial offset, and we're good.

⟨*initialize system* 45b⟩+≡ (44b) ◁115b 116b▷ [115d]
```
for (int i = 0;  i < 16;  i++) {
  // get physical address of kernel_pt_ram[i]
  memaddress physaddr = (memaddress)(&(kernel_pt_ram[i])) - 0xc0000000;
  KMAPD ( &(current_pd->ptds[832+i]), physaddr );
};
kputs ("RAM: 64 MByte, mapped to 0xD0000000-0xD3FFFFFF\n");
```
Uses current_pd 105a, kernel_pt_ram 115a, KMAPD 103c, kputs 335b, and memaddress 46c.

Since we will often have to access a physical address, we'll define a macro PHYSICAL$_{116a}$ that will translate an address from the first 64 MByte to the 0xD000.0000 ...0xD3FF.FFFF range:

[116a] ⟨*macro definitions* 35a⟩+≡ (44a) ◁113a 116c▷
```
#define PHYSICAL(x) ((x)+0xd0000000)
```
Defines:
PHYSICAL, used in chunks 116e, 117, 121–23, 165b, 166a, 171c, 209b, 211c, 293d, 294, 296, 297, 307a, 308c, 496d, 497, 549c, and 550b.

Now that we can access all of the physical addresses (including video memory) we can get rid of the video mapping for 0xb8000...0xb9000, we'll actually remove the first entry of the page directory which so far mapped part of the first (virtual) 4 MByte.

[116b] ⟨*initialize system* 45b⟩+≡ (44b) ◁115d 218c▷
```
VIDEORAM = 0xD00B8000;
// remove first page table (including the old video mapping)
fill_page_table_desc (&current_pd->ptds[0], 0, 0, 0, 0);
gdt_flush ();
```
Uses current_pd 105a, fill_page_table_desc 103b, gdt_flush 110b, and VIDEORAM 327b.

We define a macro which casts VIDEORAM$_{327b}$ into a word pointer which will later be helpful for accessing individual characters on the screen (they are encoded as words, not bytes):

[116c] ⟨*macro definitions* 35a⟩+≡ (44a) ◁116a 117▷
```
#define textmemptr ((word*)VIDEORAM)
```
Defines:
textmemptr, used in chunks 329b, 335b, and 609.
Uses VIDEORAM 327b.

4.5.2.1 MMU Emulation

Sometimes we want to find out what frame is used by a page. We present a function

[116d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁109b 119c▷
```
unsigned int pageno_to_frameno (unsigned int pageno);
```

for this purpose which basically works like the MMU when it translates addresses: It uses the fact that for a page number pageno we first look at entry pageno/1024 of the page directory, locate the referenced page table and then look at entry pageno%1024 of that page table. When the page is not mapped to a frame, the function returns −1:

[116e] ⟨*function implementations* 100b⟩+≡ (44a) ◁114a 118b▷
```
unsigned int pageno_to_frameno (unsigned int pageno) {
  unsigned int pdindex = pageno/1024;
  unsigned int ptindex = pageno%1024;
  if ( ! current_pd->ptds[pdindex].present ) {
    return -1;       // we don't have that page table
  } else {
    // get the page table
    page_table *pt = (page_table*)
                     ( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
```

```
      if ( pt->pds[ptindex].present ) {
        return pt->pds[ptindex].frame_addr;
      } else {
        return -1;      // we don't have that page
      };
    };
  };
```

Defines:
pageno_to_frameno, used in chunks 116d and 123a.
Uses current_pd 105a, page_table 101b, and PHYSICAL 116a.

Note that frame_addr holds (the upper 20 bits of) a physical address. Luckily we have a mapping of the physical address space to 0xd000.0000 and above that we can access via the PHYSICAL$_{116a}$ macro—otherwise we would have no way of accessing the page table.

4.5.3 PEEK and POKE Functions

If you remember home computers like the Commodore C64 or the Schneider/Amstrad CPC, their built-in Basic interpreters often had a way to directly access memory contents. The classical command names were PEEK$_{117}$ (for reading) and POKE$_{117}$ (for writing). Here they are again: They convert an address into a pointer to a byte and then read or write. (Credits to Dan Henry who supplied the first two lines of this code on http://www.keil.com/forum/8275/.)

⟨ *macro definitions* 35a⟩+≡ (44a) ◁116c 163b▷ [117]

```
// Peek and Poke for virtual addresses
#define PEEK(addr)           (*(byte *)(addr))
#define POKE(addr, b)        (*(byte *)(addr) = (b))
// Peek and Poke for physical addresses 0..64 MB
#define PEEKPH(addr)         (*(byte *)(PHYSICAL(addr)))
#define POKEPH(addr, b)      (*(byte *)(PHYSICAL(addr)) = (b))

// Macros for accessing unsigned ints (instead of bytes)
#define PEEK_UINT(addr)      (*(uint *)(addr))
#define POKE_UINT(addr, b)   (*(uint *)(addr) = (b))
#define PEEKPH_UINT(addr)    (*(uint *)(PHYSICAL(addr)))
#define POKEPH_UINT(addr, b) (*(uint *)(PHYSICAL(addr)) = (b))
```

Defines:
PEEK, used in chunk 612c.
POKE, used in chunks 337b and 342d.
POKE_UINT, used in chunk 567c.
Uses PHYSICAL 116a.

PEEK$_{117}$ and POKE$_{117}$ read and write bytes using virtual addresses, PEEKPH$_{117}$ and POKEPH$_{117}$ do the same with physical addresses (using our 0xD0000000 trick with the PHYSICAL$_{116a}$ macro), and finally PEEK_UINT$_{117}$, POKE_UINT$_{117}$, PEEKPH_UINT$_{117}$ and POKEPH_UINT$_{117}$ do the same as the first four functions but work with unsigned 32-bit integers instead of bytes.

With the little-endian ordering of larger integers, the following code

[118a] ⟨*peek and poke example* 118a⟩≡
```
unsigned int testvar;
unsigned int address = (unsigned int)&testvar;
POKE (address,   0x12);
POKE (address+1, 0x34);
POKE (address+2, 0x56);
POKE (address+3, 0x78);
printf ("32-bit value: 0x%x\n", PEEK_UINT (address));
```

prints

```
32-bit value: 0x78563412
```

(and not `0x12345678`).

4.5.4 Allocating and Releasing Frames

So far we have not used any dynamically generated data structures in the kernel, so there was no need for some kind of allocation function for the kernel.

When we start creating processes, we will need to reserve (virtual) memory for those processes, and there may also be areas in the kernel which need memory.

So we will start simple: with a function that requests a new frame of physical memory. It has the following definition:

[118b] ⟨*function implementations* 100b⟩+≡ (44a) ◁116e 119b▷
```
int request_new_frame () {
  ⟨find a free frame und reserve it 118c⟩
};
```
Defines:
`request_new_frame`, used in chunks 121a, 164–66, 173a, 192a, 211a, 257c, 291, 297, and 608b.

This alone will not be all too useful—only in combination with entering it in some paging table that memory will be accessible (unless code uses the mapping of the physical RAM to `0xD000.0000` and above).

Finding a free frame is simple: We look at the frame table and return the first available frame:

[118c] ⟨*find a free frame und reserve it* 118c⟩≡ (118b) 119a▷
```
unsigned int frameid;
boolean found;
start_find_frame:
found = false;
for (frameid = 0;  frameid < NUMBER_OF_FRAMES;  frameid++) {
  if ( !test_frame (frameid) ) {
    found=true;
    break;   // frame found
  };
}
```
Uses `frameid`, `NUMBER_OF_FRAMES` 112a, and `test_frame` 114a.

Then we use set_frame$_{113b}$ to mark the frame used and return the frame ID:

⟨*find a free frame und reserve it* 118c⟩+≡ (118b) ◁118c [119a]

```
if (found) {
  set_frame (frameid*4096);
  free_frames--;
  return frameid;
} else {
  ⟨page replacement: free one frame 308c⟩    // will be explained later
  goto start_find_frame;
  // return -1;    // never fail
}
```

Uses frameid, free_frames 112b, and set_frame 113b.

Note that the function clears a frame if no free one is available—we will explain the code chunk ⟨*page replacement: free one frame* 308c⟩ in Chapter 9.4.

We'll add code for releasing a frame: basically we just call clear_frame$_{113b}$, but we also need to modify free_frames$_{112b}$:

⟨*function implementations* 100b⟩+≡ (44a) ◁118b 120a▷ [119b]

```
void release_frame (int frame) {
  if ( test_frame (frame) ) {
    // only do work if frame is marked as used
    clear_frame (frame << 12);
    free_frames++;
  };
};
```

Defines:
release_frame, used in chunks 123c, 167c, 169a, 261, and 296.
Uses clear_frame 113b, free_frames 112b, and test_frame 114a.

We may call release_frame$_{119b}$ for unused frames which will have no effect.

4.6 Managing Pages in Ulix

Since we have now established a mechanism for reserving frames, we can proceed with page requests. You have already seen all the required data structures when we initialized paging in Chapter 4.4 (pp. 97 ff.).

4.6.1 Allocating Pages

Now we need to implement functions for dynamically requesting new pages and releasing them after they are no longer needed. Both are only possible via requesting and releasing frames, and we need to update existing page directories and page tables as well as occasionally create new page tables.

⟨*function prototypes* 45a⟩+≡ (44a) ◁116d 133a▷ [119c]

```
void *request_new_page  ();
void *request_new_pages (int number_of_pages);
```

Getting one page is just a special case of getting several ones:

[120a] ⟨*function implementations* 100b⟩+≡ (44a) ◁119b 120b▷
```
void *request_new_page () { return request_new_pages (1); }
```
Defines:
request_new_page, used in chunks 164a, 211a, and 608b.
Uses request_new_pages 120b.

The real work must be done here:

[120b] ⟨*function implementations* 100b⟩+≡ (44a) ◁120a 122d▷
```
void *request_new_pages (int number_of_pages) {
  ⟨find contiguous virtual memory range 120c⟩
  ⟨enter frames in page table 121a⟩
};
```
Defines:
request_new_pages, used in chunks 119c, 120a, and 608b.

That function is more useful, but there's a lot to do in the two code chunks:

contiguous virtual memory

- In ⟨*find contiguous virtual memory range* 120c⟩ we need to find a contiguous block of virtual memory addresses which are unused so far. That's important if, say, we want to reserve memory for a large array of data which will be spread across several pages: We need the virtual address range to have no "holes" so that accessing array entries by index and pointer arithmetic work properly. With 1 GByte of virtual (kernel) memory available we expect that this will always be possible. In order to find unmapped pages we use the function mmu_p$_{171c}$(), an improved version of pageno_to_frameno$_{116e}$.
- For the next step ⟨*enter frames in page table* 121a⟩ we must reserve a frame for each page we want to map and enter its address in the page table.

[120c] ⟨*find contiguous virtual memory range* 120c⟩≡ (120b)
```
unsigned int first_page = 0xc0000;  // first page
unsigned int count = 0;             // number of contiguous pages
while (count < number_of_pages && first_page+count ≤ 0xfffff) {
  if ( mmu_p (current_as, first_page + count) == -1 ) {
    count++;
  } else {
    // the block we just looked at is too small
    first_page += (count+1);        // restart search
    count = 0;
  }
}
if (count != number_of_pages)
  return NULL;   // could not find a sufficiently large area
```
Uses current_as 170b, first_page, mmu_p 171c, and NULL 46a.

(The function mmu_p$_{171c}$ does the same as pageno_to_frameno$_{116e}$, see page 116, but it works with address spaces which we have not yet defined—we'll introduce them when we implement the process system. current_as$_{170b}$ refers to the current address space, which you can ignore right now.)

There is one condition under which simply entering the data in the page table will fail: if we fill the last entry of the page table, it will afterwards be full, and the next attempt

to create a new page will find no place to store it. Then it will be too late to create a new page table (because that new page table must also have a virtual address).

So we check now whether we're attempting to fill the last entry.

⟨*enter frames in page table* 121a⟩≡ (120b) 121b▷ [121a]

```
for (int pageno = first_page;  pageno < first_page+count;  pageno++) {
  int newframeid = request_new_frame ();  // get a fresh frame for this page
  if (newframeid == -1) {                 // exit if no frame was found
    // this can only happen if the swap file is full
    return NULL;
  }
  unsigned int pdindex = pageno/1024;
  unsigned int ptindex = pageno%1024;
  page_table *pt;
  if (ptindex == 0 && !current_pd->ptds[pdindex].present) {
    // new page table!
    ⟨create new page table 122a⟩
    newframeid = request_new_frame ();    // get yet another frame
    if (newframeid == -1) {
      return NULL;                        // exit if no frame was found
      // again, this can only happen if the swap file is full
    }
  };
```

Uses current_pd 105a, first_page, NULL 46a, page_table 101b, and request_new_frame 118b.

Now we need to access the page directory and the right page table again:

⟨*enter frames in page table* 121a⟩+≡ (120b) ◁121a 121c▷ [121b]

```
  if ( !current_pd->ptds[pdindex].present ) {
    // we don't have that page table -- this should not happen!
    kputs ("FAIL! No page table entry\n");
    return NULL;
  } else {
    // get the page table
    pt = (page_table*)( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
    // enter the frame address
    KMAP ( &(pt->pds[ptindex]), newframeid * PAGE_SIZE );
    // invalidate cache entry
    asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );
  };
```

Uses current_pd 105a, KMAP 101a, kputs 335b, NULL 46a, PAGE_SIZE 112a, page_table 101b, and PHYSICAL 116a.

(The last line executes the `invlpg` instruction which invalidates the cache entry for the modified page if one exists.) `invlpg`

Finally, we clear the new page and return a pointer to the new page:

⟨*enter frames in page table* 121a⟩+≡ (120b) ◁121b [121c]

```
  memset ((void*) (pageno*4096), 0, 4096);
}
return ((void*) (first_page*4096));
```

Uses first_page and memset 596c.

If a page table does not yet exist, it has to be created and referenced from the page directory. On-the-fly creation of a new page table works like this: We have a frame at `newframeid` which we can use, its physical address is `newframeid << 12`. It is not mapped (so it has no virtual address); we'll deal with this fact later. We can use our $\texttt{PHYSICAL}_{116a}$ function to "talk" to it directly: The address is $\texttt{PHYSICAL}_{116a}\texttt{(newframeid<<12)}$. So we create the page table and fill it with zeroes:

[122a] ⟨*create new page table* 122a⟩≡ (121a) 122b▷
```
pt = (page_table*) PHYSICAL(newframeid<<12);
memset (pt, 0, PAGE_SIZE);
```
Uses memset 596c, PAGE_SIZE 112a, page_table 101b, and PHYSICAL 116a.

We need to tell the page directory that this page table is responsible for the next chunk of memory. When we calculated `pdindex` and `ptindex` above, we found that `ptindex` is 0, and `pdindex` points to the page directory entry that is currently empty. So it is just `pdindex` which we have to use:

[122b] ⟨*create new page table* 122a⟩+≡ (121a) ◁122a 122c▷
```
// KMAPD ( &(current_pd->ptds[pdindex]), newframeid << 12 );
```
Uses current_pd 105a and KMAPD 103c.

The line above is commented out; it used to work before we introduced processes (and address spaces). The following code is a variation of the above line which updates all page directories (each process has its own one). This will become clear when you reach the process chapter.

[122c] ⟨*create new page table* 122a⟩+≡ (121a) ◁122b
```
for (addr_space_id asid=0; asid<1024; asid++) {
  if (address_spaces[asid].status == AS_USED) {  // is this address space in use?
    page_directory *tmp_pd = address_spaces[asid].pd;
    KMAPD ( &(tmp_pd->ptds[pdindex]), newframeid << 12 );
  }
}
```
Uses addr_space_id 158b, address_spaces 162b, AS_USED 162a, KMAPD 103c, and page_directory 103d.

Note: these new page tables only exist physically. Their frames are marked as used, but no virtual addresses point to them. Is that a problem? We can always get their physical addresses through the page directory. So we should be fine.

4.6.2 Releasing Pages

Now we are able to request new pages, but occasionally we will also want to release them. That is much simpler: In order to release a page, we simply have to

[122d] ⟨*function implementations* 100b⟩+≡ (44a) ◁120b 123d▷
```
void release_page (unsigned int pageno) {
  ⟨remove page to frame mapping from page table 123a⟩
  ⟨release corresponding frame 123c⟩
};
```
Defines:
release_page, used in chunks 123d, 166, 167, and 169a.

First we have to get rid of the page mapping, i. e., we need to find the entry in the correct page table and replace it with a null entry. The lookup code is similar to the code for creating the new entry (in ⟨*enter frames in page table* 121a⟩). However, we first test whether a page mapping exists, because if not, we can return immediately:

⟨*remove page to frame mapping from page table* 123a⟩≡ (122d) 123b▷ [123a]
```
// int frameno = pageno_to_frameno (pageno);
int frameno = mmu_p (current_as, pageno);  // we will need this later
if ( frameno == -1 )  { return; }          // exit if no such page
```
Uses current_as 170b, mmu_p 171c, and pageno_to_frameno 116e.

(As you can see, this code originally used pageno_to_frameno$_{116e}$. However, with the introduction of address spaces, this does not work any longer since pageno_to_frameno$_{116e}$ is only aware of the first address space (that belongs to the kernel). As already mentioned, we will provide an mmu_p$_{171c}$ function which is very similar to pageno_to_frameno$_{116e}$, but takes an extra argument which lets us specify the address space. The variable current_as$_{170b}$ always stores the ID of the currently active address space.)

Next we look up the right entry and set it to zero:

⟨*remove page to frame mapping from page table* 123a⟩+≡ (122d) ◁123a [123b]
```
unsigned int pdindex = pageno/1024;
unsigned int ptindex = pageno%1024;
page_table *pt;
pt = (page_table*)( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );

// write null page descriptor
fill_page_desc ( &(pt->pds[ptindex]), false, false, false, false, 0 );

// invalidate cache entry
asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );
```
Uses current_pd 105a, fill_page_desc 100c, page_table 101b, PHYSICAL 116a, and write 429b.

We need to invalidate the cache entry for this page so that any further access to addresses inside the page lead to page faults.

Lastly, we free the frame—we have the release_frame$_{119b}$ function for that:

⟨*release corresponding frame* 123c⟩≡ (122d) [123c]
```
release_frame (frameno);  // note: this increases free_frames
```
Uses free_frames 112b and release_frame 119b.

That's all there is to it.

Sometimes we will want to release a whole consecutive range of pages, so we'll add an extra function for this purpose:

⟨*function implementations* 100b⟩+≡ (44a) ◁122d 133b▷ [123d]
```
void release_page_range (unsigned int start_pageno, unsigned int end_pageno) {
  for (int i = start_pageno;  i < end_pageno+1;  i++)  release_page (i);
};
```
Defines:
release_page_range, used in chunk 608b.
Uses release_page 122d.

4.7 Next Steps

The system is now initialized and uses paging to manage the physical memory. However, whenever something goes wrong, the system will halt or reboot, and it cannot access any hardware, except the video card: Since the video memory can be accessed like normal RAM, we can display status or error messages, but that is all. Especially the code allows no interaction, because we cannot read the keyboard's status.

In the upcoming chapter we describe the mechanism that is required to handle faults (something went wrong) and interrupts (some device wants to inform the system about an event). With interrupts we can also have timers, and that brings us one step closer to working with processes which need working timers so that we can switch back and forth between several programs that run simultaneously.

4.8 Exercises

This is the first set of exercises. You will need the development environment that you can download from the Ulix website (`http://www.ulixos.org/`). Install the virtual machine in VirtualBox and login as `ulix` (with password `ulix`).

Tutorial 1

1. On the Ulix development system, locate the directory `/home/ulix/tutorial/01/`. This directory contains parts of the code that we've presented so far; we've added the `printf` function which lets the C program write to the screen.

   Read the source code files `ulix.c` and `start.asm`. Compile the sources with `make` and run the kernel with `make run`. You should see the following output:

```
  Booting 'ULIX-i386  (c) 2008-2011 Felix Freiling & Hans-Georg Esser'

root (fd0)
 Filesystem type is fat, using whole disk
kernel /ulix.bin
   [Multiboot-elf, <0x100000:0x46:0x0>, <0x100050:0xfb0:0x0>, <0x101000:0x0:0x9
000>, shtab=0x10a190, entry=0x10002a]


_



Hello World! This is not Ulix yet :)

address of main() [ulix.c]:    c01005b0
address of start  [start.asm]: 0010002a
stack: c0101008 - c0109008
```

2. Obviously, the kernel executes the C function `main`. How does the system jump from the early assembler code (beginning with the `start` label) into the C function?
3. The file `ulix.dump` contains a listing of the generated assembler code. Here you can find all labels from the assembler file `start.asm` and also the function names from `ulix.c`. Search the file for the labels `start`, `higherhalf` and `main` and check for which memory addresses the corresponding code has been generated. (You find the memory addresses on the very left in hexadecimal format without leading "0x".)

Do you recognize the GDT trick in the assembler code? It is initiated via a "long jump" (`jmp`) with a logical address `segment:address` as jump target.

Do also search for the labels `stack_first_address` and `stack_last_address` and compare the shown addresses with those from the VM's output (in the last line)—they should match.

4. The kernel uses the `printf()` function (you can find its code in the separate `printf.c` file) for text output, but its main task to is to format the output as requested by the format strings (such as `%s` for strings or `%d` for integers) and send it to the terminal character-wise via the `kputch` function whose implementation resides in `ulix.c`.

   How does `kputch` write characters to the screen? This will also be explained later when we discuss the corresponding code section of the full Ulix sources. It may help to search the web or this book for "0xb8000" and "video". Consider how pointers can be used for accessing memory: The commands

   ```
   char *mem;  mem = (char*) 0x1234; *mem = 'a';
   ```

   can write the byte 'a' (ASCII value: `0x61`) to the memory address `0x1234`.

5. Why does the following line from `kputch()`

   ```
   screen = (char*) 0xc0000000 + 0xb8000 + posy*160 + posx*2;
   ```

   use multiplicators 160 and 2, and why does it add `0xc0000000`?

6. Use the command `objdump -h ulix.bin` to check which memory areas are used by the three sections `.setup`, `.text` and `.bss`. (You can ignore the additional sections `.comment`, `.stab` and `.stabstr`.) Compare the values with the information that the boot loader GRUB outputs in the "Multiboot-elf, ..." line when it loads the kernel.

7. The folder `tutorial/02/` in the `ulix/` directory contains an improved version of the Ulix kernel which implements paging. Tutorial 2

   Read the source code files `ulix.c` and `start.asm` and localize the code chunks which you have seen in this chapter. (There is also additional code that we have not discussed yet.)

8. Compile/assemble and link the code with `make` and start the system with `make run`.

9. In `ulix.c` the `kputch()` function has seen some slight changes, there is now the following code block:

   ```
   if (paging_ready)
     screen = (char*) 0xb8000 + posy*160 + posx*2;
   else
     screen = (char*) 0xc0000000 + 0xb8000 + posy*160 + posx*2;
   ```

   It evaluates the `paging_ready` variable which is initially set to `false` and changed to `true` after paging has been initialized. In that case, adding `0xc0000000` is no longer necessary for calculating the address (cf. exercise 5). Why does that work?

10. (Literate Programming) Convert the files `ulix.c` and `start.asm` (from `tutorial/02/`) into a literate program `tutorial02.nw`. You can restrict the documentation that you add to some keywords and roughly follow the ordering of descriptions in this chapter.
11. Test that you can reconvert the literate program into the original code files (or at least sufficiently similar versions which also compile and generate a working kernel). Also create a LaTeX file and from that a PDF version.

Tutorial 3

12. In the folder `tutorial/03/` you find yet another version of the Ulix kernel which contains an improved version of the paging code. It is a literate program (`ulix.nw`). `cd` into the folder and use `make` to extract the source code files `ulix.c` und `start.asm` from `ulix.nw`. They will automatically be compiled or assembled. Then launch the kernel with `make run`. The system performs some tests of memory management and then halts.

    Also take a look at the PDF file `ulix.pdf` (this file is written in German language, so it is likely that you want to skip this step) that you can recreate with `make pdf` if you apply changes to `ulix.nw`. In the document you can find a description of the code for frame and page management; it is also a sample solution for exercise 10. Compare how the code is broken down into code chunks in your own solution and in the sample solution.

13. We have defined the page table descriptors (`page_table_desc`$_{102}$) and the page descriptors (`page_desc`$_{100a}$) as structures. Since they are both exactly 32 bits large, there is an alternative interpretation as `unsigned int`. The goal of this exercise is to modify the literate program so that it works with these simple integer types instead of the structures.

    First, create a copy of the folder (so that you can keep the original files). If you `cd` into the `tutorial` folder, you can do that with the

    ```
    cp -r 03 03-copy
    ```

    command and make all your changes in `03-copy/`.

    a) Start with changing the type declarations for `page_table_desc` und `page_desc` to

    ```
    typedef unsigned int page_table_desc;
    typedef unsigned int page_desc;
    ```

    b) The types `page_directory` and `page_table` are not needed any longer; instead you can declare individual directories or tables like this:

    ```
    page_table_desc pd[1024] __attribute__ ((aligned (4096)));
    page_desc       pt[1024] __attribute__ ((aligned (4096)));
    ```

    That way, you can access entry `n` of `pt` via `pt[n]` instead of `pt.pds[n]`. The expression `pt` (without an index) serves as a pointer (of type `unsigned int*`) to the start of the table. If you need to hand over a single descriptor to functions like `fill_page_desc()` or `fill_page_table_desc()`, you can pass them a pointer to `pt[n]`, i. e., `&pt[n]`.

c) After the changes all functions which work with these types are broken, you need to modify them. For example, in order to fill a page descriptor in `fill_page_desc()`, you can take an address `frame_addr` and set its lowest twelve bits to 0:

```
tmpvalue = frame_addr & 0b11111111111111111111000000000000;
```

or

```
tmpvalue = frame_addr & 0xFFFFF000;
```

(The hexadecimal number combines four bits to one hex digit.) Then you add the flags in the lower twelve bits. You could define flag constants which are based on the bit positions, e. g.,

```
#define FLAG_PRESENT  1<<0   //  Bit 0: present
#define FLAG_ACCESSED 1<<5   //  Bit 5: accessed
```

etc. Then use a bitwise "or" operation (`|`), e. g.

```
tmpvalue = tmpvalue | FLAG_ACCESSED;
```

to set a specific bit. When everything is done, you can write the value with `*desc = tmpvalue;`. (You must also modify the function prototype and pass a pointer to `unsigned int` when you call the function.)

d) In order to extract the address from a descriptor, you simply set the lowest twelve bits to 0 (like above). On the other hand, you can extract single flags by performing a bitwise "and" operation (`&`) with the appropriate `FLAG_*` constant and test whether the result is 0 or not:

```
if ((descriptor & FLAG_ACCESSED) == 0) { /* flag is not set */ }
```

14. Verify that the old and new program versions work identically. In both versions you can use the $\texttt{hexdump}_{\texttt{612c}}\texttt{()}$ function which displays a memory region as hex dump. Use the starting and ending addresses of the page table or page directory as arguments, e. g.

```
hexdump ( (unsigned int)current_pd, (unsigned int)current_pd + 4096 );
```

(which will output the whole page-sized table). The output is written to the file `output.txt` (in the current directory of the development VM), you can later compare them:

```
cd 03/;       make;  make run > output.txt
cd 03-copy/;  make;  make run > output.txt
cd .. ;  diff ~/03*/output.txt
```

If `diff` creates no output, the two files are identical. (The $\texttt{hexdump}_{\texttt{612c}}\texttt{()}$ calls must occur after the test changes to the tables.)

15. Check with `make pdf` that your literate program can still be converted to a PDF file—if that does not work, identify and remove the errors.

5
Interrupts and Faults

All modern CPUs and even many of the older ones such as the Zilog Z80 8-bit processor can be interrupted: the CPU has an input line which can be triggered by an external device connected to this line. When such an interrupt occurs, the current activity is suspended, and the CPU continues operation at a specified address: it executes an interrupt handler.

In principle a device could be directly connected to the CPU, but modern machines contain many devices which want to interrupt the processor, e. g. the disk controllers, the keyboard controller, the serial ports, or the on-board clock. Thus an extra device, called the *interrupt controller*, intermediates between the other devices and the CPU. One of the advantages of such an interrupt controller is that it is programmable: it is possible to enable or disable specific interrupts whereas the CPU itself can only completely enable or disable all interrupts, using the `sti` (set interrupt flag) and `cli` (clear interrupt flag) instructions. (These machine instructions exist on Intel-x86-compatible CPUs; other chips have similar instructions.) Being programmable also means that interrupt numbers can be remapped (we will see later why this is helpful). Interrupt controllers with these features are called *programmable interrupt controllers* (PICs), and we'll use that abbreviation throughout the rest of this chapter.

interrupt controller

`sti`, `cli`

PICs

After the implementation of interrupts we will also take a look at fault handling since the involved mechanisms are very similar to those which we need for handling interrupts. As we mentioned in the introduction, the main difference between interrupts and faults is that faults occur as a direct consequence of some specific instruction that our code executes. In that sense they are *synchronous*. Interrupts on the other hand occur without any connection to the currently executing instruction, since they are not triggered (immediately) by our code but by some device. That is why they are called *asynchronous*.

synchronous

asynchronous

5.1 Examples for Interrupt Usage

Interrupt handling is a core functionality which is used in lots of places: without interrupts we would not be able to build a useful operating system.

Let's look at some example features of ULIX which depend heavily on interrupts:

Multi-tasking ULIX can execute several processes in parallel and switch between them using a simple round-robin scheduling mechanism. That is only possible because the clock chip on the motherboard regularly generates timer interrupts, and ULIX installs a timer interrupt handler which—when activated—calls the scheduler to check whether it is time to switch to a different process. If there were no interrupts, we could only implement *non-preemptive scheduling* which relies on the processes to give up the CPU voluntarily.

timer handler

Keyboard input Whenever you press or release a key on a PC, either event generates an interrupt. ULIX picks up these interrupts and the keyboard interrupt handler reads a key press or key release code from the controller.

keyboard handler

A keyboard driver does not need interrupts, but the alternative is to constantly poll (query) the keyboard controller in order to find out whether a new event has occurred. That's possible but wastes a lot of CPU time. Polling does not work well in a multi-tasking environment. (However for a single-tasking operating system it may be good enough.)

polling

Media Reading and writing hard disks and floppy disks also depends on interrupts: In the ULIX implementation of filesystems (and disk access) a process which wants to read or write makes a system call which sends a request to the drive controller. Then ULIX puts the calling process to sleep. Once the request has been served, the drive controller generates an interrupt, and the interrupt handler for the hard disk controller or the floppy disk controller (these are two separate handlers) deals with the data and wakes up the sleeping process.

ATA/FDC handlers

Again, this could be done without interrupts. But the process would have to remain active and continuously poll the controller to find out whether the data transfer has been completed.

Serial ports Finally, the serial ports are similar to the keyboard, since all of them are *character devices*: they transfer single bytes (instead of blocks of bytes).

5.2 Interrupt Handling on the Intel Architecture

Intel 8259 PIC

The classical IBM PC used the Intel 8259 Programmable Interrupt Controller, compatible descendents of which are still used in modern computers. The 8259 has eight input lines (through which up to eight separate devices may connect) and one output line which forwards received interrupt signals to the CPU. It is possible to use more than one 8259 PIC

since these controllers can be cascaded which means that a second controller's output pin is connected with one of the first controller's input pins (typically the one for device 2, see Figure 5.1). With that cascade, devices connected to the first controller keep their normal numbers (0, 1, 3–7 with 2 reserved for the second controller), and devices connected to the second controller use device numbers between 8 and 15, allowing for a total of 15 (= 16 − 1) separate device numbers. The first or primary controller is called *Master PIC*, the second one is the *Slave PIC* (as it is not directly connected to the CPU but relies on the master PIC to have its interrupts signals forwarded). The numbers 0–15 are called *Interrupt Request Numbers* (IRQs).

Cascading PICs

Master PIC

Slave PIC

IRQ

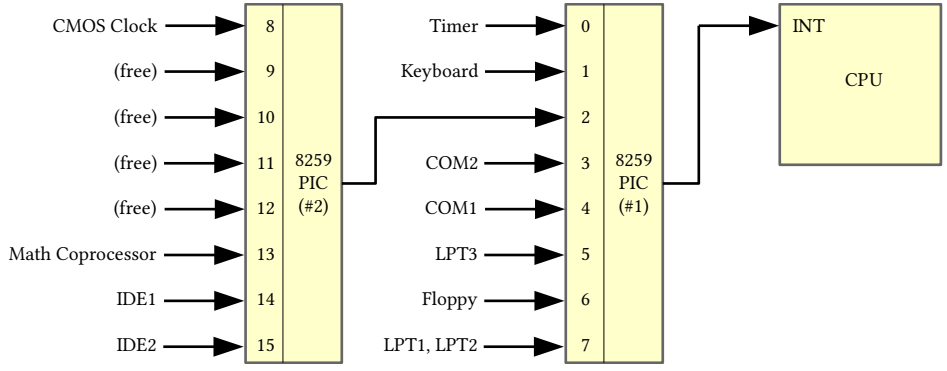


Figure 5.1: Two PICs are cascaded, which allows for 15 distinct interrupts.

As you can see from the figure, there is a fixed mapping of some devices to specific IRQs. We will use the following IRQs in the Ulix implementation:

- **0: Timer Chip**. On a PC's mainboard you can find a (programmable) timer chip which regularly generates interrupts. We will use timer interrupts to call the scheduler (besides other tasks).
- **1: Keyboard**. This is the interrupt generated by PS/2 keyboards. A USB keyboard would be handled differently, but we do not support USB devices.
- **2: Slave PIC**. As already mentioned, IRQ 2 is reserved for connecting the secondary (slave) PIC.
- **3: Serial Port 2**. The second serial port will be used for our implementation of what we've called the *serial hard disk*—you can find it in Chapter 13.4.
- **4: Serial Port 1**. We only use the first serial port for output (when running Ulix in a PC emulator), thus we will not install an interrupt handler for this IRQ.
- **6: Floppy**. This is the IRQ for the floppy controller. It can handle up to two floppy drives.
- **14: Primary IDE Controller**. And finally, 14 is the IRQ of the primary IDE controller. Many PC mainboards contain two controllers, with each of them allowing two drives to connect. The secondary IDE controller would generate the interrupt number 15,

but we're going to support only one controller.

We can define names for the IRQ numbers right now:

[132] ⟨*constants* 112a⟩+≡ (44a) ◁112a 134▷

```
#define IRQ_TIMER      0
#define IRQ_KBD        1
#define IRQ_SLAVE      2     // Here the slave PIC connects to master
#define IRQ_COM2       3
#define IRQ_COM1       4
#define IRQ_FDC        6
#define IRQ_IDE       14     // primary IDE controller; secondary has IRQ 15
```

Defines:
IRQ_COM1, used in chunk 344c.
IRQ_COM2, used in chunks 344c and 520a.
IRQ_FDC, used in chunk 552c.
IRQ_IDE, used in chunk 534b.
IRQ_KBD, used in chunk 323b.
IRQ_SLAVE, used in chunk 139b.
IRQ_TIMER, used in chunk 339a.

5.2.1 Using Ports for I/O Requests

Going where? We want to initialize the PICs, which means directly talking to these controllers. Like with most other devices we can use the machine instructions `in` and `out` to find out the PIC's current status and tell it what to do.

Here we provide the code which lets us access the controllers.

I/O Ports Access to many hardware components (including the PICs) is possible via *I/O ports*. Using `in` and `out` machine instructions it is possible to transfer bytes, words or doublewords between a CPU register and a memory location or register on some device (such as a hard disk controller).

The Intel 80386 Programmer's Reference Manual [Int86, pp. 146–147] explains:

`in`, `out`

> "The I/O instructions `IN` and `OUT` are provided to move data between I/O ports and the *EAX* (32-bit I/O), the *AX* (16-bit I/O) or *AL* (8-bit I/O) general registers. `IN` and `OUT` instructions address I/O ports either directly, with the address of one of up to 256 port addresses coded in the instruction, or indirectly via the *DX* register to one of up to 64K port addresses.
>
> `IN` (Input from Port) transfers a byte, word or doubleword from an input port to *AL*, *AX* or *EAX*. If a program specifies *AL* with the `IN` instruction, the processor transfers 8 bits from the selected port to *AL*. If a program specifies *AX* with the `IN` instruction, the processor transfers 16 bits from the port to *AX*. If a program specifies *EAX* with the `IN` instruction, the processor transfers 32 bits from the port to *EAX*.
>
> `OUT` (Output to Port) transfers a byte, word or doubleword to an output port from *AL*, *AX* or *EAX*. The program can specify the number of the port using the same methods as the `IN` instruction."

For accessing 8-bit, 16-bit and 32-bit ports, the Intel assembler language provides separate commands `inb` / `outb` (byte), `inw` / `outw` (word) and `inl` / `outl` (long: doubleword) which make it explicit what kind of transfer is wanted. We'll use them in the functions

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁119c 138b▷ [133a]

```
byte inportb (word port);
word inportw (word port);
void outportb (word port, byte data);
void outportw (word port, word data);
```

There are several possible C implementations with inline assembler code, the following code is most readable:

⟨ *function implementations* 100b ⟩+≡ (44a) ◁123d 138c▷ [133b]

```
byte inportb (word port) {
  byte retval;  asm volatile ("inb %%dx, %%al" : "=a"(retval) : "d"(port));
  return retval;
}

word inportw (word port) {
  word retval;  asm volatile ("inw %%dx, %%ax" : "=a" (retval) : "d" (port));
  return retval;
}

void outportb (word port, byte data) {
  asm volatile ("outb %%al, %%dx" : : "d" (port), "a" (data));
}

void outportw (word port, word data) {
  asm volatile ("outw %%ax, %%dx" : : "d" (port), "a" (data));
}
```

Defines:
`inportb`, used in chunks 140a, 320b, 336b, 339d, 344c, 345c, 519d, and 532–34.
`outportb`, used in chunks 135, 139e, 146a, 328a, 336b, 338c, 339d, 344c, 345c, 526, 527, 534b, and 552c.
`outportw`, used in chunk 133a.

We could provide `inportl` and `outportl` (for 32-bit values) in a similar fashion, using `inl`, `outl` and the 32-bit register *EAX* (instead of the 16-bit and 8-bit versions *AX* and *AL*), but we do not need them. (Remember that *EAX*, *AX* and *AL* are (parts of) the same register, see Figure 5.2. On a 64-bit machine, *RAX* is the 64-bit extended version of *EAX*.)

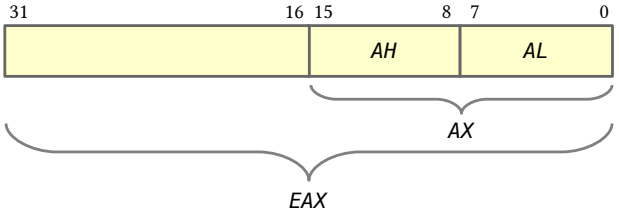


Figure 5.2: The lower half of *EAX* is *AX* which in turn is split into *AH* (high) and *AL* (low).

5.2.2 Initializing the PIC

Going where?

Now that we have functions for talking to devices we can set up the two PICs. We will configure one as master and the other as slave, and we also remap the interrupt numbers from 0–15 to 32–47 because the first 32 numbers are reserved for faults (see Section 5.3).

The PICs can be accessed via the following four ports:

[134] ⟨*constants* 112a⟩+≡ (44a) ◁132 145a▷

```
// I/O Addresses of the two programmable interrupt controllers
#define IO_PIC_MASTER_CMD   0x20  // Master (IRQs 0-7), command register
#define IO_PIC_MASTER_DATA  0x21  // Master, control register

#define IO_PIC_SLAVE_CMD    0xA0  // Slave (IRQs 8-15), command register
#define IO_PIC_SLAVE_DATA   0xA1  // Slave, control register
```

Defines:
IO_PIC_MASTER_CMD, used in chunks 135a and 146a.
IO_PIC_MASTER_DATA, used in chunks 135, 139e, and 140a.
IO_PIC_SLAVE_CMD, used in chunks 135a and 146a.
IO_PIC_SLAVE_DATA, used in chunks 135, 139e, and 140a.

They need to be initialized by sending them four "Initialization Command Words" (ICW) called ICW1, ICW2, ICW3 and ICW4 in a specific order, using specific ports. Each of the PICs has a command register and a data register. During normal operation we can write to the data register (using the ports $\texttt{IO\_PIC\_MASTER\_DATA}_{134}$ and $\texttt{IO\_PIC\_SLAVE\_DATA}_{134}$ for PIC1 or PIC2, respectively) to set the *interrupt mask*: That's a byte where each bit tells the controller whether it shall respond to a specific interrupt (1 means: mask, i. e., ignore the interrupt; 0 means: forward it to the CPU). We will start with an interrupt mask of `0xFF` for each controller (all bits are 1), thus all hardware interrupts will be ignored.

interrupt mask

The following code was taken from Bran's Kernel Development Tutorial [Fri05] (e. g. from the source file `irq.c`) and modified.

For programming the controller, we can send configuration data to the data port, but we have to initialize the programming by writing to the command port. The complete sequence is as follows:

- First we send ICW1 to both PICs. ICW1 is a byte whose bits have the following meaning [Int88, p. 11]:

  0 D_0: ICW4 needed? We set this to 1 since we want to program the controller.

  1 D_1: Single (1) / Cascade (0) mode: We set this to 0 since there's a slave.

  2 D_2: Call Address Interval (ignored), the default value is 0.

  3 D_3: Level (1) / Edge (0) Triggered Mode: we set this to 0.

  4 D_4: Initialization Bit: We set it to 1 because we want to initialize the controller.

  5,6,7 D_5, D_6, D_7: not used on x86 hardware, set to 0.

  This results in the byte `00010001` (`0x11`). The value is the same for both PICs. As mentioned before, ICW1 must be sent to the PICs' command registers.

⟨*remap the interrupts to 32..47* 135a⟩≡ (139b) 135b▷ [135a]
```
outportb (IO_PIC_MASTER_CMD,  0x11);  // ICW1: initialize; begin programming
outportb (IO_PIC_SLAVE_CMD,   0x11);  // ICW1: dito, for PIC2
```
Uses IO_PIC_MASTER_CMD 134, IO_PIC_SLAVE_CMD 134, and outportb 133b.

- In the next step we send ICW2 to the PICs' data registers. The lowest three bits specify the offset for remapping the interrupts. Since the first 32 interrupts must be reserved for processor exception handlers (e. g. "Division by Zero" and "Page Fault" handlers), we map the interrupts 0–15 to the range 32–47 (`0x20` – `0x2f`).

  remap the interrupts

  Each PIC would normally generate interrupts in the range 0–7, thus the offset is not the same for both PICs: For PIC1 it is `0x20` (32; mapping 0–7 to 32-39), and for PIC2 it is `0x28` (40; mapping 0–7 to 40–47).

⟨*remap the interrupts to 32..47* 135a⟩+≡ (139b) ◁135a 135c▷ [135b]
```
outportb (IO_PIC_MASTER_DATA, 0x20);  // ICW2 for PIC1: offset 0x20
                                      // (remaps 0x00..0x07 -> 0x20..0x27)
outportb (IO_PIC_SLAVE_DATA,  0x28);  // ICW2 for PIC2: offset 0x28
                                      // (remaps 0x08..0x0f -> 0x28..0x2f)
```
Uses IO_PIC_MASTER_DATA 134, IO_PIC_SLAVE_DATA 134, and outportb 133b.

- The next command word is ICW3. Its functionality depends on whether we send it to the master (PIC1) or the slave (PIC2): The PICs already know that they are master and slave (because we sent that information as part of ICW1) [Int88, p. 12].

  The master expects a command word byte in which each set bit specifies a slave connected to it. We have only one slave and want to make it signal new interrupts on interrupt line 2 of the master. Thus, only the third bit (from the right) must be set: $\texttt{00000100}_b$ = `0x04`.

  The slave needs a slave ID. We give it the ID 2 = `0x02`.

⟨*remap the interrupts to 32..47* 135a⟩+≡ (139b) ◁135b 135d▷ [135c]
```
outportb (IO_PIC_MASTER_DATA, 0x04);  // ICW3 for PIC1: there's a slave on IRQ 2
                                      // (0b00000100 = 0x04)
outportb (IO_PIC_SLAVE_DATA,  0x02);  // ICW3 for PIC2: your slave ID is 2
```
Uses IO_PIC_MASTER_DATA 134, IO_PIC_SLAVE_DATA 134, and outportb 133b.

- To end the sequence, we send ICW4 which is just `0x01` for x86 processors [Int88, p. 12].

⟨*remap the interrupts to 32..47* 135a⟩+≡ (139b) ◁135c [135d]
```
outportb (IO_PIC_MASTER_DATA, 0x01);  // ICW4 for PIC1 and PIC2: 8086 mode
outportb (IO_PIC_SLAVE_DATA,  0x01);
```
Uses IO_PIC_MASTER_DATA 134, IO_PIC_SLAVE_DATA 134, and outportb 133b.

With the remapping in place we can now create entries for the interrupt handler table—we need some new data structures for them.

5.2.3 Interrupt Descriptor Table

Going where?

The PICs are initialized and will do the right thing when an interrupt occurs, but we haven't told the CPU yet what to do when it receives one. This calls for a new data structure, the *Interrupt Descriptor Table*, which we must define according to the Intel standards and fill with proper values.

`lidt`

While the first Intel-8086/8088-based personal computers used a fixed address in RAM to store the interrupt handler addresses, modern machines let us place the table anywhere in memory. After preparing the table we must use the machine instruction `lidt` (load interrupt descriptor table register) to tell the CPU where to search.

The procedure we need to follow is similar to the one for activating segmentation via a GDT (see pages 90–92):

1. We first store interrupt descriptors (each of which is eight bytes large) in a table consisting of `struct idt_entry`$_{137a}$ entries,
2. then we create some kind of pointer structure `struct idt_ptr`$_{137b}$ which contains the length and the start address of the table,
3. and finally we execute `lidt` (compare this to `lgdt` for the GDT).

Figure 5.3 shows the layout of an IDT entry. The *Flags* halfbyte (second line, left in the figure) consists of

- the present flag (bit 3) which must always be set to 1,
- two bits (2 and 1) for the Descriptor Privilege Level (DPL). We will always set this to $11_b = 3$ since we want all interrupts to be available all the time (when we're in kernel or user mode) and
- a so-called "storage segment" flag (bit 0; which must be set to 0 for an "interrupt gate", see next entry).

The *Type* halfbyte declares what kind of descriptor this is: we will always set it to 1110_b, making this descriptor an

interrupt gate

- *80386 32-bit interrupt gate* descriptor (which is what we want).

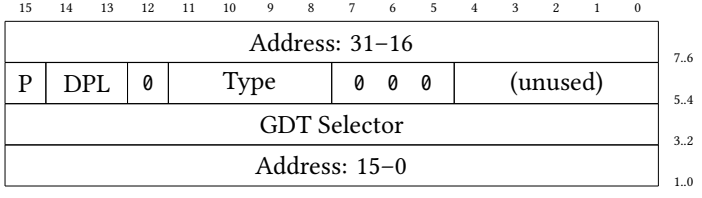


Figure 5.3: An Interrupt Descriptor contains the address of an interrupt handler and some configuration information.

Besides this type, there are alternatives:

- 0101_b for an *80386 32-bit task gate*,
- 0110_b for an *80286 16-bit interrupt gate*,
- 0111_b for an *80286 16-bit trap gate* and
- 1111_b for an *80386 32-bit trap gate*, trap gate

but we will not go into the details about these. Instead of interrupt gates we could also use trap gates, the difference between those being that "for interrupt gates, interrupts are automatically disabled upon entry and reenabled upon IRET which restores the saved *EFLAGS*" [OSD13]. We will use a trap gate for the system call handler (see Chapter 6.4).

An interrupt descriptor table entry is described by the following datatype definitions:

⟨ *type definitions* 91 ⟩+≡ (44a) ◁103d 137b▷ [137a]

```
struct idt_entry {
    unsigned int addr_low  : 16;   // lower 16 bits of address
    unsigned int gdtsel    : 16;   // use which GDT entry?
    unsigned int zeroes    :  8;   // must be set to 0
    unsigned int type      :  4;   // type of descriptor
    unsigned int flags     :  4;
    unsigned int addr_high : 16;   // higher 16 bits of address
} __attribute__((packed));
```

Defines:
idt_entry, used in chunks 138a and 146d.

The *selector* must be the number of a code segment descriptor (in the GDT); we will always set this to 0x08 since our kernel (ring 0) code segment uses that number (see code chunk ⟨ *install flat gdt* 110a ⟩ on page 110).

The IDT pointer has the same structure as the GDT pointer: it informs about the length and the location of the IDT:

⟨ *type definitions* 91 ⟩+≡ (44a) ◁137a 161▷ [137b]

```
struct idt_ptr {
    unsigned int limit   : 16;
    unsigned int base    : 32;
} __attribute__((packed));
```

Defines:
idt_ptr, used in chunk 138a.

In theory, an interrupt number can be any byte, i. e., a value between 0 and 255. We will use a full IDT with 256 entries even though most of the entries will be null descriptors—if somehow an interrupt is generated which has a null descriptor, the CPU will generate an "unhandled interrupt" exception. We will talk about exceptions right after we've finished the interrupt handling code.

[138a] ⟨*global variables* 92b⟩+≡ (44a) ◁115a 139a▷
```
struct idt_entry idt[256] = { { 0 } };
struct idt_ptr idtp;
```
Defines:
idt, used in chunks 138c and 146d.
idtp, used in chunks 146d and 147a.
Uses idt_entry 137a and idt_ptr 137b.

The variables $\texttt{idt}_{138a}$ and $\texttt{idtp}_{138a}$ will now be used in a way that is similar to how we used $\texttt{gdt}_{92b}$ (a $\texttt{struct gdt\_entry}_{91}\texttt{[]}$ array) and $\texttt{gp}_{92b}$ (a $\texttt{struct gdt\_ptr}_{92a}$ structure) when we wrote the GDT code.

We start with a function

[138b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁133a 138d▷
```
void fill_idt_entry (byte num, unsigned long address,
    word gdtsel, byte flags, byte type);
```

which writes an entry of the IDT:

[138c] ⟨*function implementations* 100b⟩+≡ (44a) ◁133b 139e▷
```
void fill_idt_entry (byte num, unsigned long address,
    word gdtsel, byte flags, byte type) {
  if (num ≥ 0 && num < 256) {
    idt[num].addr_low  = address & 0xFFFF; // address is the handler address
    idt[num].addr_high = (address >> 16) & 0xFFFF;
    idt[num].gdtsel    = gdtsel;           // GDT sel.: user mode or kernel mode?
    idt[num].zeroes    = 0;
    idt[num].flags     = flags;
    idt[num].type      = type;
  }
}
```
Defines:
fill_idt_entry, used in chunks 138b, 139b, 148b, and 202e.
Uses idt 138a.

Parts of all of our interrupt handlers will be assembler code (which we store in $\texttt{start}_{94}\texttt{.asm}$); we'll explain soon why that has to be. For the moment, let's declare 16 external function symbols $\texttt{irq0}_{144}$, $\texttt{irq1}_{144}$, ..., $\texttt{irq15}_{144}$ whose addresses we're about to enter into the IDT with $\texttt{fill\_idt\_entry}_{138c}$:

[138d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁138b 139c▷
```
extern void irq0(), irq1(), irq2(),  irq3(),  irq4(),  irq5(),  irq6(),  irq7();
extern void irq8(), irq9(), irq10(), irq11(), irq12(), irq13(), irq14(), irq15();
```

We will store the function addresses in an array which simplifies accessing them:

⟨*global variables* 92b⟩+≡ (44a) ◁138a 145b▷ [139a]
```
void (*irqs[16])() = {
  irq0, irq1, irq2,  irq3,  irq4,  irq5,  irq6,  irq7,      // store them in
  irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15      // an array
};
```
Defines:
irqs, used in chunk 139b.
Uses irq0 144, irq1 144, irq10 144, irq11 144, irq12 144, irq13 144, irq14 144, irq15 144, irq2 144, irq3 144, irq4 144, irq5 144, irq6 144, irq7 144, irq8 144, and irq9 144.

The following code chunk enters their address in the IDT:

⟨*install the interrupt handlers* 139b⟩≡ (45b) 323c▷ [139b]
```
⟨remap the interrupts to 32..47 135a⟩
set_irqmask (0xFFFF);          // initialize IRQ mask
enable_interrupt (IRQ_SLAVE);  // IRQ slave

for (int i = 0;  i < 16;  i++) {
  fill_idt_entry (32 + i,
                  (unsigned int)irqs[i],
                  0x08,
                  0b1110,     // flags: 1 (present), 11 (DPL 3), 0
                  0b1110);    // type:  1110 (32 bit interrupt gate)
}
```
Uses enable_interrupt 140b, fill_idt_entry 138c, IRQ_SLAVE 132, irqs 139a, and set_irqmask 139e.

This code chunk sets the *IRQ mask* to 0xFFFF = 1111111111111111_b via IRQ mask

⟨*function prototypes* 45a⟩+≡ (44a) ◁138d 139d▷ [139c]
```
static void set_irqmask (word mask);
```

which disables all interrupts, and then it enables the interrupt for the slave PIC with

⟨*function prototypes* 45a⟩+≡ (44a) ◁139c 139f▷ [139d]
```
static void enable_interrupt (int number);
```

—both functions have not been mentioned so far. The IRQ mask is a 16-bit value in which each bit says whether some interrupt is enabled (value 0) or not (value 1). We must talk to both PICs to set the mask, the master PIC gets the lower eight bits (for the interrupts 0–7), the slave PIC gets the upper eight bits (for the interrupts 8–15):

⟨*function implementations* 100b⟩+≡ (44a) ◁138c 140a▷ [139e]
```
static void set_irqmask (word mask) {
  outportb (IO_PIC_MASTER_DATA, (char)(mask % 256) );
  outportb (IO_PIC_SLAVE_DATA,  (char)(mask >> 8)  );
}
```
Defines:
set_irqmask, used in chunks 139 and 140b.
Uses IO_PIC_MASTER_DATA 134, IO_PIC_SLAVE_DATA 134, and outportb 133b.

We can also read the mask from the two PICs with a similar function we call

⟨*function prototypes* 45a⟩+≡ (44a) ◁139d 146b▷ [139f]
```
word get_irqmask ();
```

in which we read the two data registers instead of writing them:

[140a] ⟨*function implementations* 100b⟩+≡ (44a) ◁139e 140b▷
```
word get_irqmask () {
  return inportb (IO_PIC_MASTER_DATA)
      + (inportb (IO_PIC_SLAVE_DATA) << 8);
}
```
Defines:
  `get_irqmask`, used in chunks 139f and 140b.
Uses `inportb` 133b, `IO_PIC_MASTER_DATA` 134, and `IO_PIC_SLAVE_DATA` 134.

In the following chapters we will often enable a specific interrupt for some device after we've prepared its usage, e. g. for the floppy controller. For that purpose, we will always use `enable_interrupt`$_{140b}$`()` like we did above. It simply reads the current IRQ mask, clears a bit, and writes the new value back:

[140b] ⟨*function implementations* 100b⟩+≡ (44a) ◁140a 146a▷
```
static void enable_interrupt (int number) {
  set_irqmask (
    get_irqmask ()        // the current value
    & ~(1 << number)      // 16 one-bits, but bit "number" cleared
  );
}
```
Defines:
  `enable_interrupt`, used in chunks 139, 323b, 339a, 344c, 520a, 534b, and 552c.
Uses `get_irqmask` 140a and `set_irqmask` 139e.

5.2.4 Writing the Interrupt Handler

Going where? Everything is prepared for interrupt handlers — now we need to define them, i. e., implement the `irq0`$_{144}$`()`, … `irq15`$_{144}$`()` functions. This step requires some assembler code and some C code.

We have installed handlers for all 16 interrupts, but what do they do? We will define part of their code in the assembler file, but we start with a description of what we expect to happen in general.

When an interrupt occurs, the CPU suspends the currently running code, saves some information on the stack, and then jumps to the address that it finds in the IDT. (It also uses a different stack and switches to kernel mode if it was in user mode when the interrupt occurred.) Then the interrupt handler runs, and once it has finished its job, it returns with the `iret` instruction. `iret` is different from the regular `ret` instruction which normal functions use for returning to the calling function: it is the special "return from interrupt" instruction which restores the original state (user or kernel mode, stack, *EFLAGS* register) so that the regular code can continue as if the interrupt had never happened.

`iret`

Switching to the interrupt handler can mean a change of the privilege level that the CPU executes in: So far we've only let Ulix work in ring 0 (kernel mode), but later when we introduce processes it can happen that an interrupt occurs while the CPU runs in ring 3 (user mode). If that is the case, the privilege level changes (from 3 to 0). When such a

transition occurs, the information (return address etc.) is not written to the process' user mode stack, but on the process' kernel stack which is located elsewhere and normally used during the execution of system calls—we'll describe that in more detail later. For now, the relevant piece of information is that different information gets stored on the "target stack": In case of a privilege change the CPU first writes the contents of the *SS* and *ESP* registers on the (new) stack—this does not happen if the CPU was already operating in ring 0. Next, *EFLAGS*, *CS* and *EIP* are written to the stack: that is all we need for returning to the interrupted code. Figure 5.4 shows the different stack contents when the interrupt handler starts executing [Int86, p. 159].

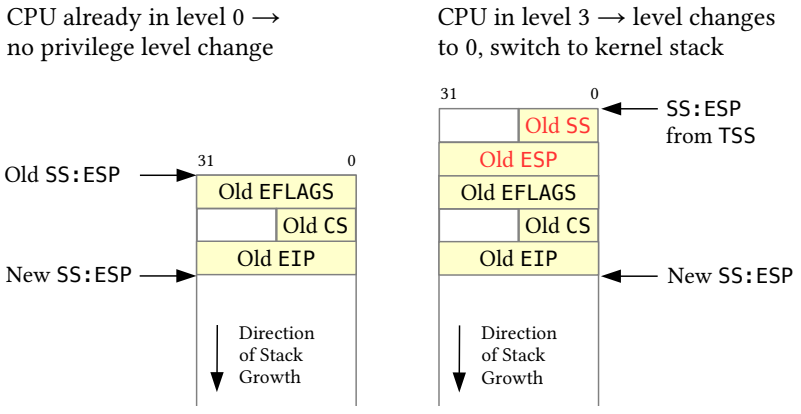


Figure 5.4: When entering the interrupt handler, the stack contains information for returning from the handler. Left: without privilege level change; right: with change from level 3 to 0, extra data marked red.

We cannot directly use a C function as an interrupt handler because once it would finish its work, it would do a regular `RET` which does not do what we want. (Of course we could use inline assembler code inside the C function to make it work anyway, but it makes more sense to directly implement parts of the handlers in assembler.)

5.2.4.1 The Context Data Structure

We want to be able to define handler functions in C which get called from the assembler code. Those functions will all have the following prototype:

```
void handler_function (context_t *r);
```

where `context_t`$_{142a}$ is a central data structure that can hold all the registers we use on the Intel machine. It will also be used in fault handlers, system call handlers and several other functions which need information about the current state. context

We define the `context_t`$_{142a}$ structure so that it matches the way in which we set up the stack in the assembler part of the handler:

[142a] ⟨*public type definitions* 142a⟩≡ (44a 48a) 175▷
```
typedef struct {
  unsigned int gs, fs, es, ds;
  unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
  unsigned int int_no, err_code;
  unsigned int eip, cs, eflags, useresp, ss;
} context_t;
```
Defines:
context_t, used in chunks 142b, 146, 151c, 174b, 175, 201d, 206d, 209c, 213d, 216b, 219c, 221–24, 234b, 255c, 258b, 260a, 276d, 282c, 289a, 299a, 310a, 319d, 328c, 331a, 332d, 342b, 370d, 372, 416, 426b, 433b, 493b, 513a, 519d, 532d, 546b, 565c, 566d, 583a, 587, 590b, and 610.

5.2.4.2 Assembler Part of the Handler

In order to have a handler function see useful values in the structure that `r` points to, we need to push the register contents in the reverse order onto the stack:

[142b] ⟨*push registers onto the stack* 142b⟩≡ (143b 144 150b 202c)
```
pusha
push ds
push es
push fs
push gs
push esp  ; pointer to the context_t
```

The first instruction `pusha` (push all general registers) pushes a lot of registers onto the stack: *EAX*, *ECX*, *EDX*, *EBX*, the old value of *ESP* (before the `pusha` execution began), *EBP*, *ESI*, and *EDI*—in that order. We add the segment registers *DS*, *ES*, *FS* and *GS*, and you can see that we've successfully handled the first two lines of the $\texttt{context\_t}_{142a}$ type definition. When the interrupt occurred, the registers *EFLAGS*, *CS* and *EIP* (and possibly also *SS* and the user mode's *ESP*) were also pushed on the stack which gives us the values in the fourth line of the $\texttt{context\_t}_{142a}$ definition.

What's missing are the values on the third line: We want to tell the handler *which interrupt* occurred so that we can use the same interrupt handler for several interrupts—for example, if we supported both IDE controllers (with interrupts 14 and 15) we could use that trick to run the same IDE handler when either of those interrupts occurred; thus, between the automatically happening push operations and the ones we perform in ⟨*push registers onto the stack* 142b⟩ we also push the interrupt number and another value `err_code` which can hold an error code. Interrupts don't have an error code, but we will recycle the same code later when we deal with faults, and some of those do provide an error code.

The final `push esp` statement in ⟨*push registers onto the stack* 142b⟩ is necessary because we cannot just place the structure contents on the stack: the handler function expects a pointer ($\texttt{context\_t}_{142a}$ `*r`), and `ESP` contains just that pointer: the start address of the structure. Figure 5.5 shows the layout of the stack after the assembler part has finished the preparations.

Later, when the handler's task is completed, we will need to pop the registers from the stack—in the reverse order:

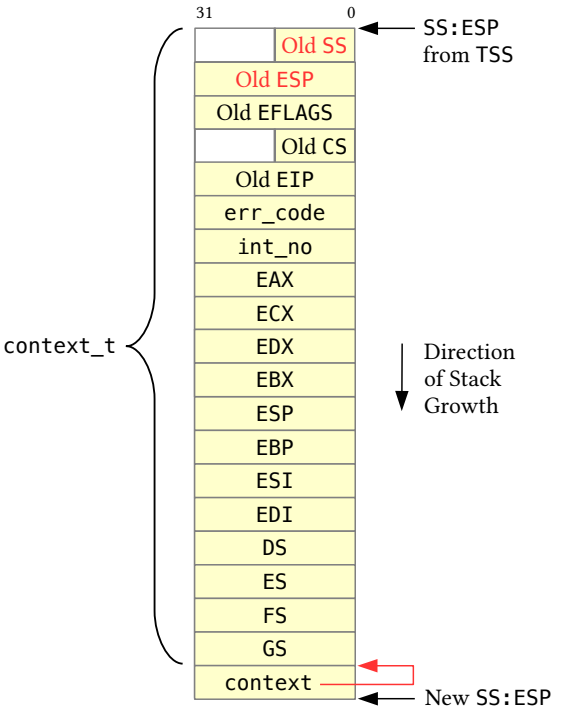


Figure 5.5: Stack after interrupt handler initialization by the assembler part.

⟨*pop registers from the stack* 143a⟩≡ (143b 144 150b 202c) [143a]

```
pop esp
pop gs
pop fs
pop es
pop ds
popa
```

Now here's an example of how we could implement the interrupt handler for IRQ 15:

⟨*irq15 example* 143b⟩≡ [143b]

```
push byte 0           ; error code
push byte 15          ; interrupt number
⟨push registers onto the stack 142b⟩
call irq_handler      ; call C function
⟨pop registers from the stack 143a⟩
add esp, 8            ; for errcode, irq no.
iret
```

Uses irq_handler 146a.

This contains all we need:

1. The two `push` commands add the error code and the interrupt number (which is 15 in this example).
2. With ⟨*push registers onto the stack* 142b⟩ we complete the `context_t`$_{142a}$ data structure and also push a pointer to it.
3. Now the stack is prepared properly to call the C function `irq_handler`$_{146a}$.
4. After returning, we first have to undo the push operations with ⟨*pop registers from the stack* 143a⟩.
5. Then we modify the stack address: we add 8, thus undoing the two `push` operations for the error code and the interrupt number.
6. Finally we return from the handler with `iret`.

We need almost the same code 16 times (for IRQs 0 to 15)—the only difference between the 16 versions is the interrupt number that we push in the second instruction. We simplify our code by having our individual handlers just push the two values (0 and the interrupt number) and then jump to an address which provides the common commands. The 0 value is a placeholder for an error code which cannot occur in interrupt handlers, but (as mentioned before) we will also implement fault handlers which shall use the same stack layout, and some of them will write a fault-specific error code into that location.

[144] ⟨*start.asm* 87⟩+≡ ◁110b 147a▷

```
global irq0, irq1, irq2,  irq3,  irq4,  irq5,  irq6,  irq7
global irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15

%macro irq_macro 1
       push byte 0          ; error code (none)
       push byte %1         ; interrupt number
       jmp irq_common_stub  ; rest is identical for all handlers
%endmacro

irq0:  irq_macro 32
irq1:  irq_macro 33
irq2:  irq_macro 34
irq3:  irq_macro 35
irq4:  irq_macro 36
irq5:  irq_macro 37
irq6:  irq_macro 38
irq7:  irq_macro 39
irq8:  irq_macro 40
irq9:  irq_macro 41
irq10: irq_macro 42
irq11: irq_macro 43
irq12: irq_macro 44
irq13: irq_macro 45
irq14: irq_macro 46
irq15: irq_macro 47
```

```
extern irq_handler          ; defined in the C source file

irq_common_stub:            ; this is the identical part
        ⟨push registers onto the stack 142b⟩
        call irq_handler    ; call C function
        ⟨pop registers from the stack 143a⟩
        add esp, 8
        iret
```

Defines:
irq0, used in chunk 139a.
irq1, used in chunk 139a.
irq10, used in chunk 139a.
irq11, used in chunk 139a.
irq12, used in chunk 139a.
irq13, used in chunk 139a.
irq14, used in chunk 139a.
irq15, used in chunk 139a.
irq2, used in chunk 139a.
irq3, used in chunk 139a.
irq4, used in chunk 139a.
irq5, used in chunk 139a.
irq6, used in chunk 139a.
irq7, used in chunk 139a.
irq8, used in chunk 139a.
irq9, used in chunks 138d and 139a.
Uses irq_handler 146a.

Our interrupt handling code is a slightly improved version of the code which Bran's Kernel Tutorial [Fri05] uses; the original code contains some extra instructions that we don't need for the ULIX kernel.

5.2.4.3 C Part of the Handler

Finally, we show what happens when the assembler code calls the external handler function $\texttt{irq\_handler}_{146a}\texttt{()}$ that we implement in the C file.

The first thing our handler needs to do is acknowledge the interrupt. For that purpose it sends the command

⟨*constants* 112a⟩+≡ (44a) ◁134 158a▷ [145a]

```
#define END_OF_INTERRUPT  0x20
```

Defines:
END_OF_INTERRUPT, used in chunk 146a.

to all PICs which are involved: In case of an interrupt number between 0 and 7 that is only the primary PIC; in case the number is 8 or higher, both controllers need to be informed. Omitting this step would stop the controller from raising further interrupts which would basically disable interrupts completely.

Next we check whether a specific handler for the current interrupt has been installed in the

⟨*global variables* 92b⟩+≡ (44a) ◁139a 148a▷ [145b]

```
void *interrupt_handlers[16] = { 0 };
```

Defines:
interrupt_handlers, used in chunk 146.

array of interrupt handlers.

[146a] ⟨*function implementations* 100b⟩+≡ (44a) ◁140b 146c▷
```
void irq_handler (context_t *r) {
  int number = r->int_no - 32;                    // interrupt number
  void (*handler)(context_t *r);                  // type of handler functions

  if (number ≥ 8) {
    outportb (IO_PIC_SLAVE_CMD, END_OF_INTERRUPT); // notify slave  PIC
  }
  outportb (IO_PIC_MASTER_CMD, END_OF_INTERRUPT);  // notify master PIC (always)

  handler = interrupt_handlers[number];
  if (handler != NULL) {
    handler (r);
  }
}
```
Defines:
irq_handler, used in chunks 143b and 144.
Uses context_t 142a, END_OF_INTERRUPT 145a, interrupt_handlers 145b, IO_PIC_MASTER_CMD 134, IO_PIC_SLAVE_CMD 134, NULL 46a, and outportb 133b.

As a last step we provide a function

[146b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁139f 146e▷
```
void install_interrupt_handler (int irq, void (*handler)(context_t *r));
```
Uses context_t 142a and install_interrupt_handler 146c.

which lets us enter (pointers to) handler functions in this array; it is pretty simple:

[146c] ⟨*function implementations* 100b⟩+≡ (44a) ◁146a 151c▷
```
void install_interrupt_handler (int irq, void (*handler)(context_t *r)) {
  if (irq ≥ 0 && irq < 16)
    interrupt_handlers[irq] = handler;
}
```
Defines:
install_interrupt_handler, used in chunks 146b, 323b, 339a, 520a, 534b, and 552c.
Uses context_t 142a and interrupt_handlers 145b.

Early in the ⟨*initialize system* 45b⟩ step of the kernel's $\texttt{main}_{44b}\texttt{()}$ function we need to load
IDTR the Interrupt Descriptor Table Register (*IDTR*) so that the CPU can find the table:

[146d] ⟨*install the interrupt descriptor table* 146d⟩≡ (45b)
```
idtp.limit = (sizeof (struct idt_entry) * 256) - 1;   // must do -1
idtp.base  = (int) &idt;
idt_load ();
```
Uses idt 138a, idt_entry 137a, idt_load 147a, and idtp 138a.

It uses the assembler function

[146e] ⟨*function prototypes* 45a⟩+≡ (44a) ◁146b 147b▷
```
extern void idt_load ();
```

which is related to gdt_flush$_{110b}$, just writing the address of idtp$_{138a}$ to the *IDTR* register via the lidt instruction instead of writing the address of gp$_{92b}$ to *GDTR* via lgdt: lidt

⟨ *start.asm* 87 ⟩+≡ ◁144 148c▷ [147a]

```
extern idtp               ; defined in the C file
global idt_load
idt_load:    lidt [idtp]
             ret
```

Defines:
idt_load, used in chunk 146.
Uses idtp 138a.

In the following chapters we will often use this function in commands similar to

```
install_interrupt_handler (IRQ_SOMEDEV, somedev_handler);
```

For comparison, once more gdt_flush$_{110b}$ and idt_load$_{147a}$:

```
extern gp   ; defined in the C file
global gdt_flush

gdt_flush: lgdt [gp]
           mov ax, 0x10
           mov ds, ax
           mov es, ax
           mov fs, ax
           mov gs, ax
           mov ss, ax
           jmp 0x08:flush2
flush2:    ret
```

```
extern idtp   ; defined in the C file
global idt_load

idt_load: lidt [idtp]
          ret
```

(The gdt_flush$_{110b}$ function does more than idt_load$_{147a}$ since it also updates all segment registers.)

5.3 Faults

As we've mentioned in the introduction to this chapter, handling a fault is very similar to handling an interrupt. Since you've just seen the interrupt code, you will recognize many concepts at once while we present the fault handling code.

Like we defined the interrupt handlers irq0$_{144}$() to irq15$_{144}$() in the assembler file start.asm, we do the same with 32 fault handler functions fault0$_{149b}$() to fault31$_{149b}$().

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁146e 151b▷ [147b]

```
extern void
  fault0(),  fault1(),  fault2(),  fault3(),  fault4(),  fault5(),  fault6(),
  fault7(),  fault8(),  fault9(),  fault10(), fault11(), fault12(), fault13(),
  fault14(), fault15(), fault16(), fault17(), fault18(), fault19(), fault20(),
  fault21(), fault22(), fault23(), fault24(), fault25(), fault26(), fault27(),
  fault28(), fault29(), fault30(), fault31();
```

Uses fault0 149b, fault1 149b, fault10 149b, fault11 149b, fault12 149b, fault13 149b, fault14 149b, fault15 149b, fault16 149b, fault17 149b, fault18 149b, fault19 149b, fault2 149b, fault20 149b, fault21 149b, fault22 149b, fault23 149b, fault24 149b, fault25 149b, fault26 149b, fault27 149b, fault28 149b, fault29 149b, fault3 149b, fault30 149b, fault31 149b, fault4 149b, fault5 149b, fault6 149b, fault7 149b, fault8 149b, and fault9 149b.

and we enter these in the IDT just like we did with the `irq*()` functions.

[148a] ⟨*global variables* 92b⟩+≡ (44a) ◁145b 151a▷

```
void (*faults[32])() = {
  fault0,  fault1,  fault2,  fault3,  fault4,  fault5,  fault6,  fault7,
  fault8,  fault9,  fault10, fault11, fault12, fault13, fault14, fault15,
  fault16, fault17, fault18, fault19, fault20, fault21, fault22, fault23,
  fault24, fault25, fault26, fault27, fault28, fault29, fault30, fault31
};
```

Defines:
faults, used in chunks 148b and 607c.
Uses fault0 149b, fault1 149b, fault10 149b, fault11 149b, fault12 149b, fault13 149b, fault14 149b, fault15 149b, fault16 149b, fault17 149b, fault18 149b, fault19 149b, fault2 149b, fault20 149b, fault21 149b, fault22 149b, fault23 149b, fault24 149b, fault25 149b, fault26 149b, fault27 149b, fault28 149b, fault29 149b, fault3 149b, fault30 149b, fault31 149b, fault4 149b, fault5 149b, fault6 149b, fault7 149b, fault8 149b, and fault9 149b.

We install those handlers in the same way that we registered the interrupt handlers earlier (see page 139):

[148b] ⟨*install the fault handlers* 148b⟩≡ (45b) 202e▷

```
for (int i = 0; i < 32; i++) {
  fill_idt_entry (i,
                  (unsigned int)faults[i],
                  0x08,
                  0b1110,   // flags: 1 (present), 11 (DPL 3), 0
                  0b1110);  // type: 1110 (32 bit interrupt gate)
}
```

Uses faults 148a and fill_idt_entry 138c.

In the assembler file we use the same trick for the `fault*()` functions that you've just seen for `irq*()`:

[148c] ⟨*start.asm* 87⟩+≡ ◁147a 149a▷

```
global fault0,  fault1,  fault2,  fault3,  fault4,  fault5,  fault6,  fault7
global fault8,  fault9,  fault10, fault11, fault12, fault13, fault14, fault15
global fault16, fault17, fault18, fault19, fault20, fault21, fault22, fault23
global fault24, fault25, fault26, fault27, fault28, fault29, fault30, fault31
```

The handlers all look similar: We push one or two bytes on the stack and then jump to `fault_common_stub`$_{150b}$. The choice of one or two arguments depends on the kind of interrupt that occurred: for some faults the CPU pushes a one-byte error code on the stack, and for some others it does not. In order to have the same stack setup (regardless of the fault) we push an extra null byte in those cases where no error code is pushed.

error code

The code always looks like one of the following two cases:

```
fault5:  push byte 0  ; error code          fault8:  ; no error code
         push byte 5                                 push byte 8
         jmp fault_common_stub                       jmp fault_common_stub
```

Since we do not want to type this repeatedly, we use `nasm`'s macro feature which lets us write simple macros for both cases. `fault_macro_0`$_{149a}$ handles the cases where we need to push an extra null byte (as in `fault5`$_{149b}$ above), and `fault_macro_no0`$_{149a}$ handles the other cases (as in `fault8`$_{149b}$ above):

⟨ *start.asm* 87 ⟩+≡ ◁ 148c 149b ▷ [149a]

```
%macro fault_macro_0 1
        push byte 0  ; error code
        push byte %1
        jmp fault_common_stub
%endmacro
```

```
%macro fault_macro_no0 1
        ;  don't push error code
        push byte %1
        jmp fault_common_stub
%endmacro
```

Defines:
`fault_macro_0`, used in chunk 149b.
`fault_macro_no0`, used in chunk 149b.
Uses `fault_common_stub` 150b.

With these macros the rest is straight-forward:

⟨ *start.asm* 87 ⟩+≡ ◁ 149a 150a ▷ [149b]

```
fault0:  fault_macro_0     0   ;  Divide by Zero
fault1:  fault_macro_0     1   ;  Debug
fault2:  fault_macro_0     2   ;  Non Maskable Interrupt
fault3:  fault_macro_0     3   ;  INT 3
fault4:  fault_macro_0     4   ;  INTO
fault5:  fault_macro_0     5   ;  Out of Bounds
fault6:  fault_macro_0     6   ;  Invalid Opcode
fault7:  fault_macro_0     7   ;  Coprocessor not available
fault8:  fault_macro_no0   8   ;  Double Fault
fault9:  fault_macro_0     9   ;  Coprocessor Segment Overrun
fault10: fault_macro_no0  10   ;  Bad TSS
fault11: fault_macro_no0  11   ;  Segment Not Present
fault12: fault_macro_no0  12   ;  Stack Fault
fault13: fault_macro_no0  13   ;  General Protection Fault
fault14: fault_macro_no0  14   ;  Page Fault
fault15: fault_macro_0    15   ;  (reserved)
fault16: fault_macro_0    16   ;  Floating Point
fault17: fault_macro_0    17   ;  Alignment Check
fault18: fault_macro_0    18   ;  Machine Check
fault19: fault_macro_0    19   ;  (reserved)
fault20: fault_macro_0    20   ;  (reserved)
fault21: fault_macro_0    21   ;  (reserved)
fault22: fault_macro_0    22   ;  (reserved)
fault23: fault_macro_0    23   ;  (reserved)
fault24: fault_macro_0    24   ;  (reserved)
fault25: fault_macro_0    25   ;  (reserved)
fault26: fault_macro_0    26   ;  (reserved)
```

```
fault27: fault_macro_0    27   ;  (reserved)
fault28: fault_macro_0    28   ;  (reserved)
fault29: fault_macro_0    29   ;  (reserved)
fault30: fault_macro_0    30   ;  (reserved)
fault31: fault_macro_0    31   ;  (reserved)
```

Defines:
fault0, used in chunks 147b and 148a.
fault1, used in chunks 147b and 148a.
fault10, used in chunks 147b and 148a.
fault11, used in chunks 147b and 148a.
fault12, used in chunks 147b and 148a.
fault13, used in chunks 147b and 148a.
fault14, used in chunks 147b and 148a.
fault15, used in chunks 147b and 148a.
fault16, used in chunks 147b and 148a.
fault17, used in chunks 147b and 148a.
fault18, used in chunks 147b and 148a.
fault19, used in chunks 147b and 148a.
fault2, used in chunks 147b and 148a.
fault20, used in chunks 147b and 148a.
fault21, used in chunks 147b and 148a.
fault22, used in chunks 147b and 148a.
fault23, used in chunks 147b and 148a.
fault24, used in chunks 147b and 148a.
fault25, used in chunks 147b and 148a.
fault26, used in chunks 147b and 148a.
fault27, used in chunks 147b and 148a.
fault28, used in chunks 147b and 148a.
fault29, used in chunks 147b and 148a.
fault3, used in chunks 147b and 148a.
fault30, used in chunks 147b and 148a.
fault31, used in chunks 147b and 148a.
fault4, used in chunks 147b and 148a.
fault5, used in chunks 147b and 148a.
fault6, used in chunks 147b and 148a.
fault7, used in chunks 147b and 148a.
fault8, used in chunks 147b and 148a.
fault9, used in chunks 147 and 148.
Uses fault_macro_0 149a and fault_macro_no0 149a.

`fault_common_stub`$_{150b}$ is—almost—a rewrite of `irq_common_stub`$_{144}$, the only difference is that we call a different C function `fault_handler`$_{151c}$`()` in the middle.

[150a] ⟨*start.asm* 87⟩+≡ ◁149b 150b▷

```
extern fault_handler
```

Uses fault_handler 151c.

The stub saves the processor state, calls the handler function and restores the stack frame:

[150b] ⟨*start.asm* 87⟩+≡ ◁150a 197c▷

```
fault_common_stub:
    ⟨push registers onto the stack 142b⟩
    call fault_handler    ; call C function
    ⟨pop registers from the stack 143a⟩
    add esp, 8            ; for errcode, irq no.
    iret
```

Defines:
fault_common_stub, used in chunk 149a.
Uses fault_handler 151c.

Initially our fault handlers will just output a message stating the cause of the fault and then halt the system; later we will provide fault handlers for some types of faults which try to solve the problem and let the operation go on. Here are the error messages:

⟨*global variables* 92b⟩+≡ (44a) ◁148a 162b▷ [151a]

```
char *exception_messages[] = {
    "Division By Zero",        "Debug",                          //  0,  1
    "Non Maskable Interrupt",  "Breakpoint",                     //  2,  3
    "Into Detected Overflow",  "Out of Bounds",                  //  4,  5
    "Invalid Opcode",          "No Coprocessor",                 //  6,  7
    "Double Fault",            "Coprocessor Segment Overrun",    //  8,  9
    "Bad TSS",                 "Segment Not Present",            // 10, 11
    "Stack Fault",             "General Protection Fault",       // 12, 13
    "Page Fault",              "Unknown Interrupt",              // 14, 15
    "Coprocessor Fault",       "Alignment Check",                // 16, 17
    "Machine Check",                                             // 18
    "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
    "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
    "Reserved", "Reserved", "Reserved"                           // 19..31
};
```

Defines:
`exception_messages`, used in chunk 152a.

We get the correct message by accessing the proper entry of the array, e. g., for a page fault (with fault number 14) it is stored in `exception_messages`$_{151a}$`[14]`.

Our C fault handler Fault Handler

⟨*function prototypes* 45a⟩+≡ (44a) ◁147b 162c▷ [151b]

```
void fault_handler (context_t *r);
```

displays some information about the problem and checks whether the fault occurred while a process was running (by testing whether `r->eip < 0xc0000000`). If not, the system switches to the kernel mode shell (and is broken).

The page fault handler is a special case: we expect to deal with page faults silently (see Chapter 9), so we check for this case before doing anything else.

⟨*function implementations* 100b⟩+≡ (44a) ◁146c 162d▷ [151c]

```
void fault_handler (context_t *r) {
  if (r->int_no == 14) {                          // fault 14 is a page fault
    page_fault_handler (r); return;
  }

  memaddress fault_address = (memaddress)(r->eip);

  if (r->int_no < 32) {
    ⟨fault handler: display status information 152a⟩

    if ( fault_address < 0xc0000000 ) {           // user mode
      ⟨fault handler: terminate process 152b⟩
    }
```

```
      ⟨disable scheduler 276b⟩                          // error inside the kernel
      ⟨enable interrupts 47b⟩
      printf ("\n");
      asm ("jmp kernel_shell");
    }
  }
```

Defines:
fault_handler, used in chunks 150 and 151b.
Uses context_t 142a, kernel_shell 610a, memaddress 46c, page_fault_handler 289a, and printf 601a.

For displaying the status information we look at the register contents which are provided by r. Especially interesting are the task number, the address space number, the address of the faulting instruction, the *EFLAGS* register and the error code which the CPU has provided upon entry into the fault handler.

[152a] ⟨*fault handler: display status information* 152a⟩≡ (151c 290a)

```
printf ("'%s' Exception at 0x%08x (task=%d, as=%d).\n",
  exception_messages[r->int_no], r->eip, current_task, current_as);
printf ("eflags: 0x%08x  errcode: 0x%08x\n", r->eflags, r->err_code);
printf ("eax: %08x  ebx: %08x  ecx: %08x  edx: %08x \n",
  r->eax, r->ebx, r->ecx, r->edx);
printf ("eip: %08x  esp: %08x  int: %8d  err: %8d \n",
  r->eip, r->esp, r->int_no, r->err_code);
printf ("ebp: %08x  cs: 0x%02x ds: 0x%02x es: 0x%02x fs: 0x%02x ss: 0x%02x \n",
  r->ebp, r->cs, r->ds, r->es, r->fs, r->ss);
printf ("User mode stack: 0x%08x-0x%08x\n", TOP_OF_USER_MODE_STACK
  - address_spaces[current_as].stacksize, TOP_OF_USER_MODE_STACK);
```

Uses address_spaces 162b, current_as 170b, current_task 192c, exception_messages 151a, printf 601a, and TOP_OF_USER_MODE_STACK 159b.

If a process was running, the fault handler terminates it:

[152b] ⟨*fault handler: terminate process* 152b⟩≡ (151c 290a 291)

```
thread_table[current_task].state = TSTATE_ZOMBIE;
remove_from_ready_queue (current_task);
r->ebx = -1;  // exit code for this process
syscall_exit (r);
```

Uses current_task 192c, remove_from_ready_queue 184c, syscall_exit 216b, thread_table 176b, and TSTATE_ZOMBIE 180a.

Since we have not talked about processes yet, you need not worry about the reference to the thread table via thread_table[current_task] or remove_from_ready_queue$_{184c}$(). We will explain this function and the thread table data structure later, and we will also show what the syscall_exit$_{216b}$() function does. You can choose to ignore the complete ⟨*fault handler: terminate process* 152b⟩ block in the code for now.

A page fault need not be a problem: it often occurs because the code attempted to access an invalid address (which is bad), but yet more often the address will be valid, but the page won't be in the physical RAM. That situation can be helped. In Chapter 9 we will implement the *page fault handler*. It requires a working hard disk since we will *page out* pages to the disk and later *page them in* again.

page fault handler

5.4 Exercises

16. **Keyboard driver: Polling**

Tutorial 4

In the folder `tutorial/04/` you find a version of the ULIX kernel which contains the new interrupt and fault handling code. It is a literate program (`ulix.nw`). You will now develop a simple keyboard driver, extending the provided code, and you should try to retain the literate programming style, i. e., integrate code and documentation in the file.

`cd` into the folder and open the file `ulix.nw`. At the end you will find a section "keyboard driver" where you can place your new code; at least most of it. The rest of the file corresponds to the literate program from the last exercise, but some new mechanisms have been added.

As a first step you can test querying the keyboard controller via polling:

a) The keyboard controller can be accessed via two ports (`0x60` and `0x64`) which you can read from via $\texttt{inportb}_{133b}\texttt{()}$. Append the port numbers to the ⟨*constants* 112a⟩ code chunk:

```
#define KBD_DATA_PORT   0x60
#define KBD_STATUS_PORT 0x64
```

The data port delivers information about pressed and released keys, and the status port lets you check whether a key was pressed (or released) at all.

b) Try to continuously read the data port in a loop and print the results (as numbers). You can query with the following code:

```
byte scancode;
scancode = inportb (KBD_DATA_PORT);
```

(We have prepared an empty code chunk ⟨*kernel main: user-defined tests* ⟩ at the end of the Noweb file which will be called after initialization.) Print the scan codes with $\texttt{printf}_{601a}\texttt{()}$. This will quickly fill the screen (even if you add no newlines to your $\texttt{printf}_{601a}$ call), so you should clear the screen when you reach the bottom line:

```
if (posy == 25) clrscr ();
```

You will notice that this approach writes a continuous (and quick) stream of data onto the screen. While the system is running, press a few keys; that will modify the output. (You may have to keep the keys pressed to recognize the changes.) The values are *keyboard scan codes*, each of them represents an action of pressing or releasing a key. You will see the same value again and again until a new press/release event occurs.

keyboard scan code

c) Improve the code by also checking the status register (via the status port). That works in the same way that you've accessed the data port, but uses the port number `KBD_STATUS_PORT`. If the lowest bit of the return value is set (which you

can check with `if ((status & 1) == 1)`), then there is a fresh scan code, and only then should you query the data port. The modified code will only generate an output when you press or release a key.

d) The scan codes for pressing and releasing a key only differ in the eighth bit (it it set for release events), so for example scan codes for the "A" key are 30 and 158 (= 30 + 128), for its left neighbor key "S" they are 31 and 159 (= 31 + 128). Create a mapping table which stores the upper case letters which correspond to a few scan codes. You need not look up an ASCII table but can simply enter characters, such as `'A'` or `'B'`, in the table. Initialize the table with zero values:

```
char scancode_table[128] = { 0 };
```

Then you can start with adding entries for the scan codes you already know, e. g. 30 and 31 for the "A" and "S" keys. (We ignore release codes in this table.)

```
scancode_table[30] = 'A';
scancode_table[31] = 'S';
```

Identify the return key's scan code. The corresponding character is `'\n'`.

Modify your existing code so that it does not only print the scan code but also the character (if it is known, i. e., the corresponding table entry is not 0). Test your program. (The $\texttt{printf}_{601a}$ format code for characters is `%c`. When you print release scan codes you get negative numbers—you can get rid of them by casting the scan code to the `int` type.)

This leaves you with a simple, polling keyboard driver.

17. **Keyboard driver: Interrupts**

In this exercise you switch to an interrupt-based keyboard driver.

a) Add the following lines to an appropriate place in the system initialization, e. g. in the ⟨ *kernel main: initialize system* ⟩ chunk:

```
install_interrupt_handler (IRQ_KBD, keyboard_handler);
enable_interrupt (IRQ_KBD);
asm ("sti");                    // enable interrupts
```

$\texttt{IRQ\_KBD}_{132}$ is already `#defined` as 1: It is the interrupt number used by the keyboard controller. Now you have to implement the keyboard handler. It has the signature

```
void keyboard_handler (struct regs *r);
```

(`struct regs` is the same as $\texttt{context\_t}_{142a}$ in the rest of the book.) It will automatically be called whenever you press or release a key.

b) Make sure that your handler gets called by letting it print a single character (e. g. `'*'`) and leaving the handler with `return;`.

c) In the next step you evaluate and print the values that you can read from the keyboard controller. It is not necessary to query the controller's data port (as you did in the above exercise), because the handler is only called when a new event has occurred. After the output (like above, using the scan code table) you can leave the handler with `return;`. Note that again, due to the simplified `printf`$_{601a}$ implementation, you will need to insert `clrscr`$_{331d}$`()` calls when you reach the screen's bottom.

The advantage of using interrupts is that the main program (in `main`$_{44b}$) need not concern itself with the keyboard.

d) Next, implement a function

```
void kreadline (char *s, int len);
```

that you call from `main`$_{44b}$`()`, e. g. with

```
char input[41];   // 40 characters plus \0 terminator
kreadline ((char*)input, 40);
```

The goal is that `kreadline()` fills the provided string (a `char` pointer) with the characters you enter (as far as they are known in the scan code table) until you either complete the input by pressing [Enter] or until the maximum number of characters (`len`) is reached. Only then shall the function return. The main program can then print the read string and start over, using an infinite loop.

The important aspect of this exercise is that the `kreadline` needs to cooperate with the interrupt handler. You will need two new global variables for an input buffer and for the next character position in the buffer:

```
char buffer[256];      // buffer for input
short int pos = 0;     // current position in the buffer
```

The interrupt handler should work as follows:

- If the scan code is larger than 127 (release event), the handler returns immediately (it simply ignores release events).
- When an unknown scan code shows up, the handler also returns at once.
- Otherwise it will print the character and also write it into the buffer.
- Then it increases `pos` and returns.

`kreadline()` performs an infinite loop and checks whether `(pos>0 && buffer[pos-1] =='\n')` is true—if so, then the function copies the entered string (from position 0 to `pos-2`) to `s`, sets `pos=0` and returns. Note that the string must be `'\0'`-terminated so that `printf`$_{601a}$`()` can later use it. You can simply replace the `'\n'` chacracter with `'\0'` if the input is terminated by pressing [Enter]; if you reach the maximum allowed number of characters, you write `'\0'` into the last byte of the string.

For copying the string you can use `strncpy`$_{594b}$`()`. That function works like the corresponding Linux function (see `man strncpy`), i. e., it expects target, source and

maximum length of the target string as arguments.

Your scan code table will need an entry for the [Enter] key in order to make this work.

18. **Backspace Support**

    Modify your code for the keyboard handler and the `kreadline` function so that it treats the backspace key appropriately: With that key you shall be able to delete the last character that was entered. It shall both be removed from the screen and from the string which `kreadline` returns.

19. **Faults**

    The current version of the mini kernel contains fault handlers. Verify that they work correctly by generating some typical faults:

    - Division by zero: Have your main program perform a division by zero, for example via `int z = 1 / 0;`—you can ignore any compiler warnings that this code will cause.
    - Try to access non-available memory, e. g., with

      ```
      char *address = (char*)0xE0000000; char tmp = *address;
      ```

    - Set the segment register *DS* to an invalid segment number, e. g.:

      ```
      asm ("mov $32, %ax;  mov %ax, %ds");
      ```

    - You can explicitly cause each fault (for fault numbers between 0 and 31) by using the assembler instruction `int $number`. For example, in order to generate an "Out of Bounds Fault" (number 5), you can use the line `asm ("int $5");`.

    The reward should be a Division by Zero Fault, a Page Fault and a General Protection Fault (and an Out of Bounds Fault in the last step).

6

Implementation of Processes

We have now written most of the code that we need to introduce the most important concept: the process. In this chapter we take a first look at the data structures and kernel functions which will let us create and schedule processes.

- In Chapter 6.1 we present the desired memory layout of a ULIX process and describe our implementation of address spaces.
- Chapter 6.2 introduces the central data structure for processes and threads, the process control block (which we will refer to as a thread control block, TCB), as well as queues for handling ready and blocked threads.
- Chapter 6.3 shows what we need to do in order to start the very first process; all further processes will be created via the $\texttt{fork}_{213g}$ mechanism.
- Since forking will require the existence of a system call interface, it is time to introduce it: we present our implementation in Chapter 6.4.
- With system calls available we can explain the implementation of the fork mechanism (and the `fork` system call) in Chapter 6.5.
- While it is an important step to bring new processes into existence, we also need to handle their termination. In Chapter 6.6 we show how to `exit` from a process.

The remaining sections of this chapter are less interesting but still required: We provide a method for requesting process information in Chapter 6.7 (which will let us write a `ps` program), describe the ELF binary format and an ELF program loader in Chapter 6.8 (so that a process can start a different application via `exec`) and finally discuss an idle process that will be activated when there is no other process that could do something useful (Chapter 8.3.3).

Everything you will see in this chapter deals with single-threaded processes. In Chapter 7 we will extend our execution model so that it also supports multiple threads inside one process. You might want to remember this whenever you wonder why we keep talking about *thread* control blocks (instead of *process* control blocks) throughout this chapter.

There's also a need to discuss how the system can switch between concurrent processes and threads: once we have more than one active task, a scheduler must take care of this. We delay this until Chapter 8.

6.1 Address Spaces for Processes

address space descriptor

We will store information about memory usage in a data structure that we call *address space descriptor*. The idea is that every process uses its own address space while several threads (of the same process) share a common address space.

Address space descriptors are stored in one large *address space table*. This table must be finite, i. e., there must exist a maximum number of address spaces for the system. This must correspond to the maximum number of threads `MAX_THREADS`$_{176a}$ that we'll soon define: While threads may share an address space, it is impossible for one thread to use more than one address space. Thus `MAX_ADDR_SPACES`$_{158a}$ has to be $\leq$ `MAX_THREADS`$_{176a}$—we give both constants the same value:

[158a] ⟨*constants* 112a⟩+≡ (44a) ◁145a 159a▷

```
#define MAX_ADDR_SPACES 1024
```

Defines:
`MAX_ADDR_SPACES`, used in chunks 162, 171a, 307a, and 308c.

As we will see later, every thread may have its own virtual address space and needs to own a reference to an address space descriptor. Even the kernel will have to do that. Since there can be so many address spaces, we need a shorthand to identify virtual address spaces. We introduce the type `addr_space_id`$_{158b}$ to do this. It is declared as `unsigned int`. Basically, an `addr_space_id`$_{158b}$ can be thought of as an index into the address space table. So rather than storing a complete address space descriptor per thread, we will rather store an address space identifier.

[158b] ⟨*public elementary type definitions* 45e⟩+≡ (44a 48a) ◁46c 178a▷

```
typedef unsigned int addr_space_id;
```

Defines:
`addr_space_id`, used in chunks 122c, 158c, 162d, 163c, 166c, 168–72, 188, 190a, 210a, and 308c.

We already note that the thread control block (which will be the central data structure for processes and threads) needs an `addr_space_id`$_{158b}$ element. That data structure is called `TCB`$_{175}$, and we will define it later in this chapter, but you will often see the code chunk ⟨*more TCB entries* 158c⟩ that lets us append entries to this structure whenever the need occurs:

[158c] ⟨*more TCB entries* 158c⟩≡ (175) 181▷

```
addr_space_id addr_space;
```

Uses `addr_space_id` 158b.

6.1.1 Memory Layout of a Process

Every process needs three (virtual) memory areas.

- **Code and Data**: We will later compile user mode binaries which expect to be loaded to the virtual addresses `0x0` and above. This area is used for the code (the machine instructions in the binary) as well as variables defined statically in the program. The *heap* will exist just behind this memory area, processes can later dynamically expand this memory area using the `sbrk`$_{174d}$ function. Heap

⟨ *constants* 112a ⟩+≡ (44a) ◁158a 159b▷ [159a]
```
#define BINARY_LOAD_ADDRESS 0x0
```
Defines:
BINARY_LOAD_ADDRESS, used in chunks 163c, 190c, and 192d.

- **User Mode Stack**: Every process needs its own stack: That is where the CPU will store return addresses and arguments whenever the process makes a function call. We'll use the virtual addresses below `0xb0000000` which will leave a lot of space between the code and data and the stack: We want the stack to grow automatically, so we'll start with just one single page of memory for the stack and increase it as needed: When you think of recursive functions where the end of the recursion depends on some calculation inside the program, it is clear that we cannot have a maximum size for the stack. Expanding the stack is a task for the page fault handler which we've already mentioned. You will see its implementation on page 287 ff.

⟨ *constants* 112a ⟩+≡ (44a) ◁159a 159c▷ [159b]
```
#define TOP_OF_USER_MODE_STACK 0xb0000000
```
Defines:
TOP_OF_USER_MODE_STACK, used in chunks 152a, 165a, 167b, 192d, 231, 289c, and 291.

- **Kernel Stack**: For several reasons we need a second stack when a process switches to kernel mode (using a system call, see Section 6.4). While it would be possible to share one kernel stack between all the processes, that would also limit us: With a single kernel stack we would run into problems when two or more processes need to enter kernel mode at the same time.

  There's also a security aspect: The kernel stack may contain kernel data that the process should not have access to.

  We'll put the kernel stack just under the kernel space of memory, at addresses below `0xc0000000`.

⟨ *constants* 112a ⟩+≡ (44a) ◁159b 162a▷ [159c]
```
#define TOP_OF_KERNEL_MODE_STACK 0xc0000000
```
Defines:
TOP_OF_KERNEL_MODE_STACK, used in chunks 192b, 196a, 211b, 257b, 261, and 280a.

Thus, the memory layout of a process is as shown in Figure 6.1. The double line below `0xc0000000` marks the barrier between process-specific and generic memory ranges: everything above `0xc0000000` is globally visible and identical in every address space, whereas the

lower addresses differ for each process, and they do not exist at all before the first process has been created.

Address ranges marked 'K' in the last column need kernel privileges to be accessed. Heap areas will only become available after they are manually requested. The '(U)' annotation of the combined stack/heap area refers to the fact that it is not allocated when a process starts but rather can grow both from the top and from the bottom, depending on the process' actions.

Address Range	Usage	Access
`0xD4000000` – `0xFFFFFFFF`	*unused*	–
`0xD0000000` – `0xD3FFFFFF`	64 MByte Physical RAM (mapped)	K
`0xC0000000` – `0xCFFFFFFF`	Kernel Code and Data	K
`0xBFFFF000` – `0xBFFFFFFF`	Kernel Stack (4 KByte = one page)	K
`0xB0000000` – `0xBFFFEFFF`	*unused*	–
… – `0xAFFFFFFF`	User Mode Stack	U
	User Mode Stack (grows automatically) Heap (can be grown with $\texttt{sbrk}_{174d}$)	(U)
`0x00000000` – …	Process Code and Data	U

Figure 6.1: This is the memory layout of a process.

We will later provide a modified version of this memory model, when we introduce several threads (of the same process), but for now this description is sufficient to understand the process implementation.

6.1.2 Creating a New Address Space

Essentially, an address space is just a fresh page directory. Its kernel memory part (addresses `0xc0000000` and higher) will be identical to the kernel's page directory which we've already set up earlier.

It is helpful to reconsider how the CPU (or the MMU) accesses the paging information: A register holds the address of the page directory which has 1024 entries, each of which is either null or points to a page table.

The upper $1024 - 768 = 256$ entries are responsible for the kernel memory (`0xC0000000` – `0xFFFFFFFF`), and the lower 768 entries are available for process memory (`0x00000000` – `0xBFFFFFFF`) with the upper part of each process' private memory (`0xBFFFF000` – `0xBFFFFFFF`) being the kernel stack which is only available in kernel mode.

We want to allow for three different situations, as far as access to process memory and kernel memory is concerned:

pure kernel mode: The kernel is actively dealing with specific kernel tasks, such as memory management or interrupt service. The kernel's view on memory in this state is as it was after we enabled paging: It sees its own memory (`0xC0000000` – `0xCFFFFFFF`) and the physical memory which is mapped to addresses starting at `0xD0000000`, but no process memory. We will not use this mode once the first process was created.

process in user mode: A process is active and running in user mode. It only sees its own memory (`0x00000000` – `0xAFFF.FFFF`: code, data, heap and user mode stack). The paging information will map the kernel stack and the kernel's memory as well, but since it will be marked non-user-mode-accessible, that will be the same as not having it at all when running in user mode. In this mode any attempt to access kernel memory (either its data or its code) will generate a page fault—even if the address is valid.

process in kernel mode: A process has entered kernel mode via a system call or an interrupt has occurred. In this situation the page tables must give access to both the current process' memory and the kernel memory. All 4 GByte of virtual memory are visible. The paging information can be the same as for the process in user mode: the current level of execution (kernel mode instead of user mode) grants the access to all of the virtual memory. (For handling an interrupt it is not necessary to see the current process' user mode memory, so we could switch to *pure kernel mode* in order to prevent interrupt handlers from looking at process memory. But since we intend to trust our interrupt handlers, we will not do that.)

We reserve memory for an address space list. This list does not hold the page directory or the page tables it points to, but just the address of the page directory and some status information:

⟨ *type definitions* 91 ⟩+≡ (44a) ◁137b 178b▷ [161]

```
typedef struct {
  void         *pd;               // pointer to the page directory
  int          pid;               // process ID (if used by a process; -1 if not)
  short        status;            // are we using this address space?
  memaddress   memstart, memend;  // first/last address below 0xc000.0000
  unsigned int stacksize;         // size of user mode stack
  memaddress   kstack_pt;         // stack page table (for kernel stack)
  unsigned int refcount;          // how many threads use this address space?
  ⟨ more address_space entries 257a ⟩
} address_space;
```

Defines:
address_space, used in chunks 162b, 173a, and 304a.

`pd` holds the (virtual) address of the page directory. `memstart` and `memend` contain the first and last user mode address (for code, data and heap), and `stacksize` tells the size of the user mode stack. We also want to keep the address of the kernel mode stack's page table handy, thus we will store it in `kstack_pt`. `refcount` lets us count how often the address space is used—for non-threaded processes this value will always be 1.

We define three possible values for the `status` field of an address space:

[162a] ⟨*constants* 112a⟩+≡ (44a) ◁159c 168b▷
```
#define AS_FREE   0
#define AS_USED   1
#define AS_DELETE 2
```
Defines:
`AS_DELETE`, used in chunk 166c.
`AS_FREE`, used in chunks 162d, 169a, 171a, and 307a.
`AS_USED`, used in chunks 122c, 162e, 163c, and 308c.

The meaning of $\texttt{AS\_FREE}_{162a}$ and $\texttt{AS\_USED}_{162a}$ is obvious, but why we need a third state $\texttt{AS\_DELETE}_{162a}$ will only become clear when you reach the section that deals with process termination. Briefly, we cannot immediately destroy the address space of a process which has actively requested its termination, so that has to happen a bit later, and in the meantime the address space will be marked as "to be deleted".

address space table

The *address space table* is an array of address space descriptors, and we will need a function that lets us search for a free entry in the table:

[162b] ⟨*global variables* 92b⟩+≡ (44a) ◁151a 168c▷
```
address_space address_spaces[MAX_ADDR_SPACES] = { { 0 } };
```
Defines:
`address_spaces`, used in chunks 122c, 152a, 162, 163c, 165–67, 169–71, 173a, 210, 211, 232c, 233c, 255b, 257b, 260a, 261, 279c, 289c, 291, 296, 297, 307a, and 308c.
Uses `address_space` 161 and `MAX_ADDR_SPACES` 158a.

[162c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁151b 163a▷
```
int get_free_address_space ();
```

It returns an integer value that serves as an index into the table.

[162d] ⟨*function implementations* 100b⟩+≡ (44a) ◁151c 163c▷
```
int get_free_address_space () {
  addr_space_id id = 0;
  while ((address_spaces[id].status != AS_FREE) && (id < MAX_ADDR_SPACES)) id++;
  if (id == MAX_ADDR_SPACES) id = -1;
  return id;
}
```
Defines:
`get_free_address_space`, used in chunks 162c and 163c.
Uses `addr_space_id` 158b, `address_spaces` 162b, `AS_FREE` 162a, and `MAX_ADDR_SPACES` 158a.

We use the first entry of the array $\texttt{address\_spaces}_{162b}$ for the kernel and let it point to the kernel page directory. We add the code for initializing this entry just after enabling paging:

[162e] ⟨*enable paging for the kernel* 109a⟩+≡ (97) ◁109a
```
address_spaces[0].status   = AS_USED;
address_spaces[0].pd       = &kernel_pd;
address_spaces[0].pid      = -1;          // not a process
```
Uses `address_spaces` 162b, `AS_USED` 162a, and `kernel_pd` 105a.

Setting `pid` to -1 marks this entry as an address space which belongs to no process.

Here is what we need to do in order to create a fresh address space: We first retrieve a new address space ID and mark its entry in the table as used. Then we reserve memory for a new page directory and copy the system's one into it. Finally we set up some user space memory and add it to the page directory:

⟨*function prototypes* 45a⟩+≡ (44a) ◁162c 164c▷ [163a]
```
int create_new_address_space (int initial_ram, int initial_stack);
```

The argument `initial_ram` defines the amount of process-private memory that should be allocated at once, similarly `initial_stack` is the initial size of the user mode stack which will always be just 4 KByte (though it can grow later). We expect the `initial_ram` and `initial_stack` values to be multiples of the page size (4 KByte)—if not, we will make them so, using this macro:

⟨*macro definitions* 35a⟩+≡ (44a) ◁117 174a▷ [163b]
```
#define MAKE_MULTIPLE_OF_PAGESIZE(x)  x = ((x+PAGE_SIZE-1)/PAGE_SIZE)*PAGE_SIZE
```
Defines:
`MAKE_MULTIPLE_OF_PAGESIZE`, used in chunk 163c.
Uses `PAGE_SIZE` 112a.

If the function call is successful, `create_new_address_space`$_{163c}$ returns the ID of the newly created address space, otherwise -1:

⟨*function implementations* 100b⟩+≡ (44a) ◁162d 165b▷ [163c]
```
int create_new_address_space (int initial_ram, int initial_stack) {
  MAKE_MULTIPLE_OF_PAGESIZE (initial_ram);
  MAKE_MULTIPLE_OF_PAGESIZE (initial_stack);
  // reserve address space table entry
  addr_space_id id;
  if ( (id = get_free_address_space ()) == -1 )  return -1;    // fail
  address_spaces[id].status    = AS_USED;
  address_spaces[id].memstart  = BINARY_LOAD_ADDRESS;
  address_spaces[id].memend    = BINARY_LOAD_ADDRESS + initial_ram;
  address_spaces[id].stacksize = initial_stack;
  address_spaces[id].refcount  = 1;  // default: used by one process
  ⟨reserve memory for new page directory 164a⟩   // sets new_pd
  address_spaces[id].pd = new_pd;
  ⟨copy master page directory to new directory 164b⟩

  int frameno, pageno;  // used in the following two code chunks
  if (initial_ram > 0)   {  ⟨create initial user mode memory 164d⟩  }
  if (initial_stack > 0) {  ⟨create initial user mode stack 165a⟩    }
  return id;
};
```
Defines:
`create_new_address_space`, used in chunks 163–65, 190a, and 210a.
Uses `addr_space_id` 158b, `address_spaces` 162b, `AS_USED` 162a, `BINARY_LOAD_ADDRESS` 159a, `get_free_address_space` 162d, and `MAKE_MULTIPLE_OF_PAGESIZE` 163b.

As usual we use `request_new_page`$_{120a}$ to get a fresh page of virtual memory which will store the new page directory: that function will also update the page directories of all

already existing address spaces if it has to create a new page table (for addresses in the kernel memory).

[164a] ⟨*reserve memory for new page directory* 164a⟩≡ (163c)
```
page_directory *new_pd = request_new_page ();
if (new_pd == NULL) {  // Error
  printf ("\nERROR: no free page, aborting create_new_address_space\n");
  return -1;
};
memset (new_pd, 0, sizeof (page_directory));
```
Uses create_new_address_space 163c, memset 596c, NULL 46a, page_directory 103d, printf 601a, and request_new_page 120a.

For copying the kernel page directory to the new directory, we simply use an assignment; this copies all references to page tables which exist in the original (kernel) page directory.

[164b] ⟨*copy master page directory to new directory* 164b⟩≡ (163c)
```
*new_pd = kernel_pd;
memset ((char*)new_pd, 0, 4);    // clear first entry (kernel pd contains
                                 // old reference to some page table)
```
Uses kernel_pd 105a and memset 596c.

Note that once we have more than one address space, we must make sure that all changes to the kernel part (the addresses starting at `0xC0000000`) will be made in each copy. The page tables of that area are all shared, but when we create a new page table, we have to write the new mapping into *every* page directory—you have already seen the code on page 122.

We modify the new page directory so that it contains information about the user mode memory, stack and kernel stack. For that purpose we will use a function

[164c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁163a 166b▷
```
int as_map_page_to_frame (int as, unsigned int pageno, unsigned int frameno);
```

which can create mappings of page numbers to frame numbers in a specific address space. We will define it just afterwards; it finds out what page table entry to modify and, if needed, also creates a new page table and updates the page directory to point to it.

[164d] ⟨*create initial user mode memory* 164d⟩≡ (163c)
```
pageno = 0;
while (initial_ram > 0) {
  if ((frameno = request_new_frame ()) < 0) {
    printf ("\nERROR: no free frame, aborting create_new_address_space\n");
    return -1;
  };
  as_map_page_to_frame (id, pageno, frameno);
  pageno++;
  initial_ram -= PAGE_SIZE;
};
```
Uses as_map_page_to_frame 165b, create_new_address_space 163c, PAGE_SIZE 112a, printf 601a, and request_new_frame 118b.

Reserving memory for the user mode stack looks almost the same, we just let the stack grow downwards whereas above the memory addresses moved upwards: so we need to modify the page number with `pageno--` instead of `pageno++`:

⟨*create initial user mode stack* 165a⟩≡ (163c) [165a]
```
pageno = TOP_OF_USER_MODE_STACK / PAGE_SIZE;
while (initial_stack > 0) {
  if ((frameno = request_new_frame ()) < 0) {
    printf ("\nERROR: no free frame, aborting create_new_address_space\n");
    return -1;
  };
  pageno--;
  as_map_page_to_frame (id, pageno, frameno);
  initial_stack -= PAGE_SIZE;
}
```
Uses as_map_page_to_frame 165b, create_new_address_space 163c, PAGE_SIZE 112a, printf 601a, request_new_frame 118b, and TOP_OF_USER_MODE_STACK 159b.

We will now describe how to enter the page-to-frame mapping in the new address space's page tables. Getting new physical memory is not a problem since we already have defined the function `request_new_frame`$_{118b}$`()` which reserves a new frame.

The function `as_map_page_to_frame`$_{165b}$ creates such a mapping in a given address space. It will basically be a rewrite of parts of ⟨*enter frames in page table* 121a⟩.

⟨*function implementations* 100b⟩+≡ (44a) ◁163c 166c▷ [165b]
```
int as_map_page_to_frame (int as, unsigned int pageno, unsigned int frameno) {
  // for address space as, map page #pageno to frame #frameno
  page_table *pt;
  page_directory *pd;

  pd = address_spaces[as].pd;           // use the right address space
  unsigned int pdindex = pageno/1024;   // calculate pd entry
  unsigned int ptindex = pageno%1024;   // ... and pt entry

  if ( ! pd->ptds[pdindex].present ) {
    // page table is not present
    ⟨create new page table for this address space 166a⟩  // sets pt
  } else {
    // get the page table
    pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
  };
  if (pdindex < 704)   // address below 0xb0000000 -> user access
    UMAP ( &(pt->pds[ptindex]), frameno << 12 );
  else
    KMAP ( &(pt->pds[ptindex]), frameno << 12 );
  return 0;
};
```
Defines:
as_map_page_to_frame, used in chunks 164, 165a, 173a, 211a, 257c, and 291.
Uses address_spaces 162b, KMAP 101a, page_directory 103d, page_table 101b, PHYSICAL 116a, and UMAP 101a.

In the last lines of this function we differentiate between user mode and kernel mode memory and use the appropriate macro (UMAP$_{101a}$ or KMAP$_{101a}$) to create an entry which allows or forbids access for processes in user mode: The address range `0x00000000` – `0xaffffffff` grants the process access in user mode, whereas `0xb0000000` – `0xffffffff` shall only be accessible in kernel mode.

Remember that every page directory entry lets us address one page table which holds the addresses of up to 1 024 pages, or $1\,024 \times 4\,096 = 4\,194\,304$ bytes (4 MByte) of memory.

$$\frac{\texttt{0xb0000000}}{1\,024 \times 4\,096} = 704$$

so we must use UMAP$_{101a}$ if `pdindex` < 704.

Now we need to explain how to create a new page table: We start with fetching a free frame and point to it from the page directory.

[166a] ⟨*create new page table for this address space* 166a⟩≡ (165b)
```
int new_frame_id = request_new_frame ();
memaddress address = PHYSICAL (new_frame_id << 12);
pt = (page_table *) address;
memset (pt, 0, sizeof (page_table));
UMAPD ( &(pd->ptds[pdindex]), new_frame_id << 12);
```
Uses memaddress 46c, memset 596c, new_frame_id, page_table 101b, PHYSICAL 116a, request_new_frame 118b, and UMAPD 103c.

6.1.3 Destroying an Address Space

When we `exit` from a process, we must also destroy its address space and release all pages used by it. For that purpose we write a function `destroy_address_space`$_{166c}$`()`:

[166b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁164c 168a▷
```
void destroy_address_space (addr_space_id id);
```

Its main task is to undo all the memory allocations that were performed when we created the address space so that no memory leaks occur: A sequence like

```
id = create_new_address_space (...);
destroy_address_space (id);
```

should return the global memory status to the same situation that it had before the creation:

[166c] ⟨*function implementations* 100b⟩+≡ (44a) ◁165b 168d▷
```
void destroy_address_space (addr_space_id id) {
  // called only from syscall_exit(), with interrupts off
  if (--address_spaces[id].refcount > 0) return;
  addr_space_id as = current_as;          // remember current address space
  current_as = id;                        // set current_as: we call release_page()

  ⟨destroy AS: release user mode pages 167a⟩ // all pages used by the process
```

```
    ⟨destroy AS: release user mode stack 167b⟩  // all its user mode stack pages
    ⟨destroy AS: release page tables 167c⟩      // the page tables (0..703)

    current_as = as;                          // restore current_as
    address_spaces[id].status = AS_DELETE;    // change AS status

    // remove kernel stack (cannot do this here, this stack is in use right now)
    add_to_kstack_delete_list (id);
  }
```

Defines:
destroy_address_space, used in chunks 166b and 216b.
Uses add_to_kstack_delete_list 168d, addr_space_id 158b, address_spaces 162b, AS_DELETE 162a, current_as 170b, release_page 122d, and syscall_exit 216b.

The comment in the above chunk's first line refers to protection of the thread table data. We will later discuss synchronization issues (in Chapter 11), and the address space table is one of the critical data structures that must be treated carefully. So, one would expect to see code for protecting it in this function, but this protection occurs elsewhere. In short, we will only modify the address space table when we hold a lock for the thread table, and that lock is already held when the kernel enters this function. (The same holds for the create_new_address_space$_{163c}$ function.)

Releasing the user mode memory is done in two simple steps:

⟨*destroy AS: release user mode pages* 167a⟩≡ (166c) [167a]

```
  for ( int i = address_spaces[id].memstart / PAGE_SIZE;
        i < address_spaces[id].memend / PAGE_SIZE;
        i++ ) {
    release_page (i);
  };
```

Uses address_spaces 162b, PAGE_SIZE 112a, and release_page 122d.

⟨*destroy AS: release user mode stack* 167b⟩≡ (166c) [167b]

```
  for ( int i = TOP_OF_USER_MODE_STACK / PAGE_SIZE - 1;
        i > (TOP_OF_USER_MODE_STACK-address_spaces[id].stacksize) / PAGE_SIZE - 1;
        i-- ) {
    release_page (i);
  };
```

Uses address_spaces 162b, PAGE_SIZE 112a, release_page 122d, and TOP_OF_USER_MODE_STACK 159b.

After releasing all the individual pages, we can also get rid of the page tables which refer to user mode memory:

⟨*destroy AS: release page tables* 167c⟩≡ (166c) [167c]

```
  page_directory *tmp_pd = address_spaces[id].pd;
  for (int i = 0;  i < 704;  i++) {
    if (tmp_pd->ptds[i].present)
      release_frame (tmp_pd->ptds[i].frame_addr);
  }
```

Uses address_spaces 162b, page_directory 103d, and release_frame 119b.

In the last code chunk the loop goes from 0 to 703 since that is the last page directory entry which points to a page table that is used in user mode (cf. the discussion of UMAP$_{101a}$ vs. KMAP$_{101a}$ usage in the implementation of as_map_page_to_frame$_{165b}$ on page 166).

We will remove the kernel stack later when we're not using it any more—doing this right now would crash the system because that memory is still in use. For that purpose we use a global variable which contains either 0 or the ID of an address space whose kernel stack needs removal. That is why we called the function

[168a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁166b 170a▷
```
void add_to_kstack_delete_list (addr_space_id id);
```

in the above code. We allow up to 1024 entries in the kernel stack delete list:

[168b] ⟨*constants* 112a⟩+≡ (44a) ◁162a 169b▷
```
#define KSTACK_DELETE_LIST_SIZE 1024
```
Defines:
KSTACK_DELETE_LIST_SIZE, used in chunks 168 and 169a.

kernel stack delete list

The *kernel stack delete list* is just an array of address space IDs that we initialize with null values.

[168c] ⟨*global variables* 92b⟩+≡ (44a) ◁162b 170b▷
```
addr_space_id kstack_delete_list[KSTACK_DELETE_LIST_SIZE] = { 0 };
```
Uses addr_space_id 158b and KSTACK_DELETE_LIST_SIZE 168b.

Entering an address space ID in the delete list is simple:

[168d] ⟨*function implementations* 100b⟩+≡ (44a) ◁166c 170c▷
```
void add_to_kstack_delete_list (addr_space_id id) {
  ⟨begin critical section in kernel 380a⟩
  int i;
  for (i = 0;  i < KSTACK_DELETE_LIST_SIZE;  i++) {
    // try to enter it here
    if (kstack_delete_list[i] == 0) {
      // found a free entry
      kstack_delete_list[i] = id;  break;
    }
  }
  ⟨end critical section in kernel 380b⟩
  if (i == KSTACK_DELETE_LIST_SIZE)
    printf ("ERROR ADDING ADDRESS SPACE TO KSTACK DELETE LIST!\n");
}
```
Defines:
add_to_kstack_delete_list, used in chunk 166c.
Uses addr_space_id 158b, KSTACK_DELETE_LIST_SIZE 168b, and printf 601a.

We have not shown the code for the scheduler yet, it is responsible for switching between the processes and is called regularly by the timer interrupt handler. Whenever the system activates the scheduler it will execute the following code chunk ⟨*scheduler: free old kernel stacks* 169a⟩ which frees those old kernel stacks that we've put into the list:

⟨*scheduler: free old kernel stacks* 169a⟩≡ (277b) [169a]
```
// check all entries in the to-be-freed list
⟨begin critical section in kernel 380a⟩
for (int entry = 0;  entry < KSTACK_DELETE_LIST_SIZE;  entry++) {
  if (kstack_delete_list[entry] != 0 && kstack_delete_list[entry] != current_as) {
    // remove it
    addr_space_id id = kstack_delete_list[entry];
    page_directory *tmp_pd = address_spaces[id].pd;
    page_table     *tmp_pt = (page_table *) address_spaces[id].kstack_pt;
    // this is the page table which maps the last 4 MB below 0xC0000000
    for (int i = 0;  i < KERNEL_STACK_PAGES;  i++) {
      int frameno = tmp_pt->pds[1023-i].frame_addr;
      release_frame (frameno);
    }
    ⟨remove extra thread kernel stacks 261⟩  // see Chapter 7
    kstack_delete_list[entry] = 0;   // remove entry from kstack delete list
    release_page (((memaddress)tmp_pt) >> 12);  // free memory for page table
    release_page (((memaddress)tmp_pd) >> 12);  // ... and page directory
    address_spaces[id].status = AS_FREE;        // mark address space as free
  }
}
⟨end critical section in kernel 380b⟩
```
Uses addr_space_id 158b, address_spaces 162b, AS_FREE 162a, current_as 170b, KERNEL_STACK_PAGES 169b, kstack, KSTACK_DELETE_LIST_SIZE 168b, memaddress 46c, page_directory 103d, page_table 101b, release_frame 119b, and release_page 122d.

We haven't defined the constant KERNEL_STACK_PAGES$_{169b}$ yet: it tells the system how many pages it shall reserve for the kernel stack.

⟨*constants* 112a⟩+≡ (44a) ◁168b 176a▷ [169b]
```
// kernel stack (per process): 1 page = 4 KByte
#define KERNEL_STACK_PAGES 4
#define KERNEL_STACK_SIZE PAGE_SIZE * KERNEL_STACK_PAGES
```
Defines:
KERNEL_STACK_PAGES, used in chunks 169a, 211, 257b, and 261.
KERNEL_STACK_SIZE, used in chunks 192b and 211b.
Uses PAGE_SIZE 112a.

We may sometimes also need to know the size of the kernel stack (KERNEL_STACK_SIZE$_{169b}$).

6.1.4 Switching between Address Spaces

In order to switch between two address spaces it is sufficient to load the new address space's page directory address into the *CR3* register.

Note that using the function activate_address_space$_{170c}$ (which we show in this section) should be avoided because it has the side effect of switching the kernel stack. Even while it is implemented as inline function, it is still not safe to call it: parameter passing creates local variables (on the kernel stack) which are lost after the context switch. We will only use it when we start the very first process.

In earlier versions of the code, ⟨*scheduler: context switch* 279c⟩ used to make a function call to `activate_address_space`$_{170c}$`()` and it caused many problems (the operating system crashed). After moving the *CR3* loading code directly into the context switch, the problems disappeared.

[170a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁168a 170d▷
```
inline void activate_address_space (addr_space_id id) __attribute__((always_inline));
```

[170b] ⟨*global variables* 92b⟩+≡ (44a) ◁168c 176b▷
```
addr_space_id current_as = 0;  // global variable: current address space
```
Defines:
`current_as`, used in chunks 120c, 123a, 152a, 166c, 169a, 170c, 173a, 210a, 216b, 232c, 233c, 255, 257, 260a, 279c, 289–91, 298a, 299a, 342b, 605c, and 614a.
Uses `addr_space_id` 158b.

[170c] ⟨*function implementations* 100b⟩+≡ (44a) ◁168d 170e▷
```
inline void activate_address_space (addr_space_id id) {
  // NOTE: Do not call this from the scheduler - where needed, replicate the code
  memaddress virt = (memaddress)address_spaces[id].pd;  // get PD address
  memaddress phys = mmu (0, virt);              // and find its physical address
  asm volatile ("mov %0, %%cr3" : : "r"(phys)); // write CR3 register
  current_as = id;                              // set current address space
  current_pd = address_spaces[id].pd;           // set current page directory
};
```
Defines:
`activate_address_space`, used in chunks 190a and 605c.
Uses `addr_space_id` 158b, `address_spaces` 162b, `current_as` 170b, `current_pd` 105a, `memaddress` 46c, `mmu` 172a, and `write` 429b.

Here we use another function called `mmu`$_{172a}$ which emulates the behavior of the memory management unit (MMU) and calculates the physical address belonging to a virtual address with respect to an address space. We will implement it soon.

We provide a helper function

[170d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁170a 171b▷
```
void list_address_spaces ();
```

which shows the list of used address spaces; it is only needed for debugging.

[170e] ⟨*function implementations* 100b⟩+≡ (44a) ◁170c 171a▷
```
void list_address_space (addr_space_id id) {
  int mem  = (memaddress) address_spaces[id].pd;
  int phys = mmu (id, (memaddress) address_spaces[id].pd);  // emulate MMU
  int memstart = address_spaces[id].memstart;
  int memend = address_spaces[id].memend;
  int stack = address_spaces[id].stacksize;
  printf ("ID: %d, MEM: %08x, PHYS: %08x  - USER: %08x, USTACK: %08x\n",
    id, mem, phys, memend-memstart, stack);
}
```
Defines:
`list_address_space`, used in chunk 171a.
Uses `addr_space_id` 158b, `address_spaces` 162b, `memaddress` 46c, `mmu` 172a, and `printf` 601a.

⟨*function implementations* 100b⟩+≡ (44a) ◁170e 171c▷ [171a]
```
void list_address_spaces () {
  addr_space_id id;
  for (id = 0;  id < MAX_ADDR_SPACES;  id++) {
    if (address_spaces[id].status != AS_FREE) {
      list_address_space (id);
    }
  }
}
```
Defines:
list_address_spaces, used in chunks 170d and 608b.
Uses addr_space_id 158b, address_spaces 162b, AS_FREE 162a, list_address_space 170e, and MAX_ADDR_SPACES 158a.

list_address_space$_{170e}$ also uses the mmu$_{172a}$ function—it is time to provide its implementation. We start with a function mmu_p$_{171c}$ which, given an address space ID and a page number, finds out whether the page is mapped in that address space and returns the frame number of the mapped frame.

⟨*function prototypes* 45a⟩+≡ (44a) ◁170d 172b▷ [171b]
```
unsigned int mmu_p (addr_space_id id, unsigned int pageno); // pageno -> frameno
memaddress mmu (addr_space_id id, memaddress vaddress); // virtual -> phys. addr.
```

mmu_p$_{171c}$ looks up the page directory and then the right page table which holds the mapping for the virtual address. Note that this function can only work if the page table is in memory—if it was paged out, it will return −1 (or actually: INT32_MAX, since it is of type unsigned int).

⟨*function implementations* 100b⟩+≡ (44a) ◁171a 172a▷ [171c]
```
unsigned int mmu_p (addr_space_id id, unsigned int pageno) {
  unsigned int pdindex = pageno/1024;
  unsigned int ptindex = pageno%1024;
  page_directory *pd = address_spaces[id].pd;
  if ( ! pd->ptds[pdindex].present ) {
    return -1;
  } else {
    page_table *pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
    if ( pt->pds[ptindex].present ) {
      return pt->pds[ptindex].frame_addr;
    } else {
      return -1;
    };
  }
};
```
Defines:
mmu_p, used in chunks 120c, 123a, 171b, 172a, 261, 293d, 294, and 614a.
Uses addr_space_id 158b, address_spaces 162b, page_directory 103d, page_table 101b, and PHYSICAL 116a.

and with mmu_p$_{171c}$ we can easily implement mmu$_{172a}$ because we just have to split a virtual address into page number and offset, then call mmu_p$_{171c}$ to find the frame number and reassemble that and the offset to form a physical address:

[172a] ⟨*function implementations* 100b⟩+≡ (44a) ◁171c 173a▷
```
memaddress mmu (addr_space_id id, memaddress vaddress) {
  unsigned int tmp = mmu_p (id, (vaddress >> 12));
  if (tmp == -1)
    return -1;  // fail
  else
    return (tmp << 12) + (vaddress % PAGE_SIZE);
}
```
Defines:
mmu, used in chunks 170, 211, and 279c.
Uses addr_space_id 158b, memaddress 46c, mmu_p 171c, and PAGE_SIZE 112a.

Note that both functions return -1 if the page or virtual address does not exist, but only when calling mmu_p$_{171c}$ we can be sure that a return value of -1 indicates a non-existing page—after all, *some* virtual address might be mapped to physical address 0xFFFFFFFF (which is the same as -1).

6.1.5 Enlarging an Address Space

We want to allow processes to increase their standard memory usage (which is 64 KByte). Unix systems provide an implementation of malloc as part of their standard library.

The brk system call (and corresponding library function) is still available on modern Unix systems, but its use is advised against. brk adds one or more pages to the calling process' data "segment". The function sbrk$_{174d}$ does the same but is more user-friendly: It takes an increment as argument, so if the process needs 16 KByte of extra memory, it can call sbrk$_{174d}$(16*1024). sbrk$_{174d}$ returns the lowest address of the new memory: After executing void *mem = sbrk$_{174d}$(incr), the address range [mem ; mem + incr − 1] is available to the process.

How can we do this in Ulix? Remember that each process uses an address_space$_{161}$ which has elements named memstart and memend (the last of which is the first address that is *not* available) and a pointer to the address space's page directory (pd). Thus, sbrk$_{174d}$ just needs to

- acquire the needed number of frames,
- modify the page directory so that the new frames are mapped just after the last old pages and
- update the memend element.

It then returns the first (virtual) address of the first new page.

We start with the kernel-internal function u_sbrk$_{173a}$; we expect that its argument is always a multiple of PAGE_SIZE$_{112a}$:

[172b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁171b 178c▷
```
void *u_sbrk (int incr);
```

⟨*function implementations* 100b⟩+≡ (44a) ◁172a 183c▷ [173a]

```
void *u_sbrk (int incr) {
  int pages = incr / PAGE_SIZE;
  address_space *aspace = &address_spaces[current_as];
  memaddress oldbrk = aspace->memend;

  for (int i = 0;  i < pages;  i++) {
    int frame = request_new_frame ();
    if (frame == -1) { return (void*)(-1); } // error!
    as_map_page_to_frame (current_as, aspace->memend/PAGE_SIZE, frame);
    aspace->memend += PAGE_SIZE;
  }
  return (void*) oldbrk;
}
```

Defines:
u_sbrk, used in chunks 172–74, 233c, and 257c.
Uses address_space 161, address_spaces 162b, as_map_page_to_frame 165b, current_as 170b, memaddress 46c, PAGE_SIZE 112a, and request_new_frame 118b.

Next we need to provide a system call for the $\texttt{u\_sbrk}_{173a}$ function so that a process can call this function. So far, you have not seen how Ulix implements system calls (we will show this in Chapter 6.4), so you might want to skip the following description and turn back to it when you reach the system call chapter. We've also put a reminder into that section.

As a brief summary, system calls are functions whose start addresses we enter in a syscall table. A process can make a system call by loading a syscall number (which serves as an index into that table) into the *EAX* register, storing arguments for the syscall in further registers (*EBX*, *ECX*, ...) and then executing the `int 0x80` assembler instruction. Filling a syscall table entry is handled by the function $\texttt{install\_syscall\_handler}_{201b}$.

The system call number $\texttt{\_\_NR\_brk}_{204c}$ is defined as 45. There is no $\texttt{sbrk}_{174d}$ system call since normally the $\texttt{sbrk}_{174d}$ function is implemented by calling a similar `brk` function. But we only implement $\texttt{u\_sbrk}_{173a}$ and reuse the `brk` system call number.

⟨*syscall prototypes* 173b⟩≡ (202a) 206c▷ [173b]

```
void syscall_sbrk (context_t *r);
```

⟨*code for* syscall_sbrk 173c⟩≡ [173c]

```
void syscall_sbrk (context_t *r) {
  // ebx: increment
  r->eax = (memaddress)u_sbrk (r->ebx);
  return;
}
```

⟨*initialize syscalls* 173d⟩≡ (44b) 206f▷ [173d]

```
install_syscall_handler (__NR_brk, syscall_sbrk);
```

Uses __NR_brk 204c, install_syscall_handler 201b, and syscall_sbrk 174b.

This is the first of many appearances of a code pattern: Most system call handlers set `r->eax` in order to provide a return value and then leave the function with `r->eax`. To simplify our code we will provide a macro `eax_return`$_{174a}$ which combines these two activities and also performs the type cast to an unsigned integer:

[174a] ⟨*macro definitions* 35a⟩+≡ (44a) ◁163b 209b▷
```
#define eax_return(retval) { r->eax = (unsigned int)((retval)); return; }
```
Defines:
`eax_return`, used in chunks 174b, 206d, 213d, 219c, 220a, 222b, 223e, 234b, 299a, 310a, 370d, 372, 426b, 433b, 566d, 583a, 587d, and 590b.

With this macro we can rewrite `syscall_sbrk`$_{174b}$ like this:

[174b] ⟨*syscall functions* 174b⟩≡ (202b) 206d▷
```
void syscall_sbrk (context_t *r) {
  // ebx: increment
  eax_return ( u_sbrk (r->ebx) );
}
```
Defines:
`syscall_sbrk`, used in chunk 173.
Uses `context_t` 142a, `eax_return` 174a, and `u_sbrk` 173a.

We also provide a user mode library function `sbrk`$_{174d}$ so that you can simply call `sbrk`$_{174d}$ in an application program (instead of manually inserting the necessary code for the system call). Again, this will become clear once you reach the description of our system call interface—which is only a few pages away. We only display the necessary code without further explanation:

[174c] ⟨*ulixlib function prototypes* 174c⟩≡ (48a) 203a▷
```
void *sbrk (int incr);
```
Uses `sbrk` 174d.

[174d] ⟨*ulixlib function implementations* 174d⟩≡ (48b) 203b▷
```
void *sbrk (int incr) { return (void*)syscall2 (__NR_brk, incr); }
```
Defines:
`sbrk`, used in chunk 174c.
Uses `__NR_brk` 204c and `syscall2` 203c.

6.2 Thread Control Blocks and Thread Queues

TCB
PCB

The *thread control block* (TCB) is the central place in the kernel where information about a thread is held. In the times when people used to speak about processes instead of threads, the TCB was called the PCB which stands for *process control block*.

One main purpose of the TCB is to store the *processor state* of a thread (sometimes also called *context*) during the times when it is not assigned to a physical processor. Note that the processor state is not the same as the thread state. As you will see soon, the state of a thread can be `running`, `blocked` etc. The processor state is all information that is necessary to pretend that the processor has never executed any other thread as the one to which the TCB belongs.

The TCB contains (among other data) the following information:

- A unique identifier of the thread. This is the so-called *thread identifier* (TID) . In previous times, the TID was often called PID for *process identifier*. We also keep a PID entry in the TCB, it will be needed when we introduce threads as part of a process. Thread ID Process ID
- Storage space to save the processor context, i. e., the registers, the stack pointer(s), etc.
- Depending on the thread state, the TCB contains an indication on what event the thread is waiting for if it is in state `blocked`.
- Information about the memory that this thread is using—we have already defined address spaces, and we will store an address space ID in the TCB.
- Any other information which may be useful to keep the system running efficiently. For example, statistical information could be stored here on how often the thread has been running in the past. This could help the scheduler make efficient scheduling decisions.

Note that the information about the address space must be handled differently in PCBs and in TCBs. In a system where multiple threads can run within one address space, there is an $n : 1$ mapping between threads and address spaces. In a classical Unix system with processes (one address space with exactly one thread), the mapping is $1 : 1$ and each thread can store the full address space information in the PCB itself. With an $n : 1$ mapping, an extra data structure is necessary to avoid having redundant information in the TCBs.

So here is the declaration of the TCB structure. We have entries for the thread ID `tid`, the process ID `pid`, a parent process ID `ppid` (so we can build a process tree), the processor context (of type $\texttt{context\_t}_{142a}$) consisting of the general purpose registers and the special purpose registers and a reference to the address space in which the thread runs (which we already added to the data structure when we discussed address spaces). We also reserve place for three memory addresses which hold the instruction pointer, stack pointer and base pointer contents). More entries will follow later.

⟨ *public type definitions* 142a⟩+≡ (44a 48a) ◁142a 254a▷ [175]

```
typedef struct {
  thread_id  pid;         // process id
  thread_id  tid;         // thread id
  thread_id  ppid;        // parent process
  int        state;       // state of the process
  context_t  regs;        // context
  memaddress esp0;        // kernel stack pointer
  memaddress eip;         // program counter
  memaddress ebp;         // base pointer
  ⟨ more TCB entries 158c⟩
} TCB;
```

Defines:
`TCB`, used in chunks 176b, 188, 190a, 210, 223e, 224f, 255c, 260a, 276c, 280a, 291, 562b, and 580–82.
Uses `context_t` 142a, `memaddress` 46c, and `thread_id` 178a.

Thread Table
Process Table

The TCBs of all threads are collected in a large table within the kernel. This table is called the *thread table*. Again, in ancient times when the table was a collection of PCBs instead of TCBs, it was called *process table*.

The thread table is simply an array of $\texttt{TCB}_{175}$s. The size of the table must be finite, so there exists a maximum number of threads which can coexist at any point in time.

[176a] ⟨*constants* 112a⟩+≡ (44a) ◁169b 190b▷

```
#define MAX_THREADS 1024
```

Defines:
MAX_THREADS, used in chunks 176b, 188a, 217b, 219c, 223e, 281, 322b, and 605d.

[176b] ⟨*global variables* 92b⟩+≡ (44a) ◁170b 180b▷

```
TCB thread_table[MAX_THREADS];
```

Defines:
thread_table, used in chunks 152b, 184–88, 190a, 206b, 210b, 216, 217, 219c, 220a, 222–24, 234b, 255c, 260a, 277, 278, 281, 322, 324a, 328–30, 332b, 334b, 335b, 368, 369c, 371a, 381a, 383a, 412b, 416b, 424, 426b, 432e, 478b, 487a, 562b, 564–66, 573b, 577c, 580–82, 587d, 588b, 605d, and 606.
Uses MAX_THREADS 176a and TCB 175.

We define the maximum number of threads here. It should somehow correspond to the maximum number of address spaces $\texttt{MAX\_ADDR\_SPACES}_{158a}$ defined earlier in Section 6.1. For example, it doesn't make sense to allow more address spaces than threads (since every thread can have at most one address space). We have set both values to 1024 which would let Ulix run 1023 processes, each of which has its individual address space. The number 0 is reserved—both in the address space table (where it refers to the kernel address space) and in the thread table (because we use that entry for a different purpose related to the scheduler).

6.2.1 Thread State

For each thread, the TCB contains a `state` field. We will define the possible values which this field can hold later (p. 180), but here we already give an overview of the what can theoretically happen to a thread.

6.2.1.1 Simple State Model for Threads

Thread State
State Model

Threads have a lifecycle. They are born, live and finally die. During their life they undergo many changes. For example, they sometimes are executed by a physical processor and sometimes not. This is what is called the *thread state* or simply *state*. The number and type of states together with the transitions which a thread can experience during its lifetime is called a *state model*.

The simplest state model which can be found in almost every textbook on operating systems consists of three states: `running`, `ready` and `blocked`. Here's what these states mean:

- A thread in state `running` is actually executing on a physical processor. If there is more than one processor in the system, more than one thread can be in this state.
- A thread in state `ready` is not currently assigned to a physical processor, but it *could* be assigned. In other words, the thread is ready to run in case a physical processor

becomes free. Many threads can be in this state at the same time; they are kept within a list called the *ready queue*. Ready Queue

- A thread in state `blocked` is waiting for a certain event. Only after this event has happened, it can become `running` or `ready` again. There can be many different events for which a thread can wait. For example, a thread could be waiting for a page fault to be serviced, i. e., waiting for an I/O operating to terminate. Another example is that a thread is waiting for a synchronization operation to be executed by another thread (see Chapter 11).

  Usually an indication of the event for which the thread is waiting is part of the `blocked` state. This can be interpreted as many different `blocked` states. For simplicity, most textbooks therefore reduce these states to just one. Many threads can be in a blocked state at the same time. They are kept internally within one (or more) *blocked queues*. Blocked Queue

We will use this state model in Ulix.

The possible transitions between thread states are depicted in Figure 6.2. We enumerate and explain them here now:

- `add`: a new thread is dynamically created and enters the set of threads in the state `ready`.
- `assign`: a new thread from state `ready` is assigned to the processor and becomes `running`.
- `block`: a running thread invokes a blocking system operation (e. g., I/O), runs into a page fault or must wait for some other event to continue operation. Now a new thread can become `running`. (Note the difference between the thread state `blocked` and the state transition `block`.)
- $\texttt{deblock}_{\texttt{i386}}$: the event for which a `blocked` thread is waiting has happened. Consequently, the `blocked` thread is transferred to the state `ready`. (Sometimes this transition is also called `ready`, but since this can be confused with the thread state `ready` we prefer to call it `deblock`.)

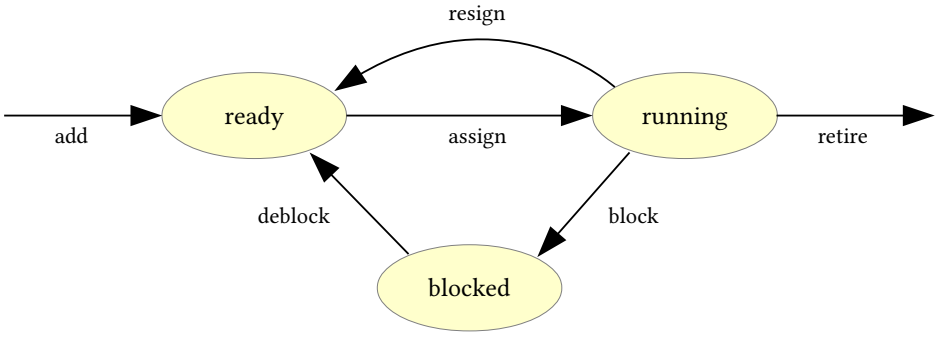


Figure 6.2: States and state transitions in the simple state model for threads.

- `resign`$_{221f}$: the thread which is currently `running` has finished executing parts of its program and leaves the physical processor. It transits from state `running` back into state `ready`. Now a new thread can become `running`.
- `retire`$_{186b}$: a currently `running` thread has finished executing its program code and terminates its lifetime.

Not all possible transitions from one state to the other exist in the state model because a reduced state model decreases the complexity of the implementation. For example it is rather uncommon to transit from `blocked` directly to `running`. Similarly, a newly created thread must be `ready` first before it may become `running`.

The transitions of a thread from one state to the other are initiated by the operating system and happen "instantaneously". Since a state change needs many machine instructions, real instantaneousness cannot be achieved, so the operating system simulates atomic transitions using synchronization operations in the kernel (see Chapter 11). In essence, the atomic transitions are implemented in such "atomic" kernel functions which carry the same name as the state transitions (e. g., `assign`, `resign`$_{221f}$, etc.). The place in the kernel where all these functions are collected is called the *dispatcher*.

Here are the forward declarations of the dispatcher functions. They will be implemented on the following pages. Note that the dispatcher operation `block` takes an indication to the event on which the thread is blocking. This indication is encoded in the particular blocked queue to which the thread should be added. The data type of `blocked_queue`$_{183a}$ will be explained below.

[178a] ⟨*public elementary type definitions* 45e⟩+≡ (44a 48a) ◁158b 560a▷

```
typedef unsigned int thread_id;
```

Defines:
`thread_id`, used in chunks 175, 178c, 181, 183–88, 190a, 192c, 209c, 210a, 216b, 255, 278a, 281, 322a, 362, 364c, 366c, 368, 391a, 424, 426b, and 546a.

[178b] ⟨*type definitions* 91⟩+≡ (44a) ◁161 194b▷

⟨*declaration of blocked queue* 183a⟩

[178c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁172b 183b▷

```
void add     (thread_id t);
void block   (blocked_queue *q, int new_state);
void deblock (thread_id t, blocked_queue *q);
void retire  (thread_id t);
```

(Instead of a `resign` function we will later provide a code chunk ⟨*resign* 221d⟩. Assigning happens only inside the scheduler, so we do not implement a specific function for this activity.)

6.2.1.2 Thread States with Swapping

In case of shortage of memory it may make sense to completely "swap out" a process with all its threads and virtual memory to external storage. In this case it is necessary to define an additional thread state `swapped`, which also leads to a more complex set of state

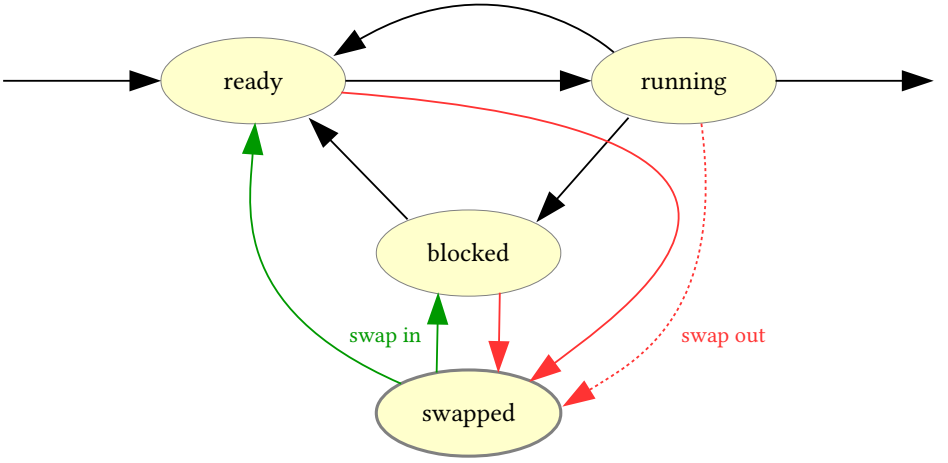


Figure 6.3: Thread states and state transitions for a system that swaps processes out and back in.

transitions since both a ready or blocked thread might be swapped out; depending on the implementation even an active process may ask for being swapped out. The system must remember the last state and restore it when it swaps the process back in (see Figure 6.3). Ulix does not implement swapping since it uses the more advanced paging model.

6.2.1.3 Thread Queues

The operating system has to perform bookkeeping of the state of threads. This can be done in several ways. One approach would be to store an entry `state` of an enumeration type in the TCB that can have the values `blocked`, `ready` or `running`. This is viable but not actually necessary. In modern operating systems the thread state is stored implicitly through the collection of linked lists. These lists contain threads and function as queues. The *ready queue* for example is a list of threads which all are in state `ready`.

Ready Queue

As global data structures we therefore need a couple of global variables:

- For every CPU in the system we need a reference to the thread that is currently assigned to the processor. For monoprocessor systems (like those that Ulix supports) it is sufficient to provide a global variable `current_task`$_{192c}$ of type `thread_id`$_{178a}$. For multiprocessor systems we would have to provide such a variable for every CPU in the system.
- A *ready queue* enumerating all threads that are in state `ready`.
- For every separate class of events which can cause a thread to go into state `blocked`, we need a *blocked queue* enumerating all threads that wait for such an event.
- In case we have a system with swapping, another list is necessary holding all swapped out threads. This is called the *swapped out queue*.

6.2.1.4 Thread States in the ULix Implementation

ULix does not support swapping but writes out individual pages to disk, so we will not need a *swapped* state. However, we will use several separate states to indicate a specific *blocked* state. (As we described above, we *could* use queue membership to indicate the specific blocked state, but using several states lets us access the state more quickly.) The following code chunk lists all the possible states that a process (or thread) can be in:

[180a] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁111c 205a▷

```
// Thread states
#define TSTATE_READY     1   // process is ready
#define TSTATE_FORK      3   // fork() has not completed
#define TSTATE_EXIT      4   // process has called exit()
#define TSTATE_WAITFOR   5   // process has called waitpid()
#define TSTATE_ZOMBIE    6   // wait for parent to retrieve exit value
#define TSTATE_WAITKEY   7   // wait for key press event
#define TSTATE_WAITFLP   8   // wait for floppy
#define TSTATE_LOCKED    9   // wait for lock
#define TSTATE_STOPPED  10   // stopped by SIGSTOP signal
#define TSTATE_WAITHD   11   // wait for hard disk
```

Defines:
TSTATE_EXIT, used in chunks 216b, 217a, 260a, 281, and 564a.
TSTATE_FORK, used in chunks 210b and 255c.
TSTATE_LOCKED, used in chunks 361c, 366a, 391, and 392.
TSTATE_READY, used in chunks 184b, 186b, 278, 563b, and 564c.
TSTATE_STOPPED, used in chunk 563.
TSTATE_WAITFLP, used in chunks 545b and 564c.
TSTATE_WAITFOR, used in chunks 217a, 219c, 281, and 564c.
TSTATE_WAITHD, used in chunks 531a and 564c.
TSTATE_WAITKEY, used in chunks 416b and 564c.
TSTATE_ZOMBIE, used in chunks 152b, 217a, and 281.

We also define a list of state names which can be used when displaying the process list:

[180b] ⟨*global variables* 92b⟩+≡ (44a) ◁176b 187c▷

```
char *state_names[12] = {
  "---",   "READY", "---",   "FORK", "EXIT", "WAIT4", "ZOMBY", "W_KEY",   // 0.. 7
  "W_FLP", "W_LCK", "STOPD", "W_IDE"                                      // 8..11
};
```

Defines:
state_names, used in chunk 605d.

Figure 6.4 shows the state transitions in ULix. The various TSTATE_∗ states are shown as a single state in order to simplify the picture.

6.2.2 Implementing Lists of Threads

Queues are standard data structures offered by almost all modern programming languages. As an example, Java provides the generic class ArrayList<E> in which objects of any type E can be stored and manipulated with standard operations like add(), size() and get(). Unfortunately, "plain" C does not offer this convenience so we have to implement queues by ourselves.

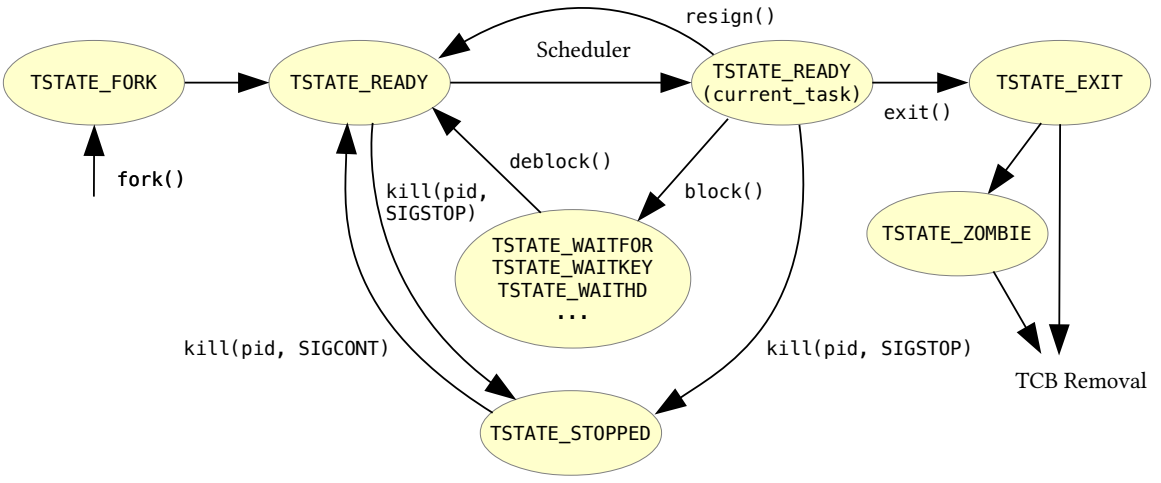


Figure 6.4: Process states and state transitions as implemented by Ulix.

As explained in Section 6.2.1.3, we have to maintain a couple of thread queues within the kernel. In Ulix we maintain only two: the ready queue and the blocked queue. In fact, the blocked queue is not a single queue but there can be multiple blocked queues, one for every event upon which a thread can wait.

When implementing such queues, we could think about using the standard implementation of a (double) linked list found in any introductory textbook on programming. However these implementations usually are examples of programming with dynamic memory allocation, e. g., in C using the `malloc` library call to allocate fresh memory on the heap. This would be a problem in Ulix since we neither have a heap nor a library to call into.

So how can we program linked lists without allocating memory? The first option is to do it like Knuth did it in TEX [Knu86] and provide both a large memory area plus functions for memory allocation and deallocation by ourselves. Since this would be overkill, we choose the second option, which is also the option taken in many operating systems: We use the thread table to implement lists. The idea is to declare two additional entries in the thread control block: one entry called `next` and one called `prev`. Both point to other entries in the thread table. So consider a thread control block $\texttt{TCB}_{175}$ `t`. The entry `t.next` points to the "next" thread in the queue that `t` belongs to. Similarly, `t.prev` points to the "previous" thread in `t`'s queue.

We define the range of these two pointers to be $\texttt{thread\_id}_{178a}$.

⟨ *more TCB entries* 158c ⟩+≡ (175) ◁158c 187b▷ [181]

```
thread_id next;       // id of the ``next'' thread
thread_id prev;       // id of the ``previous'' thread
```

Uses thread_id 178a.

An example of the semantics of the `prev` and `next` entries in the thread table is shown in Figure 6.5. It shows that the thread identifier 0 is used as an "end marker" for the lists. It

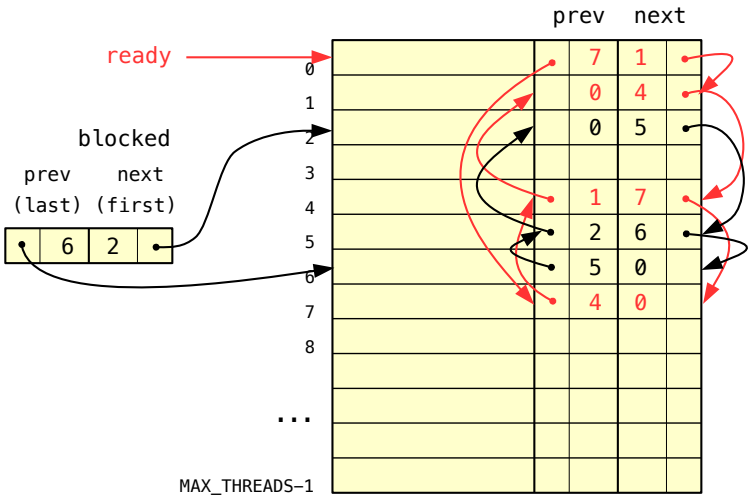


Figure 6.5: Implementation of ready queue and blocked queues. The beginning of the ready queue is implicitly defined by entry 0 in the thread table. The beginning of a blocked queue is a pair of thread identifiers pointing into the thread table from "outside".

also shows that the `prev` entry of the first entry in the queue points to the last element in the queue. In this way, it is easily possible to navigate through the queues in any way which is convenient.

Figure 6.5 also shows a small implementation trick. The thread identifier of the thread itself is always equal to the index of the thread in the thread table. Given a TID of `t`, then $\texttt{thread\_table}_{176b}\texttt{[t]}$ is the thread control block of that thread. This also means that the entry `tid` in the thread control block is more or less superfluous.

Now since we are using the value 0 to mark the end of a list, the entry 0 in the thread table has become more or less useless to store thread information. We use it instead as the "anchor" of the ready queue. So to access the first element in the ready queue, we just need to look into:

$$\texttt{thread\_table}_{176b}\texttt{[0].next}$$

The last entry in the ready queue can similarly be accessed using the following expression:

$$\texttt{thread\_table}_{176b}\texttt{[0].prev}$$

The ready queue in the figure contains threads 1, 4 and 7 (in this order).

Recalling the simple state model of threads in Section 6.2.1.1, every thread is in exactly one state at any time. This means that a thread is either running, ready or blocked. It also means that a thread can be in at most one queue at a time. In case the thread is blocked instead of ready, we can re-use the `prev` and `next` entries in the thread table to implement

the blocked list. We only need to have an anchor for this blocked list. This anchor will be a structure similar to entry 0 in the thread table, but without all the extra fields.

⟨*declaration of blocked queue* 183a⟩≡ (178b) [183a]
```
typedef struct {
  thread_id next;       // id of the ``next'' thread
  thread_id prev;       // id of the ``previous'' thread
} blocked_queue;
```
Defines:
`blocked_queue`, used in chunks 183c, 185–87, 218b, 323d, 360a, 362, 365a, 366c, 368, 391a, 522a, 529a, 544d, and 606.
Uses `thread_id` 178a.

So assume `b` is a variable of type $\texttt{blocked\_queue}_{183a}$ representing a blocked list. If both entries in `b` are 0, then the list is empty. If not, then using the thread table we can now find the first, second etc. element by following the `next` pointers. This way, we can traverse the entire list until we reach an entry in which `next==0`. That's the end of the list. Looking again at Figure 6.5, the blocked queue contains threads 2, 5 and 6 (in this order).

Finally, here's a useful function to initialize a blocked queue:

⟨*function prototypes* 45a⟩+≡ (44a) ◁178c 184a▷ [183b]
```
void initialize_blocked_queue (blocked_queue *q);
```

This is just to encapsulate the semantics of "emptiness".

⟨*function implementations* 100b⟩+≡ (44a) ◁173a 184b▷ [183c]
```
void initialize_blocked_queue (blocked_queue *q) {
  q->prev = 0;
  q->next = 0;
}
```
Defines:
`initialize_blocked_queue`, used in chunks 183b, 218c, 323e, 363d, 364b, 367b, 522b, 529b, and 544e.
Uses `blocked_queue` 183a.

6.2.2.1 Dispatcher Operations as Critical Sections

Before we actually implement queue operations we need to make a slight forward reference to Chapter 11 on synchronization and introduce the notion of a critical section, at least intuitively. Briefly spoken, a *critical section* is a sequence of code that accesses shared resources. Such a section is critical, because (as is explained in Chapter 11) concurrent accesses to a shared resource can wreak havoc with these resources, potentially making them unusable. The shared resources in question here are the shared kernel queues. What we need to ensure is that no two critical sections are executed concurrently. We refrain here from explaining how this can be achieved. What we however do at this point is to mark the beginning and end of the critical sections using the code chunk ⟨*begin critical section in kernel* 380a⟩ and ⟨*end critical section in kernel* 380b⟩ and leave the implementation to Chapter 11.

6.2.2.2 Implementing the Ready Queue

We now provide some convenient functions to add and remove threads from the queues. We start with the ready queue. The function `add_to_ready_queue`$_{184b}$`(t)` adds the thread with identifier `t` to the *end* of the ready queue. It assumes that the TCB of thread `t` has been set up and initialized already.

The function `remove_from_ready_queue`$_{184c}$`(t)` removes the thread with identifier `t` from the ready queue. It assumes that `t` is contained in the ready queue.

[184a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁183b 185a▷

```
void add_to_ready_queue (thread_id t);
void remove_from_ready_queue (thread_id t);
```

Adding to the end of the ready queue is as easy since we have a double linked list.

[184b] ⟨*function implementations* 100b⟩+≡ (44a) ◁183c 184c▷

```
void add_to_ready_queue (thread_id t) {
  ⟨begin critical section in kernel 380a⟩
  thread_id last = thread_table[0].prev;
  thread_table[0].prev = t;
  thread_table[t].next = 0;
  thread_table[t].prev = last;
  thread_table[last].next = t;
  thread_table[t].state = TSTATE_READY;  // set its state to ready
  ⟨end critical section in kernel 380b⟩
}
```

Defines:
`add_to_ready_queue`, used in chunks 186b, 192d, 212, 257c, 364c, and 563b.
Uses `thread_id` 178a, `thread_table` 176b, and `TSTATE_READY` 180a.

Removing is similarly easy.

[184c] ⟨*function implementations* 100b⟩+≡ (44a) ◁184b 185b▷

```
void remove_from_ready_queue (thread_id t) {
  ⟨begin critical section in kernel 380a⟩
  thread_id prev_thread = thread_table[t].prev;
  thread_id next_thread = thread_table[t].next;
  thread_table[prev_thread].next = next_thread;
  thread_table[next_thread].prev = prev_thread;
  ⟨end critical section in kernel 380b⟩
}
```

Defines:
`remove_from_ready_queue`, used in chunks 152b, 186b, 187a, 216b, 260a, and 564c.
Uses `thread_id` 178a and `thread_table` 176b.

We initialize the ready queue to be empty.

[184d] ⟨*initialize kernel global variables* 184d⟩≡ (44b) 306c▷

```
thread_table[0].prev = 0;
thread_table[0].next = 0;
```

Uses `thread_table` 176b.

Note that we cannot use our `initialize_blocked_queue`$_{183c}$ function. Even though we access the same elements (`prev` and `next`), the structures do not match.

6.2.2.3 Implementing a Blocked Queue

Blocked queues are implemented similar to the ready queue, except that the functions are parameterized with a blocked list anchor defined above. We provide again two functions to add and remove a thread from a blocked list. Adding to the queue happens at the "end" of the queue. An additional function allows to inspect the "front" queue element.

⟨*function prototypes* 45a⟩+≡ (44a) ◁184a 188c▷ [185a]
```
void add_to_blocked_queue          (thread_id t, blocked_queue *bq);
void remove_from_blocked_queue     (thread_id t, blocked_queue *bq);
thread_id front_of_blocked_queue   (blocked_queue bq);
```

We implement the easy inspector function to retrieve the front of a blocked queue first. It is so easy that we could have avoided writing this function altogether, but we spell it out for students who have learnt the concept of information hiding.

⟨*function implementations* 100b⟩+≡ (44a) ◁184c 185c▷ [185b]
```
thread_id front_of_blocked_queue (blocked_queue bq) {
  return bq.next;
}
```
Defines:
front_of_blocked_queue, used in chunk 364c.
Uses blocked_queue 183a and thread_id 178a.

We now implement `add_to_blocked_queue`$_{185c}$. Adding happens at the *end* of the queue. The following code is an adaption of the code for the ready queue. The conditional statement at the end is necessary since `thread_table`$_{176b}$`[0]` is not the anchor of a blocked queue.

⟨*function implementations* 100b⟩+≡ (44a) ◁185b 186a▷ [185c]
```
void add_to_blocked_queue (thread_id t, blocked_queue *bq) {
  ⟨begin critical section in kernel 380a⟩
  thread_id last = bq->prev;
  bq->prev = t;
  thread_table[t].next = 0;    // t is ``last'' thread
  thread_table[t].prev = last;
  if (last == 0) {
    bq->next = t;
  } else {
    thread_table[last].next = t;
  }
  ⟨end critical section in kernel 380b⟩
}
```
Defines:
add_to_blocked_queue, used in chunk 187a.
Uses blocked_queue 183a, thread_id 178a, and thread_table 176b.

Removal is similar to the function of the ready queue, except for again the special cases at the end.

[186a] ⟨*function implementations* 100b⟩+≡ (44a) ◁185c 186b▷

```
void remove_from_blocked_queue (thread_id t, blocked_queue *bq) {
  ⟨begin critical section in kernel 380a⟩
  thread_id prev_thread = thread_table[t].prev;
  thread_id next_thread = thread_table[t].next;
  if (prev_thread == 0) {
    bq->next = next_thread;
  } else {
    thread_table[prev_thread].next = next_thread;
  }
  if (next_thread == 0) {
    bq->prev = prev_thread;
  } else {
    thread_table[next_thread].prev = prev_thread;
  }
  ⟨end critical section in kernel 380b⟩
}
```

Defines:
`remove_from_blocked_queue`, used in chunks 186b, 364c, and 564c.
Uses `blocked_queue` 183a, `thread_id` 178a, and `thread_table` 176b.

6.2.2.4 Simple Dispatcher Operations

We now look at the implementations of the three simplest dispatcher operations. These are `add`, `retire`$_{186b}$ and `deblock`$_{186b}$. They are simple because they basically only move threads from one queue to the other.

The functions `add` and `retire`$_{186b}$ take as parameter the identifier of the thread which is newly born or about to die. The function `deblock`$_{186b}$ needs another argument: The blocked queue from which the thread is to be removed. Note that `add` and `retire`$_{186b}$ do not need to be declared as critical sections, because the queue operation already is. `deblock`$_{186b}$ and later `block` however must be executed without interruption so that kernel data structures remain consistent (see Chapter 11).

[186b] ⟨*function implementations* 100b⟩+≡ (44a) ◁186a 187a▷

```
void add (thread_id t) {
  add_to_ready_queue (t);
}

void retire (thread_id t) {
  remove_from_ready_queue (t);
}

void deblock (thread_id t, blocked_queue *q) {
  ⟨begin critical section in kernel 380a⟩
  remove_from_blocked_queue (t, q);
  add_to_ready_queue (t);
  thread_table[t].state = TSTATE_READY;
  ⟨end critical section in kernel 380b⟩
}
```

Defines:
deblock, used in chunks 217a, 281, 322a, 362, 366c, 368, 391a, 522c, 532d, and 546a.
Uses add_to_ready_queue 184b, blocked_queue 183a, remove_from_blocked_queue 186a, remove_from_ready_queue 184c, thread_id 178a, thread_table 176b, and TSTATE_READY 180a.

For blocking the current thread we provide a function `block` which takes two arguments: a blocked queue that the thread shall be moved to and the new state (e. g. $\texttt{TSTATE\_WAITHD}_{180a}$):

⟨*function implementations* 100b⟩+≡ (44a) ◁186b 188d▷ [187a]
```
void block (blocked_queue *q, int new_state) {
  if (current_task == 0) return;
  ⟨begin critical section in kernel 380a⟩
  thread_table[current_task].state = new_state;
  remove_from_ready_queue (current_task);
  add_to_blocked_queue (current_task, q);
  ⟨end critical section in kernel 380b⟩
}
```
Uses add_to_blocked_queue 185c, blocked_queue 183a, current_task 192c, remove_from_ready_queue 184c, and thread_table 176b.

Note that with the above functions we can easily write code that deblocks the "front" element from a blocked queue (if it exists) as follows:

$$\texttt{deblock}_{186b}\ \texttt{(front\_of\_blocked\_queue}_{185b}\ \texttt{(bq), \&bq);}$$

6.2.3 Allocating and Initializing a New TCB

Whenever we create a new thread or process, we will need a fresh TCB entry and initialize it. We add a `used` entry to the thread control block structure $\texttt{TCB}_{175}$

⟨*more TCB entries* 158c⟩+≡ (175) ◁181 205b▷ [187b]
```
boolean used;
```

which lets us declare an entry as used. (Since we initialize the TCB structures with null bytes, we use `used` and not `free`: remember that `false`=0.)

This will allow us to quickly find a free TCB when we create a new thread. Instead of adding such a field, we could have used a bitmap, but since we restrict ourselves to 1024 TCBs, not much space is wasted this way, and searching for a free TCB will be quick.

We will remember in a global variable

⟨*global variables* 92b⟩+≡ (44a) ◁180b 192c▷ [187c]
```
thread_id next_pid = 1;
```
Defines:
next_pid, used in chunk 188.
Uses thread_id 178a.

at which thread number we will start our search (instead of always searching from 1): this will later lead to a continuous sequence of process/thread numbers: even if we terminate a thread, its TCB will not be recycled immediately.

[188a] ⟨*find free TCB entry* 188a⟩≡ (188d) 188b▷
```
boolean tcbfound = false;
thread_id tcbid;
for (tcbid = next_pid;  ((tcbid<MAX_THREADS) && (!tcbfound));  tcbid++) {
  if (thread_table[tcbid].used == false) {
    tcbfound = true;
    break;   // leave for loop
  }
};
```
Uses MAX_THREADS 176a, next_pid 187c, thread_id 178a, and thread_table 176b.

However, once we've reached the maximum number (1023), the search for a free TCB will start over, and from that point on thread numbers will no longer indicate the order of creation of the threads.

[188b] ⟨*find free TCB entry* 188a⟩+≡ (188d) ◁188a
```
if (!tcbfound) {                                  // continue searching at 1
  for (tcbid = 1;  ((tcbid<next_pid) && (!tcbfound));  tcbid++) {
    if (thread_table[tcbid].used == false) {
      tcbfound = true;
      break;   // leave for loop
    }
  };
};

if (tcbfound) next_pid = tcbid+1;                 // update next_pid:
// either tcbfound == false or tcbid == index of first free TCB
```
Uses next_pid 187c, TCB 175, and thread_table 176b.

Once we have a free address space (or reuse one) and also have a free TCB, we can connect them:

[188c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁185a 196b▷
```
int register_new_tcb (addr_space_id as_id);
```
Uses addr_space_id 158b and register_new_tcb 188d.

This function searches for a free TCB, marks it as used and enters the address space ID which we provide as an argument. Thus, whenever we create a new thread, we always call create_new_address_space$_{163c}$ first and register_new_tcb$_{188d}$ afterwards:

[188d] ⟨*function implementations* 100b⟩+≡ (44a) ◁187a 189▷
```
int register_new_tcb (addr_space_id as_id) {
  // called by u_fork()
  ⟨find free TCB entry 188a⟩
  if (!tcbfound)  return -1;              // no free TCB!
  thread_table[tcbid].used       = true;  // mark as used
  thread_table[tcbid].addr_space = as_id; // enter address space ID
  return tcbid;
}
```
Defines:
register_new_tcb, used in chunks 188c, 190a, 210a, and 255b.
Uses addr_space_id 158b, TCB 175, thread_table 176b, and u_fork 209c.

Note that so far we have not entered the new TCB in the ready or one of the blocked queues. This will happen later when the thread has been fully initialized.

6.3 Starting the First Process

Starting the first (`init`) process is different from how all further processes are created: We have to manually set up the memory regions and data structures and load a first program from the disk. Of course, this requires filesystem support which we will implement later—for now assume that the kernel can use filesystem functions similar to the standard Unix functions `open`, `read` and `close`. `init` process

Once the `init` process is running, all further processes will be created using `fork`$_{213g}$ (see Section 6.5). This is what we need to do:

- Setup the TCB (thread control block) list and mark the first TCB as used.
- Create a new address space and reserve memory for user mode (low addresses, with user access).
- Load the process binary from disk into the new address space.
- Reserve memory for the process' kernel stack (low addresses, without user access).
- Enter all the information in the new TCB.
- Update a data structure called TSS (Task State Segment, see Section 6.3.5). TSS
- Switch from kernel mode (ring 0) to user mode (ring 3) and start executing the process' code.

⟨ *function implementations* 100b ⟩+≡ (44a) ◁188d 197a▷ [189]

```
void start_program_from_disk (char *progname) {
  ⟨ start program from disk: prepare address space and TCB entry 190a ⟩
  ⟨ start program from disk: load binary 190c ⟩
  ⟨ start program from disk: create kernel stack 192a ⟩
  ⟨ start program from disk: set uid, gid, euid, egid 573b ⟩
  ⟨ start program from disk: activate the new process 192d ⟩
};
```

Defines:
`start_program_from_disk`, used in chunk 45d.

As you can see, the start routine is rather complex. We discuss the necessary tasks step by step in the following sections, with one exception: The code chunk ⟨ *start program from disk: set uid, gid, euid, egid* 573b ⟩ will be explained much later when we discuss user and group management.

6.3.1 Preparations

We start with registering a new thread control block and fresh address space and entering useful data. The following code chunk contains a few instructions that will not make much sense to you right now since they deal with kernel components you have not seen

yet. For example, it sets the TCB element `cwd` (current working directory) to `"/"` and the file descriptors 0, 1 and 2 to standard input, standard output and standard error output—all of that will be discussed in Chapter 12 where we introduce the Ulix filesystem.

It is important that we activate the new process' address space at this step, because right after that we will load the program binary of the `init` program into the lower memory of the process, and that would be unavailable in the kernel's address space.

[190a] ⟨*start program from disk: prepare address space and TCB entry* 190a⟩≡ (189)

```
// create new address space (64 KB + 4 KB stack) and register TCB entry
addr_space_id as   = create_new_address_space (64*1024, 4096);
thread_id     tid  = register_new_tcb (as);     // get a fresh TCB
TCB           *tcb = &thread_table[tid];

// fill TCB structure
tcb->tid = tcb->pid = tid;                      // identical thread/process ID
tcb->ppid = 0;                                  // parent: 0 (none)
tcb->terminal = 0;                              // default terminal: 0
memcpy (tcb->cwd,     "/",   2);               // set current directory
memcpy (tcb->cmdline, "new", 4);               // set temporary command line
thread_table[tid].files[0] = DEV_STDIN;         // initialize standard I/O
thread_table[tid].files[1] = DEV_STDOUT;        // file descriptors
thread_table[tid].files[2] = DEV_STDERR;
for (int i = 3;  i < MAX_PFD;  i++) tcb->files[i] = -1;
activate_address_space (as);                    // activate the new address space
```

Uses `activate_address_space` 170c, `addr_space_id` 158b, `create_new_address_space` 163c, `cwd`, `DEV_STDERR` 415c, `DEV_STDIN` 415c, `DEV_STDOUT` 415c, `MAX_PFD` 424b, `memcpy` 596c, `register_new_tcb` 188d, `TCB` 175, `thread_id` 178a, and `thread_table` 176b.

6.3.2 Loading the Program

Loading the `init` program is simple because we do not use a special file format for the file, but instead link it into a "flat binary", similar to the historical `.COM` files that MS-DOS and CP/M used [Vil96, p. 171–175, 182–189]. The loader assumes that the filesize is less than 32 KByte and simply reads the whole file (or its first 32 KByte) into the virtual memory location that starts at `BINARY_LOAD_ADDRESS`$_{159a}$. We have set that constant to `0x0` in Section 6.1.1.

[190b] ⟨*constants* 112a⟩+≡ (44a) ◁176a 200a▷

```
#define PROGSIZE 32768
```

Defines:
`PROGSIZE`, used in chunk 190c.

For reading the file it uses the Ulix virtual filesystem functions `u_open`$_{412c}$, `u_read`$_{414b}$ and `u_close`$_{418a}$ which act like the well-known POSIX functions `open`, `read` and `close`.

[190c] ⟨*start program from disk: load binary* 190c⟩≡ (189)

```
int fd = u_open (progname, O_RDONLY, 0);            // open the file
u_read (fd, (char*)BINARY_LOAD_ADDRESS, PROGSIZE);  // load to virtual address 0
u_close (fd);                                       // close the file
```

Uses `BINARY_LOAD_ADDRESS` 159a, `O_RDONLY` 460b, `PROGSIZE` 190b, `u_close` 418a, `u_open` 412c, and `u_read` 414b.

The function `start_program_from_disk`$_{189}$ is called with the argument `"/init"`, so we need an `init` binary in the root directory of the root disk. That program does not do much, but only starts the `login` program via the `execv`$_{235e}$ function. `/init`

⟨*lib-build/init.c* 191a⟩≡ [191a]
```
#include "ulixlib.h"
void umain() {
  char *args[] = { "/bin/login", 0 };
  execv (args[0], args);
  printf ("exec failed\n"); for (;;);
}
```
Uses execv 235e, login 584c, and printf 601a.

For compiling this flat binary, we need a special linker configuration file that lets the GNU linker `ld` create such a format:

⟨*lib-build/process.ld* 191b⟩≡ [191b]
```
OUTPUT_FORMAT("binary")
phys = 0x00000000;
virt = 0x00000000;
SECTIONS {
  . = phys;

  .setup : AT(phys) {
    *(.setup)
  }

  .text : AT(code - virt) {
    code = .;
    *(.text)
    *(.rodata*)
    . = ALIGN(4096);
  }

  .data : AT(data - virt) {
    data = .;
    *(.data)
    . = ALIGN(4096);
  }

  .bss : AT(bss - virt) {
    bss = .;
    *(COMMON*)
    *(.bss*)
    . = ALIGN(4096);
  }
  end = .;
}
```

In the makefile for the user mode files (`lib-tools/Makefile`) the `init` program will later be compiled and linked with

```
compile:
        $(CC) $(CCOPTIONS) -g -c ulixlib.c
        $(CC) $(CCOPTIONS) -c init.c
        # link it with linker script "process.ld"
        $(LD) $(LDOPTIONS) -T process.ld -o init init.o ulixlib.o
```

6.3.3 Creating the Kernel Stack

Next we need to provide a kernel stack for the process. So far, Ulix has used the initial kernel stack defined as `_sys_stack`$_{95a}$ in the assembler file `start.asm`, but as we explained earlier, we need a separate kernel stack for every process (or thread).

We have already defined the number of kernel stack pages, KERNEL_STACK_PAGES$_{169b}$, so now we simply register as many frames and write a mapping into the page table via as_map_page_to_frame$_{165b}$.

[192a] ⟨*start program from disk: create kernel stack* 192a⟩≡ (189) 192b▷
```
unsigned int framenos[KERNEL_STACK_PAGES];        // frame numbers of kernel stack
for (int i = 0;  i < KERNEL_STACK_PAGES;  i++) {  // pages
  framenos[i] = request_new_frame ();
  as_map_page_to_frame (current_as, 0xbffff - i, framenos[i]);
}
```

After that we need to store the information about the process kernel stack into two TCB fields esp0 and ebp.

[192b] ⟨*start program from disk: create kernel stack* 192a⟩+≡ (189) ◁192a
```
char *kstack = (char*) (TOP_OF_KERNEL_MODE_STACK-KERNEL_STACK_SIZE);
memaddress adr = (memaddress)kstack;              // one page for kernel stack

tcb->esp0 = (uint)kstack + KERNEL_STACK_SIZE;     // initialize top-of-stack and
tcb->ebp  = (uint)kstack + KERNEL_STACK_SIZE;     // ebp (base pointer) values
```
Uses KERNEL_STACK_SIZE 169b, kstack, memaddress 46c, and TOP_OF_KERNEL_MODE_STACK 159c.

6.3.4 Activating the New Process

Finally we can activate the process. We've completed all the required steps, and the program sits in the memory, waiting to be started. Let's declare the varible current_task$_{192c}$ (that always holds the ID of the currently executing process or thread)

[192c] ⟨*global variables* 92b⟩+≡ (44a) ◁187c 195▷
```
thread_id current_task;
```
Defines:
current_task, used in chunks 152, 187a, 192d, 206b, 209c, 212, 216b, 217b, 219c, 222, 224c, 234b, 255a, 260a, 277b, 279c, 290a, 324a, 328–30, 332b, 334b, 335b, 366, 369c, 371a, 412b, 416b, 424, 426b, 432e, 478b, 487a, 518d, 522e, 533b, 545b, 563–66, 577c, 580–82, 587d, and 588b.
Uses thread_id 178a.

and initialize it. We also add the init process to the ready queue and enable the scheduler. Then the last step is switching to user mode.

[192d] ⟨*start program from disk: activate the new process* 192d⟩≡ (189)
```
current_task = tid;                          // make this the current task
add_to_ready_queue (tid);                    // add process to ready queue
⟨enable scheduler 276a⟩
cpu_usermode (BINARY_LOAD_ADDRESS,
              TOP_OF_USER_MODE_STACK);       // jump to user mode
```
Uses add_to_ready_queue 184b, BINARY_LOAD_ADDRESS 159a, cpu_usermode 198, current_task 192c, and TOP_OF_USER_MODE_STACK 159b.

The cpu_usermode$_{198}$ function will be written in Assembler, we discuss it in detail in the following section.

6.3.5 Configuring the TSS Structure and Entering User Mode

The Intel processor provides no command that would let us switch to user mode explicitly, but there is a way for *returning to user mode* which requires that the stack is set up properly when executing an `iret` instruction. That is what normally happens when, for example, a process already runs in user mode and an interrupt forces a jump to the interrupt handler—the CPU modifies the stack so that when the handler executes `iret`, execution will continue in the process. To make the system switch back to ring 3, the stack contains (besides other values) values which will be loaded into the `CS` and `SS` segment registers (which tell the CPU what segments to use for code and stack).

iret

The segment registers always contain a value which is a multiple of 8, making them an offset for the GDT whose entries are eight bytes long. Thus, the three lowest bits of a segment register value are always 0. What we have not mentioned yet is that the CPU modifies the lowest two bits when it pushes the register value on the stack (on interrupt entry), and it reads them when it pops the registers back from the stack (during `iret`). These two bits are then interpreted as the privilege level to which the CPU shall switch (see Figure 6.6).

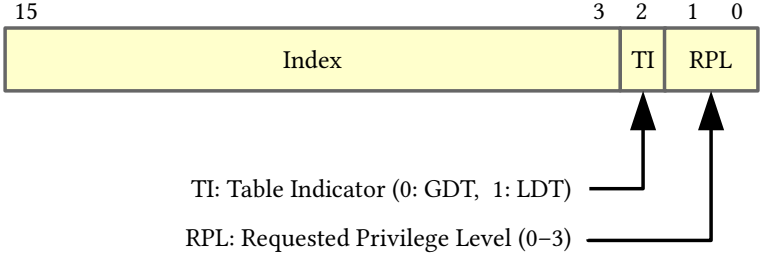


Figure 6.6: The Segment selector contains an index, a table indicator and the requested privilege level.

We can now force the system into user mode by generating a stack which looks just like the one that the CPU automatically generates when an interrupt occurs. Where the segment register contents are expected we push a value that we can calculate with `val = seg | 3;` (which will set the lowest two bits; $3 = 11_b$).

However, we cannot use the segment descriptors which we have created during the system initialization: both the code and the data segment descriptors have the entry `flags` (which contains a two-bit value *descriptor privilege level* (DPL), describing the necessary level for accessing this segment) set to 0—the system would halt, because it would switch to user mode but would not be allowed to use the memory. So we need two new segment descriptors which are designed specifically for user mode. They are identical to the old descriptors except for the `flags` entry where they have the DPL value set to 3 instead of 0.

DPL

So here's how we fill the descriptors:

[194a] ⟨*install GDTs for User Mode* 194a⟩≡ (110a) 196a▷
```
fill_gdt_entry (3, 0, 0xFFFFFFFF, 0b11111010, 0b1100);
fill_gdt_entry (4, 0, 0xFFFFFFFF, 0b11110010, 0b1100);
```
Uses fill_gdt_entry 109c.

The numbering continues with 3 since we've already filled the null descriptor (0) and the kernel mode code (1) and data (2) segment descriptors (see page 110).

For a better overview, we repeat the explanation of the old (kernel mode) GDT entries and add the two new entries:

- 10011010_b for the kernel code segment
  (present; ring 0; fixed-1; executable; exact privilege level; allow reading; not accessed)
- 10010010_b for the kernel data segment
  (present; ring 0; fixed-1; not executable; grow upwards; allow writing; not accessed)
- 11111010_b for the user mode code segment
  (present; ring 3; fixed-1; executable; exact privilege level; allow reading; not accessed)
- 11110010_b for the user mode data segment
  (present; ring 3; fixed-1; not executable; grow upwards; allow writing; not accessed)

TSS
TR, ltr

In order to enter user mode we also have to create a structure called *TSS* (*task state segment*) which is another (and final) entry in the GDT; we must load its GDT offset in a special *task register* (*TR*) using the `ltr` instruction.

The TSS is a 104 bytes long data structure [Int11, p. 303], shown in Figure 6.7. The CPU designers had intended that operating system developers would supply such a structure for each task (process or thread), and it is possible to simplify the task switch by following this suggestion. However, we decided to ignore this possibility and do the task switch without the help of the CPU because that is more instructional.

In our TSS type definition we only mention the elements which we may need and combine less interesting areas of the structure in `long long` elements (`u1`, `u2`, `u3`):

[194b] ⟨*type definitions* 91⟩+≡ (44a) ◁178b 227▷
```
typedef struct {
  unsigned int prev_tss    : 32;  // unused: previous TSS
  unsigned int esp0, ss0   : 32;  // ESP and SS to load when we switch to ring 0
  long long u1, u2         : 64;  // unused: esp1, ss1, esp2, ss2 for rings 1 and 2
  unsigned int cr3         : 32;  // unused: page directory
  unsigned int eip, eflags : 32;
  unsigned int eax, ecx, edx, ebx, esp, ebp, esi, edi, es, cs, ss, ds, fs, gs : 32;
                                  // unused (dynamic, filled by CPU)
  long long u3             : 64;  // unused: ldt, trap, iomap
} __attribute__((packed)) tss_entry_struct;
```
Defines:
tss_entry_struct, used in chunk 195.

Most of the fields are only useful when using the TSS to perform task switching: they store parts of the task context so that it is not necessary to keep track of them in the thread control block. If you are interested in this approach, you can read more about it in the Intel manual [Int11].

The `esp0` field must hold the address of the top of the kernel stack, and `ss0` must contain the segment number for kernel mode (`0x10`). The CPU will automatically set the stack pointer to that value when it switches from user mode to kernel mode (ring 0).

We use the thread control block entry to store and retrieve the process context. That is why we need only one TSS. (We cannot omit the TSS completely because the CPU demands that one exists.)

⟨ *global variables* 92b ⟩+≡ (44a) ◁192c 200b▷ [195]

```
tss_entry_struct tss_entry;
```

Defines:
`tss_entry`, used in chunk 197a.
Uses `tss_entry_struct` 194b.

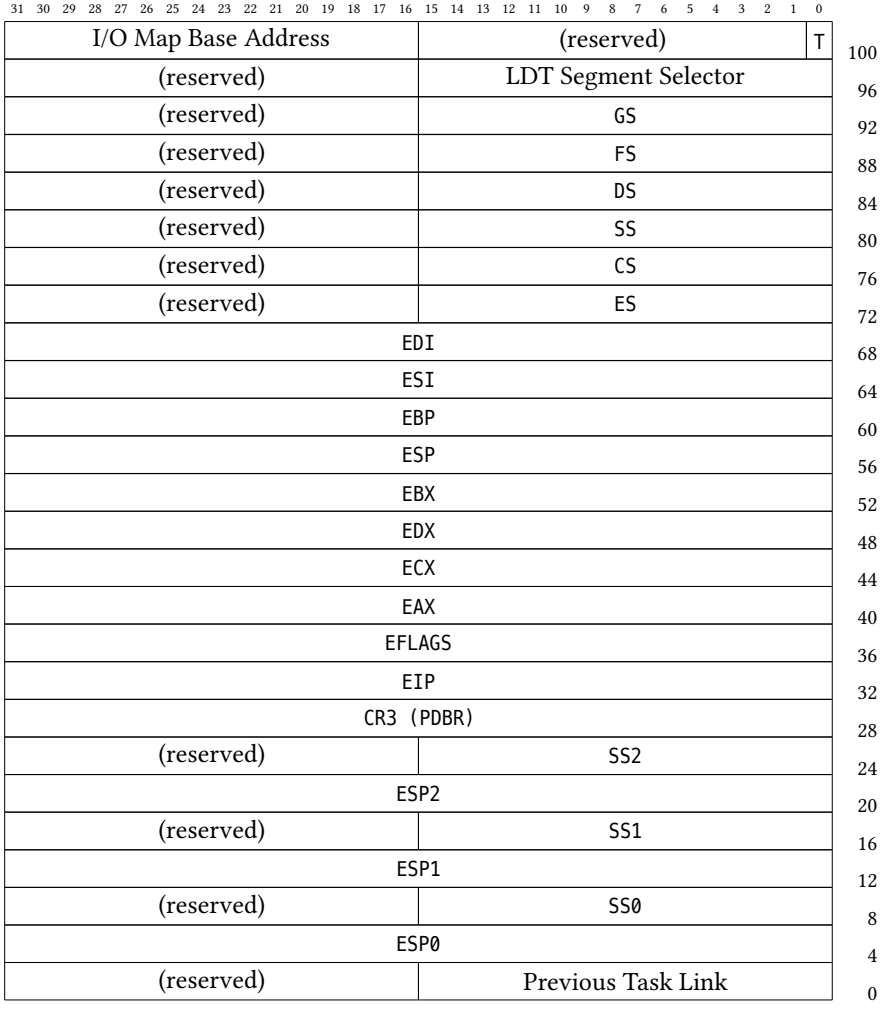


Figure 6.7: The TSS (Task State Segment) structure.

We add the data from `tss_entry`$_{195}$ to the GDT definition (see pages 110 and 193), this is the last GDT entry, and it uses number 5 since we've already used the entries 0–4.

[196a] ⟨*install GDTs for User Mode* 194a⟩+≡ (110a) ◁194a
```
write_tss (5, 0x10, (void*)TOP_OF_KERNEL_MODE_STACK);   // gdt no., ss0, esp0
```
Uses gdt 92b, TOP_OF_KERNEL_MODE_STACK 159c, and write_tss 197a.

Here's the prototype of `write_tss`$_{197a}$ which calls `fill_gdt_entry`$_{109c}$ to make the GDT entry point to `tss_entry`$_{195}$:

[196b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁188c 197d▷
```
static void write_tss (int num, word ss0, void *esp0);
```

Regular GDT entries store a base address and a limit to perform the address transformation from a logical to a linear address (which is then further translated by the paging mechanism). The TSS has a different purpose, but still gets stored in the same table. Here the base address is recycled so that it holds the address of our `tss_entry`$_{195}$ structure, and the limit field stores the size of the structure, minus 1. The GDT entry type is 0, and the required access value is `0xe9` = `11101001`$_b$.

As a reminder, Figure 6.8 shows the format of a regular segment descriptor at the top (this is the same as Figure 4.2); below, you see the slightly modified format of the TSS descriptor [Int11, p. 7-7]. `B` (bit 9 of the third word) can be 0 or 1, and we set it to 0, the value is only relevant when using several TSS structures.

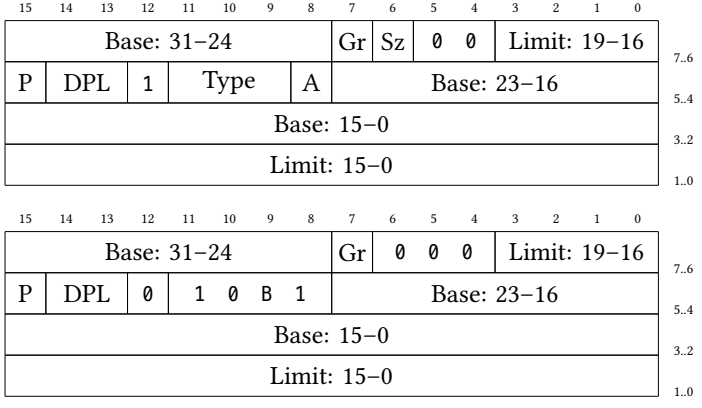


Figure 6.8: Segment descriptor (top) vs. TSS descriptor (bottom).

When we call `fill_gdt_entry`$_{109c}$, we have to set the bits 7–4 of the fourth word to `0000`$_b$ (in the last argument `gran` of the function call) and the bits 15–8 of the third word to `11101001`$_b$ (in the second last argument, `access`). This is the interpretation of the `access` bitmap:

- `11101001`$_b$ for the TSS descriptor
  (present; ring 3; fixed-0; TSS type (`1001`)).

With this information, we can implement write_tss$_{197a}$:

⟨*function implementations* 100b⟩+≡ (44a) ◁189 201b▷ [197a]
```
static void write_tss (int num, word ss0, void *esp0) {
   fill_gdt_entry (num, (uint) &tss_entry, sizeof (tss_entry) - 1,
                   0b11101001, 0b0000);               // write TSS entry to GDT
   memset (&tss_entry, 0, sizeof (tss_entry));        // fill TSS with zeros
   tss_entry.ss0  = ss0;                              // kernel stack segment
   tss_entry.esp0 = (memaddress)esp0;                 // kernel stack pointer
}
```
Defines:
write_tss, used in chunks 196, 197b, and 280a.
Uses fill_gdt_entry 109c, memaddress 46c, memset 596c, and tss_entry 195.

Thus, all five calls of fill_gdt_entry$_{109c}$ together look like this:

⟨*collection of* fill_gdt_entry *calls* 197b⟩≡ [197b]
```
//              no base      limit         access      gran
// ------------------------------------------------------------------------
fill_gdt_entry (0, 0,        0,            0,          0     );  // null descriptor
fill_gdt_entry (1, 0,        0xFFFFFFFF,   0b10011010, 0b1100);  // kernel, code
fill_gdt_entry (2, 0,        0xFFFFFFFF,   0b10010010, 0b1100);  // kernel, data
fill_gdt_entry (3, 0,        0xFFFFFFFF,   0b11111010, 0b1100);  // user, code
fill_gdt_entry (4, 0,        0xFFFFFFFF,   0b11110010, 0b1100);  // user, data
//   write_tss (5, 0x10, (void*)TOP_OF_KERNEL_MODE_STACK);   calls...
fill_gdt_entry (5, TSS_ADDR, TSS_SIZE - 1, 0b11101001, 0b0000);  // TSS descriptor
//   with TSS_ADDR = &tss_entry and TSS_SIZE = sizeof (tss_entry)
```

Finally, we add the code for loading the *task register* TR to the assembler file: the index of the TSS in the GDT is 5, so the proper value to load is $5 \times 8 = 40 =$ 0x28. We have to write the requested privilege level (RPL) into the two lowest bits:

$$\texttt{0x28} \,|\, \texttt{0x03} = \texttt{0x0b}$$

⟨*start.asm* 87⟩+≡ ◁150b 198▷ [197c]
```
[section .text]
            global tss_flush

tss_flush:  mov ax, 0x28 | 0x03
            ltr ax              ; load the task register
            ret
```
Defines:
tss_flush, used in chunks 110a, 197d, and 280a.

As always we need to tell the C compiler that the assembler function tss_flush$_{197c}$ exists elsewhere:

⟨*function prototypes* 45a⟩+≡ (44a) ◁196b 197e▷ [197d]
```
extern void tss_flush ();
```

Lastly, we present the cpu_usermode$_{198}$ routine which performs the switch from kernel mode (ring 0) to user mode (ring 3).

⟨*function prototypes* 45a⟩+≡ (44a) ◁197d 201a▷ [197e]
```
extern void cpu_usermode (memaddress address, memaddress stack);   // assembler
```

It prepares a stack that will create the first user mode process when executing `iret`. Since `iret` performs a change of privilege level (from ring 0 to ring 3), it will pop the following:

- instruction pointer `EIP`
- code segment selector `CS`
- `EFLAGS` register
- stack pointer `ESP`
- stack segment selector `SS`

The other segment selectors (`DS`, `ES`, `FS` and `GS`) can be set via `mov` instructions. When we enter the assembler function, there are three values on the stack:

- the return address in `[esp]` (in Assembler syntax, but note: we will never return to that address),
- the first argument in `[esp + 4]`,
- the second argument in `[esp + 8]`.

But each time we push data on the stack, the offsets will change. To make the following code more readable we will start with saving the current *ESP* value in *EBP*—that register will then point to the same address even while we push and pop data.

[198] ⟨*start.asm* 87⟩+≡ ◁197c 202c▷

```
                global cpu_usermode
cpu_usermode: cli                 ; disable interrupts
              mov  ebp, esp       ; remember current stack address
              mov  ax,  0x20 | 0x03 ; code selector 0x20 | RPL3: 0x03
                                  ; RPL = requested protection level
              mov  ds,  ax
              mov  es,  ax
              mov  fs,  ax
              mov  gs,  ax
              mov  eax, esp
              push 0x20 | 0x03    ; code selector 0x20 | RPL3: 0x03
              mov  eax, [ebp + 8] ; stack address is 2nd argument
              push eax            ; stack pointer
              pushf               ; EFLAGS
              pop  eax            ; trick: reenable interrupts when doing iret
              or   eax, 0x200
              push eax
              push 0x18 | 0x03    ; code selector 0x18 | RPL3: 0x03
              mov  eax, [ebp + 4] ; return address (1st argument) for iret
              push eax
              iret
```

Defines:
`cpu_usermode`, used in chunk 192d.

(There are many ways to implement this switch to user mode; the version that you see here originates from an `osdev.org` forum post by Jens Nyberg [Nyb11]. We added

comments and made some changes which make it easier to understand the code. The important point is to set up the stack properly for the final `iret` instruction.)

We're using a trick to have interrupts automatically enabled when we execute `iret`: One of the values on the stack is the *EFLAGS* register which contains the interrupt enable flag (*IF*) in bit 9. We cannot directly set that bit in *EFLAGS*, but we can modify the stack. The sequence `pop eax; or eax, 0x200; push eax` pops the *EFLAGS* (which was just pushed in the previous `pushf` instruction) from the stack, sets bit 9 ($2^9 = 512 = \texttt{0x200}$) and pushes the modified value onto the stack.

enabling interrupts

6.4 System Calls

When the operating system is in kernel mode, it has access to all its internal data and code: it may call any kernel function and, for example, read sectors from a disk or change hardware settings. Processes on the other hand cannot do the same: even though Ulix maps the kernel memory in all address spaces, processes cannot access it because the protection bits in the page tables define that this memory area may only be used when the system runs in ring 0—and processes run in ring 3 (user mode).

Even if a process was allowed to call kernel functions (by setting up the page tables differently) that would not help much since privileged machine instructions such as `in` and `out` (for talking to hardware devices) cannot be executed in ring 3.

All operating systems provide system calls as a way to access these needed kernel functions: on Ulix they allow a controlled switch from user mode to kernel mode via the `int` instruction which switches to ring 0 and executes a pre-defined interrupt handler. That handler finds out which system call the process wants to execute (by looking at the system call number that must be stored in the *EAX* register) and then proceeds by calling a system call handler function.

While we implement system calls, we will also create functions for the standard library that user mode programs must link in order to conveniently talk to the operating system via functions such as $\texttt{fork}_{213g}$, $\texttt{open}_{429b}$, $\texttt{read}_{429b}$ etc.

There are several ways to implement system calls. Let's first look at the way system calls can be called from user space. On 32-bit Intel CPUs, Linux does it via software interrupt `0x80` with arguments in registers:

⟨*example for system calls in linux* 199⟩≡ [199]

```
_start:                         ; tell linker entry point
      mov edx,len               ; message length
      mov ecx,msg               ; message to write
      mov ebx,1                 ; file descriptor (stdout)
      mov eax,4                 ; system call number (sys_write)
      int 0x80                  ; software interrupt 0x80
      mov eax,1                 ; system call number (sys_exit)
      int 0x80                  ; software interrupt 0x80

section .data
msg   db 'Hello, world!',0xa    ; the string to be printed
len   equ $ - msg               ; length of the string
```

(This example was taken from `http://asm.sourceforge.net/intro/hello.html`; the comments were modified.)

On a Linux machine you could assemble, link and run this file with

```
$ nasm -f elf test.asm
$ ld test.o -o test
$ ./test
Hello, world!
```

In this program *EAX* always holds the system call number, the other registers (in this example *EBX*, *ECX* and *EDX* are used for arguments. System call 4 is the `sys_write` syscall.

Other operating systems put arguments on the stack or into specific memory areas. We will stick with the Linux way because it is simple to use registers.

Since adding assembler code to C programs for every system call would be laborious, standard libraries make things simpler for the application developer; this can be done in two steps:

- Supplying a generic `syscall` function (that takes an arbitrary number of arguments) reduces the above code to executing

  ```
  char *msg = "Hello, world!\n";
  syscall (4, 1, msg, strlen (msg));
  ```

- But that is still unreadable, and also it is not portable because system call numbers are not identical across different Unix versions. Thus, for all standard system calls, some library provides the better known functions (such as `write`$_{429b}$) which allow the above code to be written as

  ```
  char *msg = "Hello, world!\n";
  write (STDOUT_FILENO, msg, strlen (msg));
  ```

  (with the constant `STDOUT_FILENO`$_{415b}$ set to 1).

6.4.1 System Calls in Ulix

Ulix provides functions for adding (or modifying) system calls to the system and a generic system call handler. For this purpose, we create a system call table `syscall_table`$_{200b}$ that contains pointers to functions, so for example, `syscall_table`$_{200b}$`[4]` should contain the address of Ulix's `sys_write` function. If a system call is not defined, the table entry is a null pointer, so we can initialize the whole table with null bytes:

[200a] ⟨*constants* 112a⟩+≡ (44a) ◁190b 233a▷

```
#define MAX_SYSCALLS 1024        // max syscall number: 1023
```

Defines:
`MAX_SYSCALLS`, used in chunks 200 and 201.

[200b] ⟨*global variables* 92b⟩+≡ (44a) ◁195 205c▷

```
void *syscall_table[MAX_SYSCALLS];
```

Defines:
`syscall_table`, used in chunk 201.
Uses `MAX_SYSCALLS` 200a.

Telling ULIX what function to execute when a specific system call is made is as simple as writing the address into the proper array entry. Nevertheless, we provide a function

⟨*function prototypes* 45a⟩+≡ (44a) ◁197e 202a▷ [201a]
```
void install_syscall_handler (int syscallno, void *syscall_handler);
```

which enters the handler address:

⟨*function implementations* 100b⟩+≡ (44a) ◁197a 201d▷ [201b]
```
void install_syscall_handler (int syscallno, void *syscall_handler) {
  if (syscallno ≥ 0 && syscallno < MAX_SYSCALLS)
    syscall_table[syscallno] = syscall_handler;
};
```
Defines:
`install_syscall_handler`, used in chunks 173d, 201c, 206f, 213e, 217c, 220–22, 224, 235c, 259a, 260c, 282d, 299b, 310c, 328e, 331b, 333a, 370e, 372b, 373a, 416c, 428a, 434a, 493d, 513b, 565a, 567a, 583b, 587e, 590c, and 611a.
Uses `MAX_SYSCALLS` 200a, `syscall_handler` 201d, and `syscall_table` 200b.

So if we have already defined a function `sys_write` and declared the system call number `__NR_write`$_{204c}$, we could activate the `write`$_{429b}$ system call by calling

⟨*syscall entry example* 201c⟩≡ [201c]
```
install_syscall_handler (__NR_write, sys_write);
```

The actual system call handler simply checks if there is a handler for the given system call number and (if so) calls it:

⟨*function implementations* 100b⟩+≡ (44a) ◁201b 202b▷ [201d]
```
void syscall_handler (context_t *r) {
  void (*handler) (context_t*);   // handler is a function pointer
  int number = r->eax;
  if (number != __NR_get_errno) set_errno (0); // default: no error
  if (number ≥ 0 && number < MAX_SYSCALLS)
    handler = syscall_table[number];
  else
    handler = 0;   // illegal system call number, outside 0..1023
  if (handler != 0) {
    handler (r);
  }
  else
    printf ("Unknown syscall no. eax=0x%x; ebx=0x%x. eip=0x%x, esp=0x%x. "
            "Continuing.\n", r->eax, r->ebx, r->eip, r->esp);
}
```
Defines:
`syscall_handler`, used in chunks 201 and 202c.
Uses `__NR_get_errno` 206e, `context_t` 142a, `MAX_SYSCALLS` 200a, `printf` 601a, `set_errno` 206b, and `syscall_table` 200b.

The `set_errno`$_{206b}$ function sets the `error` field of the current TCB and can be used by system call handlers to return an error code (see Section 6.4.3). We will later add system call handlers to a special code chunk named ⟨*syscall functions* 174b⟩ and put their prototypes in ⟨*syscall prototypes* 173b⟩.

[202a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁201a 202d▷
⟨*syscall prototypes* 173b⟩

[202b] ⟨*function implementations* 100b⟩+≡ (44a) ◁201d 206b▷
⟨*syscall functions* 174b⟩

We add a handler for interrupt `0x80` which looks just like our regular interrupt handlers for hardware-generated interrupts (and also like the fault handlers). The difference is that in this case we call neither `irq_handler`$_{146a}$ nor `fault_handler`$_{151c}$, but our new C function `syscall_handler`$_{201d}$. Apart from that we perform the same preparation as in the assembler code which you've already seen: We store the context in the proper order on the stack so that `syscall_handler`$_{201d}$ which takes a `context_t`$_{142a}$ `*r` as argument can evaluate and possibly change them.

[202c] ⟨*start.asm* 87⟩+≡ ◁198 213b▷

```
[section .text]
          extern syscall_handler
          global syscallh

syscallh: push byte 0       ; put 128 on the stack so it looks the same
          ; push byte 128   ; as it does after a hardware interrupt
          push byte -128    ; (getting rid of nasm error for signed byte)
          ⟨push registers onto the stack 142b⟩
          call syscall_handler
          ⟨pop registers from the stack 143a⟩
          add  esp, 8       ; undo the two "push byte" commands from the start_
          iret
```

Defines:
`syscallh`, used in chunk 202.
Uses `syscall_handler` 201d.

(In case you have forgotten it: ⟨*push registers onto the stack* 142b⟩ pushes the general purpose registers as well as *DS*, *ES*, *FS*, *GS* and *ESP* onto the stack while ⟨*pop registers from the stack* 143a⟩ pops them back in reverse order. We used this code when we introduced the interrupt handlers.)

In order to have the system jump to the `syscallh`$_{202c}$ Assembler function, we need to register its address in the interrupt descriptor table (just like we did with the interrupt and fault handlers):

[202d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁202a 206a▷

```
extern void syscallh ();
```

[202e] ⟨*install the fault handlers* 148b⟩+≡ (45b) ◁148b

```
fill_idt_entry (128,
                (unsigned int)syscallh,
                0x08,
                0b1110,   // flags: 1 (present), 11 (DPL 3), 0
                0b1110);  // type: 1110 (32 bit interrupt gate)
```

Uses `fill_idt_entry` 138c and `syscallh` 202c.

Note that we create an interrupt gate like in ⟨*install the interrupt handlers* 139b⟩ (p. 139) and ⟨*install the fault handlers* 148b⟩ (p. 148) and not a *trap gate*, so interrupts will be off when we enter a system call handler. For an *interruptible kernel* version of Ulix (see the discussion in Chapter 11.6) we would use a trap gate so that interrupts remain enabled. trap gate

6.4.2 Making System Calls

Actually making a system call works just like in the Linux example we've shown earlier:

- load the system call number in *EAX*,
- load arguments for the syscall in the next registers (*EBX*, *ECX*, ...) and
- execute `int 0x80`.

The return value of the system call can then be read from *EAX*. The following functions

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁174c 213f▷ [203a]
```
inline int syscall4 (int eax, int ebx, int ecx, int edx);
inline int syscall3 (int eax, int ebx, int ecx);
inline int syscall2 (int eax, int ebx);
inline int syscall1 (int eax);
```

standardize this process. We do not need them in the kernel, but the user mode library uses them to provide standard functions such as `open`$_{429b}$, `read`$_{429b}$, `write`$_{429b}$ or `fork`$_{213g}$:

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁174d 203c▷ [203b]
```
inline int syscall4 (int eax, int ebx, int ecx, int edx) {
  int result;
  asm ( "int $0x80" : "=a" (result) : "a" (eax), "b" (ebx), "c" (ecx), "d" (edx) );
  return result;
}
```
Defines:
`syscall4`, used in chunks 203a, 220d, 429b, and 591b.

The `asm` statement loads the *EAX* (`"a"`), *EBX* (`"b"`), *ECX* (`"c"`) and *EDX* (`"d"`) registers with the supplied values (`eax`, `ebx`, `ecx`, `edx`), then executes the instruction (`int $0x80`) and finally writes back the contents of *EAX* (`"a"`), which may have been modified, to `result`. For more information about this syntax see Appendix B.

The other functions work identically, just with less parameters which are stored in less registers:

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁203b 213g▷ [203c]
```
inline int syscall3 (int eax, int ebx, int ecx) {
  int result;
  asm ( "int $0x80" : "=a" (result) : "a" (eax), "b" (ebx), "c" (ecx) );
  return result;
}

inline int syscall2 (int eax, int ebx) {
  int result;
  asm ( "int $0x80" : "=a" (result) : "a" (eax), "b" (ebx) );
  return result;
}
```

```
inline int syscall1 (int eax) {
  int result;
  asm ( "int $0x80" : "=a" (result) : "a" (eax) );
  return result;
}
```

Defines:
syscall1, used in chunks 207b, 213g, 221f, 223b, 260e, 282f, 310e, 331d, and 513d.
syscall2, used in chunks 174d, 218a, 224f, 259c, 299d, 328g, 333c, 373e, 429b, 434c, 493f, 584c, and 587b.
syscall3, used in chunks 224f, 235e, 331d, 373e, 429b, 434c, 568b, 584c, and 591b.

System calls differ in the number of arguments. Since C provides no internal commands for accessing CPU registers and issuing `int` calls, we need inline assembler code.

As an example look at the $\texttt{write}_{429b}$ function which has the prototype

[204a] ⟨*example:* `write()` *prototype* 204a⟩≡
```
int write (int fd, const void *buf, int nbyte);
```

It takes three arguments, thus an implementation in a user mode library would look like this:

[204b] ⟨*example:* `write()` *implementation* 204b⟩≡
```
int write (int fd, const void *buf, int nbyte) {
  return syscall4 (__NR_write, fd, (int)buf, nbyte);
}
```

For increased Linux compatibility we will use the same system call numbers as Linux does—at least for those calls that Ulix does also provide.

The following definitions were taken from the 32-bit Linux[1] file `/usr/include/i386-linux-gnu/asm/unistd_32.h`:

[204c] ⟨*linux system calls* 204c⟩≡
```
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
#define __NR_close         6
#define __NR_waitpid       7
#define __NR_link          9
#define __NR_unlink       10
#define __NR_execve       11
#define __NR_chdir        12
#define __NR_chmod        15
#define __NR_lseek        19
#define __NR_getpid       20
#define __NR_sync         36
#define __NR_kill         37
#define __NR_mkdir        39
#define __NR_rmdir        40
#define __NR_brk          45
```

```
#define __NR_signal       48
#define __NR_dup2         63
#define __NR_getppid      64
#define __NR_symlink      83
#define __NR_readlink     85
#define __NR_readdir      89
#define __NR_truncate     92
#define __NR_ftruncate    93
#define __NR_stat        106
#define __NR_chown       182
#define __NR_getcwd      183
#define __NR_setreuid32  203
#define __NR_setregid32  204
#define __NR_setuid32    213
#define __NR_setgid32    214
```
(205a)

Defines:
__NR_brk, used in chunks 173d and 174d.
__NR_chdir, used in chunk 434.
__NR_chmod, used in chunks 590c and 591b.
__NR_chown, used in chunks 590c and 591b.
__NR_close, used in chunks 428a and 429b.

[1] Ubuntu 11.10, http://www.ubuntu.com/

__NR_execve, used in chunk 235.
__NR_exit, used in chunks 217c and 218a.
__NR_fork, used in chunk 213.
__NR_getcwd, used in chunk 434.
__NR_getpid, used in chunks 222e and 223b.
__NR_getppid, used in chunks 222e and 223b.
__NR_kill, used in chunks 565a and 568b.
__NR_link, used in chunks 428a and 429b.
__NR_lseek, used in chunks 428a and 429b.
__NR_mkdir, used in chunks 428a and 429b.
__NR_open, used in chunks 428a and 429b.
__NR_read, used in chunks 428a and 429b.
__NR_readdir, used in chunks 428a and 429b.
__NR_readlink, used in chunks 428a and 429b.
__NR_rmdir, used in chunks 428a and 429b.
__NR_setgid32, used in chunks 583b and 584c.
__NR_setregid32, used in chunks 583b and 584c.
__NR_setreuid32, used in chunks 583b and 584c.
__NR_setuid32, used in chunks 583b and 584c.
__NR_signal, used in chunks 567a and 568b.
__NR_stat, used in chunks 428a and 429b.
__NR_symlink, used in chunks 428a and 429b.
__NR_sync, used in chunk 513.
__NR_truncate, used in chunks 428a and 429b.
__NR_unlink, used in chunks 428a and 429b.
__NR_waitpid, used in chunk 220.
__NR_write, used in chunks 428a and 429b.
Uses __NR_ftruncate.

⟨*public constants* 46a⟩+≡ (44a 48a) ◁180a 207a▷ [205a]
⟨*linux system calls* 204c⟩
⟨*ulix system calls* 206e⟩

As we already mentioned, system calls return arguments by storing the value in *EAX*.

Now that you have seen how system calls are implemented you might want to turn back to Chapter 6.1.5 (specifically: to the implementation of the sbrk$_{174d}$ system call and library function on page 173) because we have already used the system call interface in that code and promised you a reminder once you'd get here.

6.4.3 Handling Errors with errno

Most system calls can fail: in that case they need to notify the calling process about the cause of the error. Unix systems traditionally use a special global variable named errno$_{207b}$ for this purpose; the standard behavior is to make the system call return -1 and put a specific (positive) value into errno$_{207b}$.

For Ulix we will provide the error code via a system call (and a corresponding user mode library function) called get_errno$_{206b}$(). For entering an error code into the process' TCB structure, we add the system call and function set_errno$_{206b}$(). Every user mode application must include the Ulix standard headers which will contain a macro that defines errno$_{207b}$ as the result of a system call which executes get_errno$_{206b}$. All attempts to read errno$_{207b}$ will generate that system call which reads the error field of the TCB. We haven't defined it yet, so here it is:

⟨*more TCB entries* 158c⟩+≡ (175) ◁187b 219a▷ [205b]
```
int error;
```

(In the TCB we use the name error instead of errno$_{207b}$ so that we can avoid confusion about which is which.)

We also declare a variable startup_errno$_{205c}$ which will be used in the early phase of the kernel initialization before the first process is started:

⟨*global variables* 92b⟩+≡ (44a) ◁200b 218b▷ [205c]
```
int startup_errno = 0;
```
Defines:
startup_errno, used in chunk 206b.

Inside the kernel the two functions are easy to implement:

[206a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁202d 209a▷
```
int get_errno ();
void set_errno (int err);
```

[206b] ⟨*function implementations* 100b⟩+≡ (44a) ◁202b 209c▷
```
int get_errno () {
  if (scheduler_is_active)  return thread_table[current_task].error;
  else                      return startup_errno;
}

void set_errno (int err) {
  if (scheduler_is_active)  thread_table[current_task].error = err;
  else                      startup_errno = err;
}
```
Defines:
get_errno, used in chunk 206d.
set_errno, used in chunks 201d, 206, 207c, 562b, 565c, 576, 577, and 579c.
Uses current_task 192c, scheduler_is_active 276e, startup_errno 205c, and thread_table 176b.

Now we need to turn these two functions into system calls. The system call handlers simply call the above functions; an argument (for $\texttt{set\_errno}_{206b}()$) can be found in the *EBX* register, and we store a return value (for $\texttt{get\_errno}_{206b}()$) in the *EAX* register. Both are available via the $\texttt{context\_t}_{142a}$ structure which is provided as an argument to the system call handlers:

[206c] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁173b 213c▷
```
void syscall_get_errno (context_t *r);
void syscall_set_errno (context_t *r);
```

[206d] ⟨*syscall functions* 174b⟩+≡ (202b) ◁174b 213d▷
```
void syscall_get_errno (context_t *r) { eax_return ( get_errno () ); };
void syscall_set_errno (context_t *r) { set_errno ((int)r->ebx); };
```
Defines:
syscall_get_errno, used in chunk 206f.
syscall_set_errno, used in chunk 206.
Uses context_t 142a, eax_return 174a, get_errno 206b, and set_errno 206b.

Finally we need to register the system calls:

[206e] ⟨*ulix system calls* 206e⟩≡ (205a) 221b▷
```
#define __NR_get_errno 501
#define __NR_set_errno 502
```
Defines:
__NR_get_errno, used in chunks 201d, 206f, and 207b.
__NR_set_errno, used in chunk 206f.

[206f] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁173d 213e▷
```
install_syscall_handler (__NR_get_errno, syscall_get_errno);
install_syscall_handler (__NR_set_errno, syscall_set_errno);
```
Uses __NR_get_errno 206e, __NR_set_errno 206e, install_syscall_handler 201b, syscall_get_errno 206d, and syscall_set_errno 206d.

We'll collect error codes (such as EACCES$_{577a}$ which is the code for "permission denied") in a new ⟨*error constants* 370a⟩ chunk:

⟨*public constants* 46a⟩+≡ (44a 48a) ◁205a 235a▷ [207a]
⟨*error constants* 370a⟩

and we will fill this collection with entries as we go along and opportunities for generating errors arise.

User mode programs can access the error code via the errno$_{207b}$ macro which just retrieves the value:

⟨*ulixlib constants* 207b⟩≡ (48a) [207b]
```
#define errno (syscall1(__NR_get_errno))
```
Uses __NR_get_errno 206e and syscall1 203c.

Note that most system calls do *not* set an error value because we wanted to keep the code compact. But it would be easy to change this: After all, the system call handlers *do* check for errors and simply return −1 when one occurs. By writing a macro

⟨*possible macro for readable error returns* 207c⟩≡ [207c]
```
#define err_return(retval,errno) \
  set_errno (errno); \
  return retval;
```

you could replace the `return(-1);` lines in the current code with `err_return (-1, ECODE);` lines.

6.5 Forking a Process

We're getting closer to having a multitasking operating system. We only use the function start_program_from_disk$_{189}$ for loading the first (initial) process—for everything else we want to implement the standard Unix way of creating new processes: the fork.

Figure 6.9 shows how a process and its fork proceed over time; the depiction resembles a (two-pronged) fork.

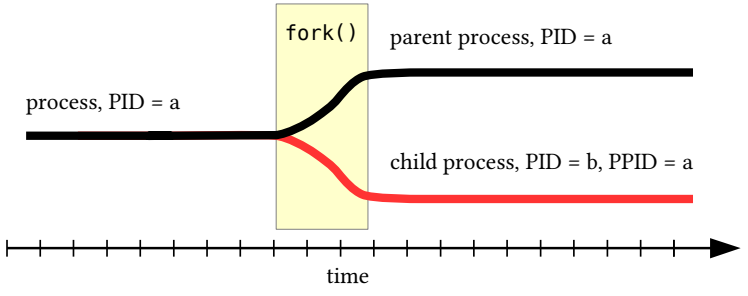


Figure 6.9: When the system forks a process, it creates an almost identical copy.

Here's an excerpt from the fork manpage on a Debian GNU/Linux 7.1 machine [Lin12a]:

```
NAME
       fork - create a child process

DESCRIPTION
       fork() creates a new process by duplicating the calling process.  The new pro-
       cess, referred to as the child, is an exact duplicate of the  calling process,
       referred to as the parent, except for the following points:

       *  The child has its own unique process ID, and this PID does not match the ID
          of any existing process group (setpgid(2)).

       *  The child's parent process ID is the same as the parent's process ID.
...

RETURN VALUE
       On success,  the PID of the child process is returned in the parent,  and 0 is
       returned in the child. On failure, -1 is returned in the parent, no child pro-
       cess is created, and errno is set appropriately.
...
```

Basically, when a Unix process calls `fork()` (and thus enters the `fork` system call), the operating system creates a duplicate of the currently running process. After a successful `fork` operation we have two processes which are almost identical. That means:

- Both processes execute the same program (i. e., the same binary is loaded in their lower memory areas),
- variables and dynamic memory have identical contents, but the memory is duplicated since both processes may make different changes to that memory once they continue running after the fork.
- They also have their own copies of the user mode and kernel mode stack.
- Most process metadata (the contents of the thread control block) are identical as well, with two important exceptions: the new process has its own process ID (and thread ID), and the new process stores the old process' thread ID in its *parent process ID* field (`ppid`).
- After the fork, both processes return from the `fork` system call and continue execution in the instruction immediately following the system call—so they need a way to find out whether they are the original process (called *parent*) or the newly forked process (called *child*). The user mode `fork` function will return 0 in the child process and the newly created process' ID in the parent process.

Parent Process
Child Process

Note that other Unix implementations do not copy the whole process memory—instead they use a technique called *copy-on-write* that only creates a copy of the page tables and marks them read-only (in both the parent and child process). This means that initially both processes use the same physical memory, but the read-only mode guarantees that no

copy-on-write

problems can occur. When a process tries to modify its memory, that will cause a page fault (due to the missing write permissions), and the fault handler will then create a copy of *that page* so that both processes have their personal copy of the faulting page. This copy (and the original) will have read and write permissions, and the faulting process can repeat its write operation. ULIX copies all of the memory which is less efficient, but allows a simpler implementation.

While we create a new process we will set its state to `TSTATE_FORK`$_{180a}$ to show that its creation is still in progress.

Our goal for this section is to implement the function

⟨ *function prototypes* 45a⟩+≡ (44a) ◁206a 213a▷ [209a]
```
int u_fork (context_t *r);
```

which will later be called from the `fork` system call handler (see page 213).

Since we will need a lot of memory copying operations, we declare two macros which let us copy physical memory areas (`phys_memcpy`$_{209b}$) and copy page frames (`copy_frame`$_{209b}$):

⟨ *macro definitions* 35a⟩+≡ (44a) ◁174a 222c▷ [209b]
```
#define phys_memcpy(target, source, size) \
  (unsigned int)memcpy ( (void*)PHYSICAL(target), (void*)PHYSICAL(source), size)
#define copy_frame(out, in)  phys_memcpy (out << 12, in << 12, PAGE_SIZE)
```
Defines:
`copy_frame`, used in chunk 211c.
`phys_memcpy`, used in chunk 211b.
Uses `memcpy` 596c, `PAGE_SIZE` 112a, and `PHYSICAL` 116a.

So, `phys_memcpy`$_{209b}$ does the same as `memcpy`$_{596c}$ but expects its first two arguments to be physical addresses (instead of virtual ones), and `copy_frame`$_{209b}$ provides a shortcut for copying physical frames since for that task the number of bytes to copy is always `PAGE_SIZE`$_{112a}$.

Next comes the definition of `u_fork`$_{209c}$. This function will be called when the `fork` system call is executed. Again, we declare everything that is done in this function as a critcal section. If you ask, why there is no ⟨ *end critical section in kernel* 380b⟩ in this chunk, see ⟨`u_fork`*: branch parent and child* 212⟩.

⟨ *function implementations* 100b⟩+≡ (44a) ◁206b 217a▷ [209c]
```
int u_fork (context_t *r) {
  ⟨ begin critical section in kernel 380a⟩
  thread_id old_tid = current_task;
  thread_id ppid    = old_tid;
  ⟨ u_fork: create new address space and TCB 210a⟩
  ⟨ u_fork: fill new TCB 210b⟩
  ⟨ u_fork: create new kernel stack and copy the old one 211a⟩
  ⟨ u_fork: copy user mode memory 211c⟩
  ⟨ u_fork: branch parent and child 212⟩
}
```
Defines:
`u_fork`, used in chunks 188d, 209a, and 213d.
Uses `context_t` 142a, `current_task` 192c, and `thread_id` 178a.

Now, here's the actual implementation. We will present it in several steps and discuss what's happening.

6.5.1 Reserving Memory and a Fresh TCB

We start by creating a new address space and cloning the current TCB into a free TCB which we first have to search for.

This step is similar to the first step in `start_program_from_disk`$_{189}$, except that memory and stack size are not free parameters, but are copied from the parent process:

[210a] ⟨`u_fork`: *create new address space and TCB* 210a⟩≡ (209c)

```
addr_space_id old_as = current_as;
// clone kernel part of PD; reserve user part of memory
addr_space_id new_as = create_new_address_space (
  address_spaces[old_as].memend - address_spaces[old_as].memstart,
  address_spaces[old_as].stacksize );
if (new_as == -1) return -1;    // error: cannot create address space

thread_id new_tid = register_new_tcb (new_as);
if (new_tid == -1) return -1;   // error: cannot create TCB entry
```

Uses `addr_space_id` 158b, `address_spaces` 162b, `create_new_address_space` 163c, `current_as` 170b, `register_new_tcb` 188d, `TCB` 175, and `thread_id` 178a.

6.5.2 Filling the Child TCB

Basically the child is an almost identical copy of the parent, so we start with copying the parent TCB to the child TCB. However, we need to modify some values, for example the process, thread and parent process ID as well as the link to the address space. We also copy the open file descriptors, but this needs more work than just copying the information in the TCB; we will explain that in the filesystem chapter where we provide the code chunk ⟨`u_fork`: *copy the file descriptors* 425a⟩. The new process is set to state `TSTATE_FORK`$_{180a}$; it will only change to `TSTATE_READY`$_{180a}$ when the fork operation is complete.

[210b] ⟨`u_fork`: *fill new TCB* 210b⟩≡ (209c)

```
TCB *t_old = &thread_table[old_tid];   // prefer to use pointers
TCB *t_new = &thread_table[new_tid];
*t_new            = *t_old;            // copy the complete TCB
t_new->state      = TSTATE_FORK;
t_new->tid        = new_tid;
t_new->addr_space = new_as;
t_new->pid        = t_new->tid;        // new process; pid = tid
t_new->ppid       = old_tid;           // set parent process ID

// copy current registers to new thread, except EBX (= return value)
t_new->regs       = *r;

// copy current ESP, EBP
asm volatile ("mov %%esp, %0" : "=r"(t_new->esp0));  // get current ESP
asm volatile ("mov %%ebp, %0" : "=r"(t_new->ebp));   // get current EBP

⟨u_fork: copy the file descriptors 425a⟩       // see filesystem chapter
```

Uses `t_new` 276c, `t_old` 276c, `TCB` 175, `thread_table` 176b, and `TSTATE_FORK` 180a.

6.5.3 The Child's Kernel Stack

The child needs a fresh kernel stack, and that also requires a new page table in which we can enter the mappings of the kernel stack pages to physical page frames.

⟨u_fork: *create new kernel stack and copy the old one* 211a⟩≡ (209c) 211b▷ [211a]

```
page_table *stackpgtable = request_new_page ();
address_spaces[new_as].kstack_pt = (memaddress)stackpgtable;
memset (stackpgtable, 0, sizeof (page_table));
page_directory *tmp_pd = address_spaces[new_as].pd;
KMAPD ( &tmp_pd->ptds[767], mmu (0, (uint)stackpgtable) );

int i, j; // counters
for (i = 0;  i < KERNEL_STACK_PAGES;  i++)
  as_map_page_to_frame (new_as, 0xbffff - i, request_new_frame () );
```

Uses address_spaces 162b, as_map_page_to_frame 165b, KERNEL_STACK_PAGES 169b, KMAPD 103c, memaddress 46c, memset 596c, mmu 172a, page_directory 103d, page_table 101b, request_new_frame 118b, and request_new_page 120a.

We use the phys_memcpy$_{209b}$ macro for copying the frames of the parent's kernel stack to the child's kernel stack, we get those physical addresses from the mmu$_{172a}$ function, using new_as for the new page table and old_as for the old table. It is not possible to simply start with an empty stack (like we did when we created the first process) because the child process, once it is fully created, will be in the middle of executing the fork system call, so the stack must be there and have the same contents as in the parent.

⟨u_fork: *create new kernel stack and copy the old one* 211a⟩+≡ (209c) ◁211a [211b]

```
// copy the physical frames
memaddress base = TOP_OF_KERNEL_MODE_STACK - KERNEL_STACK_SIZE;
for (i = 0;  i < KERNEL_STACK_PAGES;  i++)
  phys_memcpy ( mmu (new_as, base + i*PAGE_SIZE),
                mmu (old_as, base + i*PAGE_SIZE), PAGE_SIZE );
```

Uses KERNEL_STACK_PAGES 169b, KERNEL_STACK_SIZE 169b, memaddress 46c, mmu 172a, PAGE_SIZE 112a, phys_memcpy 209b, and TOP_OF_KERNEL_MODE_STACK 159c.

Note that the frames that we request here (both via request_new_page$_{120a}$ for the page table and request_new_frame$_{118b}$ for the kernel stack pages) will be released when the process exits.

6.5.4 Copying the Process' User Mode Memory

Copying the user mode memory means copying the first 3 GByte except the kernel stack which we've done already. This requires a nested loop since for each present page directory entry we look at each present page table entry and then make a copy. We have to look at the first 767 page tables (the 768th table holds the entries for the kernel stack).

⟨u_fork: *copy user mode memory* 211c⟩≡ (209c) [211c]

```
// clone first 3 GB (minus last directory entry) of address space
page_directory *old_pd = address_spaces[old_as].pd;
page_directory *new_pd = address_spaces[new_as].pd;
```

```
page_table     *old_pt, *new_pt;
for (i = 0;  i < 767;  i++) {          // only 0..766, not 767 (= kstack)
  if (old_pd->ptds[i].present) {       // page table present?
    // walk through the entries of the page table
    old_pt = (page_table*)PHYSICAL (old_pd->ptds[i].frame_addr << 12);
    new_pt = (page_table*)PHYSICAL (new_pd->ptds[i].frame_addr << 12);
    for (j = 0;  j < 1024;  j++)
      if (old_pt->pds[j].present)      // page present?
        copy_frame ( new_pt->pds[j].frame_addr, old_pt->pds[j].frame_addr );
  };
};
```

Uses address_spaces 162b, copy_frame 209b, kstack, page_directory 103d, page_table 101b, and PHYSICAL 116a.

6.5.5 A Child Is Born

All the code you have seen so far is only executed in the original (parent) process. But at some point in time there will be both the parent and the child, and the question is where the child shall start execution. We make the branch right here, as the last step in $\texttt{u\_fork}_{209c}$.

We start with querying the current instruction pointer (*EIP*) via the $\texttt{get\_eip}_{213b}$ function. This function returns the address of the instruction after the $\texttt{get\_eip}_{213b}$ call (because it retrieves the return address from the stack, and that address is not the address of the call, but of the instruction where the $\texttt{u\_fork}_{209c}$ function continues after returning from $\texttt{get\_eip}_{213b}$). That next line of code is the first line that we want to be executed by both processes, thus we store the value in the eip field of the new process' TCB. That's the whole trick behind getting the new process to start running at the correct instruction.

The rest is administrative work: In the parent process we add the new process to the ready queue, re-enable the interrupts and return the new process' thread ID. In the child process we simply return 0. We can check whether we're in the parent or child by comparing $\texttt{current\_task}_{192c}$ with the ppid variable: The latter is identical in both processes, but the comparison only evaluates to true in the parent process.

[212] ⟨u_fork: *branch parent and child* 212⟩≡ (209c)

```
memaddress eip = get_eip ();        // get current EIP
// new process begins to live right here!
if (current_task == ppid) {
  // parent tasks
  t_new->eip  = eip;
  add_to_ready_queue (new_tid);
  ⟨end critical section in kernel 380b⟩  // must be done in parent
  return new_tid;                   // in parent, fork returns child's PID
} else {
  // child tasks
  return 0;                         // in child, fork returns 0
}
```

Uses add_to_ready_queue 184b, current_task 192c, get_eip 213b, memaddress 46c, and t_new 276c.

Since

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁209a 216c▷ [213a]
```
extern memaddress get_eip ();
```

performs its trick by looking at the stack, it must be implemented in the assembler file. We simply pop the return address from the stack (storing it in *EAX*) and push it back so that the stack is as before. The contents of *EAX* are always used as functions' return values, so we're done:

⟨ *start.asm* 87 ⟩+≡ ◁202c [213b]
```
            global get_eip

get_eip:    pop eax       ; top of stack contains return address
            push eax      ; write it back
            ret
```
Defines:
get_eip, used in chunk 212.

6.5.6 The fork System Call

We can now add the `fork` system call: As usual, `syscall_fork`$_{213d}$ calls `u_fork`$_{209c}$ and stores the return value in *EAX* using `eax_return`$_{174a}$:

⟨ *syscall prototypes* 173b ⟩+≡ (202a) ◁206c 216a▷ [213c]
```
void syscall_fork (context_t *r);
```

⟨ *syscall functions* 174b ⟩+≡ (202b) ◁206d 216b▷ [213d]
```
void syscall_fork (context_t *r) { eax_return ((unsigned int) u_fork (r)); }
```
Defines:
syscall_fork, used in chunk 213.
Uses context_t 142a, eax_return 174a, and u_fork 209c.

We add the system call handler to the list:

⟨ *initialize syscalls* 173d ⟩+≡ (44b) ◁206f 217c▷ [213e]
```
install_syscall_handler (__NR_fork, syscall_fork);
```
Uses __NR_fork 204c, install_syscall_handler 201b, and syscall_fork 213d.

And here is the user mode library function:

⟨ *ulixlib function prototypes* 174c ⟩+≡ (48a) ◁203a 217d▷ [213f]
```
int fork ();
```

⟨ *ulixlib function implementations* 174d ⟩+≡ (48b) ◁203c 218a▷ [213g]
```
int fork () { return syscall1 (__NR_fork); }
```
Defines:
fork, used in chunks 213f and 214.
Uses __NR_fork 204c and syscall1 203c.

6.5.7 Testing fork

The following test program creates a process tree by calling $\texttt{fork}_{213g}$ four times:

[214] ⟨*lib-build/tools/fork2.c* 214⟩≡

```
#include "../ulixlib.h"
int main () {
  printf ("Press Return to end.\n");
  int f1 = fork (); int f2 = fork (); int f3 = fork (); int f4 = fork ();
  int pid = getpid (); int ppid = getppid (); int tid = gettid ();
  printf ("[%2d]: pid = %2d, tid = %2d, ppid = %2d, forkrets = [%2d %2d %2d %2d]\n",
    pid, pid, tid, ppid, f1, f2, f3, f4);

  long long int j; for (j = 0; j < 9999999ul; j++) ;   // wait
  if (f1!=0 && f2!=0 && f3!=0 && f4!=0) {
    char s[80]; ureadline ((char*)s, 79, false);
  }
  exit (0);
}
```

Uses exit 218a, fork 213g, getpid 223b, getppid 223b, gettid 223b, main 44b, printf 601a, and ureadline 431.

When running it, we get the following output. Figure 6.10 shows the process tree that is created by the program.

```
esser@ulix[8]:/home/esser$ fork2
Press Return to end.
[11]: pid = 11, tid = 11, ppid = 10, forkrets = [ 0 13 14 15]
[13]: pid = 13, tid = 13, ppid = 11, forkrets = [ 0  0 17 18]
[14]: pid = 14, tid = 14, ppid = 11, forkrets = [ 0 13  0 19]
[15]: pid = 15, tid = 15, ppid = 11, forkrets = [ 0 13 14  0]
[16]: pid = 16, tid = 16, ppid = 12, forkrets = [11  0  0 20]
[17]: pid = 17, tid = 17, ppid = 13, forkrets = [ 0  0  0 21]
[18]: pid = 18, tid = 18, ppid = 13, forkrets = [ 0  0 17  0]
[19]: pid = 19, tid = 19, ppid = 14, forkrets = [ 0 13  0  0]
[20]: pid = 20, tid = 20, ppid = 16, forkrets = [11  0  0  0]
[21]: pid = 21, tid = 21, ppid = 17, forkrets = [ 0  0  0  0]
[10]: pid = 10, tid = 10, ppid =  8, forkrets = [11 12 22 23]
[12]: pid = 12, tid = 12, ppid = 10, forkrets = [11  0 16 24]
[22]: pid = 22, tid = 22, ppid = 10, forkrets = [11 12  0 25]
[23]: pid = 23, tid = 23, ppid = 10, forkrets = [11 12 22  0]
[24]: pid = 24, tid = 24, ppid = 12, forkrets = [11  0 16  0]
[25]: pid = 25, tid = 25, ppid = 22, forkrets = [11 12  0  0]
esser@ulix[4]:/home/esser$
```

You will see similar code when you reach Chapter 7 where we discuss the creation of threads. Some operating systems use one kernel function that can create both new processes and threads, for example the Linux kernel has a `clone` function that handles both types. We decided against that approach because it makes the function more complex as it often has to check what type of task it is creating.

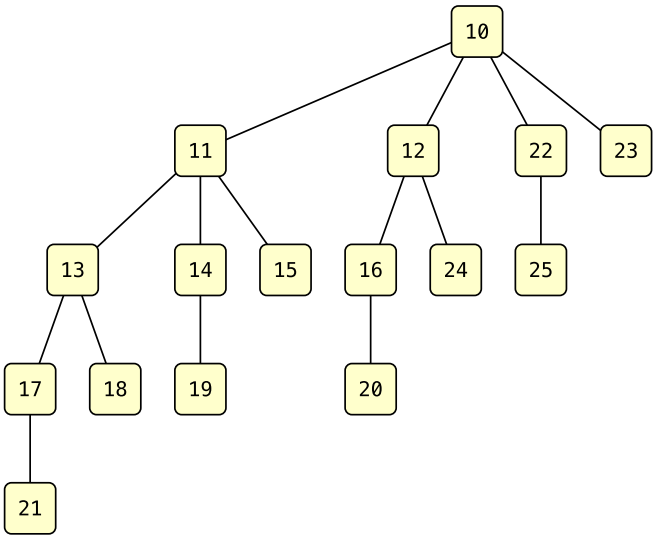


Figure 6.10: Calling `fork` four times creates this tree structure.

6.6 Exiting from a Process

In standard Unix implementations there are five ways to end the life of a process:

- The process calls `exit()` explicitly which makes it terminate immediately.
- The process executes `return` in the `main()` function or it reaches the end of that function. That will lead to an implicit call of `exit()` with the same result.
- The process receives a signal (from another process or from the kernel, see Chapter 14). If it has not installed a handler for this signal (or the signal cannot be intercepted), this causes the termination of the process (it aborts). In that case it cannot provide an exit value, instead there's an error code.
- Some kind of error occurs that causes a signal to be sent to the process (by the kernel). That is a special case of the one above.
- The process calls `abort()` which makes it send a `SIGABRT`$_{562a}$ signal to itself. The result is the same as when that signal is sent by a different process or by the kernel.

abort

In all of these cases the parent process can read the *exit status* and find out whether the process terminated normally or was aborted. The argument to `exit()` or `return` can also be used to tell the parent process whether the process finished successfully; traditionally an exit code of 0 means success, and any other value represents a problem that caused the process to (autonomously) terminate. It is not standard practice to use the exit code as some kind of return value; mainly because the exit code is typically restricted to the integer range of 0–255.

exit status

6.6.1 The exit System Call

ULIX provides an `exit` system call which terminates the process and stores the exit code (which is the single argument and available via *EBX*) in the TCB of the process.

[216a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁213c 219b▷
```
void syscall_exit (context_t *r);
```

It starts with disabling the interrupts and closing all open files of the process (this will only make sense after you've read the chapter about filesystems). Then it modifies the thread table: It removes the process from the ready queue and sets the process state to `TSTATE_EXIT`$_{180a}$. We cannot get rid of the TCB entry right now because the parent process must get a chance to read the exit code that we store in the `exitcode` field of the leaving process' TCB.

Finally, it wakes a waiting parent process, asks for destroyal of the address space (not all of that can happen at once, as we've already seen in Section 6.1.3), and updates the TCBs of any children it might have:

[216b] ⟨*syscall functions* 174b⟩+≡ (202b) ◁213d 219c▷
```
void syscall_exit (context_t *r) {
  // exit code is in ebx register:
  ⟨begin critical section in kernel 380a⟩  // access the thread table
  // close open files
  thread_id pid = thread_table[current_task].pid;
  int gfd;
  for (int pfd = 0;  pfd < MAX_PFD;  pfd++) {
    if ((gfd = thread_table[pid].files[pfd]) != -1)  u_close (gfd);
  }

  // modify thread table
  thread_table[current_task].exitcode = r->ebx;   // store exit code
  thread_table[current_task].state = TSTATE_EXIT; // mark process as finished
  remove_from_ready_queue (current_task);         // remove it from ready queue
  wake_waiting_parent_process (current_task);     // wake parent
  destroy_address_space (current_as);             // return the memory
  ⟨remove childrens link to parent 217b⟩          // notify children

  // finally: call scheduler to pick a different task
  ⟨end critical section in kernel 380b⟩
  scheduler (r, SCHED_SRC_RESIGN);
};
```
Defines:
`syscall_exit`, used in chunks 152b, 166c, 216a, 217c, and 260a.
Uses `context_t` 142a, `current_as` 170b, `current_task` 192c, `destroy_address_space` 166c, `MAX_PFD` 424b, `remove_from_ready_queue` 184c, `SCHED_SRC_RESIGN` 343a, `scheduler` 276d, `thread_id` 178a, `thread_table` 176b, `TSTATE_EXIT` 180a, `u_close` 418a, and `wake_waiting_parent_process` 217a.

We implement a function

[216c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁213a 228a▷
```
void wake_waiting_parent_process (int pid);
```

that checks whether the parent process is waiting for the current process to finish; we do not provide the code as a code chunk because it will also be used by the `u_kill`$_{562b}$ function which can terminate arbitrary processes.

If the parent is waiting, then it will be on the `waitpid_queue`$_{218b}$ queue. We can then transfer it to the ready queue by calling `deblock`$_{186b}$. If it is not waiting, we turn this process into a *zombie*: That means that the process will remain in the thread table. That way we give the parent process a chance to read the exit code, since once the TCB is gone, so is the exit code. zombie

⟨ *function implementations* 100b ⟩+≡ (44a) ◁209c 228b▷ [217a]

```
void wake_waiting_parent_process (int pid) {
  // check if we need to wake up parent process
  int ppid = thread_table[pid].ppid;
  if ( (thread_table[ppid].state == TSTATE_WAITFOR) &&
       (thread_table[ppid].waitfor == pid) ) {
    // wake up parent process
    deblock (ppid, &waitpid_queue);
    thread_table[pid].state = TSTATE_EXIT;
  } else {
    // parent is not waiting, make this process a zombie
    thread_table[pid].state = TSTATE_ZOMBIE;
  }
}
```

Defines:
`wake_waiting_parent_process`, used in chunks 216 and 564a.
Uses `deblock` 186b, `thread_table` 176b, `TSTATE_EXIT` 180a, `TSTATE_WAITFOR` 180a, `TSTATE_ZOMBIE` 180a, and `waitpid_queue` 218b.

We will remove zombie processes in the scheduler: It checks whether a zombie's parent has disappeared and (if so) deletes the zombie's TCB. You can see the code in the chunk ⟨ *scheduler: check for zombies* 281 ⟩.

We also need to inform children processes that their parent is gone. In that case we set their parent process ID `PPID` to 1 (the ID of the `init` process which becomes the `idle` process).

⟨ *remove childrens link to parent* 217b ⟩≡ (216b) [217b]

```
for (int pid = 0;  pid < MAX_THREADS;  pid++)
  if (thread_table[pid].ppid == current_task)
    thread_table[pid].ppid = 1;  // set parent to idle process
```

Uses `current_task` 192c, `MAX_THREADS` 176a, and `thread_table` 176b.

As usual, we need to enter the system call handler in the table and provide a user mode `exit`$_{218a}$ function that makes the right system call:

⟨ *initialize syscalls* 173d ⟩+≡ (44b) ◁213e 220b▷ [217c]

```
install_syscall_handler (__NR_exit, syscall_exit);
```

Uses `__NR_exit` 204c, `install_syscall_handler` 201b, and `syscall_exit` 216b.

⟨ *ulixlib function prototypes* 174c ⟩+≡ (48a) ◁213f 220c▷ [217d]

```
void exit (int exitcode);
```

[218a] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁213g 220d▷
```
void exit (int exitcode) { syscall2 (__NR_exit, exitcode); }
```
Defines:
exit, used in chunks 214, 217d, and 513e.
Uses __NR_exit 204c and syscall2 203c.

6.6.2 The waitpid System Call

Often a process wants to wait for the completion of a child process, a typical example is a shell which starts an external program by fork$_{213g}$ing, executing the program inside the child process and waiting in the parent process.

Here we implement the waitpid$_{220d}$ system call which waits for completion of a given child, the standard definition, taken from the Linux man pages, is the following:

```
pid_t waitpid (pid_t pid, int *status, int options);
```

In that prototype

- pid is the process ID of a child process (waitpid$_{220d}$ cannot be used to wait for termination of arbitrary, non-child processes),
- *status is the address of a status value which will be used to store the exit code of the child process (or an error value if the child was aborted),
- and options can be used to modify waitpid$_{220d}$'s behavior; our implementation will ignore any given options.

We need a blocked queue for processes that called waitpid$_{220d}$ since they must not be picked by the scheduler.

[218b] ⟨*global variables* 92b⟩+≡ (44a) ◁205c 276c▷
```
blocked_queue waitpid_queue;
```
Defines:
waitpid_queue, used in chunks 217–19, 281, 564c, and 606.
Uses blocked_queue 183a.

[218c] ⟨*initialize system* 45b⟩+≡ (44b) ◁116b 323e▷
```
initialize_blocked_queue (&waitpid_queue);
```
Uses initialize_blocked_queue 183c and waitpid_queue 218b.

Several things must be implemented for waitpid$_{220d}$ to work properly:

- We need the system call handler which moves the current (calling) process from the ready queue to the new waitpid_queue$_{218b}$ and calls resign$_{221f}$ (so that the scheduler picks a new process—the resign$_{221f}$ code will be shown right after waitpid$_{220d}$).
- When a process exits, it must store the exit argument in the thread control block—this TCB must remain intact until the parent process has had a chance to look up the value. (We've already shown you that part.)
- If the parent of an exiting process is in the waitpid_queue$_{218b}$ we move it back to the ready queue. (That is handled by wake_waiting_parent_process$_{217a}$, see above.)

- Once the parent process is picked by the scheduler, it will continue its execution of $\texttt{waitpid}_{220d}$ and has to read the child's exit code. After that $\texttt{waitpid}_{220d}$ it can delete the TCB entry.

As long as the parent process could not be reactivated, the child's TCB will remain intact. Note that it is not necessary for the parent process to actually look at the exitcode.

First we add `exitcode` and `waitfor` entries to the $\texttt{TCB}_{175}$ structure:

⟨ *more TCB entries* 158c ⟩+≡ (175) ◁205b 235b▷ [219a]

```
int exitcode;
int waitfor;    // pid of the child that this process waits for
```

The system call handler

⟨ *syscall prototypes* 173b ⟩+≡ (202a) ◁216a 220e▷ [219b]

```
void syscall_waitpid (context_t *r);
```

works as follows:

⟨ *syscall functions* 174b ⟩+≡ (202b) ◁216b 220a▷ [219c]

```
void syscall_waitpid (context_t *r) {
  // ebx: pid of child to wait for
  // ecx: pointer to status
  // edx: options (ignored)
  ⟨ begin critical section in kernel 380a ⟩
  int chpid = r->ebx;  // child we shall wait for

  // check errors
  if (chpid < 1 || chpid ≥ MAX_THREADS || thread_table[chpid].state == 0) {
    ⟨ end critical section in kernel 380b ⟩
    eax_return (-1);  // error
  }
  if (!thread_table[chpid].used) {
    ⟨ end critical section in kernel 380b ⟩
    eax_return (-1);  // no such process
  }
  if (thread_table[chpid].ppid != current_task) {
    ⟨ end critical section in kernel 380b ⟩
    eax_return (-1);  // not a child of mine
  }

  int *status = (int*)r->ecx;                    // address for the status
  thread_table[current_task].waitfor = chpid;
  block (&waitpid_queue, TSTATE_WAITFOR);
  ⟨ end critical section in kernel 380b ⟩
  syscall_resign (r);                            // here we resign
```

Defines:
`syscall_waitpid`, used in chunks 219b and 220b.
Uses `context_t` 142a, `current_task` 192c, `eax_return` 174a, `MAX_THREADS` 176a, `syscall_resign` 221a, `thread_table` 176b, `TSTATE_WAITFOR` 180a, and `waitpid_queue` 218b.

Calling `block` only moves the process to a different queue, but it does not stop its execution; for that purpose we must also call `syscall_resign`$_{221a}$.

When we return from `syscall_resign`$_{221a}$, the child must have finished. Unblocking this process happens in `syscall_exit`$_{216b}$`()`, here we expect to be woken up automatically.

The return value of `waitpid`$_{220d}$ is the process ID of the terminated child (`chpid`) or -1 in case of an error. Since `syscall_exit`$_{216b}$`()` has updated the `exitcode` field of the child's TCB, we can just read it.

[220a] ⟨*syscall functions* 174b⟩+≡ (202b) ◁219c 221a▷

```
    *status = thread_table[chpid].exitcode;
    thread_table[chpid].used = false;  // finally remove child process
    eax_return (chpid);                // set the return value
  }
```

Defines:
`syscall_waitpid`, used in chunks 219b and 220b.
Uses `eax_return` 174a and `thread_table` 176b.

As usual, we register the new system call:

[220b] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁217c 221c▷

```
install_syscall_handler (__NR_waitpid, syscall_waitpid);
```

Uses `__NR_waitpid` 204c, `install_syscall_handler` 201b, and `syscall_waitpid` 219c 220a.

Here is the user mode function:

[220c] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁217d 221e▷

```
int waitpid (int pid, int *status, int options);
```

[220d] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁218a 221f▷

```
int waitpid (int pid, int *status, int options) {
  return syscall4 (__NR_waitpid, pid, (uint)status, options);
}
```

Defines:
`waitpid`, used in chunks 180a, 220c, and 606.
Uses `__NR_waitpid` 204c and `syscall4` 203b.

6.6.3 Giving Up the CPU: The `resign` System Call

The `resign` system call allows a process to give up the CPU, so that the scheduler picks another process immediately.

We call the scheduler with a special argument `SCHED_SRC_RESIGN`$_{343a}$ which tells it that it was called from `syscall_resign`$_{221a}$ because we want to be able to detect how it was called. This will be explained in more detail in Chapter 8.

[220e] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁219b 222a▷

```
void syscall_resign (context_t *r);
```

⟨*syscall functions* 174b⟩+≡ (202b) ◁220a 222b▷ [221a]
```
void syscall_resign (context_t *r) {
  ⟨begin critical section in kernel 380a⟩
  scheduler (r, SCHED_SRC_RESIGN);
  ⟨end critical section in kernel 380b⟩
}
```
Defines:
syscall_resign, used in chunks 219–21.
Uses context_t 142a, SCHED_SRC_RESIGN 343a, and scheduler 276d.

We declare a syscall number for the resign system call

⟨*ulix system calls* 206e⟩+≡ (205a) ◁206e 222d▷ [221b]
```
#define __NR_resign       66
```
Defines:
__NR_resign, used in chunk 221.

and initialize the handler:

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁220b 222e▷ [221c]
```
install_syscall_handler (__NR_resign, syscall_resign);
```
Uses __NR_resign 221b, install_syscall_handler 201b, and syscall_resign 221a.

When we want to resign from inside kernel code, we simply use the following ⟨*resign* 221d⟩ code chunk which explicitly makes the system call:

⟨*resign* 221d⟩≡ (260a 361c 366a 391 392 416b 521b 531a 545b 563–65) [221d]
```
asm {
  mov eax, 66;   // System Call no. 66
  int 0x80;      // Make the System Call
}
```

and user mode processes can do the same by calling this resign$_{221f}$() function that we supply as part of the library:

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁220c 223a▷ [221e]
```
inline void resign ();
```

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁220d 223b▷ [221f]
```
inline void resign () { syscall1 (__NR_resign); }
```
Defines:
resign, used in chunk 221e.
Uses __NR_resign 221b and syscall1 203c.

6.7 Information about Processes

In this section we implement a few library functions which enable processes and threads to query their process and thread IDs, the parent process ID and information about the overall list of tasks (so that we can write a user mode ps program).

6.7.1 The gettid, getpid and getppid System Calls

Each TCB contains two IDs which describe a task: a thread ID `tid` (which is what the global variable `current_task`$_{192c}$ uses to point to the currently executing thread and which is identical to the index into the thread table) and also a process ID `pid`. Until now, thread and process IDs have always been identical, but when we introduce threads (as parts of a process) in the next chapter, we will arrive at a situation where these IDs differ. So we will provide three functions that retrieve the thread and process IDs (and also the parent process ID):

[222a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁220e 223d▷
```
void syscall_gettid (context_t *r);   // get thread ID
void syscall_getpid (context_t *r);   // get process ID
void syscall_getppid (context_t *r);  // get parent process ID
```

Getting the thread ID is simple, because the executing thread always has the thread ID stored in `current_task`$_{192c}$. For the process ID and the the parent process ID we need to access the TCB and fetch its `pid` or `ppid` entries, respectively.

[222b] ⟨*syscall functions* 174b⟩+≡ (202b) ◁221a 223e▷
```
void syscall_gettid (context_t *r)  { eax_return (current_task); }
void syscall_getpid (context_t *r)  { eax_return (current_pid); }
void syscall_getppid (context_t *r) { eax_return (current_ppid); }
```
Defines:
`syscall_getpid`, used in chunk 222e.
`syscall_getppid`, used in chunk 222e.
Uses `context_t` 142a, `current_pid` 222c, `current_ppid` 222c, `current_task` 192c, `eax_return` 174a, and `syscall_gettid`.

They use these two macros:

[222c] ⟨*macro definitions* 35a⟩+≡ (44a) ◁209b 279a▷
```
#define current_pid  (thread_table[current_task].pid)
#define current_ppid (thread_table[current_task].ppid)
```
Defines:
`current_pid`, used in chunk 222b.
`current_ppid`, used in chunk 222b.
Uses `current_task` 192c and `thread_table` 176b.

The system call numbers `__NR_getpid`$_{204c}$ and `__NR_getppid`$_{204c}$ have been defined earlier, they are standard numbers that you can also find on Linux systems. For `gettid` we need to define a number since that is no standard system call.

[222d] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁221b 223c▷
```
#define __NR_gettid  21
```
Uses `__NR_gettid`.

[222e] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁221c 224a▷
```
install_syscall_handler (__NR_gettid,  syscall_gettid);
install_syscall_handler (__NR_getpid,  syscall_getpid);
install_syscall_handler (__NR_getppid, syscall_getppid);
```
Uses `__NR_getpid` 204c, `__NR_getppid` 204c, `__NR_gettid`, `install_syscall_handler` 201b, `syscall_getpid` 222b, `syscall_getppid` 222b, and `syscall_gettid`.

The user mode $\texttt{getpid}_{223b}$, $\texttt{getppid}_{223b}$ and $\texttt{gettid}_{223b}$ functions

⟨ *ulixlib function prototypes* 174c⟩+≡ (48a) ◁221e 235d▷ [223a]
```
int gettid ();
int getpid ();
int getppid ();
```

simply make the appropriate system calls:

⟨ *ulixlib function implementations* 174d⟩+≡ (48b) ◁221f 224f▷ [223b]
```
int gettid () { return syscall1 (__NR_gettid); }
int getpid () { return syscall1 (__NR_getpid); }
int getppid () { return syscall1 (__NR_getppid); }
```
Defines:
getpid, used in chunks 214, 311b, 513e, and 568b.
getppid, used in chunk 214.
gettid, used in chunks 214 and 223a.
Uses __NR_getpid 204c, __NR_getppid 204c, __NR_gettid, and syscall1 203c.

Note that we have not implemented corresponding u_getpid, u_gettid and u_getppid functions in the kernel as we normally do; querying the current thread's IDs is too simple to justify extra functions for that purpose; if we need this information inside a kernel function, we can just use the macros $\texttt{current\_pid}_{222c}$ and $\texttt{current\_ppid}_{222c}$.

6.7.2 The getpsinfo and setpsname System Calls

The getpsinfo system call lets a process read its thread control block (the $\texttt{TCB}_{175}$ structure). That way, a non-privileged ps program can show the process list. It is not possible to modify a TCB, but the TCB may contain information that should be kept private. In a security-aware operating system the information must be filtered if some of the data are considered confidential.

⟨ *ulix system calls* 206e⟩+≡ (205a) ◁222d 224d▷ [223c]
```
#define __NR_getpsinfo 503
```
Defines:
__NR_getpsinfo, used in chunk 224.

⟨ *syscall prototypes* 173b⟩+≡ (202a) ◁222a 224b▷ [223d]
```
void syscall_getpsinfo (context_t *r);
```

⟨ *syscall functions* 174b⟩+≡ (202b) ◁222b 224c▷ [223e]
```
void syscall_getpsinfo (context_t *r) {
  unsigned int retval, pid;
  // ebx: thread ID
  // ecx: address of TCB block
  pid = r->ebx;
  if (pid > MAX_THREADS || pid < 1) {      // legal argument?
    retval = 0; goto end;
  }
  if (thread_table[pid].used == false) {   // do we have this thread?
    retval = 0; goto end;
```

```
    }

    // found a process: copy its TCB
    memcpy ((char*)r->ecx, &thread_table[pid], sizeof (TCB));
    retval = r->ecx;

    end: eax_return (retval);
  };
```

Defines:
  `syscall_getpsinfo`, used in chunks 223d and 224a.
Uses `context_t` 142a, `eax_return` 174a, `MAX_THREADS` 176a, `memcpy` 596c, `TCB` 175, and `thread_table` 176b.

[224a] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁222e 224e▷

```
  install_syscall_handler (__NR_getpsinfo, syscall_getpsinfo);
```

Uses `__NR_getpsinfo` 223c, `install_syscall_handler` 201b, and `syscall_getpsinfo` 223e.

We also allow processes to set their own name via the `setpsname` system call. In most cases this happens automatically (because `u_execv`$_{228b}$ writes the name into the appropriate field of the TCB entry, see below), but for some cases like the swapper daemon, we want to change the default name.

[224b] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁223d 234a▷

```
  void syscall_setpsname (context_t *r);
```

[224c] ⟨*syscall functions* 174b⟩+≡ (202b) ◁223e 234b▷

```
  void syscall_setpsname (context_t *r) {
    strncpy (thread_table[current_task].cmdline, (char*)r->ebx, CMDLINE_LENGTH-1);
  };
```

Defines:
  `syscall_setpsname`, used in chunk 224.
Uses `CMDLINE_LENGTH` 235a, `context_t` 142a, `current_task` 192c, `strncpy` 594b, and `thread_table` 176b.

[224d] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁223c 258c▷

```
  #define __NR_setpsname 504
```

Defines:
  `__NR_setpsname`, used in chunk 224.

[224e] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁224a 235c▷

```
  install_syscall_handler (__NR_setpsname, syscall_setpsname);
```

Uses `__NR_setpsname` 224d, `install_syscall_handler` 201b, and `syscall_setpsname` 224c.

These functions let user mode programs access the process list:

[224f] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁223b 235e▷

```
  uint getpsinfo (int pid, TCB* tcb) {
    return syscall3 (__NR_getpsinfo, pid, (uint)tcb);
  }

  uint setpsname (char *psname) {
    return syscall2 (__NR_setpsname, (uint)psname);
  }
```

Uses `__NR_getpsinfo` 223c, `__NR_setpsname` 224d, `syscall2` 203c, `syscall3` 203c, and `TCB` 175.

6.8 ELF Loader

In this section we look at Ulix's `execv`$_{235e}$ function which is able to load ELF binaries (Executable and Linking Format) [TIS95] from disk.[2] Classically, Unix systems provide several variants of exec functions (`execl`, `execle`, `execlp`, `execv`$_{235e}$, `execvp`, `execvpe` and `execve`) which differ in the way that arguments for the new program are provided. For the kernel one of these functions is sufficient, all other variants can be supplied by library functions which convert between the various syntaxes.

The standard procedure for launching an application on a Unix machine is to first `fork`$_{213g}$`()` the current process and then load a new program binary in the child process. That way, the parent process remains intact. (Note that non-Unix systems typically provide a different mechanism, for example Windows has a `CreateProcess` function which combines the creation of a new process and the loading of the program; it does not support `fork`$_{213g}$.)

6.8.1 ELF File Format

Let's look at a simple ELF binary that we create on a Linux machine. We use assembler code since that allows us to create a very compact binary:

⟨*example elf program* `test.asm` 225⟩≡ [225]

```
bits 32
global main

main:
  mov eax, 1
  mov ebx, 42
  int 0x80
```

Uses `main` 44b.

The equivalent C code would only contain `exit`$_{218a}$`(42)`: these assembler commands make a system call (with system call number 1 which is `__NR_exit`$_{204c}$) and the argument 42.

We can assemble this program with `nasm -f elf32 test.asm` which creates `test.o`; then we link it with `gcc test.o -nostdlib -e main`$_{44b}$ `-o test`, creating the binary `test`.

Let's check that this program works as expected and see what kind of information we can gather about it:

```
linux$ ./test ; echo $?
42
linux$ file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked,
BuildID[sha1]=0xa45ecc892186bae9977605e0c3d6757bdef2861b, not stripped
```

[2] Note that there is an alternative implementation of the Ulix ELF loader by Frank Kohlmann that he developed as part of his Bachelor's thesis [Koh13] which was supervised by Hans-Georg Eßer. It is available on the Ulix website and shows more details, however the language is German.

```
linux$ stat -c "%s" test    # filesize?
631
linux$ readelf -e test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80480a0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          224 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 3

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            08048074 000074 000024 00   A  0   0  4
  [ 2] .text             PROGBITS        080480a0 0000a0 00000c 00  AX  0   0 16
  [ 3] .shstrtab         STRTAB          00000000 0000ac 000034 00      0   0  1
  [ 4] .symtab           SYMTAB          00000000 0001d0 000080 10      5   4  4
  [ 5] .strtab           STRTAB          00000000 000250 000027 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x000ac 0x000ac R E 0x1000
  NOTE           0x000074 0x08048074 0x08048074 0x00024 0x00024 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .text
   01     .note.gnu.build-id
```

The ELF format and the `readelf` tool (which is available via the `binutils` package on Linux distributions) are discussed in detail in the book "Professional Linux Kernel Architecture" [Mau08, p. 1241 ff.]. `readelf`

In order to read ELF files we need to understand the two kinds of headers which they contain, the ELF header ($\texttt{Elf32\_Ehdr}_{227}$) and the ELF program header ($\texttt{Elf32\_Phdr}_{227}$). We have copied the following type definitions from the Linux header file `/usr/include/elf.h`. ELF headers

⟨ *type definitions* 91 ⟩+≡ (44a) ◁194b 292a▷ [227]

```
typedef uint16_t Elf32_Half;
typedef uint32_t Elf32_Word;
typedef uint32_t Elf32_Addr;
typedef uint32_t Elf32_Off;

typedef struct {
  byte          e_ident[16];          // Magic number and other info
  Elf32_Half    e_type;               // Object file type
  Elf32_Half    e_machine;            // Architecture
  Elf32_Word    e_version;            // Object file version
  Elf32_Addr    e_entry;              // Entry point virtual address
  Elf32_Off     e_phoff;              // Program header table file offset
  Elf32_Off     e_shoff;              // Section header table file offset
  Elf32_Word    e_flags;              // Processor-specific flags
  Elf32_Half    e_ehsize;             // ELF header size in bytes
  Elf32_Half    e_phentsize;          // Program header table entry size
  Elf32_Half    e_phnum;              // Program header table entry count
  Elf32_Half    e_shentsize;          // Section header table entry size
  Elf32_Half    e_shnum;              // Section header table entry count
  Elf32_Half    e_shstrndx;           // Section header string table index
} Elf32_Ehdr;

typedef struct {
  Elf32_Word    p_type;               // Segment type
  Elf32_Off     p_offset;             // Segment file offset
  Elf32_Addr    p_vaddr;              // Segment virtual address
  Elf32_Addr    p_paddr;              // Segment physical address
  Elf32_Word    p_filesz;             // Segment size in file
  Elf32_Word    p_memsz;              // Segment size in memory
  Elf32_Word    p_flags;              // Segment flags
  Elf32_Word    p_align;              // Segment alignment
} Elf32_Phdr;
```

Defines:
`Elf32_Ehdr`, used in chunk 228b.
`Elf32_Phdr`, used in chunk 228b.

6.8.2 Implementation of the ELF Loader

The default functions which can launch programs on Unix systems are named `exec*`, and typically there is a variety of them. They differ in the way that users can provide arguments. For example, on a Linux machine the man pages for `exec` and `execve` list the following seven functions:

```
int execl   (const char *path, const char *arg, ...);
int execlp  (const char *file, const char *arg, ...);
int execle  (const char *path, const char *arg, ..., char *const envp[]);
int execve  (const char *file, char *const argv[], char *const envp[]);
int execv   (const char *path, char *const argv[]);
int execvp  (const char *file, char *const argv[]);
int execvpe (const char *file, char *const argv[], char *const envp[]);
```

environment

The functions with an `envp[]` argument allow the caller to supply a modified *environment* (a list of exported variables) which we do not support on Ulix: neither the shell nor other application programs can set or query such environment variables.

The functions `execlp`, `execvp` and `execvpe` need not be called with the absolute path of the program but can also accept a simple program name. In that case they will scan the `$PATH`

`$PATH`

variable and search all the listed directories that can contain binaries for the program file. Again, since Ulix does not support environment variables, there is also no `$PATH` variable.

That leaves only the two basic functions `execl` and $\texttt{execv}_{235e}$. These two differ in how arguments for the program can be supplied: `execl` takes as many arguments as needed (behind the program path), whereas $\texttt{execv}_{235e}$ takes only two arguments and the second argument points to a list of arguments. For Ulix we have deviced to provide the $\texttt{execv}_{235e}$ variant, both in the kernel (as $\texttt{u\_execv}_{228b}$) and in the user mode library (as $\texttt{execv}_{235e}$):

[228a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁216c 254b▷

```
int u_execv (char *filename, char *const argv[], memaddress *newstack);
```

Our kernel function takes a third argument `newstack` that will be filled with the address of the new process' user mode stack. It also always returns (and provides the entry address of the newly loaded program if loading it was successful). Note that the user mode library function $\texttt{execv}_{235e}$ has a different semantics: it only returns if loading the program failed, otherwise the old program is gone and the loaded program starts.

[228b] ⟨*function implementations* 100b⟩+≡ (44a) ◁217a 255a▷

```
int u_execv (char *filename, char *const argv[], memaddress *newstack) {
  // returns start address of the loaded binary; or -1 if exec fails
  Elf32_Ehdr elf_header;  Elf32_Phdr program_header;
  ⟨u_execv: check that the executable exists 229a⟩
  ⟨u_execv: check permissions 580a⟩   // see chapter on Users and Groups
  ⟨u_execv: prepare arguments on stack 231⟩
  ⟨u_execv: zero out the memory 232c⟩
  ⟨u_execv: load executable, return entry address 233b⟩
}
```

Defines:
`u_execv`, used in chunks 228a and 234b.
Uses `Elf32_Ehdr` 227, `Elf32_Phdr` 227, and `memaddress` 46c.

6.8.2.1 Step 1: Checking the Executable File

It takes four steps to load and run the program inside the current process. We start with checking that the file we shall load actually is an ELF binary: We open it, read the ELF header which should be right at the beginning of the file and then check whether it contains the magic string that is used to recognize ELF files.

⟨u_execv: *check that the executable exists* 229a⟩≡ (228b) [229a]
```
int fd = u_open (filename, 0, 0);
if (fd == -1)  return -1;   // error
int sz = u_read (fd, &elf_header, sizeof (elf_header));
// check for ELF header
if (sz != sizeof (elf_header) || strncmp (elf_header.e_ident, "\x7f" "ELF", 4) != 0) {
  u_close (fd);
  return -1;
}
```
Uses strncmp 594a, u_close 418a, u_open 412c, and u_read 414b.

6.8.2.2 Step 2: Preparing the Stack

The next step is to prepare the stack: Since programs can be called with arguments, we need to push them onto the stack so that when the program initializes, it can find the arguments where they are expected. We allow up to 512 bytes for such arguments, and the user mode stack always starts at the fixed address TOP_OF_USER_MODE_STACK$_{159b}$. If the total length of the arguments is too long, the surplus arguments are lost.

Remember that the main() function of every program has this prototype:

⟨*main prototype* 229b⟩≡ [229b]
```
int main (int argc, char** argv);
```

When this function starts it expects to access its arguments on the stack like every other function does. The stack contents have to start with the return address, and then the arguments follow. Since we launch a new program we can start with an empty stack. The first address which can be used is 0xaffffffff (as we've set TOP_OF_USER_MODE_STACK$_{159b}$ to 0xb0000000). We want to reserve 512 bytes for the argument strings.

Let's assume that you start a program from the shell by issuing the command

```
esser@ulix[6]:/home/esser$ args This is an example 1 2 3 verylongstring
```

(args is a Ulix program that displays the list of all supplied arguments with their addresses.) This would mean that at the start of args, the argc parameter is set to 9, and argv points to an array of strings (i. e., an array of character pointers). What kind of data do we need to store?

- First of all, we need all the strings (argv[0] to argv[8]) which contain the characters that the arguments consist of, plus a null terminator for each argument.
- Then we need the list of addresses of these strings.
- Finally we need a pointer to the start of this list and the number of arguments.

Each address needs four bytes of storage, so in this example we need $9 \cdot 4 = 36$ bytes for the addresses, and the address of the list itself needs another four bytes.

We start with the result and show the output of `args`:

```
esser@ulix[6]:/home/esser$ args This is an example 1 2 3 verylongstring
argc: 9, &argc: 0xaffffdf8, argv: 0xaffffe00, &argv: 0xaffffdfc
len(argv[0]) =  4, &(argv[0]) = affffe24, argv[0] = args
len(argv[1]) =  4, &(argv[1]) = affffe29, argv[1] = This
len(argv[2]) =  2, &(argv[2]) = affffe2e, argv[2] = is
len(argv[3]) =  2, &(argv[3]) = affffe31, argv[3] = an
len(argv[4]) =  7, &(argv[4]) = affffe34, argv[4] = example
len(argv[5]) =  1, &(argv[5]) = affffe3c, argv[5] = 1
len(argv[6]) =  1, &(argv[6]) = affffe3e, argv[6] = 2
len(argv[7]) =  1, &(argv[7]) = affffe40, argv[7] = 3
len(argv[8]) = 14, &(argv[8]) = affffe42, argv[8] = verylongstring
esser@ulix[6]:/home/esser$
```

We can also request a hex dump of the memory area (thanks to the `hexdump` command in the kernel mode shell, see Chapter 17):

```
affffdf8  09 00 00 00 00 fe ff af  24 fe ff af 29 fe ff af  ........$...)...
affffe08  2e fe ff af 31 fe ff af  34 fe ff af 3c fe ff af  ....1...4...<...
affffe18  3e fe ff af 40 fe ff af  42 fe ff af 61 72 67 73  >...@...B...args
affffe28  00 54 68 69 73 00 69 73  00 61 6e 00 65 78 61 6d  .This.is.an.exam
affffe38  70 6c 65 00 31 00 32 00  33 00 76 65 72 79 6c 6f  ple.1.2.3.verylo
affffe48  6e 67 73 74 72 69 6e 67  00 00 00 00 00 00 00 00  ngstring........
```

little-endian

Note that the byte order is *little-endian* which means that an integer is stored in RAM with the lower bytes coming first. So, for example, the first four bytes of the second line of that hex dump, `2e fe ff af`, store the address `0xaffffe2e` (and not `0x2efeffaf`).

From there we can understand the stack layout and work backwards to arrange the stack that way. Table 6.1 shows a detailed analysis of the stack's contents. We work with a temporary variable `stack` which is a pointer to `unsigned int`; we set it to `TOP_OF_USER_MODE_STACK -512` (which is `0xaffffe00`) so that it points to the beginning of the reserved area. That way we can use pointer arithmetic (`stack--;`) to move to the next address when we want to write (four-byte) addresses to the stack. The statement `*(--stack) = address;` is a push operation: it substracts 4 from the stack address (pointer arithmetic) and *then* writes `address` to the new location. The number of arguments (`argc`) is not known yet, because $\texttt{execv}_{235e}$ accepts a null-terminated array of strings—in the example that is

```
[ "args", "This", "is", "an", "example", "1", "2", "3", "verylongstring", 0 ]
```

—so we need to walk through the list to find the number:

Address	Type	Contents	Interpretation
0xaffffdf8 – 0xaffffdfb	int	0x00000009	argc
0xaffffdfc – 0xaffffdff	int	0xfafffe00	&argv
0xafffe00 – 0xafffe03	int	0xfafffe24	&argv[0]
0xafffe04 – 0xafffe07	int	0xfafffe29	&argv[1]
0xafffe08 – 0xafffe0b	int	0xfafffe2e	&argv[2]
0xafffe0c – 0xafffe0f	int	0xfafffe31	&argv[3]
0xafffe10 – 0xafffe13	int	0xfafffe34	&argv[4]
0xafffe14 – 0xafffe17	int	0xfafffe3c	&argv[5]
0xafffe18 – 0xafffe1b	int	0xfafffe3e	&argv[6]
0xafffe1c – 0xafffe1f	int	0xfafffe40	&argv[7]
0xafffe20 – 0xafffe23	int	0xfafffe42	&argv[8]
0xafffe24 – 0xafffe28	String	"args\0"	argv[0]
0xafffe29 – 0xafffe2d	String	"This\0"	argv[1]
0xafffe2e – 0xafffe30	String	"is\0"	argv[2]
0xafffe31 – 0xafffe33	String	"an\0"	argv[3]
0xafffe34 – 0xafffe3b	String	"example\0"	argv[4]
0xafffe3c – 0xafffe3d	String	"1\0"	argv[5]
0xafffe3e – 0xafffe3f	String	"2\0"	argv[6]
0xafffe40 – 0xafffe41	String	"3\0"	argv[7]
0xafffe42 – 0xafffe50	String	"verylongstring\0"	argv[8]
0xafffe51 – 0xaffffff	—	—	(unused)

Table 6.1: Analysis of the initial stack of a process after calling `exec()`.

⟨u_execv: *prepare arguments on stack* 231⟩≡ (228b) 232a▷ [231]

```
uint *stack = (uint*) (TOP_OF_USER_MODE_STACK - 512);
// find number of arguments
word argc = 0;
while ( (memaddress)(argv+argc) < TOP_OF_USER_MODE_STACK && argv[argc] != 0 )
  argc++;
```

Uses memaddress 46c and TOP_OF_USER_MODE_STACK 159b.

Now that we know the number of arguments, we can reserve space for their addresses. We use two variables in the following loop:

- `target` always points to the memory location where the next argument (string) is to be stored. After each step we add the last argument's length to it.
- `stack` still points to the start of the reserved 512 bytes. In each step `i` we write the argument address into the location `stack + i`. Note again that due to pointer arithmetic, `stack + i` is `(int)stack + 4*i`.

[232a] ⟨u_execv: *prepare arguments on stack* 231⟩+≡ (228b) ◁231 232b▷

```
// copy arguments into the reserved 512 bytes
memaddress target = (memaddress)stack;
memaddress args_start = target;
target += argc*4;
for (int i = 0;  i < argc;  i++) {
  int size = strlen (argv[i])+1;          // string length plus terminator
  memcpy ((void*)target, argv[i], size);  // copy i-th argument
  *(stack + i) = target;                  // store its address
  target += size;
}
```

Uses memaddress 46c, memcpy 596c, and strlen 594a.

Finally, we push the arguments for `main(int argc, char **argv)` and the null return address onto the stack. These will be stored just below the reserved area.

[232b] ⟨u_execv: *prepare arguments on stack* 231⟩+≡ (228b) ◁232a

```
// finish stack preparation
*(--stack) = args_start;  // push pointer to argument list
*(--stack) = argc;        // push number of arguments
*(--stack) = 0;           // push return address (set to 0)
*newstack = (memaddress)stack;
```

Uses memaddress 46c.

If the `main()` function of a program simply returns (and does not call $\texttt{exit}_{218a}$) the normal behavior would be an implicit execution of $\texttt{exit}_{218a}$. We do not provide this feature. However, we have to store some value on the stack that tells where to return to. The start address of $\texttt{exit}_{218a}$ would be a candidate, but in our Ulix implementation we do not know that address, so we just write 0. If you write an application program that leaves `main()` via `return` you will see that it just starts over (or jumps into whatever function was compiled to address 0). Thus, Ulix programs *must* leave via an explicit $\texttt{exit}_{218a}$ call.

6.8.2.3 Step 3: Clearing the Memory

The process memory will still contain data that was stored there before the process called $\texttt{execv}_{235e}$. We do not want the new program to be able to read its predecessor's data, so we delete that data by setting the whole user mode memory to zero:

[232c] ⟨u_execv: *zero out the memory* 232c⟩≡ (228b)

```
memset ((void*)address_spaces[current_as].memstart, 0,
        address_spaces[current_as].memend - address_spaces[current_as].memstart);
```

Uses address_spaces 162b, current_as 170b, and memset 596c.

6.8.2.4 Step 4: Load the Program

Now everything is prepared for loading the program. We walk through the program headers of the ELF file and load the program code. The ELF header may point us to several ELF program headers, so we perform a loop: `elf_header.e_phnum` tells us how many ELF program headers there are.

Each such ELF program header must be read in separately, and then we have to check its type `program_header.p_type`: If it is `ELF_PT_LOAD`$_{233a}$,

⟨*constants* 112a⟩+≡ (44a) ◁200a 292b▷ [233a]
```
#define ELF_PT_LOAD 1
```
Defines:
`ELF_PT_LOAD`, used in chunk 233b.

then we read program code from the file, otherwise we ignore it.

⟨`u_execv`: *load executable, return entry address* 233b⟩≡ (228b) [233b]
```
int phoffset = elf_header.e_phoff;
for (int i = 0;  i < elf_header.e_phnum;  i++) {
  u_lseek (fd, phoffset + i * elf_header.e_phentsize, SEEK_SET);
  u_read (fd, &program_header, sizeof (program_header));
  if (program_header.p_type == ELF_PT_LOAD) {
    ⟨u_execv: reserve sufficient memory 233c⟩
    u_lseek (fd, program_header.p_offset, SEEK_SET);
    u_read (fd, (char*)program_header.p_vaddr, program_header.p_filesz);
  }
}
u_close (fd);
return elf_header.e_entry;  // success. when coming back, set EIP to entry address
```
Uses `ELF_PT_LOAD` 233a, `phoffset`, `SEEK_SET` 469b, `u_close` 418a, `u_lseek` 418a, and `u_read` 414b.

For each chunk that we need to load we find all the relevant information in the ELF program header:

- `program_header.p_offset` tells us the offset *in the ELF file*, so we can `u_lseek`$_{418a}$ to the right file location,
- `program_header.p_vaddr` contains the virtual address where the chunk is to be placed *in memory*, and
- `program_header.p_filesz` is the size of the chunk.

With these three values we can directly `u_lseek`$_{418a}$ and `u_read`$_{414b}$ the chunk without using an intermediate location.

If the loaded program is too big for the currently reserved memory or has a big BSS area (for zero-initialized variables), the loader must acquire more virtual memory via the `u_sbrk`$_{173a}$ function. It finds the total amount of required memory in the `p_memsz` element of the program header:

⟨`u_execv`: *reserve sufficient memory* 233c⟩≡ (233b) [233c]
```
int needed_memsize = program_header.p_memsz;
int current_memsize = address_spaces[current_as].memend
                    - address_spaces[current_as].memstart;
if (needed_memsize > current_memsize) {
  u_sbrk (needed_memsize-current_memsize);
}
```
Uses `address_spaces` 162b, `current_as` 170b, and `u_sbrk` 173a.

6.8.2.5 System Call Handler for execv

The system call handler is a little more complicated than usual because it has to deal with two possible situations: loading the program may succeed or fail.

- If it fails, no changes should be made to the current process, and it should receive a return value of -1 from calling execv.
- If it succeeds we need to update the process context so that it will (re-)start execution at the start address of the new program. Normally, that is 0. We also update the cmdline entry of the TCB.

Thus the function

[234a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁224b 258a▷
```
void syscall_execv (context_t *r);
```

has the following implementation:

[234b] ⟨*syscall functions* 174b⟩+≡ (202b) ◁224c 258b▷
```
void syscall_execv (context_t *r) {
  // generate command line in one string
  char *path  = (char*)r->ebx;            // path argument of execv ()
  char **argv = (char**)r->ecx;           // argv argument of execv ()
  int i = 0;  char cmdline[CMDLINE_LENGTH] = "";
  while (argv[i] != 0) {
    strncpy (cmdline + strlen(cmdline), argv[i], CMDLINE_LENGTH-strlen(cmdline)-1);
    strncpy (cmdline + strlen(cmdline), " ", CMDLINE_LENGTH-strlen(cmdline)-1);
    i++;
  }
  if (cmdline[strlen(cmdline)-1] == ' ')
    cmdline[strlen(cmdline)-1] = '\0';    // remove trailing blank

  // call u_execv()
  memaddress stack;
  memaddress startaddr = (memaddress) u_execv (path, argv, &stack);  // sets stack
  if (startaddr == -1)  eax_return (-1);  // error

  // update context and process commandline
  r->eip     = startaddr;                 // start running at address e_entry
  r->useresp = (memaddress)stack;         // from ELF header
  r->ebp     = (memaddress)stack;
  strncpy (thread_table[current_task].cmdline, cmdline, CMDLINE_LENGTH);
};
```

Defines:
syscall_execv, used in chunks 234a and 235c.
Uses CMDLINE_LENGTH 235a, context_t 142a, current_task 192c, eax_return 174a, memaddress 46c, strlen 594a, strncpy 594b, thread_table 176b, and u_execv 228b.

We have not yet defined the cmdline entry in the thread control block; we'll add it now:

⟨*public constants* 46a⟩+≡ (44a 48a) ◁207a 282b▷ [235a]
```
#define CMDLINE_LENGTH 50   // how long can a process name be?
```
Defines:
CMDLINE_LENGTH, used in chunks 224c, 234b, and 235b.

⟨*more TCB entries* 158c⟩+≡ (175) ◁219a 255d▷ [235b]
```
char cmdline[CMDLINE_LENGTH];
```
Uses CMDLINE_LENGTH 235a.

Finally we register the system call handler and provide a user mode function execv$_{235e}$ that makes the system call:

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁224e 259a▷ [235c]
```
install_syscall_handler (__NR_execve, syscall_execv);
```
Uses __NR_execve 204c, install_syscall_handler 201b, and syscall_execv 234b.

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁223a 259b▷ [235d]
```
int execv (const char *path, char *const argv[]);
```

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁224f 259c▷ [235e]
```
int execv (const char *path, char *const argv[]) {
  return syscall3 (__NR_execve, (uint)path, (uint)argv);
}
```
Defines:
execv, used in chunks 191a and 235d.
Uses __NR_execve 204c and syscall3 203c.

6.8.3 User Mode Binaries

We use a linker configuration file `process.ld` to build our user mode applications and make the compiler use it via the command line option `-T process.ld`. Here's the configuration file: linker configuration

⟨*Application Linker Config File* 235f⟩≡ [235f]
```
OUTPUT_FORMAT("elf32-i386")
ENTRY(main)
virt = 0x00000000;
SECTIONS {
  . = virt;

  .setup : AT(virt) {
    *(.setup)
  }

  .text : AT(code) {
    code = .;
    *(.text)
    *(.rodata*)
    . = ALIGN(4096);
  }

  .data : AT(data) {
    data = .;
    *(.data)
    . = ALIGN(4096);
  }

  .bss : AT(bss) {
    bss = .;
    *(COMMON*)
    *(.bss*)
    . = ALIGN(4096);
  }
  end = .;
}
```
Uses main 44b.

The file tells the compiler to create ELF binaries with virtual addresses starting at address 0. We store the C source code files for our applications in a separate folder (`lib-build/tools/`) and use the following makefile to automatically compile the binaries and copy them to a folder which will be put onto the data disk image:

[236] ⟨*lib-build/tools/Makefile* 236⟩≡

```
LD=ld
CC=/usr/bin/gcc-4.4
OBJDUMP=objdump
CCOPTIONS=-nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer   \
  -fno-function-cse -nostartfiles -mtune=i386 -momit-leaf-frame-pointer
LDOPTIONS=-Tprocess.ld -static -s --pie
OBJECTS = $(patsubst %.c, %, $(wildcard *.c))

all: $(OBJECTS) copy

%: %.c
        $(CC) $(CCOPTIONS) -g $(LDOPTIONS) $^ ../ulixlib.o -o $@
        $(OBJDUMP) -M intel -D $@ > $@.dump

clean:
        rm $(OBJECTS)

copy:
        cp $(OBJECTS) ../diskfiles/bin/
```

There's some magic in this makefile: the line `OBJECTS = $(patsubst %.c, %, $(wildcard *.c))` searches for all `*.c` files (with `wildcard`) and replaces each source filename with the filename without `.c` (e. g. `hexdump.c` with `hexdump`). Then the lines

```
%: %.c
        $(CC) $(CCOPTIONS) $(LDOPTIONS) $^ ../ulixlib.o -o $@
```

tell `make` the rule for creating binaries from source code files, and the `all` target gets a list of all the binaries that are to be created. `$^` always refers to the source file (e. g. `hexdump.c`), and `$@` refers to the target file (`hexdump`). Thus the expanded command for `hexdump` is

```
gcc-4.4 -nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer -fno-function-cse \
  -nostartfiles -mtune=i386 -momit-leaf-frame-pointer -T process.ld -static -s --pie \
  hexdump.c ../ulixlib.o -o hexdump
```

6.9 Exercises

Tutorial 5

In the `tutorial/05/` folder you find a version of the Ulix kernel which contains the new code for handling system calls and also a sample solution for the keyboard interrupt handler exercise. It is a literate program (`ulix.nw`).

In this and the following two exercises you will implement and test some system calls. While you work on the solution, try to stick to the literate programming paradigm, i. e., integrate code and documentation into the document.

20. **Writing strings with `printf`**

    The $\texttt{printf}_{601a}$`()` function is available inside the kernel, but processes cannot call it. In the restricted tutorial version of Ulix there is user mode $\texttt{printf}_{601a}$ function. You will now implement a system call handler which lets processes call the kernel's $\texttt{printf}_{601a}$ function. To make things easier, the goal is that you can later use a `userprint()` function which accepts exactly one string as an argument. ($\texttt{printf}_{601a}$ takes a format string and an arbitrary number of further arguments, but that requires more effort and is not necessary for this exercise.)

    a) Start with defining a syscall number for the `printf` system call in the ⟨ *constants* 112a ⟩ code chunk, e. g.

    ```
    #define __NR_printf   1
    ```

    b) Next you write a syscall handler with the prototype

    ```
    void syscall_printf (struct regs *r);
    ```

    which calls the kernel function $\texttt{printf}_{601a}$. Make sure that you pass the proper arguments: Which of the registers (reachable via `r->eax`, `r->ebx`, `r->ecx` und `r->edx`) holds the address of the string?

    c) Enter the new syscall handler into the system call table.

    d) Write a (user mode) function `void userprint (char *s);` which takes a string as argument and then uses one of the four `syscall*()` functions to perform the system call.

    e) Verify that your code works correctly by adding the line

    ```
    userprint ("Testausgabe\n");
    ```

    to your `main` function.

21. **Reading Memory Locations with `kpeek`**

    The goal of this exercise is to let processes look at any (existing) memory location, even those that belong to the kernel. Of course, no proper operating system would supply such a function since it completely breaks all security mechanisms. Still, it can be done and shows you how to access data structures which are invisible in user mode. With some additional code this might be turned into a useful tool that, for example, lets only the system administrator access the memory.

You will need a function `int kpeek (unsigned int address);` which takes an address as argument, reads the byte that is stored at that address (a value between 0 and 255) and returns it. If the address is not available, the function shall return −1 (which is why its type is `int` and not `unsigned char` which would otherwise be the proper type for a byte).

If this was only about writing a kernel function for the task, you could implement `kpeek` like this:

```
int kpeek (unsigned int address) {
  int page = address / 4096;
  if (pageno_to_frameno (page) == -1)
    return -1;
  else
    return  *(char*)address;
}
```

But again, this function would only be usable by the kernel (which is the same problem that we had with `printf` in the previous exercise). Instead you have to implement `kpeek` via a system call. The general steps are the same as for `printf`:

- Assign and `#define` a system call number.
- Implement a syscall handler which contains a variation of the above `kpeek` code, but which performs parameter and return value passing via registers.
- Enter the new handler in the system call table.
- Write a function `kpeek` that uses the new system call (with the help of one of the `syscall*()` functions).

You can check whether your code works properly by adding the following lines to your `main()` function:

```
unsigned int address = 0xc0000000;
*(char*)(address) = 123;
printf ("Testing kpeek: %d\n", kpeek (address));
```

The middle line writes 123 into the memory location `address`, and the last line should write "Testing kpeek: 123" to the screen. Try the same with an invalid address, e. g.

```
printf ("Testing kpeek/fail: %d\n", kpeek (0x90000000));
```

(This time you should get a "Testing kpeek/fail: -1" output.)

22. **Writing to Memory Locations with `kpoke`**

Reading is one side of the coin, writing is the other. Now you'll add a `kpoke` function that your processes can use to modify the contents of arbitrary kernel memory locations. It has the following prototype:

```
void kpoke (unsigned int address, unsigned char value);
```

If `address` is a valid address, the `value` byte shall be written to that memory location, otherwise the function shall simply return. Again, if this was only about adding functionality to the kernel, the implementation would be as simple as

```
void kpoke (unsigned int address, unsigned char value) {
  if ( ... )   // check for availability
    *(char*)(address) = value;
  return;
}
```

But again, that does not help a process. Implement `kpoke` by writing a system call handler and test the code with the following lines:

```
unsigned int address = 0xc0000000;
kpoke (address, 123);
printf ("Testing kpoke: %d\n", kpeek (address));
```

This is the same test as in the last exercise, but this time you use `kpoke`. The implementation details are very similar to those of `kpeek`, so this time we don't provide detailed steps.

Note that for *proper* testing of the new `printf`, `kpeek` and `kpoke` functions we would need user mode which is not available in this tutorial's version of the kernel. But you'll add that feature in the following exercises.

The `tutorial/06/` folder contains a version of the Ulix kernel which implements the switch to user mode. Again, it is a literate program (`ulix.nw`). Tutorial 6

23. **User Mode Applications**

With this exercise you will create the first user mode application and make it run. As usual: try to write a literate program.

a) *Test program*: First you will write a simple test program for Ulix so that you can see where all of this leads. Your program file `test.c` should only contain a `main()` routine as follows:

```
int main () {
  printf ("Hello - User Mode!\n");
  for (;;) ;   // infinite loop
}
```

You must add the implementation of `printf`: Add one of the `syscall*` functions ($\texttt{syscall1}_{\texttt{203c}}$, $\texttt{syscall2}_{\texttt{203c}}$ etc.) to the file. Our simple `printf` implementation accepts only one string argument, it works just like `userprint` (from Exercise 20). You must also `#define` the constant `__NR_printf` (1).

Note: In the user mode program source files you must always place the `main` function at the very top. All other functions that you might want to call from `main` must be declared *above* the `main` function (by writing down the prototype),

but implemented *below* `main`. If you do not follow that rule, program execution will start in the wrong function (the one whose implementation comes first). The real Ulix does not suffer from this limitation because it has an ELF binary file loader, and ELF binaries contain the start address in the header; for this exercise we're keeping things simple.

The `Makefile` already contains the necessary `gcc` invocation, you need not change it:

```
$(CC) -nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer \
-fno-function-cse -nostartfiles -mtune=i386 -momit-leaf-frame-pointer \
-T process.ld -static -o test test.c
```

By running `make` you generate the executable binary file `test` from the source code.

b) *Install the disassembler*: You will need a disassembler which translates a binary file back to readable assembler code. Install the disassembler package `x86dis` with the following command:

```
sudo apt-get install x86dis
```

c) Then disassemble the generated binary file `test` by running

```
x86dis -e 0 -s intel < test | sort -u
```

The output should begin as follows:

```
00000000 8D 4C 24 04                  lea  ecx, [esp+0x4]
00000004 83 E4 F0                     and  esp, 0xF0
00000007 FF 71 FC                     push [ecx-0x4]
0000000A 51                           push ecx
0000000B 83 EC 08                     sub  esp, 0x08
0000000E 83 EC 0C                     sub  esp, 0x0C
00000011 68 A2 00 00 00               push 0x000000A2
00000016 E8 77 00 00 00               call 0x00000092
...
```

This is the start of the translated `main()` function; the `call` instruction calls the `printf` function.

d) Since we have no filesystem in the current miniature Ulix kernel, we cannot load the program from disk. Instead we use a trick: We write the binary file directly into the kernel and later copy it into user mode memory. If you remember the `start_program_from_disk`$_{189}$ function from Section 6.3, you will soon see a similar function `start_program_from_ram` that replaces it in the absence of disk access.

For copying the binary into the kernel we use the tool `hexdump` whose output format you can set via a format string:

```
hexdump -e '8/1 "0x%02X, "' -e '8/1 """\n"' test
```

creates an output of the following form:

```
0x8D, 0x4C, 0x24, 0x04, 0x83, 0xE4, 0xF0, 0xFF,
0x71, 0xFC, 0x51, 0x83, 0xEC, 0x08, 0x83, 0xEC,
0x0C, 0x68, 0xA2, 0x00, 0x00, 0x00, 0xE8, 0x77,
[...]
0x6F, 0x64, 0x65, 0x21, 0x0A, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
*
```

(Compare these hexadecimal numbers with the numbers that the disassembler has shown: they are identical.)

You can copy the output directly to the kernel source (and add it to the ⟨*global variables* 92b⟩ chunk). Declare and initialize a variable `usermodeprog` like this:

```
unsigned char usermodeprog[] = {
  0x8D, 0x4C, 0x24, 0x04, 0x83, 0xE4, 0xF0, 0xFF,
  0x71, 0xFC, 0x51, 0x83, 0xEC, 0x08, 0x83, 0xEC,
  0x0C, 0x68, 0xA2, 0x00, 0x00, 0x00, 0xE8, 0x77,
  [...]
};
```

You only have to add the first and last line to the `hexdump` output and remove the line with the single asterisk. If there is a line at the end that has only zeroes (`0x00`), you can delete it.

Now it's time to write the function which loads the program.

e) Locate the chunk ⟨*kernel main: user-defined tests* ⟩ in the source code and add

```
start_program_from_ram ((unsigned int)usermodeprog, sizeof(usermodeprog));
```

as its new last line (before the terminating `@` line).

The function `start_program_from_ram` (which you're going to implement) will load and start the program which involves a switch from kernel mode (ring 0) to user mode (ring 3).

24. **User Mode Activation**

Now you will implement some of the functions required for loading and starting a program. You have already seen how this works in this chapter, but here you can create your own implementation of a program loader (that loads from memory, not disk). Many code parts from the regular Ulix kernel are already present in the source code file.

a) The central function is `start_program_from_ram()`: As already mentioned, it takes the place of $\texttt{start\_program\_from\_disk}_{189}\texttt{()}$ from the real kernel. A further difference between Ulix and this version is that there is no scheduler; we start one single process and have no multi-tasking.

```
void start_program_from_ram (unsigned int address, int size) {
  addr_space_id as;
  thread_id tid;
  <<start program from ram: prepare address space and TCB entry>>
  <<start program from ram: load binary>>
  <<start program from ram: create kernel stack>>
  current_task = tid;                    // make this the current task
  cpu_usermode (BINARY_LOAD_ADDRESS,
                TOP_OF_USER_MODE_STACK);   // jump to user mode
};
```

You need not type in this code, it is already included in `ulix.nw`.

b) In order to implement the missing chunks, follow these steps:

In ⟨*start program from ram: prepare address space and TCB entry* ⟩ write appropriate values into the two variables `as`, `tid` by calling `create_new_address_space`$_{163c}$`()` and `register_new_tcb`$_{188d}$`()`. These function are similar to the ones in the real kernel, they're already contained in the `ulix.nw` file for this exercise. Note that the order in which you call these two functions is important: `register_new_tcb`$_{188d}$`()` takes the address space ID as an argument. Give the new process 64 KByte of memory and a 4 KByte user mode stack.

As a further task for the first code chunk you need to assign some values to the TCB elements. Give the new process a PPID (parent process ID) of 0.

The chunk ⟨*start program from ram: create kernel stack* ⟩ is almost identical to ⟨*start program from disk: create kernel stack* 192a⟩ and is also included in `ulix.nw` already.

What's missing is the code chunk ⟨*start program from ram: load binary* ⟩: Here you can use `memcpy()` to copy the `usermodeprog` array (whose start address and length you have provided via the `address` and `size` parameters) to address 0.

`cpu_usermode()` is an assembler function that you can find in `start.asm`.

c) Test your code (calling `make` and `make run`)—after the old "Hello World" message Ulix should also print the line from the user mode program ("Hello – User Mode!").

25. **More Features for User Mode**

In the final exercise of this chapter you add an input mechanism so that your user mode programs can interact with the user. The goal is to let the application read text from the keyboard with `readline()`. This requires several additions:

a) Write a syscall handler which calls the kernel function `kreadline()`. Like all syscall handlers it has the prototype

```
void syscall_readline (struct regs *r);
```

When it is called, `r->eax` contains the syscall number (which you can ignore), and `r->ebx` holds the address of a string (that was declared in the user mode program). In the application source file `test.c` declare a string variable which can hold 256 characters. We do not provide the string length; when calling `kreadline` you can ask that the returned string be no longer than 256 characters. In the handler you have to enable interrupts before calling `kreadline()` because the system deactives them upon handler entry. Add the following line:

```
asm volatile ("sti");   // before kreadline() !
```

Assign a syscall number `__NR_readline` and register the handler function in the syscall table.

b) Now add a function `void readline (char *s);` to `test.c` that uses one of the `syscall*` functions to make the system call. You will have to define the constant `__NR_readline` in that file. Remember to put function prototypes in front of `main()` and implementations behind it so that process execution starts with the first command in `main()`.

c) Modify the `main()` function in `test.c` so that it continuously reads in text and prints it on the console, like this:

```
int main () {
  char s[256];
  printf ("Hello - User Mode!\n");
  for (;;) {
    printf ("> ");  readline (s);
    printf ("Input was: ");  printf (s);
  }
}
```

d) Test your program. After compiling `test.c` with `make` you will again have to convert the binary file into an array of hexadecimal numbers (as you did in Exercise 23 d–e) and integrate it in the kernel file `ulix.nw`. Then call `make` once more to generate the modified kernel.

The output function of this exercise's kernel is able to scroll so that you can simply go on printing even after you've reached the bottom of the screen. In order to understand how scrolling was implemented, look at the `scroll()` function and the two places from where it gets called.

e) The `scroll()` function determines the right memory address to write data to by looking at the `VIDEO` variable: Search for all the lines in the code that change `VIDEO`—the variable takes three different values (`0xc00b8000`, `0xb8000`, `0xd00b8000`) during system initialization—why is that so? As a reminder, the physical addresses that are used for the text mode framebuffer start at `0xb8000`.

f) Copy the code from `test.c` into the literate program `ulix.nw` and modify the `Makefile` so that `test.c` will be extracted from it. Name the code chunk ⟨ *test.c* ⟩. You will need an extra invocation of `notangle`, analogous to

```
notangle -Rulix.c ulix.nw >ulix.c
```

where the option `-R` is followed by the new chunk name and output redirection writes to the right file. Note that all commands in the `Makefile` must be indented by a tabulator character.

Search for code that appears identically in both `ulix.c` and `test.c` and combine it by creating code chunks which will be written to both files. As a result the literate program will be free of duplicates.

7 Implementation of Threads

In Chapters 3 and 4 we looked into a fundamental abstraction offered by the operating system: virtual memory. It abstracts physical memory, one of the main hardware resources. The second such resource is *processor time*, i. e., machine cycles or computation power offered by the CPU. The abstraction which encapsulates processor time in an operating system was traditionally called a *process*, in newer systems *threads* have taken over that job. We have just discussed the implementation of processes in the previous Chapter 6.

"process" vs. "thread"

Up to now, we used the more historic term *process* in parts of this book instead of the term *thread*. Briefly spoken, a process is a virtual address space (defined by a page directory and page tables) plus exactly one thread. Thus, the term process alludes to the classical *Unix process*. Today, modern operating systems offer multiple threads within one virtual address space, and so does Ulix.

Unix process

Thus, a summary of the differences between classical operating systems that are based on processes and newer systems with threads can be given as follows:

- In classical systems, process management provides a common abstraction for both processor time and memory. Switching from one process to another always means that the used address space changes, too.
- In modern systems, thread management and memory management are decoupled: It is possible to switch from one thread to another *without also changing the address space*. A process is simply a collection of one or more threads which share a common address space.

Note that some operating systems allow variations of the thread and process concepts which are something in-between. For example, the Linux kernel provides a `clone` function which can create new processes, new threads and other kinds of tasks which are neither.

7.1 Threads, Teams of Threads and Virtual Processors

virtual processor

A thread[1] can be regarded quite literally as an execution thread within the operating system. Threads are abstractions of processing time, *virtual processors*. They are implemented by multiplexing virtual processors (the threads) onto the physical processors (CPUs). A thread always has an associated *program*, i. e., a sequence of machine instructions which it executes. When a thread starts its operation, execution starts at a pre-defined address in this sequence.

Threads and address spaces are two abstractions which are orthogonal but nevertheless closely tied together. Whenever a virtual address space is created, a first thread is also created within the address space. This results in what is often called a *process*. In most cases (as in early Unix) this is absolutely sufficient to perform all the classical application tasks programmed on top of the operating system, and it is what you have seen when we described the `u_fork`$_{209c}$ implementation of the fork mechanism. However, it sometimes makes sense to create multiple threads within a single address space, as we now explain.

7.1.1 Teams of Threads

team of threads

We call multiple threads within a single address space a *team* (or *team of threads*). Why does it make sense to create multiple threads within one address space? There are several answers to this question.

The first block of answers refers to performance issues:

- If within an application one thread invokes a system call which blocks for an I/O operation to succeed, then the whole application will block if the application is carried on just this one single thread. If more than one thread would carry the application, these other threads could continue to operate, giving the user a better quality of service.
- Also, if an application runs on multiple threads, it is possible to distribute the machine cycles onto *physically distinct* processors. This (of course) is not an issue in a monoprocessor system. However, in a dual processor system for example an application which is carried only by a single thread will never be able to bring the power of the two CPUs into the application.

The second block of answers refers mainly to software engineering aspects, i. e., the way we write programs.

- Multiple threads within one address space allow to program those applications which contain inherent parallel activities in a much more natural way. The result is a *concurrent model of programming* which includes both the fields of distributed and parallel programming. Concurrent programming refers to programming multiple independent threads of execution in general.

[1] The theory Sections 7.1 and 7.2 of this chapter are heavily based on the "Threads" chapter of Nehmer and Sturm's book [NS01].

- Parallel programming on the one side refers rather to more dependent threads, e. g., threads which operate in strongly synchronized "lock-step" mode.
- Distributed programming on the other side refers to concurrent programming where the aspect of geographic distribution plays a role (like in the Internet).

7.1.2 Natural Concurrency

Many of today's operating systems already support multiple threads in one address space and so it is becoming more and more natural to use them. It is especially natural if the application which is implemented already contains *inherent* concurrency. As an example (taken from Nehmer and Sturm [NS01]) consider a weather reporting application. It consists of a huge database in which new measurements of humidity, temperature etc. are regularly logged from different sensing stations. From this database the application computes in a continuous manner weather reports for different areas of the country using complex weather models. Additionally, the application has a graphical interface through which users can inspect data, query weather reports and visualize measurement data.

inherent concurrency

Looking at the application from a concurrent programming viewpoint, it has three rather independent streams of activity:

1. The measurement and logging activity of data into the database.
2. The continuous weather prediction and reporting computation.
3. The graphical user interface.

Note that each stream of activity by itself is sequential.

Let's make things simple and just look at the last two activities: computation and user interface. As both are sequential activities, we can program them separately and enclose each activity within a thread. The pseudocode could look like this:

⟨*weather reporting example: thread pseudocode* 247⟩≡ [247]

```
void Compute () {                    // activity 1: computation
  while (true) {
    // do the actual computation
  }
}

void GUI() {                         // activity 2: graphical user interface
  while (true) {
    Event e = ReceiveEvent ();
    ProcessEvent (e);
  }
}

int main () {
  start_thread (Compute ());         // Start concurrent threads
  start_thread (GUI ());
}
```

Note that the sequential activities are encoded within simple sequential functions which are both started within separate threads in the `main` routine and thereafter run separately. Here we assume that the entire application exits when all of its threads have exited.

How would we program this application traditionally (i. e., without threads)? We would have to split the activities into small slices and run them alternately. Assume we can divide the function `Compute()` into small parts called `ComputeStep`. Then after computing such a step we would need to check whether user input must be handled. If yes, we handle it, if not, we compute the next step. The pseudocode could look like this:

[248] ⟨*weather reporting example: traditional pseudocode* 248⟩≡

```
int main () {
  while (true) {
    ComputeStep ();          // compute the next step
    if (QueryEvent ()) {     // do we have to process an event?
      e = ReceiveEvent ();
      ProcessEvent (e);
    }
  }
}
```

This approach should also work, but only under the assumption that we can in fact split `Compute` into `ComputeStep`. In many cases this is not as easy as it seems, sometimes it might even be impossible. Another disadvantage of the traditional approach is that the computation is interrupted regularly even if there are no events to be processed. In this case the code for `QueryEvent` should be very efficient so that it doesn't cost too many CPU cycles. It goes without saying that functions like `QueryEvent` should not block (e. g., until user input arrives) because this would block the entire application.

There are more downsides of the traditional approach. For example, the program structure without threads determines the reaction time to user input. If `ComputeStep` may take up to a couple of seconds of execution time, then reaction to user input can also take this time. The execution time of `ComputeStep` should therefore be rather short to guarantee *responsiveness*. However, a short execution time implies that the overhead of `QueryEvent` increases in relation. So we have a non-trivial tradeoff here. Finally, but this is a matter of taste, we find the traditional code much harder to read and understand than the code using threads.

responsiveness

7.1.3 Advantages of Concurrent Programming

Threads allow to create an unbounded number of virtual processors, no matter how many physical processors exist in the system. This lets us distribute applications over as many virtual processors as are necessary to serve their inherent concurrency. Threads therefore allow to abstract from the actual number of physical processors in the system and to depart from the traditional sequential programming model. If an application has inherent, natural concurrency, then it should be expressed in the program.

Threads do not only make programs with inherent concurrency easier to read and understand, they also may make the execution of the application more efficient since only

concurrent applications can exploit the power of truly concurrent hardware available in multiprocessor systems. But even on monoprocessor systems a concurrent program can be more efficient than its sequential counterpart because the periods in which one thread is blocked (e. g., due to lack of user input) can be used by other threads more effectively.

7.1.4 Virtual vs. Physical Processors

As mentioned above, a thread can be regarded as a *virtual processor*. Therefore, a team of threads can be regarded as a *virtual multiprocessor*. Ideally, every virtual processor is backed up from below by exactly one physical processor and the assignment of virtual to physical processors is fixed. However, the normal case is rather different: Many virtual processors need to be executed on few physical processors. The task of the operating system is to distribute the physical processor cycles as effectively as possible between the virtual processors in a kind of *time division multiplex* mode of operation. This is depicted in Figure 7.1.

virtual multiprocessor

As an example, consider the case where one physical processor carries two virtual processors (threads). In this case the threads would be assigned alternately to the physical processor by the operating system. The change from one thread to the other is called a *context switch*. Within the context switch, the execution of the current thread is interrupted, the processor context (registers, stack pointer etc.) is saved somewhere, the processor context of the next thread is loaded from somewhere onto the processor and the next thread then continues execution at the point in its program where it was previously interrupted.

context switch

In a sense, the operating system pretends that every thread has exclusive access to the physical processor. During a context switch, the previously running thread is "frozen" and saved somewhere. The next thread is selected and "unfrozen" by loading its state into the CPU. During the times in which they are frozen, threads do not operate. In fact, since they don't operate, they are not aware that time is passing. After unfreezing the new thread, it continues operation as if it had never been interrupted. This can remotely be compared with becoming unconscious after a knock out in boxing.

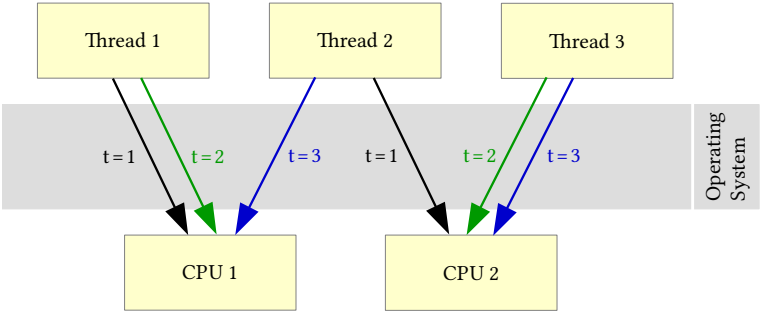


Figure 7.1: The assignment of virtual to physical processors can change over time.

scheduler

If there is more than one candidate for the next running thread, the operating system has to make a choice. The operating system component which is responsible for making this decision is called the *scheduler*. As we will see later (in Chapter 8) there exist many different strategies to make this scheduling decision.

7.2 Thread Requirements and Thread Types

Before we delve into the implementation, let's take a closer look at why threads are so useful. They help the operating system reach a higher degree of concurrency for the applications it runs.

7.2.1 Thread Requirements

If an operating system supports threads, it must offer at least two types of functionality: On the one hand, a user should be able to create a new address space with a *single* thread. On the other hand, the user should be able to assign a new program to this thread. Often these two functionalities are assembled within one single system call offered by the kernel.

To offer more flexibility, it should be possible to create *multiple* threads within one address space. Good operating systems therefore offer functionality to create a new thread within the same address space at runtime and to assign a new program to this thread.

7.2.2 Utility of Threads

Normally, several threads wait for the same processor to become free. Let's assume that each thread, once activated, uses k units of time for completion and would run until it is finished. The first thread starts at once, the second thread after k units of time, etc. That leads to an average response time of

$$\frac{1}{n}\sum_{i=1}^{n}(i-1)\cdot k = \frac{n-1}{2}\cdot k$$

units of time [NS01, p. 101]. However, this ignores that threads typically alternate between CPU and I/O bursts:

CPU burst

- A *CPU burst* is a time range during which a thread uses the CPU, i. e., it is active and executes instructions.

I/O burst

- An *I/O burst* is a time range during which a thread waits for the completion of an I/O operation that it initiated. The burst begins in the moment where the thread is put on a blocked queue (as a direct result of requesting the I/O operation) and it ends when the I/O operation completes and the thread is moved to the ready queue.

If there is only one single thread, then the system switches between CPU and I/O bursts of that thread. With several threads and a scheduler the situation becomes more complicated since the scheduler can interrupt an active thread in the middle of a CPU burst.

Also, whenever an I/O burst begins, the CPU is reassigned to a different thread (which continues its CPU burst).

If I/O was handled completely asynchronously (i. e., we ignore the times required to process I/O requests), the CPU burst times would lead to an average response time of

$$\frac{n-1}{2} \cdot t_{\text{burst}}$$

where t_{burst} is the average length of a CPU burst [NS01, p. 102].

Nehmer and Sturm measured the length of bursts and looked at their statistical distribution which resulted in the image shown in Figure 7.2 [NS01, p. 102]. This shows that typically CPU bursts are very short—much shorter than I/O bursts. Using the time that a thread waits for I/O completion for other threads (which continue their CPU bursts) creates a considerable degree of concurrency, even on a monoprocessor. Thus, it is important to pause threads which wait for I/O (which is precisely what we've been doing for processes in Chapter 6), and with threads it improves the behavior of processes by allowing CPU-burst threads of a process to continue while I/O-burst threads are blocked. This improves response times for those processes.

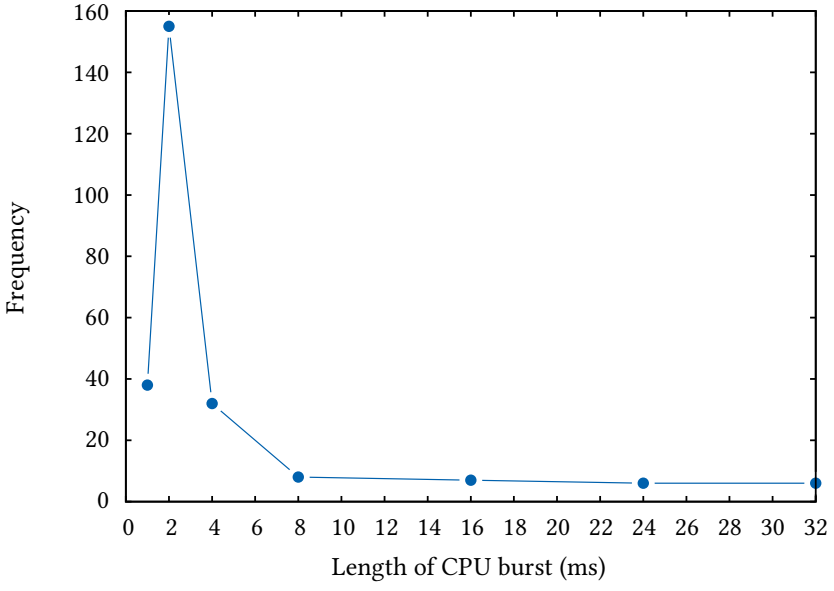


Figure 7.2: Distribution of the CPU burst length: A length of 2 ms occurred most often (as measured by Nehmer and Sturm).

However, once the number of threads becomes very large, too many of them will be in their CPU bursts simultaneously, and then the overall execution speed and responsiveness will shrink.

7.2.3 Types of Threads

Two different types of threads are usually distinguished: *kernel-level threads* and *user-level threads*. The "classical" threads (i. e., the threads in Unix processes) are kernel-level threads. The distinction is based on the mode in which the context switch is performed. In kernel-level threads the context switch happens in system mode, in user-level threads it happens in the user mode thread library.

Kernel-level threads can be regarded as virtual processors running directly on physical processors. User-level threads can be regarded as virtual processors running on kernel-level threads. In this sense, a team of kernel-level threads running in the same virtual memory can be regarded as a *virtual multiprocessor* for user-level threads.

virtual multiprocessor

The techniques used to implement and synchronize virtual processors (i. e., kernel-level threads) on physical processors are the same as those used to implement and synchronize virtual processors (user-level threads) on virtual processors (kernel-level threads). Therefore people sometimes speak of a *processor hierarchy*. Virtual processors run on virtual processors that run on physical processors. (Note that in principle it is even possible to run user-level threads on user-level threads, extending the processor hierarchy.) The multiplexing of higher-level processors to lower-level processors is performed by a software layer (see Figure 7.3). In case of kernel-level threads implemented on physical processors this software layer is the operating system; in case of user-level threads implemented on kernel-level threads this software layer is usually called the *user-level threads library*.

processor hierarchy

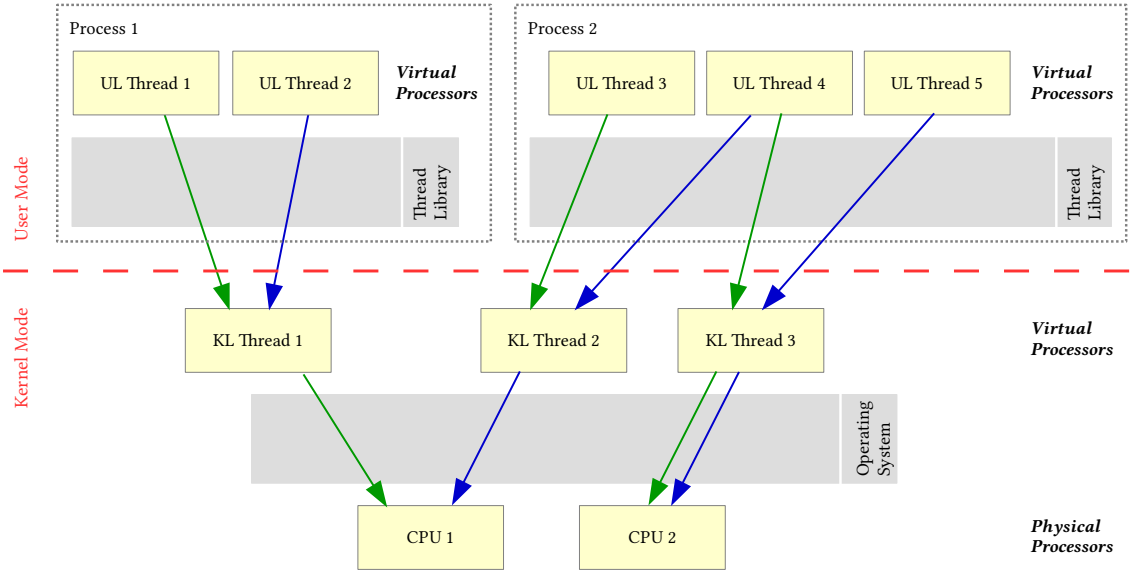


Figure 7.3: Schematic view of the processor hierarchy. Arrow colors (on one level) express the order of allocation of a virtual (top) or physical (bottom) processor, but the top and bottom arrows have their own times.

In the example shown in Figure 7.3 the green and blue arrows describe the order in which a virtual/physical processor is assigned to a user-level/kernel-level thread. If we assume that the time slices for user and kernel level threads are identical, then the example thread might execute as shown in Table 7.1, but in practice switch times for user-level threads will not be synchronized with switch times for kernel-level threads.

Time	KLT	ULT
n	1	1
$n+1$	2	3
$n+2$	3	4
$n+3$	1	2
$n+4$	2	4
$n+5$	3	5
$n+6$	1	1
$n+7$	2	3
$n+8$	3	4
$n+9$	1	2
$n+10$	2	4

Table 7.1: Possible order of execution for the threads shown in Figure 7.3.

7.2.3.1 Kernel Level Threads

Kernel-level threads are managed in system mode, i. e., the context switch of one kernel-level thread to the next requires some action by the operating system's scheduler. This is why kernel-level threads are often called *heavy-weight* threads, because entering and leaving the scheduler incurs a large performance overhead. For example, if the context switch from one kernel-level thread to the next also causes a switch of one virtual memory to another (i. e., the process changes), the effect of caching in the TLB or in any other local cache is destroyed.

heavy-weight thread

7.2.3.2 User-Level Threads

User-level threads run on kernel-level threads. From the point of view of user-level threads, kernel-level threads are virtual processors that carry the user-level thread library. In contrast to kernel-level threads, user-level threads are managed entirely in user mode. This implies that the context switch of user-level threads does not incur a context switch of the kernel-level thread that carries it. This in turn implies that there is no switch of virtual memories and no performance penalty. Therefore, user-level threads are often called *light-weight* threads.

light-weight thread

Ulix only implements kernel-level threads. We show the code in the following section.

7.3 Implementation of Threads in Ulix

So far, we have been talking about threads all the time, e. g., when we discussed the structure of the *thread* control block, but all the previous code actually dealt with processes. That changes now: we are about to introduce true threads which can be created inside processes.

7.3.1 Creating Threads Instead of Processes

Creating a kernel-level thread (belonging to an already existing process) is very similar to forking a process. The original idea was to let `u_fork`$_{209c}$ perform its usual tasks, but with two exceptions—it should not copy the user mode memory, and it should provide two new stacks (for kernel and user mode) which exist in the same address space. This approach is used in the Linux kernel's `clone` function which can generate both new threads and new processes [Lov03, pp. 22–27].

However, our first attempts led to many if-then-else constructions in the `u_fork`$_{209c}$ function which reduced the readability of the code, and we also wanted to describe process forking and thread creation separately, so we decided against the combined treatment inside `u_fork`$_{209c}$. Instead we explain thread creation in this section, and we will provide a separate function `u_pthread_create`$_{255a}$ which is loosely based on the POSIX user mode function `pthread_create`.

We're going to implement (parts of) the POSIX thread API [IEE95] inside the kernel, and the user mode library functions will simply access these kernel functions via system calls.

POSIX threads need a thread identifier of some type `pthread_t`$_{254a}$. We define this public type as a pointer to a kernel-internal data structure whose definition we will not export to user land, so we declare:

[254a] ⟨*public type definitions* 142a⟩+≡ (44a 48a) ◁175 369a▷

```
typedef void *pthread_t;
typedef void pthread_attr_t;   // attributes not implemented
```

Defines:
`pthread_attr_t`, used in chunks 255a and 259c.
`pthread_t`, used in chunks 255a and 259.

As already mentioned, a new thread differs from a new process by sharing its creator's address space, but it still needs its own kernel and user mode stacks—in the same address space which the calling process currently uses.

Our implementation of the kernel's

[254b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁228a 259d▷

```
int u_pthread_create (pthread_t *restrict thread, const pthread_attr_t *restrict attr,
                      memaddress start_address, void *restrict arg);
```

function looks similar to `u_fork`$_{209c}$:

⟨*function implementations* 100b⟩+≡ (44a) ◁228b 260a▷ [255a]
```
int u_pthread_create (pthread_t *restrict thread, const pthread_attr_t *restrict attr,
                      memaddress start_address, void *restrict arg) {
  ⟨begin critical section in kernel 380a⟩  // access the thread table
    thread_id old_tid = current_task;
    ⟨u_pthread_create: create new TCB 255b⟩
    ⟨u_pthread_create: fill new TCB 255c⟩
    ⟨u_pthread_create: create new stacks 257c⟩
  ⟨end critical section in kernel 380b⟩
  return 0;
}
```
Defines:
u_pthread_create, used in chunks 254b and 258b.
Uses current_task 192c, memaddress 46c, pthread_attr_t 254a, pthread_t 254a, and thread_id 178a.

Where u_fork$_{209c}$ starts with creating a fresh address space and reserving a TCB, the thread only needs a TCB:

⟨u_pthread_create: *create new TCB* 255b⟩≡ (255a) [255b]
```
thread_id new_tid = register_new_tcb (current_as);
address_spaces[current_as].refcount++;
address_spaces[current_as].extra_kstacks++;   // see below
```
Uses address_spaces 162b, current_as 170b, register_new_tcb 188d, and thread_id 178a.

We increase the refcount and extra_kstacks elements of the address space which lets us keep track of how often this address space is in use: As long as refcount is non-zero, we must not reuse the address space. extra_kstacks has a similar but slightly different function that we will explain further below.

Entering data in the TCB is similar to the respective step in u_fork$_{209c}$:

⟨u_pthread_create: *fill new TCB* 255c⟩≡ (255a) [255c]
```
TCB *t_old = &thread_table[old_tid];
TCB *t_new = &thread_table[new_tid];
*t_new = *t_old;     // copy the complete TCB
                     // note: this destroys data set in register_new_tcb ()
memset (&t_new->regs, 0, sizeof (context_t));
t_new->state      = TSTATE_FORK;
t_new->tid        = new_tid;
t_new->addr_space = current_as;
t_new->new        = true;         // mark new thread as new
t_new->pid        = old_tid;      // thread; pid != tid
t_new->ppid       = t_old->ppid;  // new thread has same parent as caller
```
Uses context_t 142a, current_as 170b, memset 596c, t_new 276c, t_old 276c, TCB 175, thread_table 176b, and TSTATE_FORK 180a.

We need two new TCB entries to mark a thread as new and to store its kernel stack address:

⟨*more TCB entries* 158c⟩+≡ (175) ◁235b 326d▷ [255d]
```
boolean   new;                  // is this thread new?
void      *top_of_thread_kstack; // extra kernel stack for this thread
```

We will give only one page of kernel stack memory to each new thread, and we put those just under the regular kernel stack.

Remember that we've defined `TOP_OF_KERNEL_MODE_STACK`$_{159c}$ and `KERNEL_STACK_PAGES`$_{169b}$. We can now calculate `TOP_OF_KERNEL_MODE_STACK`$_{159c}$ – `KERNEL_STACK_PAGES`$_{169b}$ * `PAGE_SIZE`$_{112a}$ to find the lowest possible address that may be used by the kernel stack. In principle we could have our first thread's kernel stack start just below, but we want to provide a protective buffer of unmapped memory: That way, whenever one of the threads exceeds its kernel stack it will generate a page fault (and in turn the process will be stopped).

protective buffer

We will provide each thread with just one page of kernel stack memory whereas the initial process always gets four pages of them—this is just intended to simplify the clean-up of a process with several threads (Figure 7.4).

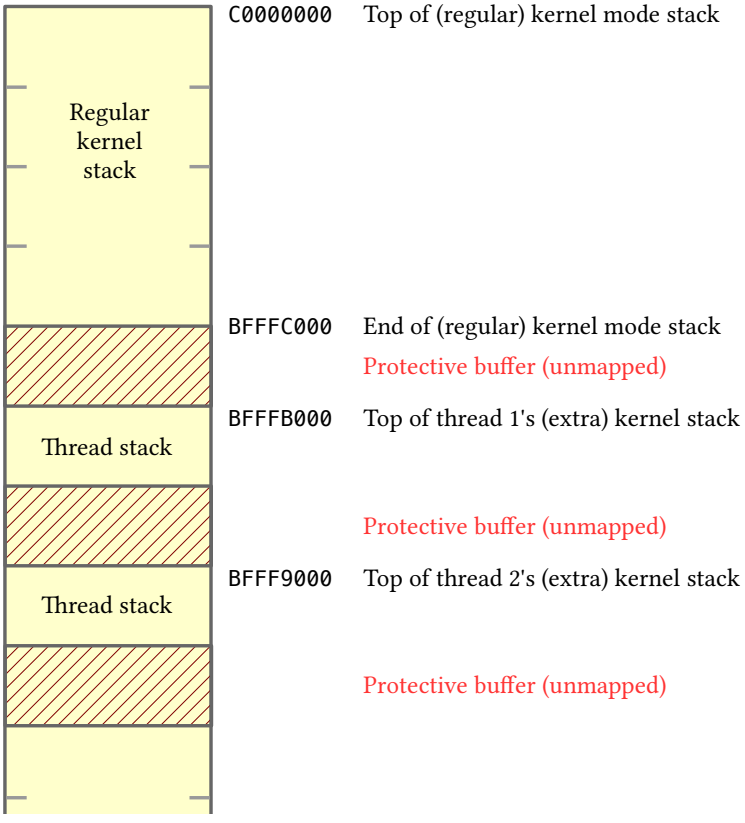


Figure 7.4: Each new thread gets a single page for its kernel stack—just below the regular kernel stack but with one-page buffers of non-mapped virtual memory between the stacks.

This leads to the address calculation

```
TOP_OF_KERNEL_MODE_STACK159c - (KERNEL_STACK_PAGES169b + 2 * thread) * PAGE_SIZE112a
```

where `thread` is 1, 2, etc. for the first, second, etc. *new* thread (and it is 0 for the original kernel thread of the process). For example, for the first new thread (and assuming that `TOP_OF_KERNEL_MODE_STACK`$_{159c}$ = `0xc0000000`, `KERNEL_STACK_PAGES`$_{169b}$ = 4) we get `0xc0000000` $-(4+2\cdot 1)\cdot 4096 =$ `0xc0000000` $-6\cdot 4096 =$ `0xbfffa000` which is the start address for the new stack.

We could use the address space's `refcount` element in our formula (but would need to subtract 1 since it starts counting with the initial, thread-less process). However, this only works if we assume that a process will create several new threads and then enters a stage in which threads will only terminate until none is left.

If thread creation is more dynamic, with new threads coming and old threads going away, we cannot use this approach because a leaving thread will decrease `refcount`. So we introduce a new `address_space`$_{161}$ entry `extra_kstacks`:

⟨ *more* `address_space` *entries* 257a⟩≡ (161) [257a]
```
byte extra_kstacks;
```

which counts the number of extra kernel stacks. In a pure process (without extra threads) its value will always be 0. We will update `refcount` by increasing and decreasing it as threads come and go, but we will only increase `extra_kstacks` when we add a new thread. That way, for every new thread we can get a fresh kernel stack.

(As a side effect all extra kernel stacks will continue to exist until the process finally terminates; more about that at the end of this chapter.)

This finally lets us calculate the bottom of the new kernel stack:

⟨ *bottom of new kernel stack* 257b⟩≡ (257c) [257b]
```
TOP_OF_KERNEL_MODE_STACK
- ( KERNEL_STACK_PAGES + 2 * (address_spaces[current_as].extra_kstacks) )
  * PAGE_SIZE
```
Uses `address_spaces` 162b, `current_as` 170b, `KERNEL_STACK_PAGES` 169b, `PAGE_SIZE` 112a, and `TOP_OF_KERNEL_MODE_STACK` 159c.

Now we have everything that we need for stack creation. For the new user mode stack we simply increase the process' heap (via `u_sbrk`$_{173a}$), and for the kernel stack we reserve a frame and update the address space.

⟨`u_pthread_create`: *create new stacks* 257c⟩≡ (255a) [257c]
```
// create user stack
void *ustack = u_sbrk (PAGE_SIZE);
memset (ustack, 0, PAGE_SIZE);

// create kernel stack
int kstack_frame = request_new_frame ();                          // get a frame
uint kstack_start = ⟨ bottom of new kernel stack 257b⟩;
as_map_page_to_frame (current_as, kstack_start >> 12, kstack_frame); // map it
```

```
uint *STACK = (uint*) (kstack_start+PAGE_SIZE);     // top of new stack
t_new->top_of_thread_kstack = STACK;

*(--STACK) = 0x20 | 0x03;               // push ss (selector 0x20 | RPL3: 0x03)
*(--STACK) = (uint)ustack + PAGE_SIZE;  // push esp (for user mode)
*(--STACK) = t_old->regs.eflags;        // push eflags
*(--STACK) = 0x18 | 0x03;               // push cs (selector 0x18 | RPL3: 0x03)
*(--STACK) = start_address;             // push eip (for user mode)

t_new->esp0 = (memaddress)STACK;
add_to_ready_queue (new_tid);
```

Uses add_to_ready_queue 184b, as_map_page_to_frame 165b, current_as 170b, kstack_frame, memaddress 46c, memset 596c, PAGE_SIZE 112a, request_new_frame 118b, t_new 276c, t_old 276c, top_of_thread_kstack, u_sbrk 173a, and ustack.

Compare the values that we push on the stack to the values we push in the assembler function cpu_usermode$_{198}$ that is invoked when the very first process starts: These are the same data, except for the start address which is 0 for the initial process and the address of the thread function in case of a new thread. The reason why we need this stack setup is the same in both cases: When the iret instruction is executed, the stack has to contain the information that lets the system go back to user mode and continue execution at the right address.

7.3.2 System Call for Thread Creation

We provide a simplified user mode implementation of thread creation which ignores thread IDs and simply provides the start address of the thread function. This is possible because we will not provide a pthread_join function (that lets a thread wait for the termination of a specific other thread). The system call handler

[258a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁234a 282a▷

```
void syscall_pthread_create (context_t *r);
```

just calls u_pthread_create$_{255a}$ with the start address in the right place and all other arguments set to NULL$_{46a}$.

[258b] ⟨*syscall functions* 174b⟩+≡ (202b) ◁234b 282c▷

```
void syscall_pthread_create (context_t *r) {
  // ebx: address of thread function
  memaddress address = r->ebx;
  u_pthread_create (NULL, NULL, address, NULL);
};
```

Uses context_t 142a, memaddress 46c, NULL 46a, syscall_pthread_create, and u_pthread_create 255a.

The next free system call number is

[258c] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁224d 260b▷

```
#define __NR_pthread_create 506
```

Defines:
__NR_pthread_create, used in chunk 259.

and we add a syscall table entry:

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁235c 260c▷ [259a]
```
install_syscall_handler (__NR_pthread_create, syscall_pthread_create);
```
Uses __NR_pthread_create 258c, install_syscall_handler 201b, and syscall_pthread_create.

For the user mode library we stick with the POSIX prototype

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁235d 260d▷ [259b]
```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
                    void *address, void *arg);
```

but as mentioned above, the only thing we pass along is the start address:

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁235e 260e▷ [259c]
```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
                    void *address, void *arg) {
  return syscall2 (__NR_pthread_create, (memaddress)address);
}
```
Uses __NR_pthread_create 258c, memaddress 46c, pthread_attr_t 254a, pthread_t 254a, and syscall2 203c.

7.3.3 Terminating Threads

In a complete POSIX thread implementation a thread can call pthread_exit$_{260e}$ to terminate, and another thread may call pthread_join to wait for that specific thread to finish. Describing pthread_join in this book would be a repetition of the code that we've shown when we discussed the process mechanisms provided by waitpid$_{220d}$ and exit$_{218a}$: We would add another blocked queue to the system and move a thread that calls pthread_join to that queue; then the pthread_exit$_{260e}$ function would check whether there is a thread that waits for this thread and wake it up.

Since no deeper understanding is gained by this repetition, we only provide the

⟨*function prototypes* 45a⟩+≡ (44a) ◁254b 275▷ [259d]
```
void syscall_pthread_exit (context_t *r);
```

function. Note that this means that many multi-threaded code examples will not work with Ulix, but from our explanation it should be clear how you could extend the Ulix code so that it supports threads properly.

last thread leaving

There is a special case we need to consider: The last thread of a multi-threaded process has two options for terminating.

- It can call the regular process exit function exit$_{218a}$. This will typically be the case if it was the "master process" that executes the main() function. In that case all other threads have already left via pthread_exit$_{260e}$.
- It can alternatively call pthread_exit$_{260e}$. The man page for pthread_exit$_{260e}$ states:

  "After the last thread in a process terminates, the process terminates as by calling exit(3) with an exit status of zero; thus, process-shared resources are released and functions registered using atexit(3) are called." [Lin12b]

  So in this special case (which we can detect by checking refcount == 1 in the current address space) we simply call syscall_exit$_{216b}$ and let it do the work.

[260a] ⟨*function implementations* 100b⟩+≡ (44a) ◁255a 276d▷
```
void syscall_pthread_exit (context_t *r) {
  if (address_spaces[current_as].refcount == 1) {
    // last thread leaves: use normal exit mechanism
    r->ebx = 0;                                  // set process exit code to 0
    syscall_exit (r); return;                    // and leave
  }

  ⟨begin critical section in kernel 380a⟩  // access the thread table
  thread_table[current_task].state = TSTATE_EXIT;
  remove_from_ready_queue (current_task);
  address_spaces[current_as].refcount--;
  thread_table[current_task].used = false;    // release TCB
  ⟨end critical section in kernel 380b⟩
  ⟨resign 221d⟩
}
```
Defines:
`syscall_pthread_exit`, used in chunks 259d and 260c.
Uses `address_spaces` 162b, `context_t` 142a, `current_as` 170b, `current_task` 192c, `remove_from_ready_queue` 184c, `syscall_exit` 216b, `TCB` 175, `thread_table` 176b, and `TSTATE_EXIT` 180a.

Again we register a system call and provide a user mode function which needs no further explanation since it simply makes the system call (and need not provide any arguments).

[260b] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁258c 310b▷
```
#define __NR_pthread_exit 507
```
Uses `__NR_pthread_exit`.

[260c] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁259a 282d▷
```
install_syscall_handler (__NR_pthread_exit, syscall_pthread_exit);
```
Uses `__NR_pthread_exit`, `install_syscall_handler` 201b, and `syscall_pthread_exit` 260a.

[260d] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁259b 282e▷
```
void pthread_exit ();
```

[260e] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁259c 282f▷
```
void pthread_exit () { syscall1 (__NR_pthread_exit); }
```
Defines:
`pthread_exit`, used in chunk 260.
Uses `__NR_pthread_exit` and `syscall1` 203c.

Note that the POSIX prototype for `pthread_exit`$_{260e}$ provides an exit value argument which we omit because in our implementation there is no way for another thread to access it.

[260f] ⟨*POSIX* `pthread_exit` *prototype* 260f⟩≡
```
void pthread_exit (void *value_ptr);
```

7.3.3.1 Getting Rid of the Extra Kernel Stacks

In ⟨*scheduler: free old kernel stacks* 169a⟩ we had included a code chunk named ⟨*remove extra thread kernel stacks* 261⟩ and given no further explanation (because at that time we only dealt with processes). Now the time has come to explain how to get rid of the extra kernel stacks.

In the overall kernel stack deletion chunk, we're in the middle of a loop (over the elements of the `kstack_delete_list`), and `id` is the ID of the current address space that we need to delete. In a regular process `address_spaces`$_{162b}$`[id].extra_kstacks` will be 0, we don't want to touch such an address space any further. But if the value is larger than 0, we remove the extra pages:

⟨*remove extra thread kernel stacks* 261⟩≡ (169a) [261]

```
if (address_spaces[id].extra_kstacks > 0)
while (address_spaces[id].extra_kstacks > 0) {
  uint stack = TOP_OF_KERNEL_MODE_STACK -
    (KERNEL_STACK_PAGES + 2 * (address_spaces[id].extra_kstacks)) * PAGE_SIZE;
  int frameno = mmu_p (id, stack/PAGE_SIZE);
  if (frameno != -1)  release_frame (frameno);
  address_spaces[id].extra_kstacks--;
}
```

Uses address_spaces 162b, KERNEL_STACK_PAGES 169b, mmu_p 171c, PAGE_SIZE 112a, release_frame 119b, and TOP_OF_KERNEL_MODE_STACK 159c.

(Note that we cannot use `release_page`$_{122d}$ because the address space is not active; the current page table does not point to the frames we want to free.)

For determining the start of each kernel stack we use the same formula that we used when we created the stack (see ⟨*bottom of new kernel stack* 257b⟩).

7.3.4 Thread Synchronization

We will also provide `pthread_mutex_*` functions for thread synchronization. You can find the code in Chapter 11.5 (which is part of the *Synchronization* chapter).

8 Scheduling

Every multi-tasking operating system needs a scheduler: It is the primary OS component that allows the quasi-parallel execution of several programs. Scheduling actually encompasses two separate tasks:

- deciding when to switch from one process or thread to another and picking that new task
- and actually performing the task switch (also called context witch).

The first problem is what scheduling strategies are about, and this is where researchers regularly develop new schedulers. In the introductory theory part (Sections 8.1 and 8.2) we look at some classical strategies.

For Ulix we will implement the *Round Robin* scheduling strategy, but picking the next process or thread according to this strategy is rather simple, so the implementation part (Section 8.3) of this chapter focuses on the context switch: Switching from one task to the other without breaking the system is complex.

Round Robin

8.1 Monoprocessor Scheduling

For this book we focus on scheduling strategies that work on systems with exactly one CPU: Ulix does not support more than one processor. With several CPUs or cores (and even with hyperthreading) things get more interesting, and multiprocessor machines can profit from specialized scheduling strategies (even though most standard schedulers can be adapted to use more than one CPU, as well).

8.1.1 Quality Metrics

Scheduling is one of the best understood parts of operating systems because it has such an important impact on system performance. However, it is not so easy to say what the "best scheduler" is because it depends very much on the definition of quality used in a particular situation. Here are several common quality metrics for scheduling algorithms, the more historical ones are listed first:

CPU usage

- The metric of *CPU usage* is one of the simplest notions of quality in the literature. It basically gives the percentage of time in which the CPU actually executed application instructions (in contrast to operating system instructions or being idle). The CPU usage is important if CPUs are very expensive (as it was in earlier times). Today, the CPU usage of common desktop computers is usually very low, since they are idle most of the time.

throughput

- The *throughput* of a system usually counts the number of tasks that the system executed per time unit. This metric depends on the definition of "task". It comes from a time in which computers did batch processing: A number of computation jobs were ready in a physical entry queue (for example in the form of punched cards). The computer then started the processing of these jobs. The throughput counted the number of such jobs that the system could execute per hour (for example).

turnaround time

- The *turnaround time* of a thread is the time it takes for the thread to be scheduled again. In other words, it is the time between two successive selections of the thread by the scheduler. The turnaround time of the entire system is the average turnaround time of all threads. It can be regarded as a refined throughput metric.

waiting time

- The *waiting time* of a thread is the average time it has to wait in the ready queue before it is scheduled. This is not the same as throughput since times when the thread is blocked do not count in the waiting time.

response time

- The *response time* of a thread is the time it takes for the thread to respond to user input. This is similar to the turnaround and waiting time, only that responses to user inputs are counted instead of being scheduled again.

real time

- Finally, a scheduler is *real time* if it manages to satisfy real time constraints. There is a further differentiation into *hard real time* and *soft real time* which is basically about the question whether it is acceptable to occasionally miss a deadline.

8.1.2 Preemptive vs. Non-preemptive Scheduling

There are two main classes of scheduling algorithms: *preemptive* ones and *non-preemptive* ones. Roughly speaking, preemptive scheduling algorithms allow that a thread is thrown off the processor even if that thread does not want to be thrown off. In practice, all scheduling algorithms are usually preemptive in order to prevent that threads (accidentally or willingly) monopolize the system.

The precise definition is as follows: A scheduling algorithm is *preemptive* if an asynchronous interrupt can cause a thread to be taken off from the processor.

8.1.3 First-Come First-Served

The most simple approach to scheduling processes is to have a single queue for all ready processes. Whenever the CPU is not busy, the scheduler picks the first process in the queue and lets it compute until it either terminates or blocks. After a process becomes unblocked, it is appended to the end of the queue.

This is a non-preemptive strategy that is called *First Come, First Served* (*FCFS*) and that can be used on old machines which do not support timer interrupts. The most important problem with this approach is that it requires cooperation of all the running processes: If one process never freely gives up the CPU, it can go on forever.

A further analysis shows that the order in which processes enter the queue influences the average *service time* (the time between entering the CPU and finishing the calculation) heavily. For example, let's assume that there are three processes P1, P2 and P3 that simultaneously come into existence. P1 needs 15 units of time, P2 and P3 need four and three units, respectively. Figure 8.1 shows three of the six possible ordering in which the processes can enter the queue (and thus start computing).

service time

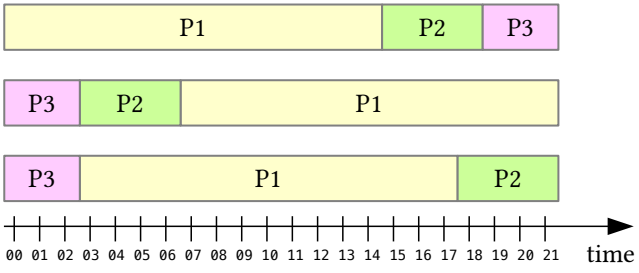


Figure 8.1: The FCFS scheduler's service time statistics depend heavily on the processes' order of system entry.

In the first execution sequence, P1, P2 and P3 finish after 15, 19 and 22 units of time, respectively. That means an average service time of $(15 + 19 + 22)/3 \approx 18.7$ units of time. In the second sequence, the times are 3, 7, 22, leading to $(3 + 7 + 22)/3 \approx 10.7$ units of time, and in the last sequence, times 3, 18, 22 result in $(3 + 18 + 22)/3 \approx 14.3$ units of time. Instead of the service time we could also look at the wait time (which would be the service time minus the burst time of the process) which gives a similar result.

So, if our goal was to minimize the average service time (or wait time), it would make sense to pick the ordering in the middle which sorts processes by their runtime.

8.1.4 Shortest Job First

There is a hypothetical strategy that does just that: The *Shortest Job First* (*SJF*) strategy always picks the job that has the shortest runtime. Thus, of all the possible orderings it will always choose the one in the middle of Figure 8.1.

Why did we call it hypothetical? The problem is that in almost every case the system has no way to find out how long the next CPU burst of some process is going to be. In some rare cases where a system only executes specially prepared applications which announce their next burst length in advance, this strategy might actually be implemented, but for multi-purpose operating systems which run arbitrary programs, it is not possible. Still, it is possible to approximate this strategy. For example the operating system could collect statistical data about each process by monitoring the length of each CPU burst. Then it could calculate averages for all the bursts of a process and order the processes by their average burst times. Programs might change their behavior over time; consider a program that performs heavy calculations on a large set of data: It would start by reading in a big chunk of data (resulting in very short CPU bursts). Once the data are there, it would start the calculation (with very long bursts). Afterwards it might write them back, returning to short bursts. So in order to cater for that variability, it would make sense to discard older statistical burst data and only use the last N burst times for calculating the average.

Also, once a process terminates, the system could store the collected statistical data about it in the filesystem: When starting the program again, it can make a better guess at what is about to happen.

Like FCFS, SJF is a non-preemptive strategy which does not interrupt processes. Both strategies are acceptable in non-interactive systems. But if a machine has a live user sitting in front of the machine who expects that several programs work seemingly in parallel, this is not good enough.

8.1.5 Round Robin

The idea behind the *Round Robin* (*RR*) scheduler is the same as that for FCFS—but with interrupts which force a process off the CPU once a time limit has been reached. The maximum time that a process is allowed to execute is called a *time slice*, and its length is an adjustable parameter of the RR strategy.

time slice

Figure 8.2 shows how an RR scheduler would treat the three sample processes from above. At the top you see the order of execution with the time slice set to four units of time, the lower part shows the sequence for a time slice of two units.

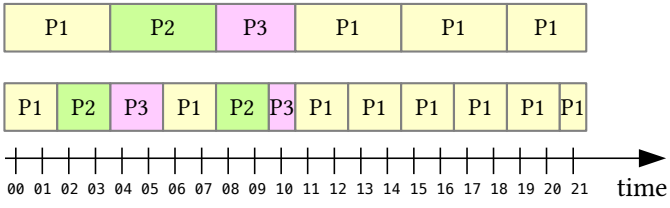


Figure 8.2: The Round Robin scheduler works with configurable time slices. Here it uses slices of four (top) and two units of time.

If a system already has an FCFS scheduler, it can easily be upgraded to an RR system: Just add a timer handler that checks how long a process has been active; if it exceeds the time slice, call the scheduler.

An open question is: What should the time slice (also called the *time slot* or the *quantum*) be? There are two adverse properties which make it non-trivial to make a decision: quantum

- On the one hand, we would like the time slice as small as possible because that guarantees that each process in the ready queue will wait only a short amount of time until it can start running. That is an important property for interactive systems that want to give their users the feeling that "things happen instantaneously". So this should lead to the rule: Make it short.
- On the other hand, after each time slice a context switch to the next process in the ready queue occurs (which is what we want). The downside is that this switch costs time. Let's assume that the context switch takes n units of time and that we have chosen the time slice to be the same n units of time. As a consequence the CPU time would be equally distributed between all the processes and the scheduler, resulting in a setup that runs with only half the possible speed because the other half is wasted by the scheduler. Obviously that is bad, so the time slice should be much larger than the context switch time. Here we get the rule: Make it long.

The answer must be some kind of compromise between the two. For interactive systems there is one property that we might be able to observe and that helps us pick the right amount: it is the typical time required to service a user interaction (like a pressed key or mouse button).

We define the *average interaction time* as follows: Assume that a process is currently blocked because it waits for an I/O event (such as a key being pressed). Once the key is actually pressed, the keyboard handler will move the process to the ready queue. The next time this process runs it will evaluate the character that was read in, and will act on that information somehow. The consequence of the pressed key will become visible, for example, an editor will display the character and move the cursor position, another program might open a menu or perform some other action. Once this observable reaction has occurred we stop the clock: The time between the reactivation of the process and now is the *interaction time* for this specific interaction. Now make a collection of several representative programs (those that are typically run on the operating system) and for each such program a collection of the typical interactions. For all those interactions measure the interaction time and then calculate the average. Add a few percent to that value to be on the safe side and use that final value as the length of the RR time slice. average interaction time

Figure 8.3 visualizes why this approach is helpful: It shows the treatment of one interaction by the RR scheduler; once with a time slice that was set as described above, once with a time slice that is just a bit too small. The yellow and green boxes represent the interactive process and a second process, and the striped black box shows the rest of a time slice which remained unused because the interactive process finished handling the interaction and blocked (waiting for the next key stroke). In the top part you can see the behavior that we want: The time slice is slightly larger than the interaction time which

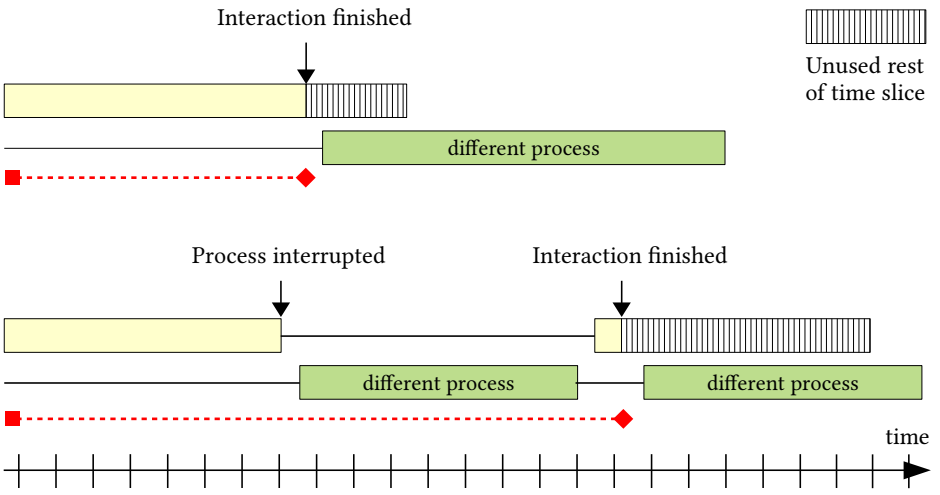


Figure 8.3: Picking a time slice that is too small has a bad influence on interactive processes (bottom). With the right choice handling an interaction can complete in one time slice.

means that the process can finish treatment of the interaction in that single slice. The bottom shows the alternative with a time slice that is just a bit too small: The process is interrupted before finishing its work on the interaction, then another process executes. We have created a benign example because there are only two processes; normally you would have to expect that there are several more so the picture would be spread much wider with *all* of the other process using their time slices before the interactive process gets its next chance to finish the task.

8.1.6 Virtual Round Robin

I/O-bound

CPU-bound

There is one problem with the RR strategy: It is unfair to *I/O-bound processes*. (We define a process to be I/O-bound if it performs I/O very often and thusly uses only small parts of each of its time slices, quickly blocking again. The opposite is a *CPU-bound process* which typically uses up its time slice completely. Obviously there is a gray are between I/O-bound and CPU-bound where is process can be called neither.)

Back to the point: RR treats I/O-bound processes unfairly because they typically use just a tiny fraction of their time slice and then block. Whatever I/O activity they perform, we can assume it to take quite some time (for example, disk access is pretty slow in comparison to CPU instructions, and waiting for a key stroke takes an indefinite amount of time). Once the I/O has completed, RR adds the process to the end of the ready queue. Then it has to wait until all other processes that stand before it in the queue have either used up their time slices or blocked. If you compare the ratio between the needed CPU time (the

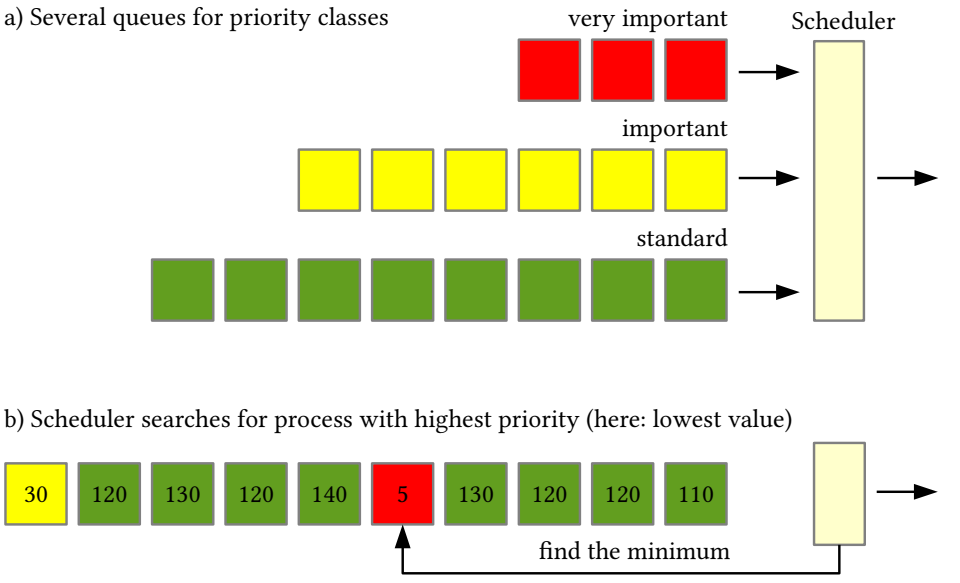


Figure 8.4: Priority-based schedulers have several possible implementations to pick from.

time that the process actually executes instructions on the CPU) with the wait time, then I/O-bound processes get a much smaller value than CPU-bound ones.

There is a modification of RR that alleviates this effect, and it is called *Virtual Round Robin* (*VRR*). It adds a second, privileged ready queue to the scheduler that only contains processes which had not fully used up their time slices when they ran the last time. If that queue is non-empty, a VRR scheduler will always pick a process from that priority queue when the current process' time slice runs out. However, it will not grant the newly chosen process a full time slice but only the rest that was not used up the last time.

8.1.7 Priority-Based Scheduling

We've already used the word "priority" when we discussed the extra queue of the VRR scheduler. Priority-based scheduling allows each process to be treated differently by making it more or less important than a process with default settings. This can be implemented in several ways. A priority-based scheduler might give an important process a longer time slice or it might pick it more often than other processes. Depending on how fine-grained priorities can be assigned to processes, there are several choices for handling the processes (see Figure 8.4):

- If the number of different priorities is low, we could manage a separate queue for each priority. In that case the scheduler would always start looking for the next process in the queue with the highest priority. Only if such a queue is empty it would look

starvation

down to the next queue. This can lead to *starvation* when one of the higher queues never empties, and there are mechanisms to solve that problem, such as increasing the priority of processes which have been waiting too long.

- Alternatively, if there are too many different priorities and we don't want the overhead of a corresponding queue collection, we can just store the priority as a numerical value in the process control block. Then the scheduler must search through the whole process table in order to find the (or one) process with the highest priority, and the queue is no longer a proper queue because the ordering in the queue is ignored by the scheduler (or only recognized if there are several processes with the highest priority). Again, such a system can lead to starvation and needs to provide a mechanism that prevents this.

static priorities

dynamic priorities

Furthermore, priorities can be static or dynamic: A scheduler with *static priorities* assigns a fixed priority to each process when it is created. It may be changed by a system call, but otherwise it remains constant throughout the lifetime of the process. When the scheduler uses *dynamic priorities* it regularly recalculates the priorities based on a set of rules. For example, such a scheduler may punish or reward a process for some specific observed behavior. Increasing the priority of processes which have been waiting for a long time (in order to avoid starvation) is an example for the use of dynamic priorities.

nice value

On Unix systems priorities are often expressed with a *nice value*. This does sometimes lead to confusion, because the "nicer" a process is, the lower is its priority. Unix provides a `nice` system call that can set an integer value that is roughly in the interval $-20 \ldots 20$. We write "roughly" because the exact values can differ from one Unix system to another; for example on a Linux machine the values -20 to 19 are valid, on OS X the range goes from -20 to 20. The nice value is used by each Unix system to calculate an internal priority, and often it is not possible to set a process to the highest internal priority by changing its nice value.

Ulix does not support priorities, but in Exercise 29 you can add that feature to the kernel.

8.1.8 Multi-Level Scheduling

Multi-level schedulers combine the characteristics of two or more other scheduling strategies. An example is a priority scheduler that uses three queues for standard, important and urgent processes (like the one in Figure 8.4, top) and then handles each of the queues like a Round Robin scheduler does.

A *Multi-Level Feedback Scheduler* for Ulix has been implemented by Markus Felsner as part of his Bachelor's thesis [Fel13] which is available online (in German). The code is based on Chapter 8 of the textbook by Remzi H. and Andrea C. Arpaci-Dusseau [ADAD14].

8.2 Multiprocessor Scheduling

If a machine has more than one CPU (or the processor has several cores, even virtual ones via hyperthreading), then things get more complicated for the scheduler. The first question is: Where should it run?

It is possible to restrict the whole kernel to run on a single, dedicated CPU that performs all the tasks which require ring 0 permissions. That would include the scheduler which would be activated regularly (via a timer, as on monoprocessor systems), and it could look at the processes running on all the CPUs and decide which processor needs a context switch to a different process. Models like these are also called *Master-Slave-Scheduling* because the dedicated (master) processor controls all the the other CPUs and distributes the workload. Such systems are useful for high-performance computing where machines sometimes have a large number of processors and applications require a specific numbers of CPUs to execute program threads simultaneously. In that case a scheduler will let the whole application (which needs to announce its processor requirements beforehand) wait until a sufficient number of slave processors becomes available and then create the requested number of threads and let them execute exclusively on the assigned CPUs. This is called *Gang Scheduling*.

master-slave-scheduling

gang scheduling

The alternative is that all processors are equals. In that case the operating system will still boot from a single processor, but during initialization it will start copies of the scheduler on each CPU. Those copies could all use the same strategy to find a new process for their local CPU, but special care must be taken so that the process queues remain consistent. For example, if two copies of the scheduler simultaneously look at the front of the ready queue, pick the same process and activate it, that process would execute twice.

Locking and other synchronization tasks become more complex when several CPUs are involved, and there's also the problem of *cache coherence* : In short, all CPUs have their own local cache and if those caches contain copies of the same memory region, then the system must make sure that changing the memory contents on one CPU invalidates the corresponding cache entries on all other processors.

cache coherence

Another important point that a multiprocessor scheduler must consider is the fact that moving a process from one processor to another one is costly because the old processor may still have parts of the process' memory in its cache (which would speed up memory access for that process) whereas the new processor's cache does not contain that memory. Modern operating systems provide the property *CPU affinity* that tells the scheduler to keep a process assigned to a CPU. However, the more constraints are added to a system (such as: process *A* must always run on CPU *X*), the harder it becomes to generate an equal load on all processors, and a load balancing algorithm is needed that may decide to move a process to a different CPU (with low load) even though this has some cost.

CPU affinity

The short introduction to scheduling on multiprocessors shall suffice for this book since we focus on monoprocessors and Ulix uses only one CPU as well.

8.3 Implementation of the Ulix Scheduler

Thanks to the forking mechanism of Section 6.5, we can already have more than one process in Ulix—but we still have to write code which lets Ulix switch between several tasks. This section is mainly about the task switch; the scheduling strategy that we use is Round Robin.

Understanding the switch basically means looking at the functions and stacks. When switching from process *A* to process *B*, we expect the following to happen:

1. Process *A* is executing, it runs in user mode, using its user mode stack.
2. A timer interrupt (IRQ 0) occurs. The CPU switches to kernel mode; this also switches the stack to the kernel stack. (Its address is stored in the TSS structure.) The CPU then jumps to the interrupt handler registered for interrupt 0 which does the following:

```
irq0:
    push  byte 0
    push  byte 32
    jmp   irq_common_stub

irq_common_stub:
    pusha
    push  ds
    push  es
    push  fs
    push  gs
    push  esp
    call irq_handler

    pop  esp
    pop  gs
    pop  fs
    pop  es
    pop  ds
    popa
    add  esp, 8
    iret
```

   So after pushing 0 (an empty error code) and 32 (that is 32+0, where 0 is the IRQ number) onto the stack, it saves all relevant registers on the stack and then calls `irq_handler`$_{146a}$ which is a C function:

```
void irq_handler (context_t *r) {
  ...
  handler = interrupt_handlers[r->int_no - 32];
  if (handler != NULL)  handler (r);
}
```

   The generic `irq_handler`$_{146a}$ looks up the correct interrupt service routine for the timer (it calculates `r->int_no-32` which in this case is $32 - 32 = 0$, finds the entry in `interrupt_handlers`$_{145b}$ (that is `timer_handler`$_{342b}$) and then calls it.
3. Next, the `timer_handler`$_{342b}$ checks whether it is time to call the scheduler and (if so) calls it:

```
if (system_ticks % 5 == 0) {
  ...
  scheduler (r, SCHED_SRC_TIMER);
```

```
  ...
}
```

The extra argument `SCHED_SRC_TIMER`$_{343a}$ tells the scheduler that is was called from the timer (which is the standard case).

4. So if it decides to call the scheduler, it enters

```
void scheduler (context_t *r, int source) {
  ...
}
```

which is the function that we have to implement.

The sequence is similar if the interrupted process was not in user mode but in kernel mode (in most cases because it was executing a system call), but in that case there is no switch from ring 3 to ring 0 since the system is already running in kernel mode. That is reflected by slightly different stack contents since an `iret` (interrupt return) from the timer handler must restore the mode that the process was operating in.

8.3.1 Stack Usage

We need to keep track of which stacks are in use and what contents are stored on these stacks. Every process has a private user mode stack and a private kernel mode stack.

1. When the current process runs (in user mode) and a timer interrupt occurs at time $t = 0$, the CPU checks the Task State Segment (TSS) to find the current top of the stack for kernel mode: it is stored in the `ESP0` entry. (It also retrieves the new value for the *SS* register; remember that we have different code/data segment descriptors for user and kernel mode.) It switches to the new stack (by changing the *ESP* and *SS* registers) and pushes the old values of *SS* and *ESP* as well as the contents of *EFLAGS*, *CS* and *EIP* onto the new stack. *EIP* is already set to the address of the next instruction of the process: the one that will be executed once we return to the process. Then it starts executing the interrupt handler code.

   The kernel stack now looks like in Figure 8.5.

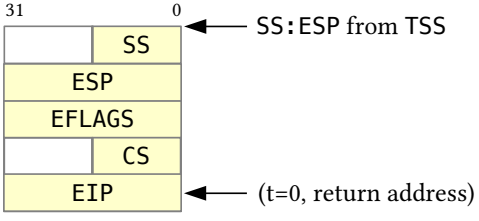


Figure 8.5: At $t = 1$ the kernel stack contains these values.

2. Then, at time $t = 1$, the interrupt handler entry function `irq0`$_{144}$ (for IRQ 0) pushes 0 and 32 onto the stack and jumps to `irq_common_stub`$_{144}$ which pushes *EAX*, *ECX*, *EDX*, *EBX*, *ESP* ($t = 1$), *EBP*, *ESI*, *EDI*, *DS*, *ES*, *FS* and *GS*.

   Finally, at time $t = 2$, it pushes the current *ESP* which is now properly set up as the address of the `context_t`$_{142a}$ structure holding all the registers (see Figure 8.6, also compare this to Figure 5.5 on page 143).

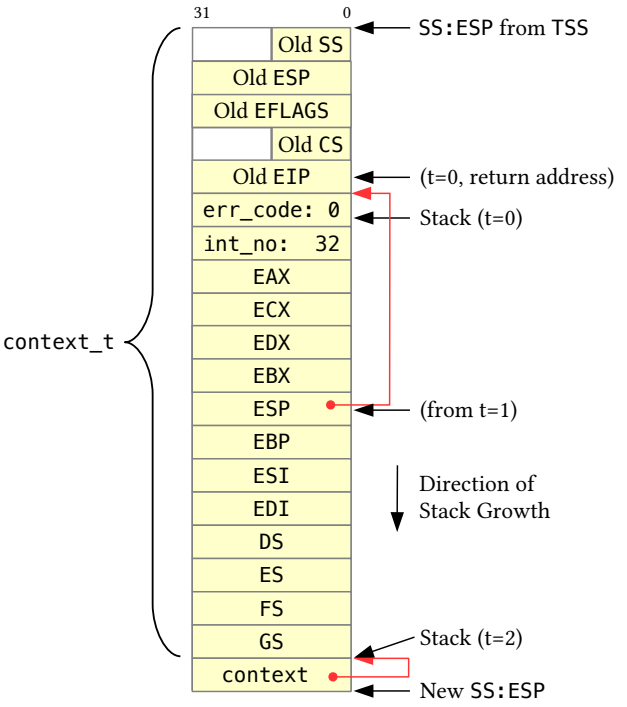


Figure 8.6: These are the stack contents just before executing `call irq_handler`.

   At $t = 3$, the instruction `call irq_handler`$_{146a}$ pushes the return address and jumps to the entry address of the C function's code.

3. When `irq_handler`$_{146a}$ starts, it expects to find two values on the stack: the return address and one argument. It takes a `context_t`$_{142a}$ * as argument and we have just prepared the stack so that it exactly fits this structure:

```
typedef struct {
  uint gs, fs, es, ds;
  uint edi, esi, ebp, esp, ebx, edx, ecx, eax;
  uint int_no, err_code;
  uint eip, cs, eflags, useresp, ss;
} context_t;
```

The function can then access the elements of the `context_t`$_{142a}$ structure.

4. `irq_handler`$_{146a}$ calls the C function `timer_handler`$_{342b}$ and passes the pointer to our `context_t`$_{142a}$ structure as its single argument.
5. Lastly, `timer_handler`$_{342b}$ calls `scheduler`$_{276d}$ (passing that pointer once more).
6. Now, `scheduler`$_{276d}$ is executing and it can access the register values which we've set up on the stack early in the assembler code and passed down all the way to the scheduler. Since we've only passed a pointer all the time (*call by reference*), the scheduler can modify the register values. After the whole call stack unwinds later and we're back in `irq_common_stub`$_{144}$, the `pop` instructions will write the modified values into the appropriate registers. Note however that somewhere inside `scheduler`$_{276d}$ an address space change will occur *that will also exchange the current kernel stack with the kernel stack that exists in the new address space* which needs to be prepared so that the whole stack unwinding works as if no switch had occurred. That is why we took special care to create the stack properly in the implementation of the forking mechanism.

Let's first assume that we do *not* enter the scheduler (because `system_ticks`$_{338a}$ `% 5` $\neq 0$). In that case we just return and do not modify anything relevant to scheduling—we expect the current process to continue running, as it does after other interrupt treatments.

If we've just entered the `scheduler`$_{276d}$, what does the stack look like right now? In comparison to Figure 8.6, there will be further return addresses and references to the `context_t`$_{142a}$ structure on the stack, since each function passes it, but we don't really have to care because all the important information is available via the pointer `r` to the `context_t`$_{142a}$.

Note again that at the beginning of the interrupt handling we stored the contents of all registers on the stack, in just the order which conveniently fits the `context_t`$_{142a}$ structure definition. We also pass the pointer to this structure to all further functions which get called (when calling `handler(r)` and then `scheduler`$_{276d}$`(r)`). So within the `scheduler`$_{276d}$ we can look at `r` to see the state as it was before the timer interrupt occurred.

Whether we've come from user mode or from kernel mode (e. g. from a process that was executing a system call when the timer interrupt fired) does not matter since all relevant registers have been saved and will be restored.

8.3.2 The Implementation

When we schedule, we select the new process and then store all registers (the data that `r` points to) in the old TCB. Then we switch the address space (the CPU's pointer to the page directory), and then we load the new TCB contents in the registers. After that we can return.

Our scheduler has the protoype

⟨*function prototypes* 45a⟩+≡ (44a) ◁259d 288▷ [275]

```
void scheduler (context_t *r, int source);
```

which—as expected—takes a `context_t`$_{142a}$ pointer as its first argument. We introduce the second argument `source` because we also call the scheduler from within other kernel func-

tions. (Right now we only discuss how it is called from the timer handler, but you have already seen us calling it from `syscall_resign`$_{221a}$ which was needed by `syscall_waitpid`$_{219c}$.)

The following two macros will allow the system to temporarily disable (and reenable) scheduling. During system initialization we've set `scheduler_is_active`$_{276e}$ to `false`; it is changed to `true` in the `start_program_from_disk`$_{189}$ function which creates the first process.

[276a] ⟨*enable scheduler* 276a⟩≡ (192d 564a 608b 610a)
```
scheduler_is_active = true;   _set_statusline ("SCH:ON ", 16);
```
Uses _set_statusline 337b and scheduler_is_active 276e.

[276b] ⟨*disable scheduler* 276b⟩≡ (151c 290b 321a 608b)
```
scheduler_is_active = false;  _set_statusline ("SCH:OFF", 16);
```
Uses _set_statusline 337b and scheduler_is_active 276e.

We declare two global variables in the kernel address space which will later come in handy when we have to remember information about the current and next process:

[276c] ⟨*global variables* 92b⟩+≡ (44a) ◁218b 276e▷
```
TCB *t_old, *t_new;
```
Defines:
t_new, used in chunks 210b, 212, 255c, 257c, 277–80, 425a, and 567.
t_old, used in chunks 210b, 255c, 257c, 277–79, and 425a.
Uses TCB 175.

These are not affected by changing the address space because they are in the region above `0xc0000000` which is identical in all address spaces. This is important! If those variables were defined locally in the `scheduler`$_{276d}$ function, they would reside in the kernel stack and get lost when the address space changes. Note that the scheduling code is a critical section, and there are two further exit points in the function at which we explicitly leave the critical section.

[276d] ⟨*function implementations* 100b⟩+≡ (44a) ◁260a 289a▷
```
void scheduler (context_t *r, int source) {
  ⟨begin critical section in kernel 380a⟩
  ⟨scheduler implementation 277a⟩
  ⟨end critical section in kernel 380b⟩
  return;
}
```
Defines:
scheduler, used in chunks 216b, 221a, 275, and 342d.
Uses context_t 142a.

Our implementation starts with checking whether there are any *zombie* processes and tries to get rid of them (see further down). Then it looks at the global variable

[276e] ⟨*global variables* 92b⟩+≡ (44a) ◁276c 292c▷
```
int scheduler_is_active = false;
```
Defines:
scheduler_is_active, used in chunks 206b, 276, 277a, 306d, 311a, 321, 329b, 334b, 335b, 412c, 416b, 509d, 510b, 512c, 518d, 521, 522, 531a, 532d, 545b, 588b, and 589a.

to determine whether it shall try to actually attempt a context switch.

⟨*scheduler implementation* 277a⟩≡ (276d) 277b▷ [277a]
```
⟨scheduler: check for zombies 281⟩      // deal with zombies if we have any
if (!scheduler_is_active) {             // check if we want to run the scheduler
  ⟨end critical section in kernel 380b⟩
  return;
}
```
Uses scheduler_is_active 276e.

With all obstacles removed, the real scheduling process can begin. The scheduler lets t_old$_{276c}$ point to the current process and then finds out which process it should switch to, storing the result in t_new$_{276c}$. Then it performs the context switch, and before returning to the new current process it checks whether that process has any pending signals and whether there is some clean-up to be done for terminated processes. (This last step could be handled elsewhere, for example via one of the ⟨*timer tasks* 306d⟩.)

⟨*scheduler implementation* 277a⟩+≡ (276d) ◁277a [277b]
```
t_old = &thread_table[current_task];
debug_printf ("SCHED: enter find next\n");
⟨scheduler: find next process and set t_new 278a⟩
debug_printf ("SCHED: leave find next\n");
if (t_new != t_old) {
  ⟨scheduler: context switch 279c⟩
}
⟨scheduler: check pending signals 567b⟩ // see chapter on signals
⟨scheduler: free old kernel stacks 169a⟩  // if there are any
```
Uses current_task 192c, debug_printf 601d, t_new 276c, t_old 276c, and thread_table 176b.

We will implement the code chunk ⟨*scheduler: check pending signals* 567b⟩ in Chapter 14 where we introduce signals. Note that the ⟨*scheduler: find next process and set* t_new 278a⟩ chunk implements the scheduling logic, whereas all the other code is about technical details of the context switch.

8.3.2.1 A Simple Round-Robin Strategy

Ulix does not attempt any sophisticated scheduling strategy; we will just use a simple Round Robin system. The search for the next process will normally use the next pointer in the current TCB since it will point to the next TCB in the ready queue. However there's a special case when the scheduler was activated because the current process called waitpid$_{220d}$; in that case the former current process has already been moved to a blocked queue and we need to start over. We can recognize that case by evaluating the source argument and looking at the state of the current process. If it is not TSTATE_READY$_{180a}$ then we must start over (with the first element of the ready queue which is stored in tid = thread_table$_{176b}$[1].next).

We also might come across the idle process (which has the thread ID 1); if so we skip it in the search of "real" processes that need some work done. (We could have left that process completely out of any queues since it will never block but always be ready.)

[278a] ⟨*scheduler: find next process and set* `t_new` 278a⟩≡ (277b) 278b▷
```
thread_id tid;
search:  // goto label
if (source == SCHED_SRC_RESIGN && t_old->state != TSTATE_READY) {
  // we cannot use the ->next pointer
  tid = thread_table[0].next;
} else {
  tid = t_old->next;
}
if (tid == 1) tid = thread_table[1].next;    // ignore idle process
```
Uses SCHED_SRC_RESIGN 343a, t_old 276c, thread_id 178a, thread_table 176b, and TSTATE_READY 180a.

If `tid` is 0, we have reached the end of the queue—or the queue may be completely empty (in which case we activate the $\texttt{idle}_{282f}$ process):

[278b] ⟨*scheduler: find next process and set* `t_new` 278a⟩+≡ (277b) ◁278a
```
if (tid == 0)                    // end of queue reached
  tid = thread_table[1].next;
if (tid == 0)                    // still 0? run idle task
  tid = 1; // idle
t_new = &thread_table[tid];
if (tid > 1 && (t_new->addr_space == 0 || t_new->state != TSTATE_READY)) {
  goto search; // continue searching
}
// found it
```
Uses t_new 276c, thread_table 176b, and TSTATE_READY 180a.

8.3.2.2 The Context Switch

Before implementing the actual context switch, let's first observe the following facts:

- We only enter the scheduler (and thus also the context switcher) via timer interrupts or when a functions resigns (i. e., freely gives up the CPU by calling $\texttt{resign}_{221f}$) or calls $\texttt{waitpid}_{220d}$.
- Once we're running inside the scheduler, we know that the kernel stack has been set up in a way that will allow the system to continue operation of the interrupted process—whether it was running in user mode or kernel mode before the interrupt occurred.
- When we switch the address space, we also switch the kernel stack. However, the stack pointer register *ESP* will still point to the top of the old process' kernel stack. We need to remedy that and have it point to the top of the new process' kernel stack.
- If we switch between threads (within one process) we need not change the address space.

To make the code more readable, we provide some macros which copy values between variables and the *ESP*, *EBP* and *CR3* registers. They require the use of inline assembler code (see Appendix B.4).

⟨ *macro definitions* 35a ⟩+≡ (44a) ◁222c 340a▷ [279a]

```
#define COPY_VAR_TO_ESP(x)  asm volatile ("mov %0, %%esp" :         : "r"(x) )
#define COPY_VAR_TO_EBP(x)  asm volatile ("mov %0, %%ebp" :         : "r"(x) )
#define COPY_ESP_TO_VAR(x)  asm volatile ("mov %%esp, %0" : "=r"(x)          )
#define COPY_EBP_TO_VAR(x)  asm volatile ("mov %%ebp, %0" : "=r"(x)          )
#define WRITE_CR3(x)        asm volatile ("mov %0, %%cr3" :         : "r"(x) )
```

Defines:
COPY_EBP_TO_VAR, used in chunk 279c.
COPY_ESP_TO_VAR, used in chunk 279c.
COPY_VAR_TO_EBP, used in chunk 279c.
COPY_VAR_TO_ESP, used in chunk 279c.
WRITE_CR3, used in chunk 279.

Now we can present the code that handles the actual context switch. It is only executed if we truly switch, i. e., if $\texttt{t\_new}_{276c} \neq \texttt{t\_old}_{276c}$. In principle, we would expect something along the lines of

⟨ *scheduler: context switch (simplified version)* 279b ⟩≡ [279b]

```
t_old->regs = *r;                   // store old:   registers
COPY_ESP_TO_VAR (t_old->esp0);      //              esp (kernel)
COPY_EBP_TO_VAR (t_old->ebp);       //              ebp

current_task = t_new->tid;          // update values of current_{task,as,pd}
current_as   = t_new->addr_space;
current_pd   = address_spaces[t_new->addr_space].pd;
WRITE_CR3 ( mmu (0, current_pd) );  // activate address space

COPY_VAR_TO_ESP (t_new->esp0);      // restore new: esp
COPY_VAR_TO_EBP (t_new->ebp);       //              ebp
*r = t_new->regs;                   //              registers
```

but it is a little more complicated since we also have to check whether we do want to change the address space (if not, then we can save some CPU time by omitting that step), and we also need to change the TSS's `esp0` entry and handle some special cases for newly created processes or threads. So the actual code is a bit more complex:

⟨ *scheduler: context switch* 279c ⟩≡ (277b) 280a▷ [279c]

```
t_old->regs = *r;                   // store old:   registers
COPY_ESP_TO_VAR (t_old->esp0);      //              esp (kernel)
COPY_EBP_TO_VAR (t_old->ebp);       //              ebp
current_task = t_new->tid;
if (current_as != t_new->addr_space) {
  // we need to change the address space (switching process, not thread)
  current_as   = t_new->addr_space;
  current_pd   = address_spaces[t_new->addr_space].pd;
  WRITE_CR3 ( mmu (0, (memaddress)current_pd) );   // activate address space
}
COPY_VAR_TO_ESP (t_new->esp0);      // restore new: esp
COPY_VAR_TO_EBP (t_new->ebp);       //              ebp       chunk continues ->
```

Uses address_spaces 162b, COPY_EBP_TO_VAR 279a, COPY_ESP_TO_VAR 279a, COPY_VAR_TO_EBP 279a, COPY_VAR_TO_ESP 279a, current_as 170b, current_pd 105a, current_task 192c, memaddress 46c, mmu 172a, t_new 276c, t_old 276c, and WRITE_CR3 279a.

We must update the TSS structure and enter the address of the kernel stack; that is either `TOP_OF_KERNEL_MODE_STACK`$_{159c}$ (for a pure process or the primary thread of a multi-threaded process, i. e., one with `pid == tid`) or it is `t_new`$_{276c}$`->top_of_thread_kstack` for a non-primary thread (`pid != tid`):

[280a] ⟨*scheduler: context switch* 279c⟩+≡ (277b) ◁279c

```
// set TSS entry esp0 to top of current kernel stack and flush TSS
if (t_new->pid != t_new->tid) {
  // thread kstack information is stored in the TCB
  write_tss (5, 0x10, t_new->top_of_thread_kstack);       // non-primary thread
} else {
  // process kstack is a fixed value
  write_tss (5, 0x10, (void*)TOP_OF_KERNEL_MODE_STACK);   // primary thread
}
tss_flush ();

// show thread ID in status line
if (t_new->tid != 1) {  // ignore switch to idle
  char msg[4]; sprintf (msg, "%03x", t_new->tid);
  _set_statusline (msg, 20);
}

⟨scheduler: switching to a fresh thread? 280b⟩     // check special case
*r = t_new->regs;                 // restore new: registers
```

Uses `_set_statusline` 337b, `kstack`, `sprintf` 601a, `t_new` 276c, `TCB` 175, `TOP_OF_KERNEL_MODE_STACK` 159c, `top_of_thread_kstack`, `tss_flush` 197c, and `write_tss` 197a.

Remember: `write_tss`$_{197a}$ sets the `ESP0` element of the TSS structure. It is only used when the CPU switches from ring 3 to ring 0 (in general: whenever it switches from ring 1–3 to ring 0, but Ulix does not use rings 1 and 2), and that means we can always start with an empty kernel stack in those situations: Thus we can always write the address of the stack's top into that element.

Finally, we need to consider the special case of switching to a freshly created thread: On page 258 we have prepared the new thread so that we can immediately leave with the `iret` instruction.

[280b] ⟨*scheduler: switching to a fresh thread?* 280b⟩≡ (280a)

```
if (t_new->new && t_new->tid != t_new->pid) {
  // new thread
  t_new->new = false;
  ⟨end critical section in kernel 380b⟩
  asm ("iret");     // return from interrupt handler, do not update r
}
```

Uses `t_new` 276c.

In this case the rest of the `scheduler`$_{276d}$ function (the code chunks ⟨*scheduler: check pending signals* 567b⟩ and ⟨*scheduler: free old kernel stacks* 169a⟩) is not executed, but that is no problem; we can handle that with the next invocation of the scheduler.

8.3.2.3 Treating Zombie Processes

We still need to check for *zombies*: A zombie is a terminated process whose parent process has not yet been able to retrieve the return value (supplied via `exit`$_{218a}$). There are two possible cases:

1. The parent is waiting. This means that it has called `waitpid`$_{220d}$ *after* this process turned into a zombie, because otherwise it would not exist anymore. In that case we deblock the parent process and delete the zombie's entry in the thread table.
2. The `ppid` ID of the zombie process was set to 1. That means that its true parent exited without calling `waitfor`. Here, we can also remove the zombie, and there is nothing else to do: No process is waiting (or will ever be) for that process.

⟨*scheduler: check for zombies* 281⟩≡ (277a) [281]

```
for (thread_id pid = 0;  pid < MAX_THREADS;  pid++) {
  if (thread_table[pid].state == TSTATE_ZOMBIE) {
    thread_id ppid = thread_table[pid].ppid;

    // case 1: parent is waiting
    if ( (thread_table[ppid].state == TSTATE_WAITFOR) &&
         (thread_table[ppid].waitfor == pid) ) {
      deblock (ppid, &waitpid_queue);
      thread_table[pid].state = TSTATE_EXIT;
      thread_table[pid].used = false;
    }

    // case 2: parent ID was set to 1 (idle_)
    if ( ppid == 1 ) {
      thread_table[pid].state = TSTATE_EXIT;
      thread_table[pid].used = false;
    }
  }
}
```

Uses deblock 186b, MAX_THREADS 176a, thread_id 178a, thread_table 176b, TSTATE_EXIT 180a, TSTATE_WAITFOR 180a, TSTATE_ZOMBIE 180a, and waitpid_queue 218b.

8.3.3 Letting the `init` Process Idle

As a last topic for this chapter we discuss the `idle` process with process ID 1. It starts as the `init` process, but once it has spawned some other processes, it will turn into the `idle` process. If it becomes active it shall do nothing. However, if we interpret "nothing" as an empty infinite loop (`for(;;);`) then the system will always actively spin in this loop whenever no other process is ready—that uses processor power. `init` → `idle`

There is a better way: The assembler instruction `hlt` (halt) can stop the CPU until the next interrupt occurs. We provide a system call that executes this instruction and use it in the `idle` process:

[282a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁258a 298c▷
```
void syscall_idle (context_t *r);
```

[282b] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁235a 298b▷
```
#define __NR_idle 505
```
Defines:
`__NR_idle`, used in chunk 282.

[282c] ⟨*syscall functions* 174b⟩+≡ (202b) ◁258b 299a▷
```
void syscall_idle (context_t *r) {
  ⟨enable interrupts 47b⟩
  asm ("hlt");
  ⟨disable interrupts 47a⟩
}
```
Defines:
`syscall_idle`, used in chunk 282.
Uses `context_t` 142a.

[282d] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁260c 299b▷
```
install_syscall_handler (__NR_idle, syscall_idle);
```
Uses `__NR_idle` 282b, `install_syscall_handler` 201b, and `syscall_idle` 282c.

In the user mode library we add an `idle`$_{282f}$ function:

[282e] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁260d 299c▷
```
inline void idle ();
```

[282f] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁260e 299d▷
```
inline void idle () { syscall1 (__NR_idle); }
```
Defines:
`idle`, used in chunk 282.
Uses `__NR_idle` 282b and `syscall1` 203c.

When the system starts, `start_program_from_disk`$_{189}$ loads the `/init` program from disk whose source code we have already shown on page 191. It starts the program `/bin/login` (via `execv`$_{235e}$) which in turn launches some new processes (`/bin/swapper`, see page 311, and a few login processes that let users log in on the virtual consoles). When that is done it becomes the `idle` process via

[282g] ⟨*init to idle transformation* 282g⟩≡
```
setpsname ("[idle]");
for (;;) {
  idle ();   // if we don't call idle, we will have 100% CPU usage
}
```
Uses `idle` 282f.

8.4 Exercises

26. **Scheduling with the [Esc] Key**

Tutorial 7

The `tutorial/07/` folder contains a new version of the mini ULIX kernel which includes the `fork()` and `schedule()` functions. Again, it is a literate program (`ulix.nw`). However the scheduler is never called: Normally the timer handler would have to do that, but it is not part of that kernel version. In this exercise you introduce a manual scheduling that can be initiated by pressing the [Esc] key.

a) Look at the user mode program `test.c` whose machine code version is already part of the kernel sources. Its `main()` function creates a new process via `fork()`, afterwards parent and child write "P" (for parent) or "C" (child) on the screen in infinite loops:

```
int main () {
  printf ("Hello - User Mode!\n");
  int pid = fork ();
  for (;;) {
    if (pid == 0) printf ("C");  // child
    else          printf ("P");  // parent
  }
}
```

Start the ULIX version (using `make` and `make run`). You will see that the system displays only "V"s, the child process does not run. (If you compile the same program for Linux and run it, you see alternating sequences of "P"s and "C"s.)

b) The scheduler is implemented in the `scheduler()` function, just like it is in the real ULIX kernel. You can test it by adding a check for a pressed [Esc] key to the keyboard handler. That key has scan code 1. Locate the `keyboard_handler()` function and add the following test for scan code 1 right after it was read with `inportb()`:

```
if (s == 1) {
  scheduler (r, 0);
  return;
}
```

When you recompile and start the kernel, you will see "P"s again. But pressing [Esc] will switch to the child process, so that the output sequence changes to "C"s. Every further time you press [Esc], the process will switch again.

27. **Calling the Scheduler from the Timer Handler**

Now you add a timer handler which will regularly call the scheduler and thus automatize the multi-tasking.

a) Start with removing the scheduler call from the keyboard handler that you added in the previous exercise; you can simply turn it into a comment.

b) We also need the timer handler to update a global `ticks` variable so that we can see how long the system has been running. Define it like this:

```
unsigned long int ticks
```

and initialize it to 0. (Which code chunk to you have to append to in order to declare and initialize variables?) Don't add this code in earlier definitions of the chunks but create your own section at the end of the document where you put additions to the earlier chunk definitions.

Implement a function `timer_handler` with the same prototype that all the other interrupt handlers have, e. g., the keyboard handler. Its first task is to increment the `ticks` variable. Afterwards call the scheduler:

```
scheduler (r, 0);
```

With these changes the timer is already functional, but you still have to register it and enable the timer interrupt. Write an addition to the code chunk ⟨*kernel main: initialize system* ⟩ where you put appropriate function calls. (You can check how Ulix does that for the keyboard handler.) The interrupt number for the timer is defined as $\texttt{IRQ\_TIMER}_{132}$ (and has the value 0).

Compile and start the modified kernel. Now the output of "V"s und "S"s should switch automatically: Every time the hardware generates a timer interrupt, Ulix will switch between the two processes.

c) As we discussed in this chapter, task switches cost time. It makes sense to let a process run for a longer time before the scheduler takes away the CPU and gives it to another process. You can achieve this behavior by only calling the scheduler if `ticks` is some multiple of a given number, e. g. 5. With

```
(ticks % 5 == 0)
```

you can check whether `ticks` is a multiple of 5. Call the scheduler only when that condition evaluates to true. That reduces the amount of scheduler invocations (per time) to a fifth of what it was before. You can also test how different values (e. g. 25, 100) change the overall behavior.

d) The current code in `scheduler()` supports exactly two processes which use the TCBs 1 and 2. That makes it rather simple to determine which process comes next. How would you have to modify the function so that it supports exactly four processes instead of two?

Tutorial 8

Until now all exercises used a stripped-down version of the Ulix kernel which evolved from exercise to exercise, mimicking the progress throughout the book. For the following two tasks you will work with real kernel sources. This is not the version that is presented in the book (because the exercises were developed before the Ulix implementation was complete), but it is a usable system with full user mode and filesystem support. You will now add new capabilities to the system.

Use the code that you find in the `/home/ulix/ulix/` folder for the following two exercises.

28. **Boost: Run Only This Process**

The scheduler which is called by `timer_handler` every five ticks chooses the next process in the ready queue. In this exercise you give a process a chance to prevent this switch (so that it can go on longer). By using the function

```
void boost (int n);
```

it shall be able to set a global variable (global in the ULIX kernel). The timer handler shall then use one of the ⟨*timer tasks* 306d⟩) to check whether this value is 0. If not, it decrements the variable and performs no context switch.

a) Declare a global variable `int boost_count` and initialize it to 0. As in the last exercise you need to find the appropriate code chunks for these two actions.

b) Move the ⟨*timer tasks* 306d⟩ code chunk that calls `scheduler()` into your own section of the literate program file so that you can document the changes to the chunk. It is this chunk:

```
<<timer tasks>≥
// Every 5 clocks call the scheduler
if (system_ticks % 5 == 0) {
  [...]
  scheduler (r, SCHED_SRC_TIMER);  // defined in the process chapter
  [...]
```

c) Modify the `if` clause; the extra condition `boost_count == 0` must also be fulfilled. You will also need to add an `else` case which decrements `boost_count`.

d) Write a syscall handler

```
void syscall_boost (context_t *r);
```

that reads the right processor register (which one is that?) and copies its value to `boost_count` (if it is positive or 0). Define a syscall number constant `__NR_boost` (using a syscall number that is not yet in use) and add an `install_syscall_handler`$_{201b}$`()` call to the right code chunk.

e) Write a user mode library function

```
void boost (int n);
```

that uses `syscall2`$_{203c}$`()` to make the system call. You can look at the implementation of `open()` to see how that can be done.

f) Finally, write a small user mode application that lets you test the effect of calling `boost()`.

(Note that you need not implement a corresponding `u_boost` function which gets called by `syscall_boost`: It would just contain the one line

```
void u_boost (int n) { boost_count = n; }
```

so there is no point in writing an extra function for that task.)

29. **Process Prioritization**

All ULIX processes (or threads) receive equal treatment by the ULIX scheduler because the system does not know priorities. In this exercise you add the classical priority mechanism present in all Unix systems.

a) Add a nice value (`int nice`) to the thread control block that will hold values between -20 and 19, the default value is 0. If a process forks, it will pass the nice value on to the child process.

b) Implement a function `int u_setpriority (int nice)` that can be used for changing the current process' nice value to the supplied value. You will also need corresponding functions `syscall_setpriority` in the kernel and `int setpriority (int nice)` in the user mode library. (The return value of `u_setpriority` or `setpriority` is the new nice value.) The prototype differs from the `setpriority()` function on Linux systems, but looking at the man page on a Linux box can still be helpful.

c) In the ⟨*timer tasks* 306d⟩ chunk, modify the code block

```
if (system_ticks % 5 == 0) {
  [...]
  scheduler (r, SCHED_SRC_TIMER);  // defined in the process chapter
  [...]
```

so that the decision whether the scheduler is called depends on the current process' nice value (that you can find in $\texttt{thread\_table}_{176b}\texttt{[current\_task}_{192c}\texttt{].nice}$). You have several options for doing this, but the result should be that a process with a lower nice value receives a longer time slice than a process with a larger nice value.

d) Write a test program that lets you check whether changing the nice values really changes the behavior of the process. You will need at least two processes (one that calls `setpriority` and one that does not) in order to observe any effect.

9
Handling Page Faults

Recall what happens every time the machine accesses a memory address, either for receiving the next instruction or for reading or storing data: the address that the processor asks for is a virtual address, and it must first be translated to a physical address by the MMU. Special register `CR3` tells the MMU what page directory to use, and that directory reveals the location of a page table which is in turn accessed to (finally) find the physical frame number and calculate the complete physical address by adding the offset (within the page).

In many cases this procedure will fail at some point, typically because either the page directory or the page table contains an entry which tells the MMU that the page does not exist in memory. This means that address translation cannot continue, and the CPU raises a *page fault*.

page fault

Every operating system needs to handle such page faults, the minimum action that is required is either killing a process (which has tried to access an illegal address) or halting the operating system (if the faulting instruction occurred inside code which was not executed on behalf of some process). But we expect our fault handler to be better than that and also handle the following situations:

- **User mode stack grows**: When a user mode program uses recursion or makes extensive use of the stack for other reasons, it will soon cross into a memory range just below the reserved stack space and cause a page fault. In that case the process can rightly expect the stack to grow automatically. Thus, ULIX will check whether a memory area just below the current end of the stack was accessed—and if so, increase the stack by one page. Afterwards, execution of the process can resume.

automatic stack growth

- **Access to page which was paged out**: In Sections 9.2 ff. we will introduce a swap file[1] and show code for moving pages of memory to disk and back. That will become necessary when the whole physical memory is in use and there are no further free page frames.

9.1 The Ulix Page Fault Handler

In this section we present the page fault handler that we implemented for Ulix. The handler function has the prototype of all other fault handlers:

[288] ⟨*function prototypes* 45a⟩+≡ (44a) ◁275 293c▷
```
void page_fault_handler (context_t *regs);
```

and we originally started by taking code from James Molloy's kernel tutorial [Mol08, Ch. 6.4.5] (which just gives some information about the faulting reason and address) and then added some features which make it usable in Ulix.

CR2, err_code

The *CR2* register holds the virtual address which caused the page fault, and the `err_code` element of the context tells us more about why the access to this address failed, so this is the first information we need to gather. Individual bits of `err_code` describe whether the page that was accessed is present, read-only or only accessible in kernel mode.

Then we check the possible reasons for a page fault and handle them as well as we can:

- First, if the page was paged out to disk (see next section), we bring the page back in. In that case we can simply leave the fault handler with `return`, and the running process will resume its operation, repeating the instruction that caused the fault at the first attempt. (Actually `return` jumps back into the generic $\texttt{fault\_handler}_{151c}$ function which then returns to $\texttt{fault\_common\_stub}_{150b}$ in `start.asm`, and the transition back to the process occurs in the assembler code via the `iret` instruction.)
- Then we check whether the process tried to access an address directly below the user mode stack. That will often occur when a function calls itself recursively. In that case we increase the stack and also `return`.

These two conditions are recoverable, but there are other situations in which we either have to kill the faulting process or—worst case—permanently disable user mode and jump to a safe kernel function such as the kernel shell that we have included for debugging purposes.

- If a process tried to access an invalid address, and we can neither bring it back by paging in a paged-out page nor can we help the situation by increasing the user mode stack, then it was likely caused by a programming error and we need to kill the process. We check that condition by testing whether the faulting instruction's address

[1] Note that we use the term "swap file" and not "page file". Ulix does not implement swapping which is an older technique where whole processes are removed from memory. Instead we move single pages to disk and back, but historically a file or a partition used for swapping was called swap file/swap partition, and that name lives on in paging systems, e. g. in Linux and other Unix-like systems.

is below the kernel's memory address range (`r->eip < 0xc0000000`). In that case the system can continue running, minus the killed process.

- Last, we must assume that an error in the kernel caused the page fault. Then there's nothing to do since the failed instruction is part of the kernel code and there is no simple way to resume after such a problem. We might try to write a memory dump to disk or provide some other means that might help debugging the code, but here we just jump to the kernel shell (from which we cannot return to user mode anymore). If that kernel shell was not available, we would simply halt the machine completely.

Our implementation of the page fault handler

⟨*function implementations* 100b⟩+≡ (44a) ◁276d 293d▷ [289a]
```
void page_fault_handler (context_t *r) {
  ⟨page fault handler implementation 289b⟩
}
```
Defines:
page_fault_handler, used in chunks 151c and 288.
Uses context_t 142a.

starts with gathering all the available information:

⟨*page fault handler implementation* 289b⟩≡ (289a) 289c▷ [289b]
```
memaddress faulting_address;
asm volatile ("mov %%cr2, %0" : "=r" (faulting_address));   // read address
int present   = !(r->err_code & 0x1);   // page present?
int rw = r->err_code & 0x2;             // attempted to write_?
int us = r->err_code & 0x4;             // CPU in user-mode (ring 3)?
int reserved = r->err_code & 0x8;       // overwritten CPU-reserved bits of
                                        // page entry?
int id = r->err_code & 0x10;            // caused by an instruction fetch?

⟨page fault handler: check if page was paged out 298a⟩   // see next section
```
Uses memaddress 46c.

In the last line from above it checks whether the page was paged out to disk; we will describe that in the following section where we introduce the swap file.

The second benign case of a page fault occurs when the user mode stack needs to grow. We can check that condition by looking at the current end of the user mode stack and calculating whether adding one extra page would solve to problem—if so, we give the process that extra page. Otherwise (if the address is too far away from the stack) we cannot help this process:

⟨*page fault handler implementation* 289b⟩+≡ (289a) ◁289b 290a▷ [289c]
```
if (faulting_address ≤ TOP_OF_USER_MODE_STACK &&
    faulting_address ≥
    TOP_OF_USER_MODE_STACK-address_spaces[current_as].stacksize - PAGE_SIZE) {
  ⟨page fault handler: enlarge user mode stack 291⟩   // user mode, stack
  return;
}
```
Uses address_spaces 162b, current_as 170b, PAGE_SIZE 112a, and TOP_OF_USER_MODE_STACK 159b.

Note that this restricts user mode processes in the way they use the stack. For example, you cannot write a function that takes so many (or so large) arguments that its invocation would increase the stack by more than one page. So depending on the kinds of applications you want to run on Ulix, you might want to modify the above code chunk.

If neither of the first two cases is applicable, then there's a real problem. With the rest of the code we try to determine how big that problem is: It may be sufficient to kill the current process, or we may have to halt the system. First we write an error message to the screen. Here, we also use the old code chunk ⟨*fault handler: display status information* 152a⟩ from the generic fault handler that we implemented earlier in Chapter 5.3.

The same chapter also contains the ⟨*fault handler: terminate process* 152b⟩ chunk that we recycle when we decide to kill the current process. We do that if the faulting address is in the user space of virtual memory, i. e., below `0xc0000000`. The old code chunk simply removes the process from the ready queue and calls $\texttt{syscall\_exit}_{216b}$. The `exit` system call handler will then return the process' resources and launch the scheduler which finally picks another process.

[290a] ⟨*page fault handler implementation* 289b⟩+≡ (289a) ◁289c 290b▷

```
printf ("Page fault! ( ");              // write error message.
if (present) printf ("present ");    if (rw) printf ("re" "ad-only ");
if (us)      printf ("user-mode ");  if (id) printf ("instruction-fetch ");
printf (")\n");
⟨fault handler: display status information 152a⟩
printf ("address = 0x%08x. current_task = %d. current_as = %d.\n",
        faulting_address, current_task, current_as);
hexdump (r->eip & 0xFFFFFFF0, (r->eip & 0xFFFFFFF0)+128 );

if ((memaddress)(r->eip) < 0xc0000000) { ⟨fault handler: terminate process 152b⟩ }
```

Uses `current_as` 170b, `current_task` 192c, `hexdump` 612c, `memaddress` 46c, and `printf` 601a.

(If you are curious why the string `"read-only"` is split into `"re" "ad-only"` in the above chunk, read the footnote.[2])

Finally, if the page fault was not caused by a process that tried to access an illegal address, then we must assume we've come across a kernel bug. There's no way to recover, because where should the system continue execution? Our last remaining option is to stop the system. Ulix provides a kernel mode shell that can be used for debugging, instead of a real full stop we jump into that function, but the user mode is gone for good.

[290b] ⟨*page fault handler implementation* 289b⟩+≡ (289a) ◁290a

```
// error inside the kernel; cannot fix, leave user mode
⟨disable scheduler 276b⟩
⟨enable interrupts 47b⟩
printf ("\n"); asm ("jmp kernel_shell");   // jump to the kernel shell
```

Uses `kernel_shell` 610a and `printf` 601a.

[2] Sometimes we need to outwit the automatic cross-referencer of the literate programming software. For example, it would detect `read` and misinterpret it as a reference to the $\texttt{read}_{429b}$ function and thus add an entry to the "Uses:" block. We avoid this by either splitting strings or, in case of comments, adding an underscore: `read_` will not be (mis-)detected as $\texttt{read}_{429b}$.

9.1.1 Enlarging the Stack

When we notice that the stack's size is causing the problem, we can grow it. In the above code detection of necessary stack growth uses the fact that the stack grows linearly; we assume that an illegal access to the stack (i. e., to a page that has not been mapped to a frame yet) always occurs directly below the last valid stack page. So, the address has to be below the top of the stack, but above the lowest valid address minus PAGE_SIZE$_{112a}$.

We can then simply grow the stack by mapping the next page (remembering that the stack grows downwards) to a fresh frame:

⟨*page fault handler: enlarge user mode stack* 291⟩≡ (289c) [291]

```
memaddress new_stack = TOP_OF_USER_MODE_STACK;
new_stack -= address_spaces[current_as].stacksize;
int pageno = new_stack / PAGE_SIZE - 1;
int frameno;
if ((frameno = request_new_frame ()) < 0) {
  printf ("\nERROR: no free frame, cannot grow user mode stack\n");  // error
  ⟨fault handler: terminate process 152b⟩
};

as_map_page_to_frame (current_as, pageno, frameno);   // update page table and
address_spaces[current_as].stacksize += PAGE_SIZE;    // TCB stack size entry
```

Uses address_spaces 162b, as_map_page_to_frame 165b, current_as 170b, memaddress 46c, PAGE_SIZE 112a, printf 601a, request_new_frame 118b, TCB 175, and TOP_OF_USER_MODE_STACK 159b.

Note that this code only works for a thread-less process. If a process consists of several threads, and one of the non-primary threads exceeds its user mode stack, this code will not be executed because it only checks for problems with the primary thread's stack. Our design does not allow growable stacks for the extra stacks because we have placed those stacks close to each other in virtual memory. We could increase the free spaces between thread stacks to make the problem a little smaller, but in the end there must always be a limit to the threads' stack sizes—after all we do not know beforehand how many threads a process is going to create.

The Ulix disk provides

- a `fault-mem` application which accesses an illegal address (and will subsequently be killed) and
- a `recurse` program that recursively calls a function (and thus forces stack growth): In early versions of the Ulix kernel it would eventually run out of memory and then be killed; with the final Ulix code it can go on rather long because the kernel will start paging out memory in order to free frames.
- There is also a `tp` program that explicitly pages out a page of its memory and then accesses it (so that it will be brought back in; see the next sections).

9.2 The Swap File

Ulix uses a 64 MByte swap file which is stored on the hard disk. This is also just a little less than the maximum file size that our Minix filesystem implementation supports (see Chapter 12): With six direct block addresses, one single indirect block (leading to 256 blocks) and one double indirect block (leading to 256×256 blocks) as well as a block size of 1 KByte, files can be no larger than 65798 KByte $\approx$ 64.26 MByte.

The swap file stores only the contents of pages, all administrative data is kept in memory. 64 MByte allow Ulix to double the available RAM (since it works with a fixed amount of 64 MByte of RAM as well), providing up to 128 MByte of virtual memory to the system and processes.

Internally we will store information about pages which have been written to disk. For each such page we need to know the address space ID and the page number, thus an internal paging record has the following form:

[292a] ⟨*type definitions* 91⟩+≡ (44a) ◁227 295c▷

```
struct paging_entry {
  int as         : 10;    // 10 bits for address space, values from [0..1023]
  int pageno     : 20;    // 20 bits for the page number
  int used       :  1;    // 2 bits for two flags
  int reserved   :  1;
} __attribute__((packed));
```

Defines:
`paging_entry`, used in chunk 292c.

This is just small enough to fit in a 32-bit integer. As the page size is 4 KByte, we need 64 MByte / 4 KByte = 16 384 such entries:

[292b] ⟨*constants* 112a⟩+≡ (44a) ◁233a 306a▷

```
#define MAX_SWAP_FRAMES 16384
```

Defines:
`MAX_SWAP_FRAMES`, used in chunks 292–94.

[292c] ⟨*global variables* 92b⟩+≡ (44a) ◁276e 293a▷

```
struct paging_entry paging[MAX_SWAP_FRAMES] = { { 0 } };
```

Defines:
`paging`, used in chunks 293d, 294, and 306c.
Uses `MAX_SWAP_FRAMES` 292b and `paging_entry` 292a.

If $\texttt{paging}_{292c}\texttt{[i].used}$ is 0 (`false`), the corresponding swap file entry `i` is free which fits our initialization of the data structure.

We assume that a swap file `/tmp/swap` of size 64 MByte already exists.

```
root@ulix[7]:/root# ls -l /tmp
[...]
 5 -rw-------  1    0    0 67108864  3 May 11:53 swap
```

During the system initialization we open this file and keep it open throughout the whole system runtime.

⟨*global variables* 92b⟩+≡ (44a) ◁292c 306b▷ [293a]
```
int swap_fd;
```
Defines:
swap_fd, used in chunks 293 and 294.

⟨*initialize swap* 293b⟩≡ (44b) [293b]
```
swap_fd = u_open ("/tmp/swap", O_RDWR, 0);
if (swap_fd != -1) {
  int size = u_lseek (swap_fd, 0, SEEK_END);
  printf ("swapon: enabling /tmp/swap (%d MByte)\n", size/1024/1024);
  u_lseek (swap_fd, 0, SEEK_SET);
} else
  printf ("swapon: error opening /tmp/swap!\n");
```
Uses O_RDWR 460b, printf 601a, SEEK_END 469b, SEEK_SET 469b, swap_fd 293a, u_lseek 418a, and u_open 412c.

We provide two simple functions which write a page to the file and read it back in. Both need to walk through (parts of) the paging array in order to find out whether the page is (already) on the disk and which free entry can be used if that is not the case.

⟨*function prototypes* 45a⟩+≡ (44a) ◁288 295a▷ [293c]
```
int write_swap_page (int as, int pageno, int frameno);
int read_swap_page  (int as, int pageno, int frameno);
```

We give these functions an extra argument `frameno`: If we already know the physical address where a page is stored in memory, we will provide its frame number so that the functions need not calculate it. (Actually we're not using the feature where write_swap_page$_{293d}$ or read_swap_page$_{294}$ would manually calculate the frame number. However it would be useful for an enhanced paging mechanism that might page out pages but still keep them in memory as well or page in pages from the swap file and still keep them on the disk. We do neither in our implementation.)

We do not expect these functions to alter a page table of the involved process—that happens in the functions page_out$_{296}$ and page_in$_{297}$ which we present in the next section.

⟨*function implementations* 100b⟩+≡ (44a) ◁289a 294▷ [293d]
```
int write_swap_page (int as, int pageno, int frameno) {
  // get frame number, if it was not supplied
  if (frameno == -1) frameno = mmu_p (as, pageno);
  if (frameno == -1) return -1;                    // error: page not available

  // get index
  int index      = -1;
  int free_index = -1;
  for (int i = 0;  i < MAX_SWAP_FRAMES;  i++) {
    if (free_index == -1 && !paging[i].used)  free_index = i;
    if (paging[i].used && paging[i].as == as && paging[i].pageno == pageno) {
      index = i;  // already on disk!
      break;
```

```
    }
  }
  if (index == -1 && free_index == -1) return -1;  // not found + no free space
  if (index == -1 && free_index != -1) {
    index = free_index;                            // create new entry
    paging[index].used   = true;
    paging[index].as     = as;
    paging[index].pageno = pageno;
  }
  // note: if (index != -1) we do not modify paging[]; this is an update

  // write to disk
  u_lseek (swap_fd, index*PAGE_SIZE, SEEK_SET);
  u_write (swap_fd, (char*)PHYSICAL(frameno*PAGE_SIZE), PAGE_SIZE);
  return 0;                                        // success
}
```

Defines:
write_swap_page, used in chunks 293c and 296.
Uses MAX_SWAP_FRAMES 292b, mmu_p 171c, PAGE_SIZE 112a, paging 292c, PHYSICAL 116a, SEEK_SET 469b, swap_fd 293a, u_lseek 418a, and u_write 415a.

Note that u_write$_{415a}$ will use the buffer cache (see Chapter 13.3), thus calling the function write_swap_page$_{293d}$ may at first only result in copying a page to a different memory area.

Reading a page back in is simpler because we need not distinguish between updates and initial write operations: When we try to read, the page is either there or it is not. We do not check the case that a requested page might be missing in the swap file because we only call this function when we know that the page must be there.

[294] ⟨*function implementations* 100b⟩+≡ (44a) ◁293d 296▷

```
int read_swap_page (int as, int pageno, int frameno) {
  // get frame number, if it was not supplied
  if (frameno == -1) frameno = mmu_p (as, pageno);
  if (frameno == -1) return -1;       // error: page not available

  int index = -1;                                          // get index
  for (int i = 0;  i < MAX_SWAP_FRAMES;  i++) {
    if (paging[i].used && paging[i].as == as && paging[i].pageno == pageno) {
      index = i;  // found the entry!
      break;
    }
  }

  u_lseek (swap_fd, index*PAGE_SIZE, SEEK_SET);            // read from disk
  u_read (swap_fd, (char*)PHYSICAL(frameno*PAGE_SIZE), PAGE_SIZE);
  return 0;                           // success
}
```

Defines:
read_swap_page, used in chunk 297.
Uses MAX_SWAP_FRAMES 292b, mmu_p 171c, PAGE_SIZE 112a, paging 292c, PHYSICAL 116a, read 429b, SEEK_SET 469b, swap_fd 293a, u_lseek 418a, and u_read 414b.

9.2.1 Paging Out and In

The `write_swap_page`$_{293d}$ and `read_swap_page`$_{294}$ functions simply copy a page frame from memory to the swap file or vice versa. But real paging requires more than that: we need to modify a page table whenever we remove or add a page. This is what the two functions

⟨*function prototypes* 45a⟩+≡ (44a) ◁293c 306e▷ [295a]
```
int page_out (int as, int pageno);
int page_in  (int as, int pageno);
```

are for. Some other kernel function (which we will describe soon) makes the decision to remove page X of process Y and then calls `page_out`$_{296}$ which in return saves the page (via `write_swap_page`$_{293d}$) and updates the relevant process' page table to indicate that the page is no longer in RAM (but could be gotten from the swap file). When that process tries to access the page the next time, a page fault will occur which must be handled by the page fault handler which brings the needed page back in and lets the process reattempt the last instruction.

Let us first recall the data structure `page_desc`$_{100a}$ for the page descriptor:

⟨`page_desc` *structure definition (repeated)* 295b⟩≡ [295b]
```
typedef struct {
  unsigned int present         :  1;  //  0
  unsigned int writeable       :  1;  //  1
  unsigned int user_accessible :  1;  //  2
  unsigned int pwt             :  1;  //  3
  unsigned int pcd             :  1;  //  4
  unsigned int accessed        :  1;  //  5
  unsigned int dirty           :  1;  //  6
  unsigned int zeroes          :  2;  //  8.. 7
  unsigned int unused_bits     :  3;  // 11.. 9
  unsigned int frame_addr      : 20;  // 31..12
} page_desc;
```

If the `present` bit is set to 0, any attempt to access the page will lead to a page fault. Thus, when we want to page out a page, we can simply reset the page descriptor's `present` bit. But how is the fault handler to know whether a "genuine" page fault has occurred (i. e., the page does not exist at all) or whether the kernel paged out the page and is capable of getting it back? The bits 9–11 of the descriptor, called `unused_bits` above, are completely ignored by the MMU. This is our starting point: We use one of these three bits for keeping the paged out state:

⟨*type definitions* 91⟩+≡ (44a) ◁292a 318b▷ [295c]
```
typedef struct {
  unsigned int present         :  1;  //  0
  unsigned int writeable       :  1;  //  1
  unsigned int user_accessible :  1;  //  2
  unsigned int pwt             :  1;  //  3
  unsigned int pcd             :  1;  //  4
  unsigned int accessed        :  1;  //  5
  unsigned int dirty           :  1;  //  6
```

```
    unsigned int zeroes          :  2;  //  8.. 7
    unsigned int paged_out       :  1;  //  9       <- new
    unsigned int unused_bits     :  2;  // 11..10
    unsigned int frame_addr      : 20;  // 31..12
  } new_page_desc;
```

Defines:
new_page_desc, used in chunks 296 and 297.

Since page_desc$_{100a}$ and new_page_desc$_{295c}$ have the same layout, we can cast them into one another without corrupting data. So when we want to find out whether the new paged_out bit is set, we can check that with `if ( ((new_page_desc*)pd)->paged_out ) { ... }`.

Now we can present the page_out$_{296}$ and page_in$_{297}$ functions which perform the same calculations as the mmu_p$_{171c}$ function which we introduced earlier. page_out$_{296}$ does four things:

- it calls write_swap_page$_{293d}$ to do the transfer from memory to disk,
- it resets the page descriptor's present bit and sets its paged_out bit,
- invlpg it invalidates the TLB entry with the invlpg instruction in order to make sure that the next access to this page will cause a page fault (instead of accessing the old frame which may no longer hold the page) [Int08, p. 21] (or [Int11, p. 4-56–4.57]),
- and it releases the frame, thus increasing the free physical memory.

[296] ⟨*function implementations* 100b⟩+≡ (44a) ◁294 297▷

```
int page_out (int as, int pageno) {
  uint pdindex = pageno/1024; uint ptindex = pageno%1024;
  page_directory *pd = address_spaces[as].pd;
  if ( ! pd->ptds[pdindex].present ) {
    return -1;                                  // fail: page table not found
  } else {
    page_table *pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
    if ( pt->pds[ptindex].present ) {           // found the page
      new_page_desc *pdesc = (new_page_desc*) &pt->pds[ptindex];
      int frameno = pdesc->frame_addr;
      write_swap_page (as, pageno, frameno);    // write to swap file
      pdesc->present = false;                   // mark page non-present
      pdesc->paged_out = true;                  // mark page paged-out
      asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );
      release_frame (frameno);                  // mark phys. frame as free
      return 0;                                 // success
    } else {
      return -1;                                // fail: page not found
    };
  }
}
```

Defines:
page_out, used in chunks 295a, 299a, and 308c.
Uses address_spaces 162b, new_page_desc 295c, page_directory 103d, page_table 101b, PHYSICAL 116a, release_frame 119b, and write_swap_page 293d.

The page_in$_{297}$ function expects that we try to page in a page which was paged out with page_out$_{296}$ earlier. That is, the page descriptor must exist and have its paged_out bit set. It will then do the following three things:

- it reserves a new physical frame which will soon hold the page and writes its address to the page descriptor,
- it calls read_swap_page$_{294}$ to do the transfer from disk to memory,
- and it resets the page descriptor's paged_out bit,

⟨*function implementations* 100b⟩+≡ (44a) ◁296 319d▷ [297]

```
int page_in (int as, int pageno) {
  uint pdindex = pageno/1024;
  uint ptindex = pageno%1024;
  page_directory *pd = address_spaces[as].pd;
  if ( ! pd->ptds[pdindex].present ) {
    printf ("DEBUG: page_in: page table not present\n");
    return -1;                                  // fail: page table not found
  } else {
    page_table *pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
    if ( !pt->pds[ptindex].present ) {
      // found the page descriptor
      new_page_desc *pdesc = (new_page_desc*) &pt->pds[ptindex];
      if (!pdesc->paged_out) {
        printf ("DEBUG: page_in: page 0x%0x not marked paged out!\n", pageno);
        return -1;                              // fail: page was not paged out
      }
      int frameno = request_new_frame ();       // reserve a phys. frame
      if (frameno == -1) return -1;             // fail: no free memory
      read_swap_page (as, pageno, frameno);     // read from swap file
      pdesc->present = true;                    // mark page present
      pdesc->paged_out = false;                 // mark page not paged-out
      pdesc->frame_addr = frameno;              // write new phys. frame number
      // asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );   // not needed
      return 0;                                 // success
    } else {
      printf ("DEBUG: page_in: page not found\n");
      return -1;                                // fail: page not found
    };
  }
}
```

Defines:
page_in, used in chunk 298a.
Uses address_spaces 162b, new_page_desc 295c, page_directory 103d, page_table 101b, PHYSICAL 116a, printf 601a, read_swap_page 294, and request_new_frame 118b.

When we page in, we need not invalidate a TLB entry since it does not store information about non-present pages [Int08, p. 21] (or [Int11, p. 4-58]):

> If a paging-structure entry is modified to transition the present bit from 0 to 1, no invalidation is necessary. This is because no TLB entry or paging-structure cache entry will be created with information from a paging-structure entry that is marked "not present". (If it is also the case that no invalidation was performed the last time the present bit was transitioned from 1 to 0, the processor may use a TLB entry or paging-structure cache entry that was created when the present bit had earlier been 1.)

This is all the code we need for paging in and out a page. Now we need to decide when to page out a page and how to pick that page.

9.2.2 Letting the Page Fault Handler Page In a Page

We can now add the missing code chunk ⟨*page fault handler: check if page was paged out* 298a⟩. Recall that the faulting address is stored in `faulting_address`. All we need to do here is attempt to page in the corresponding page (`faulting_address / PAGE_SIZE`$_{112a}$)—if we are successful we can leave the page fault handler, and the process will re-execute the last instruction.

[298a] ⟨*page fault handler: check if page was paged out* 298a⟩≡ (289b)

```
int pageno = faulting_address / PAGE_SIZE;
if (page_in (current_as, pageno) == 0) {
  return;           // success, leave fault handler
}
```

Uses `current_as` 170b, `page_in` 297, and `PAGE_SIZE` 112a.

9.2.3 Testing

At this step of the implementation task we should check whether our code works as intended. For that purpose we will provide a temporary system call with which a process can force the kernel to page out a page. (Note that in general it is a bad idea to allow processes to take that kind of control over the memory management.)

A test program will then try to access data in the paged-out page which should in turn have the page fault handler bring the page back. We know that we're successful if the program executes without errors.

[298b] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁282b 315▷

```
#define __NR_page_out 508
```

Defines:
`__NR_page_out`, used in chunk 299.

[298c] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁282a 309▷

```
void syscall_page_out (context_t *r);
```

⟨*syscall functions* 174b⟩+≡ (202b) ◁282c 310a▷ [299a]
```
void syscall_page_out (context_t *r) {
  // ebx: page number
  eax_return (page_out (current_as, r->ebx) );
}
```
Defines:
  `syscall_page_out`, used in chunks 298c and 299b.
Uses `context_t` 142a, `current_as` 170b, `eax_return` 174a, and `page_out` 296.

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁282d 310c▷ [299b]
```
install_syscall_handler (__NR_page_out, syscall_page_out);
```
Uses `__NR_page_out` 298b, `install_syscall_handler` 201b, and `syscall_page_out` 299a.

The user mode program will use the following library function

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁282e 310d▷ [299c]
```
int lib_page_out (int pageno);
```

which just makes the system call

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁282f 310e▷ [299d]
```
int lib_page_out (int pageno) { return syscall2 (__NR_page_out, pageno); }
```
Defines:
  `lib_page_out`, used in chunk 299c.
Uses `__NR_page_out` 298b and `syscall2` 203c.

Here is the code for a simple test program:

⟨*lib-build/tools/tp.c* 299e⟩≡ [299e]
```
#include "../ulixlib.h"
char test[4096] __attribute__ ((aligned (4096)));

int main () {
  printf ("Testing paging\n");
  test[5] = 'X';
  unsigned int address = (unsigned int)(&test[5]);
  printf ("test[5] = '%c', address = 0x%x\n", test[5], address);
  lib_page_out (address >> 12);
  printf ("test[5] = '%c', address = 0x%x\n", test[5], address);
  exit (0);
}
```

Now we need to discuss *when* the operating system should page out a page and *which* page it should choose. We enter the realm of *page replacement strategies*—the topic of the following section.

9.3 Page Replacement Strategy

When memory gets full, eventually the system will have to move pages to the disk in order to make room for other processes' memory demands. Paging out a page (i. e., writing it to disk and releasing the page frame that held the page) and assigning a different process' page to this page frame is called page replacement. Paging the information in and out of main memory is extremely simple because of the fixed size data chunks—as you have seen in the implementation of `page_in`$_{297}$ and `page_out`$_{296}$.

Access to secondary storage is very slow, while access to main memory is rather fast. At time of writing, good hard disks have an average access time of about eight milliseconds, main memory of about eight nanoseconds. This is a difference of 10^6, i. e., six orders of magnitude. To make this huge difference more evident, assume that access to main memory needs one *second*. Then the access to secondary storage would have to take 10^6 seconds, which is roughly 11.5 days, to stay in the same relation.

Well-tuned paging systems can achieve a performance which is very close to the speed of main memory. The decisive parameter is the probability p of not finding the requested information in RAM (i. e., the probability of a page fault). Given that t_{mm} is the time necessary to access main memory and t_{pf} the time to handle a page fault, the average time t_{vm} to access virtual memory using a paging system is:

$$t_{vm} = (1 - p) \cdot t_{mm} + p \cdot t_{pf}$$

Since t_{pf} is dominated by the access time to secondary storage, we need to keep p as low as possible.

The algorithm that decides which page to page out is called a page replacement algorithm, and it implements a page replacement strategy. The chosen strategy is a part of the memory management system's design, and several choices are available.

One possible choice would be a random selection: Whenever there is need for a free page frame (and none available) just pick any odd page frame and page out its contents. This strategy would not be much good, but we can think of even worse ones, e. g. always pick the very first page frame in the RAM.

The selection process has no consequences on the overall functioning of memory management: Even the worst strategy (and "pick the first page frame" is a good candidate for that) will lead to a working memory management system. However, the selection process decides how efficiently the resulting system behaves.

Before going into details, let us note that there is no direct equivalent to page replacement in filesystems—unless you had another layer of the memory hierarchy that is above disk access, e. g. an automatic tape backup system with a tape robot that can write files to a tape and delete them on disk when disk space gets low. If you had such a setup, you would move from a CPU–cache–RAM–disk memory hierarchy to a CPU–cache–RAM–disk–tape one, and accessing a file currently on tape would cause something that could be called a file access fault, resulting in the system automatically fetching the file back from tape (and keeping the requesting process blocked during all the time until the file becomes available again). Strategies for deciding which files to temporarily transfer from the disk

to a tape would be called a file replacement strategy and be somewhat similar to the page replacement strategies. Distributed filesystems (or distributed network filesystems) that allow files to either exist on a local machine or on a remote host's disk do something similar if they make file access transparent, no matter whether things are stored locally or remotely. We will not look any further into this. A true analog of page replacement would operate on disk block level, i. e. remove individual blocks from the disk in order to store them elsewhere, and that is something that does not make much sense since files are typically accessed fully when they are accessed at all. It might however make sense to keep the first block of a file on disk when removing the file, because often only the first block of a file is read in order to find out its filetype (think of "magic numbers").

A way of measuring a page replacement strategy is the average number of page faults that it causes. It is not possible to truly calculate this number, because it depends on so many things, e. g.:

- The absolute memory demands depend on all the processes currently running on a system.
- Even if sample situations (test cases) are created that consist of predefined processes with fixed start times and memory requirements (such as: process will access its page number n at instruction i) it is not possible to predict when precisely this process will execute this instruction—scheduling the processes will always result in slightly different orders of execution each time the test case is run.

So all we can do is think of theoretical properties of replacement strategies and, when implementing a strategy, observe its effects on a number of test cases which are tested several times in order to calculate an average number of page faults for each test case. Looking at the design of a strategy will however allow us to make some principle predictions.

9.3.1 Page Locking

Page replacement is a good idea, but some pages must sometimes be protected from being paged out. For example, certain parts of the operating system are so critical that they should never be paged out to secondary storage. The most striking example is the code that contains the interrupt handlers. If a page fault occurs and the code for the page fault handler is not be present, then we are be in big trouble. Also, most parts of the page tables for the kernel should always be present, as well as pages that are located in special frames for memory-mapped I/O. In such cases the pages should be *locked* into their frames and page replacement algorithms should ignore these pages.

Ulix does not explicitly support page locking, but it considers the upper 1 GByte of each address space as locked: kernel memory will never be paged out, so we only have to deal with the memory of processes which avoids all the problems that otherwise call for page locking.

9.3.2 Page Replacement Strategies

We describe a few classical replacement strategies before showing you the implementation in the Ulix kernel.

9.3.2.1 FIFO Page Replacement

A simple approach to page replacement is using a FIFO (first in, first out) list that keeps record of pages as they come into memory (either by being newly created, e. g. because a new process was started, or by being brought back in from disk after they had been paged out earlier). For Ulix we could modify the `as_map_page_to_frame`$_{165b}$ and `page_in`$_{297}$ functions in order to keep track of new pages.

The list can grow up to a size that is determined by the number of available page frames in the system's memory. When this limit is reached, the list will be chopped from the top: The page that is first in the list is removed from the list and also paged out. If the owning process tries to access this (paged-out) page again, a page fault occurs, and the memory manager has to page it back in, adding it at the end of the FIFO list.

This approach is simple because administering a FIFO list is simple, and selecting the next page to be paged out only requires reading the list head and removing it. However it has the problem of totally ignoring that some pages are accessed much more frequently than others. All pages travel from the list end to the list head at equal speed as pages are continuously paged out and back in, and for constantly and frequently used pages this means they will be paged out and in very often. It would make sense to be informed about the access frequency and keep the more frequently used pages in memory all the time, resulting in a much increased overall performance (with less page faults).

9.3.2.2 Second Chance Algorithm

access bit

An attempt to bring the frequency of page access into the FIFO strategy is the introduction of a "second chance": The idea is to set an *access bit* for a page each time it is accessed by its owning process. This is something that the MMUs of most processors do automatically—which is important because otherwise it would be very hard to detect memory access manually.

The modification of the FIFO strategy is the following:

- A simple FIFO list of all pages works in principle as in the FIFO case.
- The MMU sets bits for each page access, as described above.
- When a page frame has to be freed, the system looks at the first list entry (as before). If that page has its access bit set, it is *not* paged out, but instead moved to the end of the list, and its access bit is cleared: it gets a second chance.

So the Second Chance algorithm selects the first page in the FIFO list that does not have its access bit set. "Not using the chance" then means that after being spared when first found at the list head, it will travel all the way from the end to the head of the list without being accessed one single time. Then the memory manager will page it out.

9.3.2.3 Clock

The Clock algorithm does the same as the Second Chance algorithm but does not require the reordering of the list (by taking away the head element and appending it to the list). Instead it uses a circular list (where the last element points back to the head) and uses a "clock hand" which points to the current head of the list.

When the algorithm needs to pick the candidate it starts with the list element that the clock hand points to. If its access bit is not set, that page is paged-out and removed from the list; the clock hand turns forward to the next element in the list.

If, however, the access bit is set, the algorithm clears it, moves the clock hand to the next location and starts over. It may eventually come full circle and arrive at the element whose access bit it had just reset; then it will pick that page.

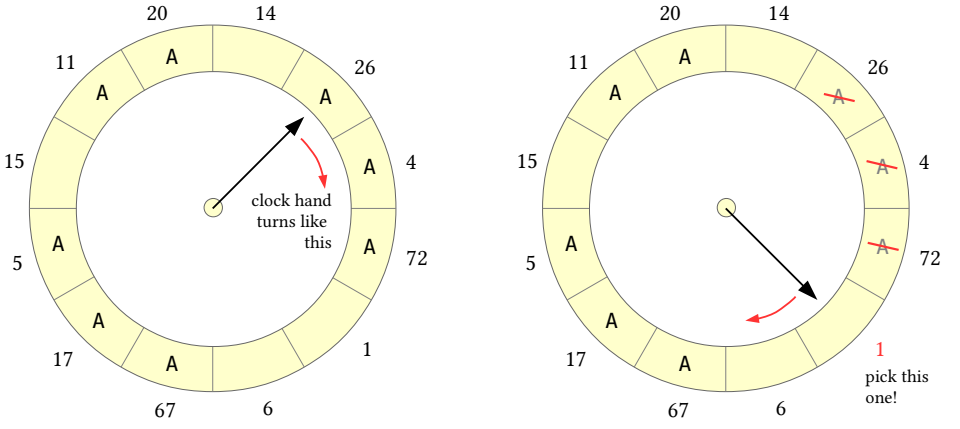


Figure 9.1: The Clock algorithm resets the access bits in the first three entries (pages 26, 4, 72) and picks the fourth entry (page 1) because its access bit is not set.

Figure 9.1 shows an example of the Clock algorithm at work: When it starts, its clock hand points to page 26 (left part of figure). It sees that the page has its access bit set, so it resets it and moves on (clock-wise). The same repeats twice for pages 4 and 72, but when it reaches page 1 which does not have its access bit set, it picks that page as the candidate for removal.

9.3.2.4 Least-Frequently Used

The Second Hand or Clock strategy suffers from the fact that the only observed property of a page is whether it has been accessed recently or not. However some pages are used much more often than others, and those much-used pages should be avoided when choosing a candidate for paging because they will be used again soon with high probability.

What we would like to have is an access log for each page so that we can pick a page which both was not accessed recently and in general was not accessed a lot further ago. A

true access log (that picks up every single access) is very hard to implement. For example, one could modify all page descriptors so that they cause a page fault. Then every access would generate a page fault, the page fault handler could temporarily grant access to that page and resume the process, only to remove the access permissions as soon as possible. While we would still not register all page accesses (since a process might access the same page several times in short sequence) it would give a good overview of the actual usage patterns. But this would be extremely expensive in terms of CPU time as the system would permanently generate page faults.

We can, however, do something that approximates such an access log: The Least-Frequently Used strategy counts page accesses by regularly checking and resetting the access bit. Every time it notices a set access bit it increments the access counter for that page. From time to time the counters need to be scaled down so that they don't exceed the limit of their datatype. When the time comes to page out a page, the page with the lowest counter value is chosen.

For Ulix, with its fixed 64 MByte of RAM, this would mean keeping records for up to 16 384 pages. If we used the maximum possible amount of RAM (4 GByte) and all its frames were in use, there would be more than a million pages to look after, and we might want to grow or shrink the list dynamically (according to the number of existing pages) so that we don't waste too much memory for it. Also, the larger the list of pages becomes, the longer it takes to search for a minimum.

9.4 Page Replacement Implementation in Ulix

We will use access counters, but not for individual (process/page) pairs, but for *hashes* of them. This will let us use a fixed-size counter table which need not grow or shrink over time when new processes are created or old ones removed.

Each page can be identified by an (`as`, `pageno`) pair. `pageno` is a 20 bit number, and `as` is a 10 bit number (since we only allow up to $1024 = 2^{10}$ address spaces).

We map this to an array index by calling

[304a] ⟨*pseudo code for calculating the index into the hash table* 304a⟩≡

```
index = hash ((address_space << 20) | pageno)
```

Uses `address_space` 161 and `hash` 306f.

This `index` number points to entry `counter_table`$_{306b}$`[index]` which stores a used flag and a counter. We will regularly update the counter table …

[304b] ⟨*pseudo code for counter updates* 304b⟩≡

```
for (as in used_address_spaces, pageno in user_page_numbers(as)) {
  n = get_and_reset_referenced_bit (as, pageno);  // 0 or 1
  index = hash ((address_space << 20) | pageno);
  if (n==1) {
    counter_table[index].used = true;
    counter_table[index].count++;
  }
}
```

...and (less often) rescale the counters by halving them if the maximum counter is above some threshold:

⟨*pseudo code for counter rescaling* 305a⟩≡ [305a]
```
// get maximum count
themax = 0;
for (index in 0..maxindex)
  if (counter_table[index].used)
    themax = max (themax, counter_table[index].count);
if (themax < THRESHOLD) return;  // do nothing

// halve all counters
for (index in 0..maxindex)
  if (counter_table[index].used) {
    counter_table[index].count /= 2
    counter_table[index].count += 1;  // add 1 to avoid 0 value
  }
```

This automatically leads to some kind of aging: when the maximum reaches the threshold value, all entries will be halved.

Now, picking a page with minimum counter for replacement goes like this:

⟨*pseudo code for picking a page* 305b⟩≡ [305b]
```
pick = NULL;
for (as in used_address_spaces, pageno in user_page_numbers(as)) {
  index = hash ((address_space << 20) | pageno);
  if (pick==NULL && counter_table[index].used) {
    // initialize minimum, pick
    pick = (as, pageno);
    themin = counter_table[index].count;
  } else {
    if (counter_table[index].count < themin) {
      themin = counter_table[index].count
      pick = (as, pageno);
    }
  }
}
if (pick != NULL) page_out (pick.as, pick.pageno);
```

dirty page

This algorithm does not check whether a page is *dirty*, i. e., modified. In more advanced paging systems, a page may simultaneously exist in memory and on the disk (for example when it was paged out and paged back in but the swap file entry was not deleted). In that case it would make sense to pick a page which is still on disk *and* has not been changed in memory since it was brought back in the last time. Since our implementation does not keep pages both in RAM and on disk, we can ignore the dirty flag (or consider every page dirty; we always have to write to disk, whatever page we pick).

Note that the algorithm may pick a wrong page if there are pages with the same hash as the chosen one. In that case we have no way to decide which of those pages had the fewest accesses, but all of them at least had very few accesses, so this is good enough.

Here's the actual implementation. We define the counter table as an array of simple structures:

[306a] ⟨*constants* 112a⟩+≡ (44a) ◁292b 308b▷
```
#define PG_MAX_COUNTERS 1024
```
Defines:
PG_MAX_COUNTERS, used in chunks 306–8.

[306b] ⟨*global variables* 92b⟩+≡ (44a) ◁293a 310f▷
```
struct { boolean used; int count; } counter_table[PG_MAX_COUNTERS] = { { 0 } };
lock paging_lock;
```
Defines:
counter_table, used in chunks 307 and 308.
paging_lock, used in chunks 306–8.
Uses lock 365a and PG_MAX_COUNTERS 306a.

We also provide a lock to protect access to that table:

[306c] ⟨*initialize kernel global variables* 184d⟩+≡ (44b) ◁184d 310g▷
```
paging_lock = get_new_lock ("paging");
```
Uses get_new_lock 367b, paging 292c, and paging_lock 306b.

And we regularly update the table via timer tasks.

[306d] ⟨*timer tasks* 306d⟩≡ (342b) 311a▷
```
if (scheduler_is_active && ((system_ticks % 10) == 0)) {
  ⟨page replacement: update counters 307a⟩   // Every 10 ticks (~ 0.1 seconds)
}
if (scheduler_is_active && ((system_ticks % 50) == 5)) {
  ⟨page replacement: rescale counters 308a⟩   // Every 50 ticks (~ 0.5 seconds)
}
```
Uses scheduler_is_active 276e and system_ticks 338a.

hash function

As mentioned above, we need a *hash function* for mapping all the possible (address space, page number) combinations onto our array. Hashing is a science in its own right, and we do not attempt to provide a clever or useful hashing algorithm in this book. Instead we implement our hash function

[306e] ⟨*function prototypes* 45a⟩+≡ (44a) ◁295a 306f▷
```
int hash (int val, int maxval);
```

in a very simple fashion: We assume that the `val` argument was created from an address space ID `as` and a page number `pageno` by calculating `(as << 20) | pageno`. Our hash function can then restore the original values via the formulas `as = val >> 20` and `pageno = val & 0b1111111111`. We multiply the address space ID with 32 and add the page number. Since that sum may exceed PG_MAX_COUNTERS$_{306a}$, we use a modulo operation to make it fit:

[306f] ⟨*function prototypes* 45a⟩+≡ (44a) ◁306e 319a▷
```
int hash (int val, int maxval) {
  // return val % maxval;   // ridiculous hash
  return ((val >> 20)*32 + (val & 0b1111111111)) % maxval;
}
```
Defines:
hash, used in chunks 304 and 306–8.

The update code does not disable interrupts or use a lock; we do not really care if data are changed while we assemble the statistical data, since a small error in the statistics (which might result from parallel access to a page table entry) will not change the overall behavior.

The double loop over address space IDs and page numbers that we've shown above in the ⟨*pseudo code for counter updates* 304b⟩ code chunk turns into a triple loop (over address space IDs, page table descriptors and page descriptors) since we cannot directly access the page descriptor for some page n without inspecting the right page table (number $n/1024$) first. We only look at the first 768 page tables—beyond that kernel memory starts, and we have decided to never page out memory that belongs to the kernel. That way we need not deal with *sticky bits* (*locked* bits) in the page descriptors. Instead, the simple rule is: If a page belongs to process memory, it is a candidate for removal; otherwise not.

sticky bit, locked

⟨*page replacement: update counters* 307a⟩≡ (306d) [307a]

```
if (mutex_try_lock (paging_lock)) {
  for (int as = 1;  as < MAX_ADDR_SPACES;  as++) {
    if (address_spaces[as].status != AS_FREE) {
      page_directory *pd = address_spaces[as].pd;
      for (int i = 0;  i < 100;  i++) {      // < 768: only work on process memory
        if (pd->ptds[i].present) {           // directory entry in use
          page_table *pt = (page_table*)(PHYSICAL ((pd->ptds[i].frame_addr)<<12));
          for (int j = 0;  j < 1024;  j++) {
            if (pt->pds[j].present) {        // table entry in use
              ⟨page replacement: update counter for page i · 1024 + j 307b⟩
            }
          }
        }
      }
    }
  }
  mutex_unlock (paging_lock);
}
```

Uses address_spaces 162b, AS_FREE 162a, MAX_ADDR_SPACES 158a, mutex_try_lock 366b, mutex_unlock 366c, page_directory 103d, page_table 101b, paging_lock 306b, and PHYSICAL 116a.

For updating the counter for page $1024 \cdot i + j$ we look at its page descriptor. If the accessed bit is set

⟨*page replacement: update counter for page i · 1024 + j* 307b⟩≡ (307a) [307b]

```
int pageno = i*1024 + j;
int n = pt->pds[j].accessed;   // get and ...
pt->pds[j].accessed = false;   // reset access flag
int index;
if (n == 1 &&
    (index = hash ((as << 20) | pageno, PG_MAX_COUNTERS)) < PG_MAX_COUNTERS) {
  counter_table[index].used = true;
  counter_table[index].count++;
}
```

Uses counter_table 306b, hash 306f, and PG_MAX_COUNTERS 306a.

The implementation of the rescaling operation is only slightly more complex than the pseudocode:

[308a] ⟨*page replacement: rescale counters* 308a⟩≡ (306d)

```
// get the maximum count
int themax = 0;
if (mutex_try_lock (paging_lock)) {
  for (int index = 0; index < PG_MAX_COUNTERS; index++) {
    if (counter_table[index].used) {
      int val = counter_table[index].count;
      if (val > themax) themax = val;
    }
  }

  if (themax > PG_COUNTER_THRESHOLD) {
    // rescale all counters
    for (int index = 0; index < PG_MAX_COUNTERS; index++) {
      if (counter_table[index].used) {
        counter_table[index].count /= 2;
        counter_table[index].count += 1;  // avoid 0 value
      }
    }
  }
  mutex_unlock (paging_lock);
}
```

Uses `counter_table` 306b, `mutex_try_lock` 366b, `mutex_unlock` 366c, `paging_lock` 306b, `PG_COUNTER_THRESHOLD` 308b, and `PG_MAX_COUNTERS` 306a.

We still need to define the counter threshold:

[308b] ⟨*constants* 112a⟩+≡ (44a) ◁306a 318a▷

```
#define PG_COUNTER_THRESHOLD 100000
```

Defines:
`PG_COUNTER_THRESHOLD`, used in chunk 308a.

Once at least one page has been access counted more than 100 000 times, the counter values will be rescaled.

Finally this is the code which frees a frame. It looks at all the pages in all address spaces, generates the hash and looks up the counter for that hash (if it exists). It initializes `pick_as` and `pick_pageno` to the first address space and page number for whose hash it finds a counter and then updates these variables whenever it finds a smaller counter.

[308c] ⟨*page replacement: free one frame* 308c⟩≡ (119a 310a)

```
addr_space_id pick_as      = -1;
int           pick_pageno, themin;
while (!mutex_try_lock (paging_lock)) ;    // active waiting for lock
for (int as = 1;  as < MAX_ADDR_SPACES;  as++) {
  if (address_spaces[as].status == AS_USED) {
    page_directory *pd = address_spaces[as].pd;
    for (int i = 0;  i < 768;  i++) {      // < 768: only work on process memory
      if (pd->ptds[i].present) {           // directory entry in use
```

```
          page_table *pt = (page_table*) (PHYSICAL ((pd->ptds[i].frame_addr) << 12));
          for (int j = 0;  j < 1024;  j++) {
            if (pt->pds[j].present) {        // table entry in use
              int pageno = i*1024 + j;
              int index = hash ((as << 20) | pageno, PG_MAX_COUNTERS);
              if (pick_as==-1 && counter_table[index].used) {
                // initialize minimum, pick
                pick_as     = as;
                pick_pageno = pageno;
                themin = counter_table[index].count;
              } else {
                if (counter_table[index].count < themin) {
                  themin = counter_table[index].count;
                  pick_as     = as;
                  pick_pageno = pageno;
                }
              }
            }
          }
        }
      }
    }
  }
  mutex_unlock (paging_lock);

  if (pick_as != -1) {
    mutex_lock (paging_lock);
    page_out (pick_as, pick_pageno);
    mutex_unlock (paging_lock);
  } else {
    printf ("\nERROR: cannot pick a page to evict!\n");
  }
```

Uses addr_space_id 158b, address_spaces 162b, AS_USED 162a, counter_table 306b, hash 306f, lock 365a, MAX_ADDR_SPACES 158a, mutex_lock 366a, mutex_try_lock 366b, mutex_unlock 366c, page_directory 103d, page_out 296, page_table 101b, paging_lock 306b, PG_MAX_COUNTERS 306a, PHYSICAL 116a, pick_pageno, printf 601a, and themin.

Again, instead of the double loop from the ⟨*pseudo code for picking a page* 305b⟩ code chunk, we need a triple loop to access all page tables referenced by all page directories for all address spaces.

9.4.1 The Swapper Process

We provide two system calls that retrieve the number of free frames and issue a request to free a page:

⟨*syscall prototypes* 173b⟩+≡ (202a) ◁298c 330b▷ [309]

```
  void syscall_get_free_frames (context_t *r);
  void syscall_free_a_frame (context_t *r);
```

[310a] ⟨*syscall functions* 174b⟩+≡ (202b) ◁299a 331a▷
```
void syscall_get_free_frames (context_t *r) {
  // no parameters
  mutex_lock (swapper_lock);    // lock_, see below
  eax_return (free_frames);
}

void syscall_free_a_frame (context_t *r) {
  // no parameters
  ⟨page replacement: free one frame 308c⟩
}
```
Defines:
syscall_free_a_frame, used in chunk 310c.
syscall_get_free_frames, used in chunks 309 and 310c.
Uses context_t 142a, eax_return 174a, free_frames 112b, mutex_lock 366a, and swapper_lock 310f.

[310b] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁260b 330c▷
```
#define __NR_get_free_frames   509
#define __NR_free_a_frame      510
```
Uses __NR_free_a_frame and __NR_get_free_frames.

[310c] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁299b 328e▷
```
install_syscall_handler (__NR_get_free_frames, syscall_get_free_frames);
install_syscall_handler (__NR_free_a_frame,    syscall_free_a_frame);
```
Uses __NR_free_a_frame, __NR_get_free_frames, install_syscall_handler 201b, syscall_free_a_frame 310a, and syscall_get_free_frames 310a.

We also need user mode library functions which can make the two system calls:

[310d] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁299c 328f▷
```
int get_free_frames ();
void free_a_frame ();
```

[310e] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁299d 328g▷
```
int get_free_frames () { return syscall1 (__NR_get_free_frames); }
void free_a_frame ()   { syscall1 (__NR_free_a_frame); }
```
Defines:
free_a_frame, used in chunks 311b and 513e.
get_free_frames, used in chunks 310d, 311b, and 513e.
Uses __NR_free_a_frame, __NR_get_free_frames, and syscall1 203c.

The swapper process should not work permanently, so we use a trick: We let it block on a lock and add a timer task that unlocks that lock every 0.1 seconds.

[310f] ⟨*global variables* 92b⟩+≡ (44a) ◁306b 316▷
```
lock swapper_lock;
```
Defines:
swapper_lock, used in chunks 310 and 311a.
Uses lock 365a.

[310g] ⟨*initialize kernel global variables* 184d⟩+≡ (44b) ◁306c 363d▷
```
swapper_lock = get_new_lock ("swapper");
```
Uses get_new_lock 367b and swapper_lock 310f.

⟨*timer tasks* 306d⟩+≡ (342b) ◁306d 342c▷ [311a]

```
if (scheduler_is_active && ((system_ticks % 10) == 0)) {
  // Every   10 clocks (approx. 0.1 seconds)
  if (swapper_lock->bq.next)
    mutex_unlock (swapper_lock);
}
```

Uses mutex_unlock 366c, scheduler_is_active 276e, swapper_lock 310f, and system_ticks 338a.

The swapper program switches to the last virtual console. In an infinite loop it queries the number of free frames using $\texttt{get\_free\_frames}_{310e}$, and that function will block because $\texttt{syscall\_get\_free\_frames}_{310a}$ locks the $\texttt{swapper\_lock}_{310f}$. The function returns after the timer handler releases the lock, so the loop is only executed every 0.1 seconds.

If the number of frames gets too low, the program calls $\texttt{free\_a\_frame}_{310e}$.

⟨*lib-build/tools/swapper.c* 311b⟩≡ 513e▷ [311b]

```
#include "../ulixlib.h"
#define THRESHOLD init_frames - 500
int main () {
  setterm (9);
  int init_frames = get_free_frames ();
  int last_free_frames;
  int free_frames = init_frames;
  int pid = getpid ();
  unsigned int counter = 0;

  for (;;) {
    last_free_frames = free_frames;
    free_frames = get_free_frames ();
    if (free_frames != last_free_frames) {
      printf ("[%d.%d] swapper: %d free frames. threshold = %d.",
        pid, counter++, free_frames, THRESHOLD);
      if (free_frames < THRESHOLD) {
        printf ("calling free_a_frame (%d < %d)\n", free_frames, init_frames - 500);
        free_a_frame ();
      } else {
        printf ("\n");
      }
    }
  }
}
```

Uses free_a_frame 310e, free_frames 112b, get_free_frames 310e, getpid 223b, main 44b, printf 601a, setterm 328g, and THRESHOLD.

We will start this swapper process right from the `init` process; it will run with process ID 2. In order to stop arbitrary processes from calling $\texttt{free\_a\_frame}_{310e}$, the system call handler should verify that it was called by this process (and no other one).

10 Talking to the Hardware

In this chapter we provide the code which talks to various kinds of hardware. In most cases this will include an interrupt handler which gets called when a device generates a hardware interrupt.

10.1 Keyboard

Ulix does not provide a graphical user interface, and it does not recognize a mouse. Thus, the keyboard is the only available input device. Since there will be up to ten virtual consoles (on which users can log on with different user accounts), we need several keyboard input buffers and keep track of where to store a new character when a key was pressed.

10.1.1 Scan Code Tables

The keyboard interrupt handler must recognize which key was pressed, while also checking if any of the *modifier keys* (such as shift, control or alt) was held down at the same time.

modifier keys

The array `scancode_table`$_{316}$ maps the *key codes* of a standard US keyboard (as generated by the keyboard controller) to ASCII characters. We started with the code in Bran's Kernel Development tutorial [Fri05] (the table is on the `http://www.osdever.net/bkerndev/Docs/keyboard.htm` page), but modified it.

key codes

Similarly, `scancode_up_table`$_{316}$ holds the characters for the same key codes, but with one of the shift keys pressed. Since we alternated between US and German keyboards during the development of Ulix, we also provide corresponding tables for the German layout which you can find in `scancode_DE_table`$_{317}$ and `scancode_DE_up_table`$_{317}$.

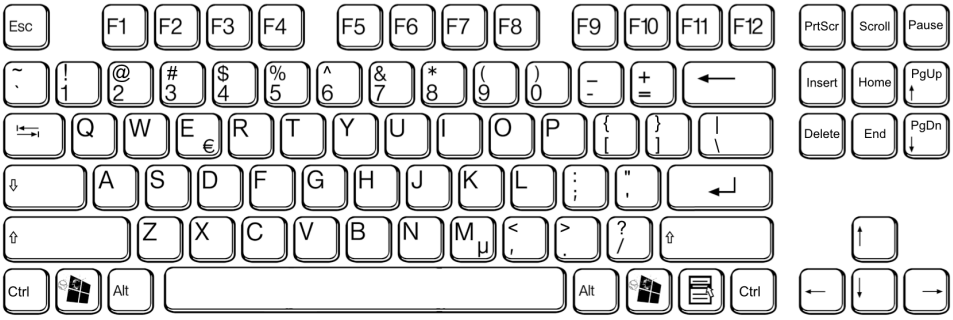


Figure 10.1: Layout of a US keyboard with additional Windows keys (without the number pad)

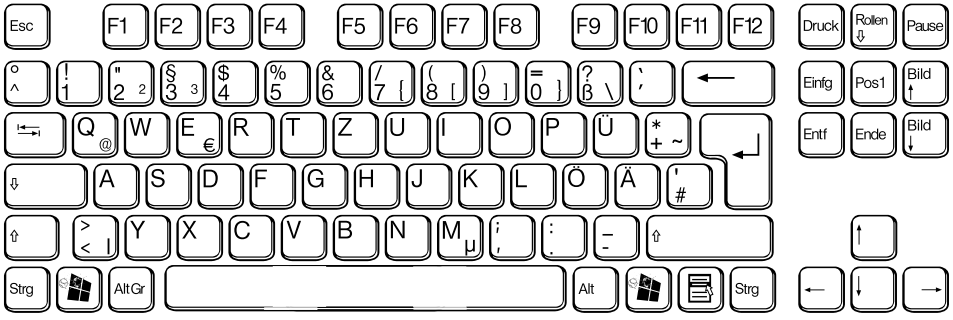


Figure 10.2: Layout of a German keyboard with additional Windows keys (without the number pad)

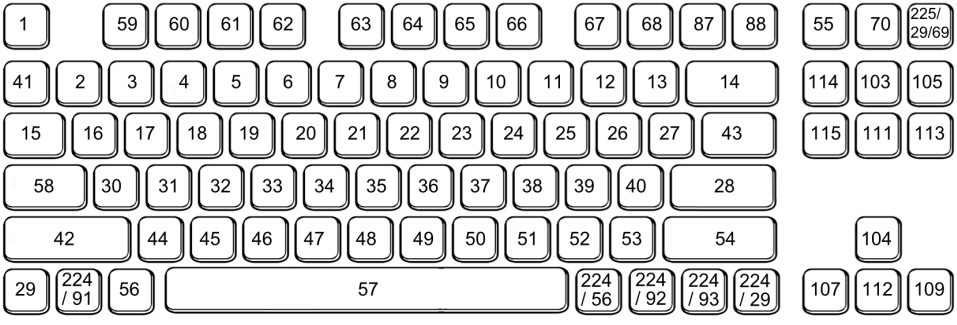


Figure 10.3: Scancodes for the US keyboard; on a German keyboard “<” generates the key code 41.

Figure 10.1 shows the layout of a standard US-American PC keyboard, and Figure 10.2 shows the German layout. In the third figure (Figure 10.3) you can find the key codes.

Both pressing and releasing a key generate a key code (that way the operating system can see whether the user holds a key pressed). The key codes for pressing and releasing any specific key are identical except for the upper bit: If a key was pressed, the key code's upper bit is unset (0); if it was released it is set (1). Thus `scancode & 0x80` is 0 if the event is a key press event, it is non-zero otherwise. In the latter case `scancode-0x80` (or `scancode & ~0x80`) calculates the key code of the corresponding key press event.

There are some exceptions for newer keys which did not exist on the original PC XT keyboard [IBM83, pp. 1-65–1-69], and they use combinations which are initiated with an escape character (`0xe0` = 224 or `0xe1` = 225). Figure 10.3 shows this for the two Windows keys, the (Windows) menu key and the right Alt and Ctrl keys. Note how Left-Alt and Right-Alt (or Left-Ctrl and Right-Ctrl) only differ in that the right keys generate the escape code and then the same code as the corresponding left key, e. g., 29 for Left-Ctrl and 224 / 29 for Right-Ctrl. This way a driver that is unaware of escape codes will just ignore the escape code and interpret the second code (almost) correctly.

In the "Keyboard scancodes" list [Bro09], Brouwer describes the newer keys, too. He also notes:

> "The prefix `e0` was originally used for the grey duplicates of keys on the original PC/XT keyboard. These days `e0` is just used to expand code space. The prefix `e1` used for Pause/Break indicated that this key sends the make/break sequence at make time, and does nothing upon release."

The terms *scan codes* and *key codes* are sometimes used interchangeably, but there are other encodings of key-press and key-release events. We only discuss the key codes that are transmitted by the keyboard controller. They are also called "set 1" or "IBM PC XT" scan codes. A complete overview of "set 1" and "set 2" scan codes can also be found in a Microsoft specification document [Mic00a]. scan codes

All 0 entries in the map make Ulix ignore a key. We also enter 0 in the map for modifier keys (Shift, Ctrl, Alt etc.) since we handle them separately. For the Escape and cursor keys we provide names because we will use them later:

⟨ *public constants* 46a ⟩+≡ (44a 48a) ◁298b 326b▷ [315]

```
#define KEY_ESC     27
#define KEY_UP     191
#define KEY_DOWN   192
#define KEY_LEFT   193
#define KEY_RIGHT  194
```

Defines:
KEY_DOWN, used in chunks 316 and 317.
KEY_LEFT, used in chunks 316 and 317.
KEY_RIGHT, used in chunks 316 and 317.
KEY_UP, used in chunks 316 and 317.
Uses KEY_ESC.

This is the table for the US keyboard:

[316] ⟨*global variables* 92b⟩+≡ (44a) ◁310f 317▷

```
byte scancode_table[128] = {
  /*  0.. 9 */    0,  KEY_ESC, '1', '2', '3', '4', '5', '6', '7', '8',
  /* 10..19 */   '9', '0', '-', '=', '\b',       /* Backspace */
                 '\t', /* Tab */   'q', 'w', 'e', 'r',
  /* 20..29 */   't', 'y', 'u', 'i', 'o', 'p', '[', ']',
                 '\n', /* Enter */  0, /* Control */
  /* 30..39 */   'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',
  /* 40..49 */   '\'', '`', 0, /* Left shift */  '\\', 'z', 'x', 'c', 'v', 'b', 'n',
  /* 50..59 */   'm', ',', '.', '/', 0, /* Right shift */
                 '*', 0, /* Alt */  ' ', /* Space bar */
                 0, /* CapsLock */  0, /* F1 */
  /* 60..69 */   0, 0, 0, 0, 0, 0, 0, 0, 0, /* F2..F10 */  0, /* NumLock */
  /* 70..79 */   0, /* Scroll Lock */  0, /* Home */  KEY_UP, 0, /* Page Up */
                 '-', KEY_LEFT, 0, KEY_RIGHT,  '+', 0, /* End */
  /* 80..89 */   KEY_DOWN, 0, /* Page Down */  0, /* Insert */  0, /* Delete */
                 0, 0, 0, 0, /* F11 */  0, /* F12 */   0,
  /* 90..127     not defined */
};

byte scancode_up_table[128] = {
  /*  0.. 9 */    0,  KEY_ESC, '!', '@', '#', '$', '%', '^', '&', '*',
  /* 10..19 */   '(', ')', '_', '+', '\b',       /* Backspace */
                 '\t', /* Tab */   'Q', 'W', 'E', 'R',
  /* 20..29 */   'T', 'Y', 'U', 'I', 'O', 'P', '{', '}',
                 '\n', /* Enter */  0, /* Control */
  /* 30..39 */   'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', ':',
  /* 40..49 */   '"', '~', 0, /* Left shift */  '|', 'Z', 'X', 'C', 'V', 'B', 'N',
  /* 50..59 */   'M', '<', '>', '?', 0, /* Right shift */
                 '*', 0, /* Alt */  ' ', /* Space bar */
                 0, /* CapsLock */  0, /* F1 */
  /* 60..69 */   0, 0, 0, 0, 0, 0, 0, 0, 0, /* F2..F10 */  0, /* NumLock */
  /* 70..79 */   0, /* Scroll Lock */   0, /* Home */  KEY_UP, 0, /* Page Up */
                 '-', KEY_LEFT, 0, KEY_RIGHT, '+', 0, /* End */
  /* 80..89 */   KEY_DOWN, 0, /* Page Down */  0, /* Insert */  0, /* Delete */
                 0, 0, 0, 0, /* F11 */  0, /* F12 */   0,
  /* 90..127     not defined */
};
```

Defines:
scancode_table, used in chunk 319d.
scancode_up_table, used in chunk 319d.
Uses KEY_DOWN 315, KEY_ESC, KEY_LEFT 315, KEY_RIGHT 315, and KEY_UP 315.

ULIX does not support German special characters (äöüÄÖÜß§), so the keys which would generate those characters are mapped to standard ASCII characters which can then be entered via two keys, for example, pressing [Ä] or [Shift-Ä] will generate the ' and " characters. Users can switch between the US and German layouts by pressing [Ctrl-L].

⟨*global variables* 92b⟩+≡ (44a) ◁316 318c▷ [317]

```
byte scancode_DE_table[128] = {
  /*  0.. 9 */   '^',  KEY_ESC, '1', '2', '3', '4', '5', '6', '7', '8',
  /* 10..19 */   '9', '0', '-', '\'', '\b',      /* Backspace */
                 '\t', /* Tab */  'q', 'w', 'e', 'r',
  /* 20..29 */   't', 'z', 'u', 'i', 'o', 'p', '[', '+',
                 '\n', /* Enter */  0, /* Control */
  /* 30..39 */   'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',
  /* 40..49 */   '\'', '<', 0, /* Left shift */  '#', 'y', 'x', 'c', 'v', 'b', 'n',
  /* 50..59 */   'm', ',', '.', '-', 0, /* Right shift */
                 '*', 0, /* Alt */  ' ', /* Space bar */
                 0, /* CapsLock */  0, /* F1 */
  /* 60..69 */   0, 0, 0, 0, 0, 0, 0, 0, 0, /* F2..F10 */  0, /* NumLock */
  /* 70..79 */   0, /* Scroll Lock */   0, /* Home */  KEY_UP, 0, /* Page Up */
                 '-', KEY_LEFT, 0, KEY_RIGHT, '+', 0, /* End */
  /* 80..89 */   KEY_DOWN, 0, /* Page Down */  0, /* Insert */    0, /* Delete */
                 0, 0, 0, 0, /* F11 */  0, /* F12 */   0,
  /* 90..127     not defined */
};

byte scancode_DE_up_table[128] = {
  /*  0.. 9 */   '^',  KEY_ESC, '!', '"', '#', '$', '%', '&', '/', '(',
  /* 10..19 */   ')', '=', '?', '`', '\b',       /* Backspace */
                 '\t', /* Tab */   'Q', 'W', 'E', 'R',
  /* 20..29 */   'T', 'Y', 'U', 'I', 'O', 'P', '{', '*',
                 '\n', /* Enter */  0, /* Control */
  /* 30..39 */   'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', ':',
  /* 40..49 */   '"', '>', 0, /* Left shift */  '\'', 'Z', 'X', 'C', 'V', 'B', 'N',
  /* 50..59 */   'M', ';', ':', '_', 0, /* Right shift */
                 '*', 0, /* Alt */  ' ', /* Space bar */
                 0, /* CapsLock */  0, /* F1 */
  /* 60..69 */   0, 0, 0, 0, 0, 0, 0, 0, 0, /* F2..F10 */  0, /* NumLock */
  /* 70..79 */   0, /* Scroll Lock */   0, /* Home */  KEY_UP, 0, /* Page Up */
                 '-', KEY_LEFT, 0, KEY_RIGHT, '+', 0, /* End */
  /* 80..89 */   KEY_DOWN, 0, /* Page Down */  0, /* Insert */    0, /* Delete */
                 0, 0, 0, 0, /* F11 */  0, /* F12 */   0,
  /* 90..127     not defined */
};
```

Defines:
`scancode_DE_table`, used in chunk 319d.
`scancode_DE_up_table`, used in chunk 319d.
Uses `KEY_DOWN` 315, `KEY_ESC`, `KEY_LEFT` 315, `KEY_RIGHT` 315, and `KEY_UP` 315.

10.1.2 Virtual Consoles

We provide ten virtual consoles (terminals), each of which has its own keyboard buffer. Such a buffer can store up to 32 characters—if an application does not react fast enough to key-press events, the buffer can become full: in that case further key-presses are lost.

[318a] ⟨*constants* 112a⟩+≡ (44a) ◁308b 319c▷
```
#define SYSTEM_KBD_BUFLEN 32
#define TERMINALS 10
```
Defines:
SYSTEM_KBD_BUFLEN, used in chunks 318b, 321b, 324a, and 416b.
TERMINALS, used in chunks 318c and 324a.

For each buffer we also store the current position (where the next character will be entered) and the last read position (which character was last read):

[318b] ⟨*type definitions* 91⟩+≡ (44a) ◁295c 325b▷
```
typedef struct {
  char kbd[SYSTEM_KBD_BUFLEN+1];
  int kbd_pos;
  int kbd_lastread;
  int kbd_count;
} terminal_t;
```
Defines:
terminal_t, used in chunks 318c, 321b, 324a, and 416b.
Uses SYSTEM_KBD_BUFLEN 318a.

The kbd_count field is redundant but makes checking the buffer status simpler.

[318c] ⟨*global variables* 92b⟩+≡ (44a) ◁317 318d▷
```
terminal_t terminals[TERMINALS] = { { { 0 } } };
```
Defines:
terminals, used in chunks 318, 321b, 324a, 326c, 381b, and 416b.
Uses terminal_t 318b and TERMINALS 318a.

Terminal 0 is also used as the system terminal. ULIX provides a kernel mode shell that can be activated with Shift-Esc. It always uses the first terminal and keeps its own set of position variables.

[318d] ⟨*global variables* 92b⟩+≡ (44a) ◁318c 319b▷
```
char *system_kbd = terminals[0].kbd;
int system_kbd_pos;
int system_kbd_lastread;
int system_kbd_count;
```
Defines:
system_kbd_count, used in chunks 318e and 610a.
system_kbd_lastread, used in chunks 318e and 610a.
system_kbd_pos, used in chunks 318e and 610a.
Uses terminals 318c.

They need to be initialized when the system boots:

[318e] ⟨*setup keyboard* 318e⟩≡ (44b)
```
system_kbd_pos = 0;
system_kbd_lastread = -1;
system_kbd_count = 0;
for (int i = 0;  i < 10;  i++)
  terminals[i].kbd_lastread = -1;
```
Uses system_kbd_count 318d, system_kbd_lastread 318d, system_kbd_pos 318d, and terminals 318c.

10.1.3 Keyboard Interrupt Handler

The keyboard handler deals will all press and release events:

⟨*function prototypes* 45a⟩+≡ (44a) ◁306f 323a▷ [319a]
```
void keyboard_handler (context_t *r);
```

It checks the variable `LANG_GERMAN`$_{319b}$ to decide whether it shall use the German or the US keyboard layout:

⟨*global variables* 92b⟩+≡ (44a) ◁318d 323d▷ [319b]
```
boolean LANG_GERMAN = 1;   // default: german keyboard
```
Defines:
`LANG_GERMAN`, used in chunks 319d and 321a.

The implementation is rather simple, the function is only long because it needs to handle key presses differently when one of the modifier keys (Left-Shift, Right-Shift, Ctrl, Alt) is held while another key is pressed. Other than that, the handler reads a scan code from the keyboard I/O port.

⟨*constants* 112a⟩+≡ (44a) ◁318a 320a▷ [319c]
```
#define IO_KEYBOARD 0x60
```
Defines:
`IO_KEYBOARD`, used in chunk 320b.

and interprets it. For standard keys it looks up the assigned character using one of the scan code tables. Key-release events are ignored unless one of the modifier keys was released: in that case the status of `shift_pressed`, `alt_pressed` etc. must be updated. We declare those variables as `static` in the function so that they keep their values between several invocations of the handler.

⟨*function implementations* 100b⟩+≡ (44a) ◁297 323b▷ [319d]
```
void keyboard_handler (context_t *r) {
  char *lower_table; char *upper_table;
  if (LANG_GERMAN) {
    lower_table = scancode_DE_table; upper_table = scancode_DE_up_table;
  } else {
    lower_table = scancode_table;    upper_table = scancode_up_table;
  }

  static boolean shift_pressed = false;  static boolean left_shift_pressed  = false;
  static boolean alt_pressed   = false;  static boolean right_shift_pressed = false;
  static boolean ctrl_pressed  = false;
  ⟨keyboard handler implementation 320b⟩
}
```
Defines:
`keyboard_handler`, used in chunks 319a and 323b.
Uses `context_t` 142a, `LANG_GERMAN` 319b, `scancode_DE_table` 317, `scancode_DE_up_table` 317, `scancode_table` 316, and `scancode_up_table` 316.

After initializing the keyboard mapping and the states of the modifier keys the real work begins. We read the scan code from the I/O port `IO_KEYBOARD`$_{319c}$. Then we check if the scan code corresponds to a key release event (i. e., the highest bit is set). That situation

is only of interest for the modifier keys: If Shift, Ctrl or Alt were released, we update the corresponding static variable and return immediately. The release of regular keys is ignored. We also give the modifier key numbers names to make the code more readable:

[320a] ⟨*constants* 112a⟩+≡ (44a) ◁319c 325a▷

```
#define KEY_CTRL     29
#define KEY_L_SHIFT  42
#define KEY_R_SHIFT  54
#define KEY_ALT      56
```

Defines:
KEY_ALT, used in chunk 320.
KEY_CTRL, used in chunk 320.
KEY_L_SHIFT, used in chunk 320.
KEY_R_SHIFT, used in chunk 320.

[320b] ⟨*keyboard handler implementation* 320b⟩≡ (319d) 320c▷

```
byte scancode = inportb (IO_KEYBOARD);   // read scan code from keyboard
if (scancode & 0x80) {                   // release key event
  switch (scancode & ~0x80) {
    case KEY_CTRL:    ctrl_pressed        = false; break;
    case KEY_L_SHIFT: left_shift_pressed  = false; break;
    case KEY_R_SHIFT: right_shift_pressed = false; break;
    case KEY_ALT:     alt_pressed         = false; break;
  }
  shift_pressed = left_shift_pressed || right_shift_pressed;
  return;
}
```

Uses inportb 133b, IO_KEYBOARD 319c, KEY_ALT 320a, KEY_CTRL 320a, KEY_L_SHIFT 320a, and KEY_R_SHIFT 320a.

Otherwise we deal with a key press event. To keep things ordered nicely, we start with checking whether one of the modifier keys was pressed: Again, we can update a state variable and return from the handler. If that was not the case, we look up the character in the right scan code table (either upper_table or lower_table):

[320c] ⟨*keyboard handler implementation* 320b⟩+≡ (319d) ◁320b 321a▷

```
// press key event
switch (scancode) {
  case KEY_CTRL:    ctrl_pressed  = true;                          return;
  case KEY_L_SHIFT: shift_pressed = left_shift_pressed  = true; return;
  case KEY_R_SHIFT: shift_pressed = right_shift_pressed = true; return;
  case KEY_ALT:     alt_pressed   = true;                          return;
}

byte c = (shift_pressed ? upper_table[scancode] : lower_table[scancode]);
```

Uses KEY_ALT 320a, KEY_CTRL 320a, KEY_L_SHIFT 320a, and KEY_R_SHIFT 320a.

kernel mode shell

Then we check for special key combinations: Alt-0 to Alt-9 let us switch to a different terminal, Ctrl-C kills the current process, Ctrl-L changes the keyboard layout, and Shift-Escape starts the *kernel mode shell* (which can be used for debugging, see Chapter 17).

⟨*keyboard handler implementation* 320b⟩+≡ (319d) ◁320c 321b▷ [321a]

```
// Alt-0 to Alt-9: switch terminal
if (alt_pressed && '0' ≤ c && c ≤ '9') {
  vt_activate ((int)((c-'0')+9)%10);   // activate virtual console
  vt_move_cursor ();                   // update cursor on new terminal
  return;
};

// Ctrl-C: kill and reset input
if (ctrl_pressed && c == 'c') {
  ⟨keyboard handler: find active process, set target_pid 322b⟩
  u_kill (target_pid, SIGKILL);        // kill the process
  return;
}

// Ctrl-L: change keyboard layout
if (ctrl_pressed && c == 'l') {
  switch (LANG_GERMAN) {
    case 0:  LANG_GERMAN = 1;  _set_statusline ("de", 44);  return;
    case 1:  LANG_GERMAN = 0;  _set_statusline ("en", 44);  return;
  }
}

// Shift-Escape: start kernel mode shell
if (shift_pressed && c == KEY_ESC && scheduler_is_active) {
  ⟨disable scheduler 276b⟩
  printf ("\nGoing to kernel shell\n");
  vt_activate (0);                     // must run on vt0
  kernel_shell ();
  printf ("returning from kernel shell\n");
  return;
};
```

Uses _set_statusline 337b, kernel_shell 610a, KEY_ESC, kill 568b, LANG_GERMAN 319b, printf 601a, scheduler_is_active 276e, SIGKILL 562a, target_pid, u_kill 562b, vt_activate 327a, and vt_move_cursor 328a.

With all special cases handled, only the default case remains: If a regular character was entered, we need to store it in one of the keyboard buffers—as long as it is not filled already. So we first check whether the buffer can carry the new character:

⟨*keyboard handler implementation* 320b⟩+≡ (319d) ◁321a [321b]

```
terminal_t *term = &terminals[cur_vt];
if (term->kbd_count < SYSTEM_KBD_BUFLEN) {
  if (ctrl_pressed && c ≥ 'a' && c ≤ 'z') c -= 96;  // Ctrl
  term->kbd[term->kbd_pos] = c;
  term->kbd_pos = (term->kbd_pos + 1) % SYSTEM_KBD_BUFLEN;
  term->kbd_count++;
  if (scheduler_is_active) { ⟨keyboard handler: wake sleeping process 322a⟩ }
}
```

Uses cur_vt 326a, scheduler_is_active 276e, SYSTEM_KBD_BUFLEN 318a, terminal_t 318b, and terminals 318c.

We still need to discuss what happens when a process is sleeping (while waiting for input from its terminal). We search the `keyboard_queue`$_{323d}$ (that we will define in the following section) for a process which waits for input and uses the currently active terminal `cur_vt`$_{326a}$. If we find one (and we assume that for each terminal at most one process can wait for key entry) we wake it up, i. e., move it to the ready queue using the `deblock`$_{186b}$ function:

[322a] ⟨*keyboard handler: wake sleeping process* 322a⟩≡ (321b 381b)

```
thread_id start_pid = keyboard_queue.next;
if (start_pid != 0) {              // only if the queue is not empty
  thread_id search_pid = start_pid;
  do {
    if (thread_table[search_pid].terminal == cur_vt) {
      deblock (search_pid, &keyboard_queue);
      break;
    } else {
      search_pid = thread_table[search_pid].next;
    }
  } while (search_pid != start_pid && search_pid != 0);
}
```

Uses `cur_vt` 326a, `deblock` 186b, `keyboard_queue` 323d, `thread_id` 178a, and `thread_table` 176b.

A Ctrl-C key combination should make the system deliver a `SIGKILL`$_{562a}$ signal to the process that uses the current terminal. There may be several such processes; we will pick the first one which has no child process:

[322b] ⟨*keyboard handler: find active process, set* `target_pid` 322b⟩≡ (321a)

```
int target_pid = 0;
for (int i = 3;  i < MAX_THREADS;  i++) {
  if (thread_table[i].used && (thread_table[i].terminal == cur_vt)) {
    int is_candidate = true;
    for (int j = 3;  j < MAX_THREADS;  j++) {
      if (thread_table[j].used && (thread_table[j].ppid == i)) {
        // thread j has parent i - not a candidate
        is_candidate = false;
        break;   // leave inner loop
      }
    }
    if (is_candidate) {
      target_pid = i;
      goto end_of_search;
    }
  }
}
end_of_search:
;  // label needs a statement
```

Uses `cur_vt` 326a, `MAX_THREADS` 176a, `target_pid`, and `thread_table` 176b.

During system initialization we register the keyboard handler:

⟨*function prototypes* 45a⟩+≡ (44a) ◁319a 323f▷ [323a]
```
void keyboard_install ();
```

⟨*function implementations* 100b⟩+≡ (44a) ◁319d 324a▷ [323b]
```
void keyboard_install () {
  install_interrupt_handler (IRQ_KBD, keyboard_handler);
  enable_interrupt (IRQ_KBD);
}
```
Defines:
`keyboard_install`, used in chunk 323.
Uses `enable_interrupt` 140b, `install_interrupt_handler` 146c, `IRQ_KBD` 132, and `keyboard_handler` 319d.

We add calling $\texttt{keyboard\_install}_{323b}$ to the general chunk that installs interrupt handlers:

⟨*install the interrupt handlers* 139b⟩+≡ (45b) ◁139b [323c]
```
keyboard_install ();
```
Uses `keyboard_install` 323b.

10.1.4 The Keyboard Queue

We provide several blocked queues—one for each different reason that a process may block for. Here we define the queue for processes that wait for a keystroke (on their terminal).

⟨*global variables* 92b⟩+≡ (44a) ◁319b 326a▷ [323d]
```
blocked_queue keyboard_queue;   // processes which wait for a keystroke
```
Defines:
`keyboard_queue`, used in chunks 322a, 323e, 416b, 564c, and 606.
Uses `blocked_queue` 183a.

We must initialize the queue:

⟨*initialize system* 45b⟩+≡ (44b) ◁218c 326c▷ [323e]
```
initialize_blocked_queue (&keyboard_queue);
```
Uses `initialize_blocked_queue` 183c and `keyboard_queue` 323d.

Now we can provide two functions which read in a character or a whole string:

⟨*function prototypes* 45a⟩+≡ (44a) ◁323a 326e▷ [323f]
```
void kgetch    (char *c);
void kreadline (char *s, int maxlength);
```

They are only used for the kernel mode shell, processes have their own way of reading characters from the keyboard; they use the regular file $\texttt{read}_{429b}$ function with the $\texttt{STDIN\_FILENO}_{415b}$ file descriptor because their input might be redirected to a file. We will describe this in Chapter 12.

In kernel mode we can just run a loop that waits for a new character to appear in the keyboard buffer.

[324a] ⟨*function implementations* 100b⟩+≡ (44a) ◁323b 324b▷

```
void kgetch (char *c) {
  int t = thread_table[current_task].terminal;
  if (t < 0 || t > TERMINALS-1) {
    t = 0;  printf ("ERROR: terminal not set! setting to 0\n");
  }
  terminal_t *term = &terminals[t];

  *c = 0;
  while (*c == 0) {
    if (term->kbd_count > 0) {
      term->kbd_count--;
      term->kbd_lastread = (term->kbd_lastread+1) % SYSTEM_KBD_BUFLEN;
      *c = term->kbd[term->kbd_lastread];
    } else {
      *c = 0;
    };
  };
};
```

Defines:
kgetch, used in chunk 324b.
Uses current_task 192c, printf 601a, SYSTEM_KBD_BUFLEN 318a, terminal_t 318b, TERMINALS 318a, terminals 318c, and thread_table 176b.

The kreadline$_{324b}$ function repeatedly calls kgetch$_{324a}$ until a newline character is read (which terminates the input).

[324b] ⟨*function implementations* 100b⟩+≡ (44a) ◁324a 327a▷

```
void kreadline (char *s, int maxlength) {
  char c;
  int pos = 0;
  for (;;) {
    ⟨enable interrupts 47b⟩
    kgetch (&c);                                  // read one character
    if (c == 0x08 && pos > 0) {                   // backspace
      pos--;
      kputch (c); kputch (' '); kputch (c);
    } else if ( c == '\n' ) {                     // newline: end of input
      kputch ('\n');
      s[pos] = (char) 0;
      return;
    } else if (c != 0x08 && pos < maxlength) {  // other character
      kputch (c);
      s[pos++] = c;
    };
  };
};
```

Defines:
kreadline, used in chunks 323f and 610a.
Uses kgetch 324a and kputch 335b.

10.2 Terminals

We want Ulix to provide *several terminals* so that we can run a few login shells and execute programs on them.

Conceptually, providing terminals is not complicated: we need

- memory to store the contents of the terminals – roughly 80 x 25 x 2 bytes per terminal (the size of the textmode video buffer),
- a way to make Ulix switch the active terminal,
- a modification of the `write`$_{429b}$`()` functions so that they will either write to the current terminal or a specified terminal.
- When writes to a terminal occur, the terminal's screen buffer is updated—if it is the active terminal, the screen is updated at the same time.
- When switching to a different terminal, its screen buffer is copied to the screen.

We start with the required memory. Since Ulix uses the last line on the screen for displaying a status line, we consider it not to be part of any terminal buffer; for example scrolling shall always ignore the last line, and from a process' point of view the 25th line does not exist. So we can define

⟨ *constants* 112a ⟩+≡ (44a) ◁320a 325c▷ [325a]

```
#define VT_WIDTH  (80)
#define VT_HEIGHT (24)
#define VT_SIZE   (VT_WIDTH * VT_HEIGHT * 2)
```

Defines:
VT_HEIGHT, used in chunk 334.
VT_SIZE, used in chunks 325–27, 329b, 332b, and 337b.
VT_WIDTH, used in chunks 329b and 334a.

⟨ *type definitions* 91 ⟩+≡ (44a) ◁318b 360a▷ [325b]

```
typedef struct {
  char mem[VT_SIZE];
  int x,y;
} term_buffer;
```

Defines:
term_buffer, used in chunks 326a, 334b, and 335b.
Uses VT_SIZE 325a.

Two bytes are required for each character; the first one holds the ASCII value of the symbol to be displayed, the second is used for foreground and background colors.

We want the system to use up to ten virtual consoles (numbered from 0 to 9), so we create an array for them:

⟨ *constants* 112a ⟩+≡ (44a) ◁325a 327c▷ [325c]

```
#define MAX_VT 9
```

Defines:
MAX_VT, used in chunks 326–28.

[326a] ⟨*global variables* 92b⟩+≡ (44a) ◁323d 327b▷
```
term_buffer vt[MAX_VT+1];
int cur_vt = 0;
```
Defines:
cur_vt, used in chunks 321, 322, 327a, 329b, 330a, 332b, 334b, 335b, and 342b.
vt, used in chunks 326–30, 332b, 334b, and 335b.
Uses MAX_VT 325c and term_buffer 325b.

$\texttt{vt}_{326a}\texttt{[i].mem}$ is the buffer of console `i`, and $\texttt{vt}_{326a}\texttt{[i].x}$ and $\texttt{vt}_{326a}\texttt{[i].y}$ hold the current cursor position in console `i`. We initialize the current terminal to number 0.

To start with proper contents, we initialize each of the ten text buffers with blanks. A blank character is actually a `word` with the low byte containing the ASCII value of the blank symbol (`0x20`) and the high byte containing the color information (`0x0F` for white on black).

[326b] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁315 328d▷
```
#define VT_NORMAL_BACKGROUND (0x0F << 8)
#define VT_BLUE_BACKGROUND   (0x1F << 8)
#define VT_RED_BACKGROUND    (0x4F << 8)
```
Defines:
VT_BLUE_BACKGROUND, used in chunks 329b and 609.
VT_NORMAL_BACKGROUND, used in chunks 326c, 329b, and 333–35.
VT_RED_BACKGROUND, used in chunk 609.

[326c] ⟨*initialize system* 45b⟩+≡ (44b) ◁323e 509b▷
```
int vtno;
word *memptr;
unsigned blank = 0x20 | VT_NORMAL_BACKGROUND;  // blank character
for (vtno = 1;  vtno < 10;  vtno++) {
  memptr = (word*)vt[vtno].mem;
  memsetw (memptr, blank, VT_SIZE/2);
}
printf ("VT:  Initialized ten terminals (press [Alt-1] to [Alt-0])\n");
```
Uses memsetw 596c, printf 601a, terminals 318c, vt 326a, VT_NORMAL_BACKGROUND 326b, and VT_SIZE 325a.

Note that we do *not* initialize the first terminal's buffer $\texttt{vt}_{326a}\texttt{[0]}$ because it will obtain a copy of the current screen when we switch to a different terminal.

We also need a way to tell a process what terminal it runs on, so we add a new $\texttt{TCB}_{175}$ entry:

[326d] ⟨*more TCB entries* 158c⟩+≡ (175) ◁255d 424c▷
```
int terminal;
```

A regular Unix system would allow for a more complex setup, but for Ulix we restrict ourselves to using ten text consoles.

Activating a console via

[326e] ⟨*function prototypes* 45a⟩+≡ (44a) ◁323f 327d▷
```
int vt_activate (int i);
```

is the simplest of all the operations:

⟨*function implementations* 100b⟩+≡ (44a) ◁324b 328a▷ [327a]
```
int vt_activate (int new_vt) {
  if (new_vt < 0 || new_vt > MAX_VT)  return -1;        // no such console
  else {
    memcpy (vt[cur_vt].mem, (void*)VIDEORAM, VT_SIZE);  // save old contents
    vt[cur_vt].x = csr_x; vt[cur_vt].y = csr_y;
    memcpy ((void*)VIDEORAM, vt[new_vt].mem, VT_SIZE);  // load new contents
    cur_vt = new_vt;
    csr_x = vt[new_vt].x; csr_y = vt[new_vt].y;
    vt_move_cursor ();
    return 0;
  }
}
```
Defines:
  vt_activate, used in chunks 321a and 326e.
Uses cur_vt 326a, MAX_VT 325c, memcpy 596c, VIDEORAM 327b, vt 326a, vt_move_cursor 328a, and VT_SIZE 325a.

Here we're using the address VIDEORAM$_{327b}$ to access the text mode frame buffer of the graphics card; csr_x and csr_y store the cursor position on the visible terminal. We have not defined the variables yet, so here they are:

⟨*global variables* 92b⟩+≡ (44a) ◁326a 328b▷ [327b]
```
uint VIDEORAM = 0xB8000;
byte csr_x = 0; byte csr_y = 0;   // Cursor position
```
Defines:
  VIDEORAM, used in chunks 116, 327a, 332b, 334b, 337b, and 342d.

It is initially set to 0xb8000 but changes its value to 0xd00b8000 during system initialization (when we set up paging). We can also use textmemptr$_{116c}$ which was #defined as ((word*)VIDEORAM$_{327b}$).

update cursor location

With vt_move_cursor$_{328a}$ we update the cursor location, since it will need to be in a different position on the new terminal. The cursor location can be controlled by sending $80 \cdot x + y$ (where x is the line number and y is the column number) to the VGA cursor location register. This is a 16 bit value—it must be sent in two chunks. First the control code IO_VGA_CURSOR_LOC_HIGH$_{327c}$ is sent to the IO_VGA_TARGET$_{327c}$ port (which signals that the high byte of the cursor location follows), then that high byte is sent to the IO_VGA_VALUE$_{327c}$ port. A similar sequence follows, using IO_VGA_CURSOR_LOC_LOW$_{327c}$ and the lower byte.

⟨*constants* 112a⟩+≡ (44a) ◁325c 338d▷ [327c]
```
#define IO_VGA_TARGET           0x3D4
#define IO_VGA_VALUE            0x3D5
#define IO_VGA_CURSOR_LOC_HIGH  14
#define IO_VGA_CURSOR_LOC_LOW   15
```
Defines:
  IO_VGA_CURSOR_LOC_HIGH, used in chunk 328a.
  IO_VGA_CURSOR_LOC_LOW, used in chunk 328a.
  IO_VGA_TARGET, used in chunk 328a.

⟨*function prototypes* 45a⟩+≡ (44a) ◁326e 329a▷ [327d]
```
void vt_move_cursor ();
```

[328a] ⟨*function implementations* 100b⟩+≡ (44a) ◁327a 328c▷
```
void vt_move_cursor () {
  unsigned position = csr_y * 80 + csr_x;
  // high byte:
  outportb (IO_VGA_TARGET, IO_VGA_CURSOR_LOC_HIGH);
  outportb (0x3D5, position >> 8);
  // low byte:
  outportb (IO_VGA_TARGET, IO_VGA_CURSOR_LOC_LOW);
  outportb (0x3D5, position & 0xff);  // low byte
}
```
Defines:
vt_move_cursor, used in chunks 321a, 327, 329b, 330a, and 335b.
Uses IO_VGA_CURSOR_LOC_HIGH 327c, IO_VGA_CURSOR_LOC_LOW 327c, IO_VGA_TARGET 327c, and outportb 133b.

Let's define what terminal we expect to display kernel messages. We initialize the variable KERNEL_VT$_{328b}$ to 0 (for the first terminal), though it may later be changed.

[328b] ⟨*global variables* 92b⟩+≡ (44a) ◁327b 338a▷
```
short int KERNEL_VT = 0;
```
Defines:
KERNEL_VT, used in chunks 334b and 335b.

Back to terminal selection, we provide a system call that lets a process choose which terminal to use.

[328c] ⟨*function implementations* 100b⟩+≡ (44a) ◁328a 329b▷
```
void syscall_setterm (context_t *r) {
  int vt = r->ebx;                          // argument in ebx register
  if (vt<0 || vt>MAX_VT) { return; }        // check if proper number...
  thread_table[current_task].terminal = vt;
};
```
Defines:
syscall_setterm, used in chunk 328e.
Uses context_t 142a, current_task 192c, MAX_VT 325c, thread_table 176b, and vt 326a.

We define the system call number and register the syscall:

[328d] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁326b 415b▷
```
#define __NR_setterm  511
```
Defines:
__NR_setterm, used in chunk 328.

[328e] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁310c 331b▷
```
install_syscall_handler (__NR_setterm, syscall_setterm);
```
Uses __NR_setterm 328d, install_syscall_handler 201b, and syscall_setterm 328c.

The user mode library gains a new function as well:

[328f] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁310d 331c▷
```
void setterm (int vt);
```

[328g] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁310e 331d▷
```
void setterm (int vt) { syscall2 (__NR_setterm, (uint) vt); }
```
Defines:
setterm, used in chunks 311b and 513e.
Uses __NR_setterm 328d, syscall2 203c, and vt 326a.

We also need functions to clear the screen, set the cursor and get the current cursor location:

⟨*function prototypes* 45a⟩+≡ (44a) ◁327d 332a▷ [329a]
```
void vt_clrscr ();
void vt_get_xy (char *x, char *y);
void vt_set_xy (char x, char y);
```

clearing the screen

Clearing the screen means writing a blank character to each location. We need to consider that each character byte is followed by a format byte and—if calling the function from the kernel—we want to format the last line with a blue background so that the status line can be recognized.

`vt_clrscr`$_{329b}$ just overwrites the terminal buffer of the current process with blank characters and then calls `vt_set_xy`$_{330a}$ to set the cursor to the top left position. If the current process is also working on the currently visible terminal, the function updates the physical screen as well. (Otherwise the change will only become visible when the user switches to that terminal.)

⟨*function implementations* 100b⟩+≡ (44a) ◁328c 330a▷ [329b]
```
void vt_clrscr () {
  word blank    = 0x20 | VT_NORMAL_BACKGROUND;
  word blankrev = 0x20 | VT_BLUE_BACKGROUND;
  int process_term;
  if (scheduler_is_active) {
    process_term = thread_table[current_task].terminal;
    word *memptr = (word*)vt[process_term].mem;
    memsetw (memptr, blank, VT_SIZE/2);                    // lines 1–24
    vt_set_xy (0, 0);
  }

  // current terminal?
  if ((!scheduler_is_active) || (scheduler_is_active && process_term == cur_vt))
    memsetw (textmemptr, blank, VT_SIZE/2);                // lines 1–24

  // kernel mode? clear status line, set cursor
  if (!scheduler_is_active) {
    memsetw (textmemptr + VT_SIZE/2, blankrev, VT_WIDTH);  // line  25
    csr_x = csr_y = 0;
    vt_move_cursor ();
  }
}
```
Defines:
`vt_clrscr`, used in chunks 331a, 337c, and 608b.
Uses `cur_vt` 326a, `current_task` 192c, `memsetw` 596c, `scheduler_is_active` 276e, `textmemptr` 116c, `thread_table` 176b, `vt` 326a, `VT_BLUE_BACKGROUND` 326b, `vt_move_cursor` 328a, `VT_NORMAL_BACKGROUND` 326b, `vt_set_xy` 330a, `VT_SIZE` 325a, and `VT_WIDTH` 325a.

The vt_get_xy$_{330a}$ and vt_set_xy$_{330a}$ read respectively set the x and y members of the current terminal's term_buffer$_{325b}$ structure. We will only call them from processes, so we need not check for as many special cases as we did in vt_clrscr$_{329b}$. The only condition we have to check is whether we're changing the cursor location of the currently active terminal—then we also need to update the hardware cursor.

[330a] ⟨*function implementations* 100b⟩+≡ (44a) ◁329b 332b▷

```
void vt_get_xy (char *x, char *y) {
  int process_term = thread_table[current_task].terminal;
  *x = vt[process_term].x;
  *y = vt[process_term].y;
}

void vt_set_xy (char x, char y) {
  int process_term = thread_table[current_task].terminal;
  vt[process_term].x = x;
  vt[process_term].y = y;

  // current terminal?
  if (process_term == cur_vt) {
    csr_x = x; csr_y = y;
    vt_move_cursor ();
  }
}
```

Defines:
vt_get_xy, used in chunk 331a.
vt_set_xy, used in chunks 329 and 331a.
Uses cur_vt 326a, current_task 192c, thread_table 176b, vt 326a, and vt_move_cursor 328a.

We provide three system calls

[330b] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁309 370c▷

```
void syscall_clrscr (context_t *r);
void syscall_get_xy (context_t *r);
void syscall_set_xy (context_t *r);
```

for these functions:

[330c] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁310b 332c▷

```
#define __NR_clrscr   512
#define __NR_get_xy   513
#define __NR_set_xy   514
```

Defines:
__NR_clrscr, used in chunk 331.
__NR_get_xy, used in chunk 331.
__NR_set_xy, used in chunk 331.

As usual, the system call handlers evaluate the parameters by looking at the registers *EBX* and *ECX* (if there are any), then they call the above functions.

⟨ *syscall functions* 174b ⟩+≡ (202b) ◁310a 332d▷ [331a]

```
void syscall_clrscr (context_t *r) {
  // no parameters, no return value
  vt_clrscr ();
}

void syscall_get_xy (context_t *r) {
  // ebx: address of x position (char)
  // ecx: address of y position (char)
  vt_get_xy ((char*)r->ebx, (char*)r->ecx);
}

void syscall_set_xy (context_t *r) {
  // ebx: x position (char)
  // ecx: y position (char)
  vt_set_xy ((char)r->ebx, (char)r->ecx);
}
```

Defines:
`syscall_clrscr`, used in chunk 331b.
`syscall_get_xy`, used in chunk 331b.
`syscall_set_xy`, used in chunks 330b and 331b.
Uses `context_t` 142a, `vt_clrscr` 329b, `vt_get_xy` 330a, and `vt_set_xy` 330a.

And we add those system calls to the system:

⟨ *initialize syscalls* 173d ⟩+≡ (44b) ◁328e 333a▷ [331b]

```
install_syscall_handler (__NR_clrscr, syscall_clrscr);
install_syscall_handler (__NR_get_xy, syscall_get_xy);
install_syscall_handler (__NR_set_xy, syscall_set_xy);
```

Uses `__NR_clrscr` 330c, `__NR_get_xy` 330c, `__NR_set_xy` 330c, `install_syscall_handler` 201b, `syscall_clrscr` 331a, `syscall_get_xy` 331a, and `syscall_set_xy` 331a.

Via the user mode library we provide the functionality to processes:

⟨ *ulixlib function prototypes* 174c ⟩+≡ (48a) ◁328f 333b▷ [331c]

```
void clrscr ();
void get_xy (char *x, char *y);
void set_xy (char x, char y);
```

⟨ *ulixlib function implementations* 174d ⟩+≡ (48b) ◁328g 333c▷ [331d]

```
void clrscr ()                { syscall1 (__NR_clrscr);                }
void get_xy (char *x, char *y) { syscall3 (__NR_get_xy, (int) x, (int) y); }
void set_xy (char x, char y)   { syscall3 (__NR_set_xy, (int) x, (int) y); }
```

Defines:
`set_xy`, used in chunk 331c.
Uses `__NR_clrscr` 330c, `__NR_get_xy` 330c, `__NR_set_xy` 330c, `syscall1` 203c, and `syscall3` 203c.

To make life easier for the application programmer (who cannot access the screen memory directly) we also provide functions which allow reading or writing the whole screen (that is: 24 lines of 80 characters; the last line on the 80×25 display is reserved for the operating system). For this purpose we implement the `read_screen`$_{332b}$ and `write_screen`$_{332b}$ functions and let applications call them via system calls.

[332a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁329a 335a▷
```
void read_write_screen (char *buf, boolean read_flag);
void read_screen (char *buf);
void write_screen (char *buf);
```

[332b] ⟨*function implementations* 100b⟩+≡ (44a) ◁330a 334a▷
```
void read_write_screen (char *buf, boolean read_flag) {
  // if read_flag == true: read from screen, otherwise write
  int process_term = thread_table[current_task].terminal;
  char *video_address = (char*) vt[process_term].mem;

  if (read_flag) {
    memcpy (buf, video_address, VT_SIZE);    // read the screen
  } else {
    memcpy (video_address, buf, VT_SIZE);    // write the screen
    // current terminal?
    if (process_term == cur_vt)
      memcpy ((char*)VIDEORAM, video_address, VT_SIZE);
  }
}

void read_screen (char *buf)  { read_write_screen (buf, true);  }
void write_screen (char *buf) { read_write_screen (buf, false); }
```
Defines:
read_screen, used in chunk 333e.
read_write_screen, used in chunk 332d.
write_screen, used in chunks 332 and 333.
Uses cur_vt 326a, current_task 192c, memcpy 596c, thread_table 176b, VIDEORAM 327b, vt 326a, and VT_SIZE 325a.

In the system call handlers we call read_write_screen$_{332b}$ instead of read_screen$_{332b}$ and write_screen$_{332b}$ to save the extra function call:

[332c] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁330c 370b▷
```
#define __NR_read_screen   515
#define __NR_write_screen  516
```
Uses __NR_read_screen and __NR_write_screen.

[332d] ⟨*syscall functions* 174b⟩+≡ (202b) ◁331a 370d▷
```
void syscall_read_screen (context_t *r) {
  // ebx: buffer address
  read_write_screen ((char *) r->ebx, true);
}

void syscall_write_screen (context_t *r) {
  // ebx: buffer address
  read_write_screen ((char *) r->ebx, false);
}
```
Defines:
syscall_read_screen, used in chunk 333a.
syscall_write_screen, used in chunk 333a.
Uses context_t 142a and read_write_screen 332b.

⟨ *initialize syscalls* 173d ⟩+≡ (44b) ◁331b 370e▷ [333a]
```
install_syscall_handler (__NR_read_screen,  syscall_read_screen);
install_syscall_handler (__NR_write_screen, syscall_write_screen);
```
Uses __NR_read_screen, __NR_write_screen, install_syscall_handler 201b, syscall_read_screen 332d, and syscall_write_screen 332d.

Again, we add these to the library:

⟨ *ulixlib function prototypes* 174c ⟩+≡ (48a) ◁331c 333d▷ [333b]
```
void read_screen (char *buf);
void write_screen (char *buf);
```

⟨ *ulixlib function implementations* 174d ⟩+≡ (48b) ◁331d 333e▷ [333c]
```
void read_screen (char *buf)   { syscall2 (__NR_read_screen, (uint) buf);  }
void write_screen (char *buf)  { syscall2 (__NR_write_screen, (uint) buf); }
```
Defines:
read_screen, used in chunk 333e.
write_screen, used in chunks 332 and 333.
Uses __NR_read_screen, __NR_write_screen, and syscall2 203c.

scrolling in user mode

Applications can use read_screen$_{332b}$ and write_screen$_{332b}$ for scrolling. Here's a simple scroll function which scrolls the user mode part of the screen (lines 1–24) one line "up" (that means: the first lines disappears, and all other lines move up one line, leaving one blank line at the bottom)

⟨ *ulixlib function prototypes* 174c ⟩+≡ (48a) ◁333b 373d▷ [333d]
```
void scroll_up ();
void scroll_down ();
```

⟨ *ulixlib function implementations* 174d ⟩+≡ (48b) ◁333c 373e▷ [333e]
```
void scroll_up () {
  char buffer[80*25*2];  // we reserve space for 25 (!) lines
  word blank = 0x20 | VT_NORMAL_BACKGROUND; // blank character
  read_screen ((char*)buffer);
  memsetw ((word*)((char*)buffer + 80*24*2), blank, 80);
  write_screen ((char*)buffer + 160);
}

void scroll_down () {
  char buffer[80*25*2];  // we reserve space for 25 (!) lines
  word blank = 0x20 | VT_NORMAL_BACKGROUND; // blank character
  read_screen ((char*)buffer + 160);
  memsetw ((word*)((char*)buffer), blank, 80);
  write_screen ((char*)buffer);
}
```
Defines:
scroll_up, used in chunk 333d.
Uses memsetw 596c, read_screen 332b 333c, VT_NORMAL_BACKGROUND 326b, and write_screen 332b 333c.

scrolling in kernel mode

For scrolling from inside the kernel, we provide a helper function that can "scroll" any screen-sized chunk of memory. In our case that is an area of 24 lines à 80 characters, each of which is 2 bytes large ($24 \times 80 \times 2 = 3840$ bytes), and scrolling it means to move lines 2–24 to lines 1–23 and empty line 24.

[334a] ⟨*function implementations* 100b⟩+≡ (44a) ◁332b 334b▷

```
void vt_scroll_mem (word *address) {
  word blank = ' ' | VT_NORMAL_BACKGROUND;  // space + format
  memcpy  (address, address + VT_WIDTH, (VT_HEIGHT-1) * VT_WIDTH * 2);
  memsetw (address + (VT_HEIGHT-1) * VT_WIDTH, blank, VT_WIDTH);
}
```

Defines:
vt_scroll_mem, used in chunk 334b.
Uses memcpy 596c, memsetw 596c, VT_HEIGHT 325a, VT_NORMAL_BACKGROUND 326b, and VT_WIDTH 325a.

Note that this function uses pointer arithmetic: `address` is of type `word*`, i. e., a pointer to a 16-bit wide integer. That means that when we add e. g. $\texttt{VT\_WIDTH}_{325a}$ to `address`, the resulting address is actually `2 *` $\texttt{VT\_WIDTH}_{325a}$ higher.

[334b] ⟨*function implementations* 100b⟩+≡ (44a) ◁334a 335b▷

```
void vt_scroll () {
  term_buffer *term;
  short int target_vt;
  if (scheduler_is_active) {
    target_vt = thread_table[current_task].terminal;
    term = &vt[target_vt];
  } else {
    target_vt = KERNEL_VT;   // kernel: default write to 0
  }

  if (cur_vt == target_vt && csr_y ≥ VT_HEIGHT) {
    vt_scroll_mem ((word*)VIDEORAM);
    csr_y = VT_HEIGHT-1;
  }

  if (scheduler_is_active && term->y ≥ VT_HEIGHT) {
    vt_scroll_mem ((word*)term->mem);
    term->y = VT_HEIGHT-1;
  }
}
```

Defines:
vt_scroll, used in chunk 335b.
Uses cur_vt 326a, current_task 192c, KERNEL_VT 328b, scheduler_is_active 276e, term_buffer 325b, thread_table 176b, VIDEORAM 327b, vt 326a, VT_HEIGHT 325a, and vt_scroll_mem 334a.

10.2.1 Terminal Output

The next two functions

⟨*function prototypes* 45a⟩+≡ (44a) ◁332a 336a▷ [335a]
```
void kputch (byte c);
void kputs (char *text);
```

write a character or a string to the screen.

The $\texttt{kputch}_{335b}$ function is based on the `scrn.c` function of Bran's kernel tutorial [Fri05] but was modified a lot.

⟨*function implementations* 100b⟩+≡ (44a) ◁334b 336b▷ [335b]
```
void kputch (byte c) {
  // check if we're writing to current terminal
  term_buffer *term;
  short int target_vt;
  word *where;
  if (scheduler_is_active) {
    target_vt = thread_table[current_task].terminal;
    term = &vt[target_vt];
  } else {
    target_vt = KERNEL_VT;   // kernel: default write to 0
  }

  switch (c) {
    case '\b': // backspace, move cursor back
               if (cur_vt == target_vt) { if (csr_x   != 0) csr_x--;   }
               if (scheduler_is_active) { if (term->x != 0) term->x--; }
               break;

    case '\r': // carriage return, go back to first column
               if (cur_vt == target_vt) { csr_x = 0;   }
               if (scheduler_is_active) { term->x = 0; }
               break;

    case '\n': // newline, go to next line, first column
               if (cur_vt == target_vt) { csr_x = 0;    csr_y++;   }
               if (scheduler_is_active) { term->x = 0;  term->y++; }
               break;
  }

  if (c ≥ ' ') {   // normal character
    if (cur_vt == target_vt) {
      where = textmemptr + (csr_y * 80 + csr_x);
      *where = c | VT_NORMAL_BACKGROUND;
      csr_x++;
    }
    if (scheduler_is_active) {
      where = (word*)term->mem + (term->y * 80 + term->x);
```

```
        *where = c | VT_NORMAL_BACKGROUND;
        term->x++;
      }
    }

    if (csr_x ≥ 80) {   // end of line reached
      if (cur_vt == target_vt) { csr_x = 0;    csr_y++;   }
      if (scheduler_is_active) { term->x = 0;  term->y++; }
    }

    vt_scroll ();   // scroll if necessary
    if (cur_vt == target_vt) { vt_move_cursor (); };

    // write to serial console
    if (c == '\b') {  //  backspace
      uartputc ('\b'); uartputc (' '); uartputc ('\b');
    } else uartputc (c);
  }

  void kputs (char *text) {
    while (*text != 0)
      kputch (*(text++));
  }
```

Defines:
kputch, used in chunks 324b, 417, 598a, 605b, 611b, and 613b.
kputs, used in chunks 108, 115d, 121b, 335a, 603, 604b, 608b, and 610–13.
Uses cur_vt 326a, current_task 192c, KERNEL_VT 328b, scheduler_is_active 276e, term_buffer 325b, textmemptr 116c, thread_table 176b, uartputc 336b, vt 326a, vt_move_cursor 328a, VT_NORMAL_BACKGROUND 326b, and vt_scroll 334b.

For writing to the serial console, kputch$_{335b}$ uses the helper function

[336a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁335a 337a▷

```
void uartputc (int c);
```

which sends a character to the I/O port IO_COM1$_{344a}$. This is useful for running Ulix in the qemu PC emulator [B$^+$14] which can display serial line output in the terminal window of the host machine, see also Section 10.4 on serial ports.

[336b] ⟨*function implementations* 100b⟩+≡ (44a) ◁335b 337b▷

```
void uartputc (int c) {
  // taken from the xv6 operating system [CKM12], uart.c
  if (!uart[0]) return;      // leave if we have no first serial port
  // wait until COM1 is ready to receive another byte
  for (int i = 0; i < 128 && !(inportb (IO_COM1+5) & 0x20); i++) ;
  outportb (IO_COM1+0, c);   // write the byte
}
```

Defines:
uartputc, used in chunks 335b, 336a, and 598a.
Uses inportb 133b, IO_COM1 344a, outportb 133b, and uart 344b.

10.2.2 Status Line Management

The last line on the screen is reserved for the Ulix status line. We provide two functions which let the kernel display status messages: kernel status messages

⟨*function prototypes* 45a⟩+≡ (44a) ◁336a 338b▷ [337a]
```
void set_statusline (char *text);
void _set_statusline (char *text, int offset);
```

The first function, `set_statusline`$_{337b}$, always writes to the start of the status line, whereas `_set_statusline`$_{337b}$ takes an extra position argument and can be used to update a small location somewhere in the middle of the line.

⟨*function implementations* 100b⟩+≡ (44a) ◁336b 338c▷ [337b]
```
void set_statusline (char *text) { _set_statusline (text, 0); }

void _set_statusline (char *text, int offset) {
  int i = 0;
  uint videoaddress = VIDEORAM + VT_SIZE+2*offset;  // last line of video
  while ((*text != 0) && (i < 80)) {
    POKE (videoaddress + 2*i, *text);
    i++; text++;
  }
}
```
Defines:
`_set_statusline`, used in chunks 276, 280a, 321a, 342b, 343b, and 512b.
`set_statusline`, used in chunks 337 and 608–10.
Uses `POKE` 117, `VIDEORAM` 327b, and `VT_SIZE` 325a.

10.2.3 Initializing the Screen

When we described the kernel initialization in the `main`$_{44b}$ function, we promised to define the code chunk ⟨*setup video* 337c⟩ in this chapter—here it is: We use `vt_clrscr`$_{329b}$ to clear the screen and `set_statusline`$_{337b}$ to display the OS name and version.

⟨*setup video* 337c⟩≡ (44b) [337c]
```
vt_clrscr ();
set_statusline (UNAME);
printf ("%s                          Build: %s\n", UNAME, BUILDDATE);
```
Uses `BUILDDATE` 35a, `printf` 601a, `set_statusline` 337b, `UNAME` 35a, and `vt_clrscr` 329b.

Remember that we've set `UNAME`$_{35a}$ and `BUILDDATE`$_{35a}$ at the very beginning.

10.3 System Timer

clock chip
timer interrupt

A central hardware component is the *clock chip* which regularly causes a *timer interrupt*. By adding a timer handler, the kernel can regularly check whether some administrative action is necessary. The most important action is calling the scheduler: Without the timer handler we would have to live without preemptive multi-tasking.

There are many other tasks for the timer handler, for example we will use it to keep track of time in our system. Every time the timer handler runs, we will increment a `system_ticks`$_{338a}$ variable that must be initialized at system start. `system_time`$_{338a}$ will be set to Unix time (the seconds since the *Unix epoch*, 1 January 1970, 00:00:00 UTC) so that we can properly display the date and time and update timestamps in the filesystem.

Unix epoch

[338a] ⟨*global variables* 92b⟩+≡ (44a) ◁328b 339c▷

```
unsigned int system_ticks = 0; // updated 100 times a second
unsigned int system_time;      // unix time (in seconds)
```

Defines:
`system_ticks`, used in chunks 306d, 311a, 342, and 343b.
`system_time`, used in chunks 342c, 343b, 475c, 478b, and 605a.

10.3.1 Setting the Frequency

timer frequency

Initially the clock chip is pre-set to a weird frequency (≈ 18.222 Hz), we change that to 100 Hz when the system starts. The function

[338b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁337a 340d▷

```
void timer_phase (int hz);
```

adjusts the timer's frequency: It first announces that it wants to set the frequency by sending `0x36` to port `IO_CLOCK_COMMAND`$_{338d}$ and then sends the lower and the higher eight bits of the divisor `1193180 / hz` to the port `IO_CLOCK_CHANNEL0`$_{338d}$ which is responsible for configuring timer 0 (the system timer) [vG94, p. 794].

[338c] ⟨*function implementations* 100b⟩+≡ (44a) ◁337b 340e▷

```
void timer_phase (int hz) {
  // source: http://www.osdever.net/bkerndev/Docs/pit.htm
  int divisor = 1193180 / hz;                    // calculate divisor
  outportb (IO_CLOCK_COMMAND,  0x36);            // set command byte 0x36
  outportb (IO_CLOCK_CHANNEL0, divisor & 0xFF);  // set low byte of divisor
  outportb (IO_CLOCK_CHANNEL0, divisor >> 8);    // set high byte of divisor
};
```

Defines:
`timer_phase`, used in chunks 338b and 339a.
Uses `IO_CLOCK_CHANNEL0` 338d, `IO_CLOCK_COMMAND` 338d, and `outportb` 133b.

with

[338d] ⟨*constants* 112a⟩+≡ (44a) ◁327c 339b▷

```
#define IO_CLOCK_COMMAND   0x43
#define IO_CLOCK_CHANNEL0  0x40
```

Defines:
IO_CLOCK_CHANNEL0, used in chunk 338c.
IO_CLOCK_COMMAND, used in chunk 338c.

When the kernel runs through the initialization steps, we let it set the frequency and install the timer interrupt handler (whose implementation we will discuss soon) for interrupt number $\texttt{IRQ\_TIMER}_{132}$ (0). While we're at it, we also query the current date and time.

⟨*install the timer* 339a⟩≡ (45b) [339a]
```
timer_phase (100);   // set timer to 100 Hz (100 interrupts/second)
install_interrupt_handler (IRQ_TIMER, timer_handler);
enable_interrupt (IRQ_TIMER);
⟨read date and time from CMOS 339d⟩
```
Uses enable_interrupt 140b, install_interrupt_handler 146c, IRQ_TIMER 132, timer_handler 342b, and timer_phase 338c.

10.3.2 Reading the Date and Time

Since it is a somewhat related task, we also query the PC's CMOS chip to find out what date and time it is [vG94, p. 746–747]:

⟨*constants* 112a⟩+≡ (44a) ◁338d 343a▷ [339b]
```
#define IO_CMOS_CMD  0x70
#define IO_CMOS_DATA 0x71
```
Defines:
IO_CMOS_CMD, used in chunks 339d and 552c.
IO_CMOS_DATA, used in chunks 339d and 552c.

⟨*global variables* 92b⟩+≡ (44a) ◁338a 344b▷ [339c]
```
unsigned long system_start_time = 0;
```
Defines:
system_start_time, used in chunks 340b and 342c.

⟨*read date and time from CMOS* 339d⟩≡ (339a) 340b▷ [339d]
```
// code adapted from http://wiki.osdev.org/CMOS
outportb (IO_CMOS_CMD,    0); byte second =  inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD,    2); byte minute =  inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD,    4); byte hour =    inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD,    7); byte day =     inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD,    8); byte month =   inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD,    9); word year =    inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD, 0x32); word century = inportb (IO_CMOS_DATA);
```
Uses hour, inportb 133b, IO_CMOS_CMD 339b, IO_CMOS_DATA 339b, and outportb 133b.

The values that the CMOS chip returns are *BCD*-encoded (binary-coded decimal; each half-byte encodes a decimal digit, from 0 = $\texttt{0000}_b$ to 9 = $\texttt{1001}_b$), so they have to be converted so that they make sense: To convert one BCD byte into a proper number, take the upper half times 10 (`(bcd >> 4) * 10`) and add the lower half (`bcd & 0x0f`): BCD

[340a] ⟨*macro definitions* 35a⟩+≡ (44a) ◁279a 471d▷
```
#define CONVERT_BCD(bcd) (((bcd >> 4) * 10) + (bcd & 0x0f))
```
Defines:
CONVERT_BCD, used in chunk 340b.

[340b] ⟨*read date and time from CMOS* 339d⟩+≡ (339a) ◁339d
```
second = CONVERT_BCD (second);  minute  = CONVERT_BCD (minute);
hour   = CONVERT_BCD (hour);    day     = CONVERT_BCD (day);
month  = CONVERT_BCD (month);   century = CONVERT_BCD (century);
year   = CONVERT_BCD (year) + 100 * century;
system_start_time = unixtime (year, month, day, hour, minute, second);
printf ("Current time: %4d/%02d/%02d %02d:%02d:%02d\n",
        year, month, day, hour, minute, second);
```
Uses CONVERT_BCD 340a, hour, printf 601a, system_start_time 339c, and unixtime 340e.

The year is only stored with two digits (e. g., 14 for the year 2014), so we have to add `100 * century`. Some CMOS chips return the hour in "12 hour time". For example, they would represent the hour value 23 as 11 and set the highest bit to indicate "pm" time. A formula that can cope with both types of BIOS is

[340c] ⟨*alternative hour transformation* 340c⟩≡
```
hour = ( (hour & 0x0F) + (((hour & 0x70) / 16) * 10) ) | (hour & 0x80);
```

—we do not use it since qemu returns "24 hour time". We need functions

[340d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁338b 342a▷
```
ulong unixtime (int year, int month, int day, int hour, int minute, int second);
void rev_unixtime (ulong unixtime, short *year, char *month, char *day,
                   char *hour, char *minute, char *second);
```

that convert between Unix time (seconds since 01/01/1970) and a time structure with year, month, day, hour, minute and second. You can skip the implementation of the following two functions since they are neither pretty to look at nor do they tell you anything about operating systems.

[340e] ⟨*function implementations* 100b⟩+≡ (44a) ◁338c 341▷
```
ulong unixtime (int year, int month, int day, int hour, int minute, int second) {
  // Source code taken from http://de.wikipedia.org/wiki/Unixzeit,
  // variable and function names translated to english
  const short days_since_start_of_year[12] =
    {0,31,59,90,120,151,181,212,243,273,304,334};
  unsigned long years=year-1970;
  int leapyears=((year-1)-1968)/4 - ((year-1)-1900)/100 + ((year-1)-1600)/400;

  ulong unix_time = second + 60*minute + 60*60*hour +
                    (days_since_start_of_year[month-1]+day-1)*60*60*24 +
                    (years*365+leapyears)*60*60*24;
  if ( (month>2) && (year%4==0 && (year%100!=0 || year%400==0)) )
    unix_time+=60*60*24;   // leap day?
  return unix_time;
}
```

Defines:
unixtime, used in chunk 340.
Uses hour and ulong 46b.

The function $\texttt{rev\_unixtime}_{341}$ is not used in the ULIX kernel at all, however the user mode program `ls` uses it, so we show it here for completeness. $\texttt{yearlength}_{341}$ is a helper function that returns the length of a year (either 364 or 365 for a leap year).

⟨*function implementations* 100b⟩+≡ (44a) ◁340e 342b▷ [341]

```
short yearlength (short year) {
  int res = 364;
  if ( ((year % 4 == 0) && ( year % 100 != 0)) || (year % 400 == 0) )  res++;
  return res;
}

void rev_unixtime (ulong utime, short *year, char *month, char *day,
                   char *hour, char *minute, char *second) {
  char days_per_month[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  int days = utime / (60*60*24);    char sec = utime % 60;
  char min = (utime/60) % 60;       char hou  = (utime/(60*60)) % 24;

  int yy = 1970;
  if (days > 15706) {   // speed up calculation for 2013 or later
    days -= 15706;
    yy += 43;
  }

  for (;;) {
    int l = yearlength (yy);
    if (days ≥ l) {
      yy++;
      days -= (l+1);    // distance between two years is l+1, not l
    } else break;
  }

  int mon = 1;
  for (;;) {
    int l = days_per_month[mon];
    if ((l == 2) && (yearlength (yy) == 365)) l++;
    if (days ≥ l) {
      mon++;
      days -= l;
    } else break;
  }

  days++;
  *year = yy;   *month  = mon; *day    = days;   // return results
  *hour = hou;  *minute = min; *second = sec;
}
```

Uses hour, min, sec, and ulong 46b.

10.3.3 Implementation of the Timer Handler

This is our timer interrupt handler:

[342a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁340d 345b▷
```
void timer_handler (context_t *r);
```

It executes all the timer tasks (as defined in the code chunk ⟨*timer tasks* 306d⟩ and updates the status line.

[342b] ⟨*function implementations* 100b⟩+≡ (44a) ◁341 344c▷
```
void timer_handler (context_t *r) {
  char buf[80];   // temporary buffer, can be used by all timer tasks
  ⟨timer tasks 306d⟩

  // show current terminal, free frames, current_as
  sprintf ((char*)&buf, "tty%d  FF=%04x  AS=%04d", cur_vt, free_frames, current_as);
  _set_statusline ((char*)&buf, 48);
}
```
Defines:
`timer_handler`, used in chunks 339a and 342a.
Uses `_set_statusline` 337b, `context_t` 142a, `cur_vt` 326a, `current_as` 170b, `free_frames` 112b, and `sprintf` 601a.

10.3.4 Tasks for the Timer

We need the timer handler to do several things which we collect in the ⟨*timer tasks* 306d⟩ code chunk. The first and easiest task is to modify the system uptime:

[342c] ⟨*timer tasks* 306d⟩+≡ (342b) ◁311a 342d▷
```
system_ticks++;    // one more timer interrupt
system_time = (uint)(system_ticks/100) + system_start_time;  // frequency: 100 Hz
```
Uses `system_start_time` 339c, `system_ticks` 338a, and `system_time` 338a.

Next, it calls the scheduler. It also displays a quickly changing progress character in the right top corner of the screen so that users can check that the scheduler is still active. If those signs stop spinning, something has gone wrong.

[342d] ⟨*timer tasks* 306d⟩+≡ (342b) ◁342c 343b▷
```
char sched_chars[] = "|/-\\";   // scheduler activity
static short sched_c = 0;       // next character to display

if (system_ticks % 5 == 0) {
  // cycle |/-\|- to show scheduler calls in upper right corner
  POKE (VIDEORAM + 79*2, sched_chars[sched_c]);
  sched_c++; sched_c %= 4;
  scheduler (r, SCHED_SRC_TIMER);
};
```
Uses `POKE` 117, `sched_chars`, `SCHED_SRC_TIMER` 343a, `scheduler` 276d, `system_ticks` 338a, and `VIDEORAM` 327b.

As mentioned earlier, the timer handler does not call `scheduler`$_{276d}$ if a resign action is currently active. When it does call the scheduler, it provides a second `SCHED_SRC_TIMER`$_{343a}$ argument to indicate that it was him who called. (The alternative is that `syscall_resign`$_{221a}$ called the scheduler which it announces by using `SCED_SRC_RESIGN`.)

⟨ *constants* 112a⟩+≡ (44a) ◁339b 344a▷ [343a]

```
#define SCHED_SRC_TIMER   0
#define SCHED_SRC_RESIGN  1
```

Defines:
SCHED_SRC_RESIGN, used in chunks 216b, 221a, and 278a.
SCHED_SRC_TIMER, used in chunk 342d.

In the status line at the bottom of the screen we display the current time; we want to update this display approximately every other second:

⟨ *timer tasks* 306d⟩+≡ (342b) ◁342d 546e▷ [343b]

```
short int sec,min,hour;

if (system_ticks % 100 == 0) {          // Every 100 clocks (approx. 1 second)
  hour = (system_time/60/60)%24;        // display the time
  min = (system_time/60)%60;
  sec = system_time%60;
  sprintf ((char*)&buf, "%02d:%02d:%02d", hour, min, sec);
  _set_statusline ((char*)&buf, 72);
}
```

Uses `_set_statusline` 337b, `hour`, `min`, `sec`, `sprintf` 601a, `system_ticks` 338a, and `system_time` 338a.

There are only two further places in the book where ⟨ *timer tasks* 306d⟩ gets an addition; we have decided not to place them in this chapter but at the places where the need for them arises. These are the chunks:

- Updating the counters for the page replacement code, p. 306
- Releasing the `swapper_lock`$_{310f}$ so that the `swapper` process can enter the next loop, p. 311

10.4 Serial Ports

Ulix supports two serial ports. It uses the first one to copy the regular output to a serial console and also writes kernel debug messages to that console. When running in a PC emulator which can redirect serial ports, these messages can be displayed in the terminal window from which Ulix was started. The `Makefile` in the `bin-build/` directory calls `qemu` with a `-serial` option and a pipe into the `tee` command

```
-serial mon:stdio | tee ulix.output
```

to simultaneously display the serial output in the terminal window and write it into a log file `ulix.output`.

The following code is borrowed from the xv6 operating system [CKM12], especially from source files `uart.c` and `console.c`. We modified the $\texttt{uartinit}_{344c}$ function so that it can deal with two serial ports.

Several I/O ports are used for sending data to the serial ports or reading from them, the base port numbers are the following:

[344a] ⟨*constants* 112a⟩+≡ (44a) ◁343a 363a▷

```
#define IO_COM1    0x3f8
#define IO_COM2    0x2f8
```

Defines:
`IO_COM1`, used in chunks 336b and 344c.
`IO_COM2`, used in chunks 344c, 345c, and 519d.

We use the array $\texttt{uart}_{344b}$ to keep track of available ports.

[344b] ⟨*global variables* 92b⟩+≡ (44a) ◁339c 363b▷

```
static int uart[2];    // do we have serial ports?
```

Defines:
`uart`, used in chunks 336b, 344c, 345c, and 519d.

[344c] ⟨*function implementations* 100b⟩+≡ (44a) ◁342b 345c▷

```
void uartinit (int serport) {
  char *p;
  word io_com, irq;
  switch (serport) {
    case 1: io_com = IO_COM1; irq = IRQ_COM1; break;
    case 2: io_com = IO_COM2; irq = IRQ_COM2; break;
    default: return;
  }

  outportb (io_com+2, 0);       // Turn off the FIFO
  // set 9600 baud, 8 data bits, 1 stop bit, parity off.
  outportb (io_com+3, 0x80);    // Unlock divisor
  outportb (io_com+0, 115200/9600);
  outportb (io_com+1, 0);
  outportb (io_com+3, 0x03);    // Lock divisor, 8 data bits.
  outportb (io_com+4, 0);
  outportb (io_com+1, 0x01);    // Enable receive interrupts.

  // If status is 0xFF, no serial port.
  if (inportb (io_com+5) == 0xFF) { return; }
  uart[serport-1] = 1;

  // Acknowledge pre-existing interrupt conditions; enable interrupts.
  inportb (io_com+2);
  inportb (io_com+0);
  enable_interrupt (irq);
}
```

Defines:
`uartinit`, used in chunk 345.
Uses `enable_interrupt` 140b, `inportb` 133b, `IO_COM1` 344a, `IO_COM2` 344a, `IRQ_COM1` 132, `IRQ_COM2` 132, `outportb` 133b, and `uart` 344b.

We start the serial port when booting:

⟨*setup serial port* 345a⟩≡ (44b) [345a]
```
uartinit (1);
```
Uses uartinit 344c.

To simplify disk access, we provide something we call a *serial hard disk* (see Chapter 13.4). Using the second serial port (of a virtual machine) we allow ULIX to connect to an external process which imitates a hard disk controller. ULIX can send simple commands to that process and in return will be served with data (1024 byte sectors) out of a hard disk image file.

serial hard disk

⟨*function prototypes* 45a⟩+≡ (44a) ◁342a 361a▷ [345b]
```
void uart2putc (int);
```

⟨*function implementations* 100b⟩+≡ (44a) ◁344c 361c▷ [345c]
```
void uart2putc (int c) {
  // taken from the xv6 operating system [CKM12], uart.c
  if (!uart[1]) return;      // leave if we have no second serial port
  // wait until COM2 is ready to receive another byte
  for (int i = 0;  i < 128 && !(inportb (IO_COM2+5) & 0x20);  i++) ;
  outportb (IO_COM2+0, c);   // write the byte
}
```
Defines:
uart2putc, used in chunks 345b, 517, 518, and 521a.
Uses inportb 133b, IO_COM2 344a, outportb 133b, and uart 344b.

(This function is almost identical to uartputc$_{336b}$ except that it uses IO_COM2$_{344a}$ instead of IO_COM1$_{344a}$, see page 336.)

⟨*setup serial hard disk* 345d⟩≡ (45c) 520a▷ [345d]
```
uartinit (2);
```
Uses uartinit 344c.

The interrupt handler for the second serial port will be implemented in Chapter 13.4. We need no handler for the first port because we only write to it.

11 Synchronization

In previous chapters, we had a look at the basic abstractions implemented by the operating system: virtual memory abstracting physical memory and virtual processors (threads) abstracting physical processors. Virtual processors may now execute concurrent programs in which the concurrent threads often have to interact in some specific way. There are two basic interaction patterns:

- A *competitive interaction pattern* occurs when two threads want to perform the same operation, however only one of them is permitted to do so at the same time. This means that one thread must go first and the other must wait until the first has finished his operation. Classic examples of this interaction pattern are accesses to exclusive resources or critical code sections. In this interaction pattern, the competing threads often don't know of each other so that some mediator (i. e., the operating system) has to synchronize the threads in a convenient and fair manner. competitive interaction
- A *cooperative interaction model* occurs when two threads know each other and want to exchange information in a well-defined way. This interaction pattern occurs for example in client/server-type systems where one thread requests information which another thread provides. cooperative interaction

The question for Ulix is: Which thread synchronization abstractions make sense and how can they be implemented?

Depending on the basic implementation mechanisms, we distinguish between memory-based synchronization abstractions and message-based synchronization abstractions. The most relevant ones for us are the former ones which are based on the availability of shared memory between threads. They can therefore be utilized in those operating systems using service combinations which primarily depend on the availability of shared memory.

Outline

We first present the central abstraction of competitive thread synchronization in Section 11.1. Then we go through different ways of implementing critical sections. While there are some purely software-based methods for achieving mutual exclusion that require no hardware support, they are rarely used; thus we immediately turn to more low-level and more practical synchronization techniques based on special hardware operations in Section 11.2.

We then climb again up the abstraction ladder and look at a higher-level concepts: *Semaphores* can be regarded as an operating system service which is more useable than low-level hardware. They are treated in Section 11.3. We will discuss the implementation of these concepts in ULIX as we go along. We present a standard implementation of semaphores based on atomic hardware operations.

Then we look at a specialization of semaphores, the *mutexes* (or *locks*), and show their implementation in ULIX in Sections 11.4 (for the kernel) and 11.5 (for threads in user mode). Finally, Section 11.6 discusses the important topic of kernel-level synchronization which is needed in situations where interrupt handlers and threads share common data—in those situations we cannot use blocking mutexes or semaphores because an interrupt handler must not block.

11.1 Critical Sections

We start with explaining the central concept: the *critical section*.

11.1.1 The Case of the Lost List Element

Consider an implementation of a linked list. This *could* for example be the implementation of the ready queue within the dispatcher (for the real implementation see Section 6.2.2). Imagine a list element consists of the real content of the element together with a pointer to the next list element. Adding an item to the front of the queue is usually implemented like `add_to_front` does it in the following example program and as is illustrated in Figure 11.1.

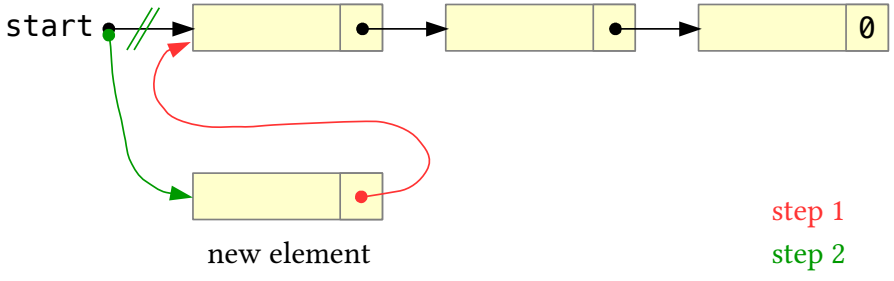


Figure 11.1: Example of adding an element to the front of a linked list.

⟨*code example: adding an item to a linked list* 349⟩≡ [349]

```
typedef struct element {
  int            value;
  struct element *next;
} element;

void add_to_front (element **first, element *e) {
  e->next = *first;  // operation 1
  *first  = e;       // operation 2
}

void show (element *first) {
  int i = 0; while (first != 0) {
    printf ("%d: contains %d\n", i++, first->value);  first = first->next;
  }
}

int main () {
  element a, b, c;  a.value = 100; a.next = &b; b.value = 101; b.next = 0;
  element *list = &a;        show (list);  // list = [ 100, 101 ]
  c.value = 102;
  add_to_front (&list, &c);  show (list);  // list = [ 102, 100, 101 ]
}
```

First, the `next` pointer of the new element is set to the "old" front of the list. Second, the global `list` pointer is set to the "new" front of the list. If you follow the final pointer structure you will see that the new list element has been correctly inserted at the front of the list.

The claim is now that the list implementation from above can cause problems if multiple threads try to put different elements into the list at the same time. To see this, consider Figure 11.2. There, two threads T_1 and T_2 invoke the implementation of `add_to_front` from above at almost the same time. The scheduling of the two threads is somewhat unfortunate in that T_1 is interrupted after the first operation, then T_2 adds its element, and then thread T_1 can finalize its insertion by executing the second operation. In total there are four pointer assignments, which are reflected in the figure. If you follow the final pointer structure of the ready queue, you will see that one element (namely that of thread T_2) has been lost: It is not contained in the list anymore.

The reason for this is the unfortunate scheduling of the machine instructions. Operation 2 of thread T_1 overwrites the effect of the two operations of thread T_2, because it implicitly assumes that nothing has happened after it executed its own operation 1. These problems would have been avoided if there were a guarantee that whenever some thread executes operation 1 it can also execute operation 2 without being interrupted.

11.1.2 Defining Critical Sections

A *critical section* is a sequence of instructions of a program which access shared resources. In the example above, the linked list is the *shared resource*. Manipulation of shared re- shared resource

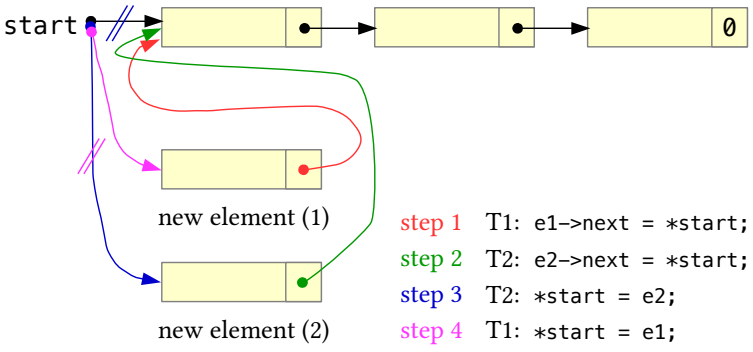


Figure 11.2: Concurrent threads trying to add two elements to a linked list: The list can lose elements when operations are interleaved in a special way.

entry/exit protocol

sources should be protected by an *entry* and *exit protocol*. These should guarantee mutual exclusion between critical sections. This is defined as follows:

Definition 1 (mutual exclusion) *At any time there is at most one thread executing within its critical section.*

Note that critical sections are something very abstract. They have a meaning at almost any level of abstraction, be it operating system, user program or programming language level. When dealing with critical sections it is merely necessary to mark the beginning and the end of the critical sections. The runtime system must then guarantee that no two critical sections at the same level of abstraction are executed concurrently.

In the following code examples, we mark beginning and end of critical sections with the two macros `ENTER_MUTEX` and `EXIT_MUTEX`. This is an abbreviation for entering and exiting mutual exclusion. So if we write our list operation from above again, we should mark the critical section in the following way:

[350] ⟨*code example: adding an item to a linked list within a critical section* 350⟩≡

```
void add_to_front (element **first, element *e) {
  ENTER_MUTEX ();
  e->next = *first;  // operation 1
  *first  = e;       // operation 2
  EXIT_MUTEX ();
}
```

We will learn about many ways to implement critical sections in this chapter. For the time being, imagine a global token which must be acquired before a thread can enter its critical section.

What we want to achieve is the behavior that you can see in Figure 11.3: Assume that there are two threads which share a resource, e. g., a memory location in the process that both threads belong to. Both threads contain code that performs an update on that memory address: It reads the value stored at the address, performs some calculation and

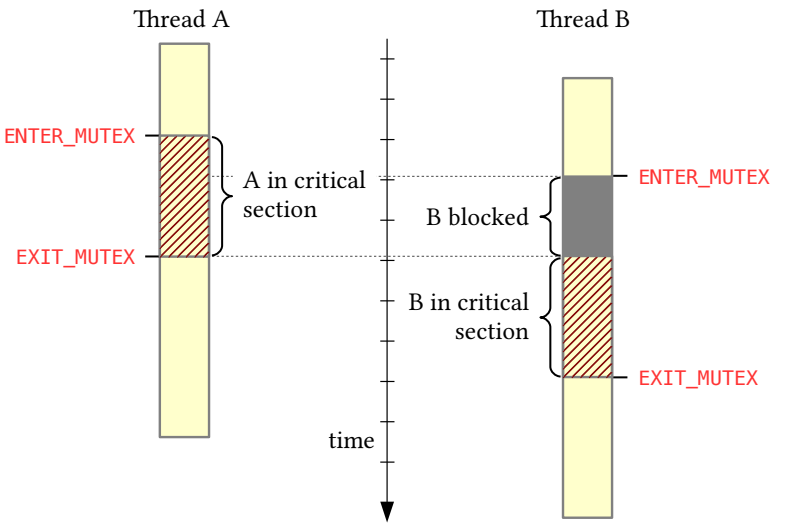


Figure 11.3: Simple example of mutual exclusion between two threads. Using a global token, one thread has to wait until the other thread returns the token to enter its critical section.

then writes back a new value to the same location. The whole code range from reading it in to writing it back is the critical section, and we want to make sure that they cannot overlap, turning the code block into an atomic action.

One of the threads (in the figure, it is thread A) will first reach the entry point of its critical section. Before it enters, it calls `ENTER_MUTEX`. Since at that time no other thread is in its own critical section, it can enter. Shortly afterwards, the other thread (thread B) also arrives at the entry point of its critical section: It must not pass, because thread A is still executing inside the critical section. Since it cannot continue, it will block.

After some time has passed, thread A finishes the work in the critical section and calls `EXIT_MUTEX`. Now we can let thread B pass and enter its critical section. Later it finishes the work and also leaves, calling `EXIT_MUTEX`, too.

You can think of the mutex as some global token that only one of the threads can possess and which is required to enter the critical section. Application programmers arrive at this situation all the time when they write multi-threaded programs, they use the synchronization features that the operating system provides, and our task is to implement this mechanism.

11.2 Hardware-based Synchronization

In this section we will consider synchronization based on explicit hardware support. We will look at the simplest thinkable mechanism first and then at more refined ways which are based on special CPU operations.

11.2.1 Disabling Interrupts

interrupt masking

The simplest way to achieve mutual exclusion on a single CPU is to switch off the interrupts, which is also often called *interrupt masking*. Every modern multi-purpose CPU which has an interrupt mechanism allows to disable certain or all interrupts. The effect is that the interrupt handler is not invoked when the interrupt is signaled. Hence, asynchronous interrupts, which are usually the source for non-atomicity in critical sections, can be effectively eliminated.

Conceptually, every CPU should offer instructions like `INTERRUPTS_OFF` and `INTERRUPTS_ON`. We have already defined code chunks ⟨*disable interrupts* 47a⟩ and ⟨*enable interrupts* 47b⟩ which perform these tasks on an Intel x86 processor via the assembler instructions `cli` (clear interrupt flag) and `sti` (set interrupt flag).

cli, sti

Whenever a kernel programmer needs to ensure mutual exclusion of a critical section in system mode, he will now have to write the following.

[352] ⟨*example: mutual exclusion using interrupt masking* 352⟩≡

```
⟨disable interrupts 47a⟩
// critical section
⟨enable interrupts 47b⟩
```

Note that masking interrupts can only be performed in system mode. (If normal programs could invoke the interrupt masking operations in user mode then they could monopolize the CPU.) Also, interrupts should only be disabled for relatively short periods of time. Otherwise, interrupts which are only flagged for a certain period of time like asynchronous I/O interrupts could be missed, leading to a possible *lost wakeup* (discussed later in Section 11.6.4.6). So overall, disabling interrupts is only advisable for rather short code sections within the kernel. Another disadvantage of this mechanism is that it only works for monoprocessor systems since turning off interrupts on one CPU does not affect the code executed on another CPU which could access shared memory data structures concurrently.

A trick makes is possible to improve the situation slightly: Using a global bit `busy` as a lock, we can extend the duration within a critical section without losing interrupts. The idea is to use the global lock bit as an indication whether some thread is within the critical section and just use interrupt masking to access this bit. The entry and exit protocols `ENTER_MUTEX` and `EXIT_MUTEX` for critical sections can then be programmed as follows.

⟨*example: mutual exclusion using global lock bit and interrupt masking* 353⟩≡ [353]

```
global boolean busy = false;      // no thread in critical section

void ENTER_MUTEX () {
  ⟨disable interrupts 47a⟩
  while (busy == true) {          // someone else in critical section
    ⟨enable interrupts 47b⟩
    NOP;                          // briefly leave interrupts on
    ⟨disable interrupts 47a⟩
  }
  busy = true;                    // I am in the critical section
  ⟨enable interrupts 47b⟩
}

void EXIT_MUTEX () {
  busy = false;                   // I've left the critical section
}
```

Two processes wishing to enter their critical sections will "race" for the lock bit during the entry protocol. The process which is able to switch off interrupts first will be able to grab the lock bit (in case it is free). If it is not free, some other process is in its critical section. So we have to turn on the interrupts at least for a short period of time to allow that process to interrupt and exit the critical section.

11.2.2 Using Special Hardware Instructions

Most processors today offer machine instructions which are specially tailored towards synchronization so that it can be achieved without having to mess around with the interrupts. The most common such instructions are either called *test-and-set* or *lock*. They are designed is such a way that mutual exclusion can be achieved by "grabbing a token"—that is similar to the use of the global lock bit above.

11.2.2.1 Test-and-Set

Assume you have a global lock bit `busy` which is initially `false`. The test-and-set instruction takes two arguments: the first is the name (or address) of the global lock bit, the second is a local variable. Invoking `Test-and-Set (&busy, &local)` then results in the following two actions performed as one atomic (i. e., uninterrupted) operation:

1. The value of `locked_bit` is copied into `local` ("test"), and
2. the `locked_bit` is set to `true` ("set").

In pseudocode this can be expressed as:

```
void Test-and-Set (*busy, *local) {
  *local = *busy;
  *busy = true;
}
```

The idea of this operation is that after it has been performed you can safely check whether you have "grabbed the token" or not. If you have grabbed the token, then the result of the operation (i. e., the value stored in `local`) should be `false` since it reflects the value of `busy` *before* it was set to `true`.

As an example for a "real" Test-and-Set operation, here is the description of the `gcc` built-in function `__sync_lock_test_and_set`[354a] [Int01, section 7.4.5, p. 61] which can be used in the way described above. The semantics of the function is as follows:

[354a] ⟨*compiler-internal functions* 354a⟩≡ 391b ▷

```
int __sync_lock_test_and_set (int *variable, int value) {
  int tmp = *variable;   // save old value
  *variable = value;     // set new value
  return tmp;            // return old value
}
```

The compiler (and in the end the processor) guarantees that all of this is executed atomically.

11.2.2.2 Lock Instruction

Another common machine instruction you can find is called `Lock`. Our presentation here follows Nehmer and Sturm [NS01] who introduce it in the form of a boolean function which implicitly refers to the global lock bit `busy`.

In pseudocode, `Lock` does the following:

```
boolean Lock () {
  tmp = busy;
  busy = true;
  return tmp;
}
```

In effect, `Lock` does the same as `Test-and-Set` in that it copies the value of `busy` *before* it is set to `true` and then returns this value. Note again that all this is done in an uninterruptible way.

11.2.2.3 Spin Locks

Now we can implement `ENTER_MUTEX` and `EXIT_MUTEX` without having to use privileged hardware instructions. We first have a look at the implementations based on `Test-and-Set`.

[354b] ⟨*example: synchronization using* `Test-and-Set` 354b⟩≡

```
boolean busy = false;

void ENTER_MUTEX() {
  repeat {
    Test-and-Set (&busy, &local);
  } until (local == false);
}
```

```
void EXIT_MUTEX() {
  busy = false;
}
```

Here is the implementation based on Lock.

⟨*example: synchronization using* Lock 355⟩≡ [355]

```
void ENTER_MUTEX() {
  while (Lock () == true) ; // loop over empty instruction
}

void EXIT_MUTEX() {
  busy = false;
}
```

Note that both implementations do not require privileged machine instructions, thus such a mechanism could be implemented completely in user mode, for example as part of a library that supplies service functions for threads.

The construction using Test-and-Set and Lock in the preceding examples is called a *spin lock*. In such a spin lock, threads waiting to enter their critical section must "spin" in a loop until they are allowed to enter. A spin lock is a form of *busy waiting* which is often encountered in low-level synchronization. Busy waiting however is a very inefficient form of waiting since CPU cycles are used up without actually contributing to any form of computation. Imagine how many machine instructions a 3 GHz CPU spinning in a loop could have donated to some computation. So similar to interrupt masking, spin locks are only allowed if critical sections are relatively short. The advantage of spin locks over interrupt masking however is that they can be performed without switching to kernel mode.

spin lock

busy waiting

11.2.3 Monoprocessor vs. Multiprocessor Synchronization

To achieve mutual exclusion on a monoprocessor system, it is sufficient to turn interrupts off when entering and turning them on again when leaving the critical section. We will illustrate this strategy for performing mutual exclusion within Ulix in Section 11.6. As noted above, however, simply turning interrupts off is not sufficient on a multiprocessor system because disabling interrupts on one CPU does not prevent another CPU from accessing a shared data structure. In a multiprocessor system we additionally have to use spin locks. The strategy is as follows:

1. First we achieve *local mutual exclusion per CPU* by disabling interrupts.
2. Then we go into a spin lock to achieve *global mutual exclusion over all CPUs*.

Is achieving mutual exclusion at the lowest level in multiprocessor systems necessarily as complicated as this? Find out yourself by solving exercise 30.

It is generally advisable to avoid busy waiting whenever possible. At the lowest level of abstraction (e. g., synchronizing CPUs on the hardware level) busy waiting cannot be totally avoided. However, on higher levels of abstraction it generally can be avoided using more abstract synchronization primitives like semaphores.

11.2.4 Nested Critical Sections

In processor systems that support multiple *interrupt levels* turning off interrupts is not as easy as it may seem because interrupt handlers (and therefore critical sections) can be invoked in a nested fashion.

Assume for example, an interrupt handler at interrupt level 3 is executed on a CPU. During execution of that handler all interrupts at level 3 or lower are disabled, however, an interrupt at higher priority 5 may kick in and its handler be executed. It disables interrupts up to level 5. However, when this interrupt handler returns, it would be a bad idea to enable interrupts altogether. It rather must *restore the interrupt level* that was active *before* the interrupt occured.

A similar situation happens if (on purpose or by accident) one critical section is declared within another such as in the following code example:

[356] ⟨*example: nested critical sections* 356⟩≡

```
f() {
  // higher level critical section
  ENTER_MUTEX ();
  g();
  EXIT_MUTEX ();
}

g() {
  // lower level critical section
  ENTER_MUTEX ();
  // do something critical
  EXIT_MUTEX ();
}
```

If we use interrupt masking as synchronization primitive and these markers are nested, it must be assured that `EXIT_MUTEX` enables interrupts only when it is called in `f()` and not in `g()`. The general rule is: The `EXIT_MUTEX` must restore the interrupt level that was active before its corresponding `ENTER_MUTEX` was called. In systems such as Ulix that do not distinguish interrupt levels (i. e., interrupts are either on or off completely), inner critical sections are superseded by outer critical sections, i. e., the outermost critical section disables interrupts and finally enables them again. All inner critical sections do not change anything with the interrupt settings.

We now show how to realize such a "nestable" `ENTER_MUTEX` and `EXIT_MUTEX` in code. We assume a system (such as Ulix) that does not distinguish interrupt levels (i. e., interrupts are either on or off completely). In such environments storing the prior interrupt level burns down to storing a counter representing the nexting level. We use a global variable to store the current level of nesting.

⟨*global variables (unused)* 357a⟩≡ [357a]
```
int if_nested_level = 0;
```
Defines:
if_nested_level, used in chunk 357.

Whenever we enter and exit a critical section, we flag that code appropriately. Here is a first implementation that assumes that interrupts are on when calling the chunk for the first time. It disables the interrupts and increments the nesting level. Note that disabling interrupts using `cli` is idempotent. We only turn interrupts back on again if the nesting level has reached its initial value again.

⟨*nestable begin critical section (first version)* 357b⟩≡ [357b]
```
⟨disable interrupts 47a⟩        // invoke cli
if_nested_level++;
```
Uses if_nested_level 357a.

⟨*nestable end critical section (first version)* 357c⟩≡ [357c]
```
if_nested_level--;
if (if_nested_level == 0) {
  ⟨enable interrupts 47b⟩       // invoke sti
}
```
Uses if_nested_level 357a.

In case we do not know whether interrupts were on or off before we invoke the chunk for the first time, we can use the following (slightly better) code. It remembers the state of the first invocation of the chunk and sets the value that was active during that instance when returning for the last time.

⟨*nestable begin critical section* 357d⟩≡ [357d]
```
{  // create scope for scope-local variable eflags
  int eflags;
  asm volatile (
    "pushf                \n"     // push EFLAGS
    "cli                  \n"     // disable interrupts
    "movl   (%%esp), %0   \n"     // copy to eflags variable
    "addl   $4,      %%esp \n"    // restore stack pointer
  : "=r"(eflags) );
  if (if_nested_level == 0)
    if_state = (eflags >> 9) & 1;  // bit 9 of EFLAGS is IF
  if_nested_level++;
}
```
Uses if_nested_level 357a and if_state 383b.

⟨*nestable end critical section* 357e⟩≡ [357e]
```
if_nested_level--;
if (if_nested_level == 0 && if_state == 1) {
  ⟨enable interrupts 47b⟩
}
```
Uses if_nested_level 357a and if_state 383b.

In exercise 31 we discuss the case of nestable critical sections on multiprocessor systems.

11.3 Semaphores

As seen above in Section 11.2.2, the simplest way of implementing blocking is *busy waiting*. However, that is a rather inefficient way to implement blocking as it consumes CPU cycles that could have been used for threads that are ready to run. We can avoid it by offering the right synchronization abstractions within the operating system. The operations `ENTER_MUTEX` and `EXIT_MUTEX` can, for example, then directly influence the state of threads. This is depicted in Figure 11.4 where three threads are scheduled onto two CPUs. In the example, thread *A* enters its critical section while running on CPU 1 and thread *B* running on CPU 2 calls `ENTER_MUTEX`. Because *A* is already in its critical section, *B* must block. Instead of waiting actively in a loop, it could go to sleep (change its state to `blocked`) and allow a different ready-to-run thread *C* to run on CPU 2.

semaphore

The most popular abstraction for synchronization in operating systems is the *semaphore*. The name stems from a special type of signal used in railway systems. There, a critical section is a single track railway line. At any time, at most one train is allowed to run on such a line and so entering and exiting this part is governed by special signals. Note that designing proper semantics for such signals is not as easy at it seems because the signals at both ends must be synchronized. For example, it must be ensured that of two trains concurrently approaching the signals from opposite ends only one is allowed to pass. Also, leaving the critical section on one end must allow another follow-up train waiting at the opposite end to enter the track, too.

Inspired by real-world semaphores, Edsger W. Dijkstra introduced semaphores as a synchronization abstraction in his "THE" operating system in 1968 [Dij68]. The name "THE" stands for "Technische Hogeschool Eindhoven" (Eindhoven University of Technol-

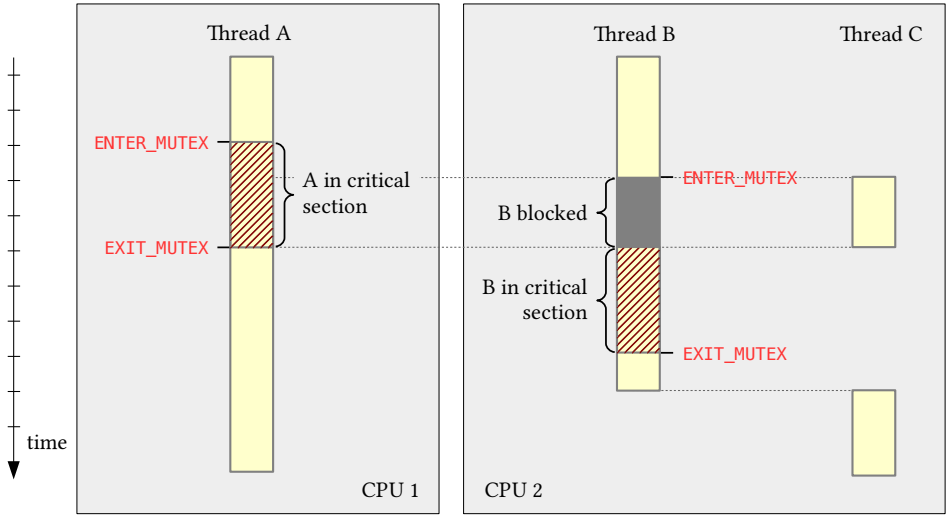


Figure 11.4: Avoiding busy waiting by running a different thread.

ogy) where Dijkstra was a professor at that time. Ever since, Dijkstra evolved into one of the most prominent and fascinating figures in computer science. Not only did he influence many of today's programming languages through his work on the language ALGOL (Algorithmic Language), but he also invented a lot of clever algorithms (like the famous shortest path algorithm for graphs).

11.3.1 Semantics of Semaphores

A semaphore is an operating system abstraction offering two primitive operations called P and V. The operation P (which can be read as "pass", originally: dutch *prolaag, probeer te verlagen*; try to reduce) [Wik, Dij] is invoked by a thread when it wishes to enter its critical section. Conversely, the operation V (which can be read as "leaVe", originally: dutch *verhogen*, increase) is invoked when a process leaves its critical section.

A semaphore guarantees *k-mutual exclusion*. The formal statement of this concept can be defined as follows.

k-mutual exclusion

Definition 2 (*k*-mutual exclusion) *If all threads properly encapsulate their critical section with P and V, then the semaphore guarantees that at most k threads are in their critical sections at the same time.*

The concept of k-mutual exclusion is a generalization of simple *mutual exclusion* for which $k = 1$. The actual value of k must be passed to the semaphore upon initialization.

mutual exclusion

It is possible to break down k-mutual exclusion to specific semantics of the individual semaphore operations P and V as follows:

Definition 3 (semantics of P and V) *Assume semaphore S is initialized with k. Then the operations P and V on S, written $P(S)$ and $V(S)$, have the following meaning:*

- *$P(S)$ blocks in case exactly k threads have passed $P(S)$ without passing $V(S)$.*
- *$V(S)$ deblocks a thread which is blocked at a $P(S)$ in case such a thread exists.*

For $k = 1$, the operations P and V therefore clearly resemble the semantics of a mechanism necessary to protect a single track railway line. More generally, they resemble the semantics for protection signals of a k-track railway line segment.

11.3.2 Single Mutual Exclusion

The notion of k-mutual exclusion for $k = 1$ is often simply called *mutual exclusion* or *mutex* for short. The name "mutex" is also used for the semaphore that protects a simple critical section where at most one thread is allowed to enter. Such a critical section can be implemented easily with semaphores as follows.

mutex

⟨*example: classical mutual exclusion with semaphores* 359⟩≡ [359]

```
Semaphore Mutex = 1; // initialization
// code of the thread
P (Mutex);              // enter critical section
```

```
// critical section
V (Mutex);              // leave critical section
// continued code of the thread
```

Incidentally, this is exactly what we would need if we want to implement `ENTER_MUTEX` and `EXIT_MUTEX` without busy waiting at the operating system level. The question of course is: Can we implement semaphores without busy waiting?

11.3.3 Initialization of Semaphores

kernel level semaphores

We now look at a simple implementation of semaphores at the operating system level. These semaphores are consequently called *kernel level semaphores*. Semaphores at that level basically encapsulate a counter and a list of threads. This list can be thought of as being at the same level as the ready queue in the dispatcher. In fact, it implements one type of blocked queue in the system (see Section 6.2.1.3).

We first declare the semaphore type, a structure consisting of a counter and a queue. All declarations and functions on this type of semaphore are prefixed with `kl_` to identify them as kernel level semaphores and clearly separate them from user level semaphores. We allow for additional "implementation" fields at the end of the semaphore structure which are not of interest for the general idea of semaphores.

[360a] ⟨*type definitions* 91⟩+≡ (44a) ◁325b 360b▷

```
typedef struct {
  int counter;
  blocked_queue bq;
  ⟨more kl_semaphore entries 363c⟩
} kl_semaphore;
```

Defines:
`kl_semaphore`, used in chunks 361–63 and 391a.
Uses `blocked_queue` 183a.

The structure $\texttt{kl\_semaphore}_{360a}$ is the internal representation of semaphores. In later code we will refer to semaphores by a unique identifier instead of a pointer to such a structure. Basically, this identifier will serve as a pointer into a global semaphore table implemented later.

[360b] ⟨*type definitions* 91⟩+≡ (44a) ◁360a 365a▷

```
typedef int kl_semaphore_id;
```

Defines:
`kl_semaphore_id`, used in chunks 361c, 362, 364, and 391a.

The function $\texttt{get\_new\_semaphore}_{364b}\texttt{(k)}$ returns the identifier of a new semaphore initialized with `k`. The return value -1 is used as an error code meaning that something went wrong during allocation. This usually means that the internal table is full, which is a bad sign.

We also provide a function to release a semaphore. Releasing it implies that all threads which may be blocked on that semaphore are deblocked.

Here are the prototypes:

⟨*function prototypes* 45a⟩+≡ (44a) ◁345b 361b▷ [361a]
```
kl_semaphore_id get_new_semaphore (int k);
void release_semaphore (kl_semaphore_id s);
```

11.3.4 Implementing P and V

The idea of the implementation of P and V is as follows. The counter of the semaphore represents the remaining "potential" of the semaphore, i. e., the number of threads which are still allowed to pass without being blocked. To actually block and deblock threads, we can simply use the operations provided by the kernel level dispatcher (see Section 6.2.1.1 for the general description and Section 6.2.2.3 for the implementation).

We will name the functions `wait_semaphore`$_{361c}$ (for P) and `signal_semaphore`$_{362}$ (for V):

⟨*function prototypes* 45a⟩+≡ (44a) ◁361a 365d▷ [361b]
```
void wait_semaphore (kl_semaphore_id sid);
void signal_semaphore (kl_semaphore_id sid);
```

The idea of the operation P is to check the remaining potential of the semaphore and block in case the potential is used up. To understand the condition under which a thread is blocked (`counter < 0`), remember that a semaphore initialized with 1 allows one thread to pass. Since the counter is decremented before the check, a condition `counter < 1` would not be correct.

Note that we mark the body of the implementations of P and V as critical sections. To see why, you should explore the case of interrupts happening after the counter manipulation and the test of the counter value (or solve exercise 32). Since semaphores are synchronization techniques on a higher level of abstraction than hardware mechanisms, it would also be counter-intuitive if we could implement them without referring to *any* lower-level synchronization primitive.

⟨*function implementations* 100b⟩+≡ (44a) ◁345c 362▷ [361c]
```
void wait_semaphore (kl_semaphore_id sid) {
  kl_semaphore sem = ⟨semaphore structure with identifier sid 363e⟩;
  ⟨begin critical section in kernel 380a⟩
  sem.counter--;
  if (sem.counter < 0) {
    block (&sem.bq, TSTATE_LOCKED);
    ⟨resign 221d⟩
  }
  ⟨end critical section in kernel 380b⟩
}
```
Defines:
`wait_semaphore`, used in chunks 361b and 391a.
Uses `kl_semaphore` 360a, `kl_semaphore_id` 360b, and `TSTATE_LOCKED` 180a.

The operation V is slightly simpler than P because there is no danger of a context switch. The function just checks whether a thread is still blocked on the semaphore queue and deblocks this thread if there is one.

[362] ⟨*function implementations* 100b⟩+≡ (44a) ◁361c 364b▷

```
void signal_semaphore (kl_semaphore_id sid) {
  kl_semaphore sem = ⟨semaphore structure with identifier sid 363e⟩;
  ⟨begin critical section in kernel 380a⟩
  sem.counter++;
  if (sem.counter < 1) {
    blocked_queue *bq = &(sem.bq);
    thread_id head = bq->next;
    if (head != 0) {
      deblock (head, bq);
    }
  }
  ⟨end critical section in kernel 380b⟩
}
```

Defines:
`signal_semaphore`, used in chunk 391a.
Uses `blocked_queue` 183a, `deblock` 186b, `kl_semaphore` 360a, `kl_semaphore_id` 360b, and `thread_id` 178a.

The above implementation of semaphores is probably the simplest one but still leaves some room for variation. For example, the implementation of the semaphore queue can be performed in different ways, e. g. as a simple FIFO queue, but priority queues can be used, too. The FIFO processing order is likely the one which is implicitly assumed most often when using semaphores, because it guarantees that the thread which has waited longest is deblocked first.

11.3.5 User-Level Semaphores

Until now, we have discussed the implementation of *kernel-level* semaphores, i. e., semaphores that have the power to block and deblock *kernel-level* threads. If you are implementing a user-mode thread library to realize user-level threads, you also may need a synchronization abstraction such as semaphores. You could use kernel-level semaphores for this purpose, but it is also possible to realize synchronization primitives that are tailored to handle *user-level* threads: *user-level semaphores*.

To recapitulate the difference between kernel-level threads and user-level threads, have a look again at Figure 7.3 on page 252. Kernel-level threads are *virtual processors* for user programs, and it is even possible to map a user program to multiple kernel-level threads (resulting in a *virtual multiprocessor*). But the power of such a virtual multiprocessor can only be unfolded if the user mode program supports a multiprogramming abstraction in user mode, such as a user-level thread library.

User-level semaphores are semaphores that block and deblock *user-level* threads. Their design and implementation is equivalent to those at kernel level, however, they are implemented in user mode, i. e., one step up the abstraction hierarchy. User-level semaphores have counters, blocked queues etc. in a similar way as kernel-level semaphores. However, these counters are simple integers in user space and the queues are the queues manipulated in user space by the thread library. To build user-level semaphores, you can therefore simply take the implementation of kernel-level semaphores and copy them into your user

program, at least in large parts. The only notable difference is the implementation of critical sections. This will be discussed later when we discuss the general notion of kernel synchronization.

11.3.6 Semaphores in ULix

The main effort to implement kernel level semaphores has already been done above. Here we fill in the final gaps.

Although semaphores (like threads and virtual address spaces) can be allocated and freed, we want to implement them without dynamic memory. Thus semaphores are held in a large semaphore table called `kl_semaphore_table`$_{363b}$. It is an array of `kl_semaphore`$_{360a}$ structures.

⟨*constants* 112a⟩+≡ (44a) ◁344a 365b▷ [363a]
```
#define MAX_SEMAPHORES 1024
```
Defines:
`MAX_SEMAPHORES`, used in chunks 363 and 364b.

⟨*global variables* 92b⟩+≡ (44a) ◁344b 364a▷ [363b]
```
kl_semaphore kl_semaphore_table[MAX_SEMAPHORES];
```
Defines:
`kl_semaphore_table`, used in chunks 363 and 364.
Uses `kl_semaphore` 360a and `MAX_SEMAPHORES` 363a.

There's a maximum number of semaphores that can be allocated in the kernel.

Since both used and unused semaphores are held in a table, we need additional information to distinguish both. So each semaphore has a counter and a queue, but it also has an additional field storing the semaphore state. The value `false` (0) means that the semaphore entry is free.

⟨*more* `kl_semaphore` *entries* 363c⟩≡ (360a) [363c]
```
boolean used;
```

Now it's also clear how we can initialize the fields.

⟨*initialize kernel global variables* 184d⟩+≡ (44b) ◁310g 516c▷ [363d]
```
for (int i = 0;  i < MAX_SEMAPHORES;  i++) {
  kl_semaphore_table[i].counter = 0;
  initialize_blocked_queue (&kl_semaphore_table[i].bq);
  kl_semaphore_table[i].used = false;
}
```
Uses `initialize_blocked_queue` 183c, `kl_semaphore_table` 363b, and `MAX_SEMAPHORES` 363a.

Since we didn't mention the semaphore table earlier, we need to fill in the mapping between the semaphore identifier `sid` and the semaphore structure in the table.

⟨*semaphore structure with identifier* `sid` 363e⟩≡ (361c 362 391a) [363e]
```
kl_semaphore_table[sid]
```
Uses `kl_semaphore_table` 363b.

Finally, we have to implement the two functions `get_new_semaphore`$_{364b}$ for acquiring and `release_semaphore`$_{364c}$ for releasing semaphores. Allocation is done in a round robin fashion (like in the FIFO allocation scheme for pages). We use a counter `next_kl_semaphore`$_{364a}$ to point to the next semaphore entry in the table which can be allocated.

[364a] ⟨*global variables* 92b⟩+≡ (44a) ◁363b 365c▷

```
kl_semaphore_id next_kl_semaphore = 0;
```

Defines:
`next_kl_semaphore`, used in chunk 364b.
Uses `kl_semaphore_id` 360b.

To allocate a new semaphore we check the next table entry and use it if it is free. While looking for a free entry in the table we use a check counter to catch the case where the semaphore table is full.

[364b] ⟨*function implementations* 100b⟩+≡ (44a) ◁362 364c▷

```
kl_semaphore_id get_new_semaphore (int k) {
  int check = MAX_SEMAPHORES;
  while (kl_semaphore_table[next_kl_semaphore].used == true) {
    next_kl_semaphore = (next_kl_semaphore + 1) % MAX_SEMAPHORES;
    check--;
    if (check ≤ 0)  return -1;
  }
  kl_semaphore_table[next_kl_semaphore].used    = true;
  kl_semaphore_table[next_kl_semaphore].counter = k;
  initialize_blocked_queue (&kl_semaphore_table[next_kl_semaphore].bq);
  return next_kl_semaphore;
}
```

Uses `initialize_blocked_queue` 183c, `kl_semaphore_id` 360b, `kl_semaphore_table` 363b, `MAX_SEMAPHORES` 363a, and `next_kl_semaphore` 364a.

Releasing a semaphore is a little tricky. Just resetting the state field in the semaphore table is not enough since threads may be blocked in the semaphore queue. These threads must be released to the ready queue.

[364c] ⟨*function implementations* 100b⟩+≡ (44a) ◁364b 366a▷

```
void release_semaphore (kl_semaphore_id s) {
  kl_semaphore_table[s].used = false;
  while (front_of_blocked_queue (kl_semaphore_table[s].bq) != 0) {
    thread_id t = front_of_blocked_queue (kl_semaphore_table[s].bq);
    remove_from_blocked_queue (t, &kl_semaphore_table[s].bq);
    add_to_ready_queue (t);
  }
}
```

Defines:
`release_semaphore`, used in chunk 361a.
Uses `add_to_ready_queue` 184b, `front_of_blocked_queue` 185b, `kl_semaphore_id` 360b, `kl_semaphore_table` 363b, `remove_from_blocked_queue` 186a, and `thread_id` 178a.

11.4 ULIX LOCKS

Locks could be treated as a special case of semaphores (which are initialized to 1), but we have decided to provide a separate implementation for kernel locks and a user mode interface.

For locks, we use the following type which resembles the definition of the semaphore type `kl_semaphore`$_{360a}$:

⟨*type definitions* 91⟩+≡ (44a) ◁360b 405a▷ [365a]

```
typedef struct {
  short int     l;            // the lock
  boolean       used;         // are we using this lock?
  blocked_queue bq;           // queue for this lock
  char          lockname[20]; // name
} lock_t;
typedef lock_t *lock;
```

Defines:
`lock`, used in chunks 306b, 308c, 310f, 366–69, 371a, 373c, 509a, 516b, 530a, 547b, and 552c.
`lock_t`, used in chunk 365c.
Uses `blocked_queue` 183a.

As we make a lot of use of locks in the kernel, we provide the `lockname` field so that we can generate more helpful debugging output.

We allow up to 1024 locks

⟨*constants* 112a⟩+≡ (44a) ◁363a 410a▷ [365b]

```
#define MAX_LOCKS 1024
```

Defines:
`MAX_LOCKS`, used in chunks 365c, 367b, and 606.

and reserve a table for them—as with the semaphores, we want to avoid dynamic allocation of memory in the kernel.

⟨*global variables* 92b⟩+≡ (44a) ◁364a 383b▷ [365c]

```
lock_t kernel_locks[MAX_LOCKS];
```

Defines:
`kernel_locks`, used in chunks 367b, 368, and 606.
Uses `lock_t` 365a and `MAX_LOCKS` 365b.

We reserve the first lock (`kernel_locks`$_{365c}$`[0]`) for locking the lock table itself.

11.4.1 Locking, Unlocking and Just Wishing

We provide three functions that can be used by user mode processes or threads (via corresponding library functions that we introduce in the next section):

⟨*function prototypes* 45a⟩+≡ (44a) ◁361b 367a▷ [365d]

```
void    mutex_lock     (lock lockvar);
boolean mutex_try_lock (lock lockvar);
void    mutex_unlock   (lock lockvar);
```

`mutex_lock`$_{366a}$ and `mutex_unlock`$_{366c}$ have the expected behavior: The first function tries to acquire the lock, and if that fails, it will block the active process and place it on the queue that belongs to this lock. The unlocking function returns the lock and wakes the first waiting process. In addition to these, `mutex_try_lock`$_{366b}$ does what its name implies: It *tries* to acquire the lock, but does not block if that goes wrong. It always returns, and its return value indicates whether the lock was acquired or not. If it failed, it returns `false`. So programs can use this to attempt to get a lock, but they have to check the return value and must not enter the critical section, if `false` was returned.

Regarding the placement of ⟨*begin critical section in kernel* 380a⟩ see exercises 34 and 35.

[366a] ⟨*function implementations* 100b⟩+≡ (44a) ◁364c 366b▷

```
void mutex_lock (lock lockvar) {
  if (current_task == 0) { return; }        // no process
  ⟨begin critical section in kernel 380a⟩
  while ( lockvar->l == 1 ) {
    block (&(lockvar->bq), TSTATE_LOCKED);  // put process to sleep
    ⟨resign 221d⟩
  }
  lockvar->l = 1;
  ⟨end critical section in kernel 380b⟩
}
```

Defines:
`mutex_lock`, used in chunks 308c, 310a, 367b, 368, 371a, 509d, 510b, 512b, 516d, 517c, 520c, 530, and 549d.
Uses `current_task` 192c, `lock` 365a, and `TSTATE_LOCKED` 180a.

[366b] ⟨*function implementations* 100b⟩+≡ (44a) ◁366a 366c▷

```
boolean mutex_try_lock (lock lockvar) {
  ⟨begin critical section in kernel 380a⟩
  int tmp = lockvar->l;  lockvar->l = 1;
  ⟨end critical section in kernel 380b⟩
  return (tmp == 0);
}
```

Defines:
`mutex_try_lock`, used in chunks 307, 308, and 371a.
Uses `lock` 365a.

For unlocking, we reset `lockvar->l` to `0` and then check whether we can wake up a waiting thread:

[366c] ⟨*function implementations* 100b⟩+≡ (44a) ◁366b 367b▷

```
void mutex_unlock (lock lockvar) {
  if (current_task == 0) { return; }    // no process
  if (lockvar->l == 0) {
    debug_printf ("NOTICE: unlocking unlocked LOCK: %s\n", lockvar->lockname);
  }
  ⟨begin critical section in kernel 380a⟩
  lockvar->l = 0;
  // wake a process
  blocked_queue *bq = &(lockvar->bq);
  thread_id head = bq->next;
```

```
    if (head != 0) {            // If one thread is waiting, deblock and resign
      deblock (head, bq);
    }
    ⟨end critical section in kernel 380b⟩
  }
```

Defines:
  mutex_unlock, used in chunks 307, 308, 311a, 365d, 367b, 368, 371a, 509d, 510b, 512b, 516d, 517c, 520c, 530, and 549e.
Uses blocked_queue 183a, current_task 192c, deblock 186b, debug_printf 601d, lock 365a, and thread_id 178a.

If you compare the $\texttt{mutex\_lock}_{366a}$ and $\texttt{mutex\_unlock}_{366c}$ functions with $\texttt{wait\_semaphore}_{361c}$ and $\texttt{signal\_semaphore}_{362}$ you will notice a big similarity. The main difference is that semaphores are more general, thus allowing k-mutual exclusion, whereas locks can only be used for single-mutual exclusion. However, in none of the ULIX code we came across a situation where a semaphore with $k > 1$ would have been useful. So now, if we want to achieve mutual exclusion between kernel level threads, we can simply acquire a kernel lock. This strategy is more elegant than using hardware mechanisms directly and also more efficient on multi-processor systems where we can avoid effort spent spinning in spin locks.

11.4.1.1 Lock Administration

Finally, we need to provide functions that allow to create a new lock and release it when it is no longer needed. They have these prototypes:

⟨*function prototypes* 45a⟩+≡ (44a) ◁365d 369b▷ [367a]

```
lock get_new_lock (char *name);
void release_lock (lock l);
```

The $\texttt{get\_new\_lock}_{367b}$ function has an argument via which we can give the lock a name. If you enable the kernel mode shell, you can type `locks` to see a list of all the locks, with their names and the threads on their blocked queues.

⟨*function implementations* 100b⟩+≡ (44a) ◁366c 368▷ [367b]

```
lock get_new_lock (char *name) {
  mutex_lock (kernel_locks); // lock the list of kernel locks, we use kernel_locks[0]
    for (int i = 1;  i < MAX_LOCKS;  i++) {
      if (!kernel_locks[i].used) {
        kernel_locks[i].used = true;
        initialize_blocked_queue (&kernel_locks[i].bq); // initialize blocked queue
        strncpy (kernel_locks[i].lockname, name, 20);
        mutex_unlock (kernel_locks);                    // unlock access to  list
        return &kernel_locks[i];
      }
    }
  mutex_unlock (kernel_locks);
  return NULL;
}
```

Defines:
get_new_lock, used in chunks 306c, 310g, 369c, 509b, 516c, 530b, and 552c.
Uses initialize_blocked_queue 183c, kernel_locks 365c, lock 365a, MAX_LOCKS 365b, mutex_lock 366a, mutex_unlock 366c, NULL 46a, and strncpy 594b.

For freeing a lock we set the used entry to false and unlock all threads on the blocked list (if there are any):

[368] ⟨*function implementations* 100b⟩+≡ (44a) ◁367b 369c▷

```
void release_lock (lock l) {
  mutex_lock (kernel_locks);  // lock the list of kernel locks
    l->used = false;
    blocked_queue *bq = &(l->bq);
    thread_id head = bq->next;
    while (head != 0) {
      thread_id next = thread_table[head].next;
      deblock (head, bq);
      head = next;
    }
  mutex_unlock (kernel_locks);
}
```

Defines:
release_lock, used in chunks 367a and 373c.
Uses blocked_queue 183a, deblock 186b, kernel_locks 365c, lock 365a, mutex_lock 366a, mutex_unlock 366c, thread_id 178a, and thread_table 176b.

11.5 Pthread Mutexes for Threads

In order to provide user space programs with mutexes, it is not necessary to interface the kernel—the code that we used for the implementation of kernel locks would also work in user mode since it requires no privileged instructions. However, we want to queue threads which try to acquire a mutex, and that is a task for the kernel. So instead of duplicating parts of the already existing locking code, we provide a user mode interface to the kernel functions $\texttt{get\_new\_lock}_{367b}$, $\texttt{mutex\_lock}_{366a}$, $\texttt{mutex\_unlock}_{366c}$ and $\texttt{release\_lock}_{368}$.

If you search for POSIX mutex functions on a Linux machine, you will find several functions, including the following ones:

```
pthread_mutex_init(3)    - create a mutex
pthread_mutex_lock(3)    - lock a mutex
pthread_mutex_trylock(3) - attempt to lock a mutex without blocking
pthread_mutex_unlock(3)  - unlock a mutex
pthread_mutex_destroy(3) - free resources allocated for a mutex
```

Our implementation only supports the essential features, so for example, you cannot use mutex attributes. We only include the corresponding argument in the function call so that the functions have a similar look and feel as the regular POSIX functions. You've already seen the same comment when you looked at the implementation of POSIX threads.

Also note that our kernel locks are valid globally and can be used across process borders. That means that in a ULIX program a process can create a mutex and then fork; afterwards both processes can be synchronized via the mutex. POSIX mutexes forbid this.

11.5.1 Creating a New Mutex

Before we start, we define two types that simplify attempts to port programs to ULIX:

⟨*public type definitions* 142a⟩+≡ (44a 48a) ◁254a 489b▷ [369a]
```
typedef int pthread_mutex_t;
typedef int pthread_mutexattr_t;
```
Defines:
pthread_mutex_t, used in chunks 369–73.
pthread_mutexattr_t, used in chunks 369c, 370d, and 373.

For mutex creation we implement the function

⟨*function prototypes* 45a⟩+≡ (44a) ◁367a 370f▷ [369b]
```
int u_pthread_mutex_init (pthread_mutex_t *restrict mutex,
         const pthread_mutexattr_t *restrict attr);
```

that reserves a fresh kernel lock via get_new_lock$_{367b}$ and returns the memory address of the lock data structure. This serves as a unique identifier for the lock when used in user space. (Since that address is in the kernel's private memory range, it will also be valid across process borders; see our earlier comment about cross-process use of mutexes.)

Since kernel locks have names, the function generates a name that consists of the string "lock, pid=" and the thread ID. Note that this is not unique if the same thread creates several mutexes.

⟨*function implementations* 100b⟩+≡ (44a) ◁368 371a▷ [369c]
```
int u_pthread_mutex_init (pthread_mutex_t *restrict mutex,
         const pthread_mutexattr_t *restrict attr) {
  char s[20];
  sprintf ((char*)s, "lock, pid=%d", thread_table[current_task].pid);
  lock tmp = get_new_lock (s);
  if (tmp != NULL) {
    *mutex = (pthread_mutex_t)tmp;
    return 0;        // success
  } else {
    thread_table[current_task].error = EAGAIN;
    return -1;       // error
  }
}
```
Defines:
u_pthread_mutex_init, used in chunks 369b and 370d.
Uses current_task 192c, EAGAIN 370a, get_new_lock 367b, lock 365a, NULL 46a, pthread_mutex_t 369a, pthread_mutexattr_t 369a, sprintf 601a, and thread_table 176b.

If `get_new_lock`$_{367b}$ was unsuccessful, we set the `error` field in the TCB to `EAGAIN`$_{370a}$ (it can be queried via the `errno`$_{207b}$ macro in the process).

[370a] ⟨*error constants* 370a⟩≡ (207a) 371b▷
```
#define EAGAIN 35
```
Defines:
`EAGAIN`, used in chunk 369c.

We add a system call:

[370b] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁332c 371c▷
```
#define __NR_pthread_mutex_init 517
```
Defines:
`__NR_pthread_mutex_init`, used in chunks 370e and 373e.

[370c] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁330b 371d▷
```
void syscall_pthread_mutex_init (context_t *r);
```

[370d] ⟨*syscall functions* 174b⟩+≡ (202b) ◁332d 372a▷
```
void syscall_pthread_mutex_init (context_t *r) {
  // ebx: mutex id
  // ecx: attributes, not implemented
  eax_return ( u_pthread_mutex_init ( (pthread_mutex_t*)r->ebx,
                                      (pthread_mutexattr_t*)r->ecx ) );
}
```
Defines:
`syscall_pthread_mutex_init`, used in chunk 370.
Uses `context_t` 142a, `eax_return` 174a, `pthread_mutex_t` 369a, `pthread_mutexattr_t` 369a, and `u_pthread_mutex_init` 369c.

and enter it in the syscall table:

[370e] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁333a 372b▷
```
install_syscall_handler (__NR_pthread_mutex_init, syscall_pthread_mutex_init);
```
Uses `__NR_pthread_mutex_init` 370b, `install_syscall_handler` 201b, and `syscall_pthread_mutex_init` 370d.

11.5.2 Locking and Unlocking a Mutex

We define the three locking and unlocking functions

[370f] ⟨*function prototypes* 45a⟩+≡ (44a) ◁369b 373b▷
```
int u_pthread_mutex_lock    (pthread_mutex_t *mutex);
int u_pthread_mutex_trylock (pthread_mutex_t *mutex);
int u_pthread_mutex_unlock  (pthread_mutex_t *mutex);
```

which "convert" POSIX-compliant mutexes into kernel mutexes and call the `mutex_lock`$_{366a}$, `mutex_try_lock`$_{366b}$ and `mutex_unlock`$_{366c}$ functions. If `u_pthread_mutex_trylock`$_{371a}$ cannot acquire the mutex via `mutex_try_lock`$_{366b}$, it will set the `error` field of the TCB to `EBUSY`$_{371b}$ (as expected by the POSIX standard) and return -1. The other two functions cannot fail, they simply return.

⟨*function implementations* 100b⟩+≡ (44a) ◁369c 373c▷ [371a]

```
int u_pthread_mutex_lock (pthread_mutex_t *mutex) {
  lock l = (lock)*mutex;
  mutex_lock (l);
  return 0;
}

int u_pthread_mutex_trylock (pthread_mutex_t *mutex) {
  lock l = (lock)*mutex;
  if (mutex_try_lock (l))
    return 0;    // success
  else {
    thread_table[current_task].error = EBUSY;
    return -1;   // error
  }
}

int u_pthread_mutex_unlock (pthread_mutex_t *mutex) {
  lock l = (lock)*mutex;
  mutex_unlock (l);
  return 0;
}
```

Defines:
u_pthread_mutex_lock, used in chunk 372a.
u_pthread_mutex_trylock, used in chunk 372a.
u_pthread_mutex_unlock, used in chunks 370f and 372a.
Uses current_task 192c, EBUSY 371b, lock 365a, mutex_lock 366a, mutex_try_lock 366b, mutex_unlock 366c, pthread_mutex_t 369a, and thread_table 176b.

⟨*error constants* 370a⟩+≡ (207a) ◁370a 561c▷ [371b]

```
#define EBUSY           16          // device / resource busy
```

Defines:
EBUSY, used in chunk 371a.
Uses busy.

Again, we add system calls for the new functions:

⟨*ulix system calls* 206e⟩+≡ (205a) ◁370b 372e▷ [371c]

```
#define __NR_pthread_mutex_lock    518
#define __NR_pthread_mutex_unlock  519
#define __NR_pthread_mutex_trylock 526
```

Defines:
__NR_pthread_mutex_lock, used in chunks 372b and 373e.
__NR_pthread_mutex_trylock, used in chunk 372b.
__NR_pthread_mutex_unlock, used in chunks 372b and 373e.

⟨*syscall prototypes* 173b⟩+≡ (202a) ◁370c 372c▷ [371d]

```
void syscall_pthread_mutex_lock    (context_t *r);
void syscall_pthread_mutex_trylock (context_t *r);
void syscall_pthread_mutex_unlock  (context_t *r);
```

They simply evaluate the mutex ID (found in the *EBX* register) and return the result by setting the *EAX* register:

[372a] ⟨*syscall functions* 174b⟩+≡ (202b) ◁370d 372d▷

```
void syscall_pthread_mutex_lock (context_t *r) {
  // ebx: mutex id
  eax_return ( u_pthread_mutex_lock ((pthread_mutex_t*)r->ebx) );
}

void syscall_pthread_mutex_trylock (context_t *r) {
  // ebx: mutex id
  eax_return ( u_pthread_mutex_trylock ((pthread_mutex_t*)r->ebx) );
}

void syscall_pthread_mutex_unlock (context_t *r) {
  // ebx: mutex id
  eax_return ( u_pthread_mutex_unlock ((pthread_mutex_t*)r->ebx) );
}
```

Defines:
`syscall_pthread_mutex_lock`, used in chunk 372b.
`syscall_pthread_mutex_unlock`, used in chunks 371d and 372b.
Uses `context_t` 142a, `eax_return` 174a, `pthread_mutex_t` 369a, `syscall_pthread_mutex_trylock`, `u_pthread_mutex_lock` 371a, `u_pthread_mutex_trylock` 371a, and `u_pthread_mutex_unlock` 371a.

[372b] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁370e 373a▷

```
install_syscall_handler (__NR_pthread_mutex_lock,    syscall_pthread_mutex_lock);
install_syscall_handler (__NR_pthread_mutex_trylock,syscall_pthread_mutex_trylock);
install_syscall_handler (__NR_pthread_mutex_unlock, syscall_pthread_mutex_unlock);
```

Uses `__NR_pthread_mutex_lock` 371c, `__NR_pthread_mutex_trylock` 371c, `__NR_pthread_mutex_unlock` 371c, `install_syscall_handler` 201b, `syscall_pthread_mutex_lock` 372a, `syscall_pthread_mutex_trylock`, and `syscall_pthread_mutex_unlock` 372a.

11.5.3 Destroying a Mutex

Finally, we need to be able to destroy a mutex when it is no longer needed.

[372c] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁371d 416a▷

```
void syscall_pthread_mutex_destroy (context_t *r);
```

[372d] ⟨*syscall functions* 174b⟩+≡ (202b) ◁372a 416b▷

```
void syscall_pthread_mutex_destroy (context_t *r) {
  // ebx: mutex id
  eax_return ( u_pthread_mutex_destroy ((pthread_mutex_t*)r->ebx) );
}
```

Defines:
`syscall_pthread_mutex_destroy`, used in chunks 372c and 373a.
Uses `context_t` 142a, `eax_return` 174a, `pthread_mutex_t` 369a, and `u_pthread_mutex_destroy` 373c.

[372e] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁371c 415d▷

```
#define __NR_pthread_mutex_destroy 520
```

Defines:
`__NR_pthread_mutex_destroy`, used in chunk 373.

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁372b 416c▷ [373a]
```
install_syscall_handler (__NR_pthread_mutex_destroy, syscall_pthread_mutex_destroy);
```
Uses __NR_pthread_mutex_destroy 372e, install_syscall_handler 201b, and syscall_pthread_mutex_destroy 372d.

Here is the implementation of the kernel function

⟨*function prototypes* 45a⟩+≡ (44a) ◁370f 405d▷ [373b]
```
int u_pthread_mutex_destroy (pthread_mutex_t *mutex);
```

⟨*function implementations* 100b⟩+≡ (44a) ◁371a 406▷ [373c]
```
int u_pthread_mutex_destroy (pthread_mutex_t *mutex) {
  lock l = (lock)*mutex;
  release_lock (l);
  return 0;
}
```
Defines:
u_pthread_mutex_destroy, used in chunks 372d and 373b.
Uses lock 365a, pthread_mutex_t 369a, and release_lock 368.

11.5.4 The Library Functions

As usual we provide a set of library functions that let user mode processes make these system calls:

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁333d 429a▷ [373d]
```
int pthread_mutex_init    (pthread_mutex_t *mutex,
                           const pthread_mutexattr_t *attr);
int pthread_mutex_lock    (pthread_mutex_t *mutex);
int pthread_mutex_unlock  (pthread_mutex_t *mutex);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁333e 429b▷ [373e]
```
int pthread_mutex_init    (pthread_mutex_t *mutex,
                           const pthread_mutexattr_t *attr) {
  return syscall3 (__NR_pthread_mutex_init, (unsigned int)mutex, (unsigned int)attr);
}

int pthread_mutex_lock    (pthread_mutex_t *mutex) {
  return syscall2 (__NR_pthread_mutex_lock, (int)mutex);
}

int pthread_mutex_unlock  (pthread_mutex_t *mutex) {
  return syscall2 (__NR_pthread_mutex_unlock, (int)mutex);
}

int pthread_mutex_destroy (pthread_mutex_t *mutex) {
  return syscall2 (__NR_pthread_mutex_destroy, (int)mutex);
}
```

Uses __NR_pthread_mutex_destroy 372e, __NR_pthread_mutex_init 370b, __NR_pthread_mutex_lock 371c, __NR_pthread_mutex_unlock 371c, pthread_mutex_t 369a, pthread_mutexattr_t 369a, syscall2 203c, and syscall3 203c.

11.5.5 Testing

In order to test the synchronization of threads, you can run the `/bin/thread` program. It starts three threads, two of which add to or subtract from a shared variable. After 250 additions and 250 subtractions, the variable should have the initial value. The program accepts a parameter: If you start it as `thread 0` it will work without synchronization (and return random results due to the two threads concurrently entering their critical sections). If, however, you start it as `thread 1` it will use a pthread_mutex_t$_{369a}$ to protect those sections and consistently return the correct result.

11.6 Kernel Synchronization

Until now, we have dealt with a couple of synchronization mechanisms that are suitable for different levels of abstraction:

1. Hardware-based mechanisms, such as interrupt masking and spin locks, that can establish mutual exclusion on a monoprocessor or multiprocessor system.
2. Kernel-level semaphores that can be used to block and deblock kernel level threads. Kernel-level locks are a special instance of such semaphores.
3. User-level semaphores that can be used to block and deblock user-level threads.

So far, our discussion of synchronization has focused on threads and with semaphores and locks we have provided nice abstractions to synchronize them. Whenever we need to prevent a thread from accessing some resource we simply block it; later when the resource becomes available again, we unblock such a thread so that it can continue execution. This works for both threads in user mode and in kernel mode.

However, there is other code in the kernel which is not executed on behalf of some particular thread: Interrupt handlers are activated whenever an interrupt occurs, and while such a handler function runs in the context of the thread which was active when the interrupt was signaled, it is not related to that thread in any manner. Getting this right is called *kernel synchronization*.

Kernel synchronization is a messy business because machine operations run in kernel mode and interrupts and context switches make code and execution sequences unintuitive. At its core, kernel synchronization deals with correctly implementing critical sections at the lowest level (the kernel). This section deals with kernel synchronization and discusses how this issue is solved in Ulix.

11.6.1 Overview

In general, synchronization may be needed on different levels of abstraction within an operating system. Figure 11.5 (a modified version of Figure 7.3 on page 252) shows those levels; the dotted black lines represent situations where shared data may be used, thus requiring synchronization. Differing from Figure 7.3, it also includes an interrupt handler which always runs in kernel mode.

We now discuss the different synchronization issues in context from top (user-level threads) to bottom (CPUs).

11.6.1.1 Synchronizing User-Level Threads

First we consider the following case: Two user-level threads which are mapped to one or multiple kernel-level threads may use the same data and need to perform mutual exclusion. In Figure 11.5, for example, Process 1 runs on one kernel-level thread (virtual monoprocessor) and Process 2 runs on two kernel-level threads (virtual multiprocessor). In both cases, the user-level thread library has to take care of synchronization since the kernel does not recognize the two threads as separate entities.

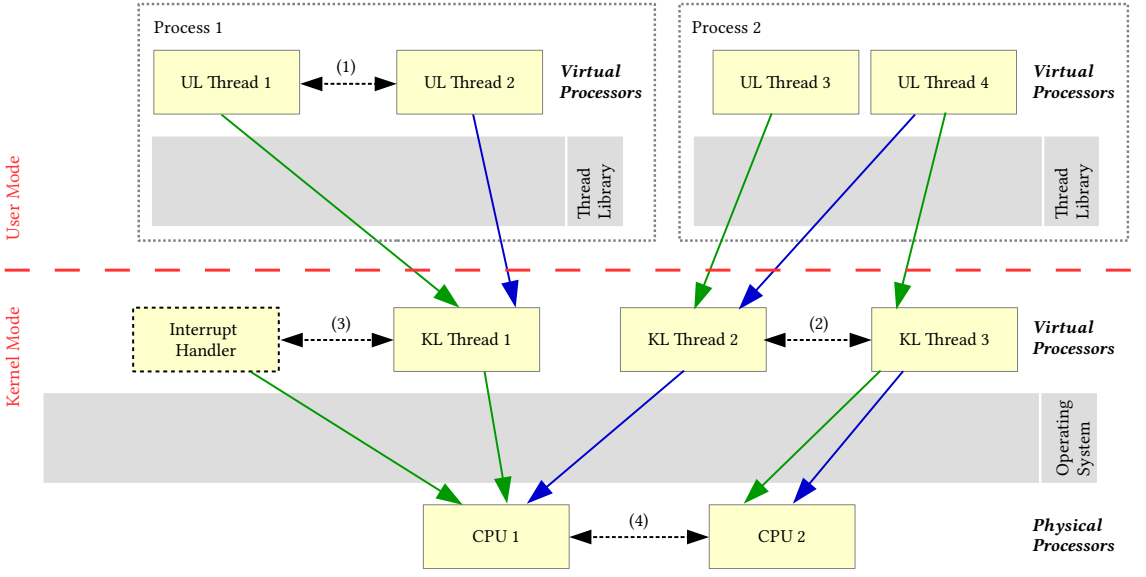


Figure 11.5: Synchronization in user and kernel mode. Arrow colors (on one level) express the order of allocation of a virtual (top) or physical (bottom) processor; black dotted arrows show where synchronization can be supported between user-level threads (1), between kernel-level threads (2), between kernel-level threads and interrupt handlers (3), and between two CPUs (4).

The mechanism of choice to synchronize user-level threads are user-level semaphores. They can be used to block and deblock user-level threads for synchronization. And if the thread library runs on a single kernel-level thread and the library does not support a signal mechanism (a "user-level interrupt" mechanism, see Chapter 14), only one concurrent activity will update the critical thread library data structures at any time. So while there might be critical sections in the thread (library) code, the entry and exit protocols may be empty.

However, if user-level threads can be interrupted (using signals for example) or if user-level threads run on a virtual multiprocessor (multiple kernel-level threads), we have to ensure mutual exclusion of concurrent activities in the threads library again. For the case of "interruptible" user-level threads (via signals), we need to enusure that any invocation of a signal handler does not violate mutual exclusion. A valid method would be to "mask" signals (i. e., ensure that during certain times user-defined signal handlers are not executed). This situation is analogous to interrupt-masking at lower levels.

For the case of a virtual multiprocessor (multiple kernel-level threads), signal masking is not enough since signals target kernel-level threads and so if one kernel-level thread masks signals another can still receive them and invoke a signal handler, thus violating mutual exclusion. So how could mutual exclusion be achieved here?

A naive approach would be to use a spin lock in addition to signal masking. Since the necessary machine instructions like Test-and-Set are not privileged, the user-level thread library can use them to achieve "global" mutual exclusion. However, spin locks imply busy waiting, so the question is: Can we do better?

Fortunately, the answer is yes: We can use a kernel-level semaphore, or more precisely a kernel lock as follows. During initialization of the thread library we allocate a kernel lock. Whenever a user-level thread wishes to run exclusively, if will lock the mutex and proceed. When it finishes its critical section, it releases the lock. To see why this works, consider again Figure 11.5 and look at user-level thread 3 and user-level thread 4 in the top right. Both threads might be running on different virtual processors (kernel-level thread 2 and kernel-level thread 3, the green assignment). Now assume that user-level thread 3 enters a critical section and acquires the lock. If user-level thread 4 attempts to enter its own critical section, it will try to acquire the same lock, but because it is taken, then *kernel-level* thread 3 is blocked. It is automatically deblocked when user-level thread 3 releases the lock. Busy waiting is completely avoided!

Remember however, that the above solution still needs to take care of user-level "interrupts" (signals). If signal handlers can be run at any time, also note that kernel locks are not useful for mutual exclusion if the thread library is running on a virtual monoprocessor. If one of the user-level threads acquires the lock, is interrupted by a signal handler and switches to another thread that also tries to acquire the lock, the kernel-level thread (i. e., the entire user-level thread library) would block and the situation could never be resolved.

Ulix (like most other systems) does not implement the mapping of several user-level threads to one kernel-level thread, thus we need not consider this case—the user-mode functions `pthread_mutex_lock`$_{373e}$ and `pthread_mutex_unlock`$_{373e}$ use system calls to call the kernel functions `u_pthread_mutex_lock`$_{371a}$ and `u_pthread_mutex_unlock`$_{371a}$.

11.6.1.2 Synchronizing Kernel-Level Threads

Now we consider the case where two kernel-level threads wish to synchronize because they use the same data. This is case (2) in Figure 11.5. The method of choice here is obviously to use kernel-level semaphores, or more specifically (for mutual exclusion) kernel locks. If such locks can be acquired and released atomically, kernel locks can be used to achieve mutual exclusion between kernel-level threads, just as discussed previously for the virtual multiprocessor that runs a user-level thread library.

The property we need to ensure, however, is in fact the atomicity of lock acquisition. Remember that kernel-level threads may be interrupted at any time and a context switch might schedule another kernel-level thread. So the problem of achieving mutual exclusion between kernel-level threads can be reduced to the problem of ensuring the atomicity of the locking procedure. Unfortunately, we cannot use kernel mutexes for this since this is the mechanism we are trying to implement.

We have handled exactly this case in the previous sections where we have discussed the implementation of `u_pthread_mutex_lock`$_{371a}$ and `u_pthread_mutex_unlock`$_{371a}$. The trick to solve this problem was to declare the main parts of `mutex_lock`$_{366a}$, `mutex_try_lock`$_{366b}$ and `mutex_unlock`$_{366c}$ as critical sections in the kernel, but we have not yet discussed how to actually implement this. However, we have all necessary ingredients ready to deal with it now. This is what kernel synchronization is all about.

Many operating systems also support synchronization across processes which is also possible in Ulix if two processes share a `pthread_mutex_t`$_{369a}$ variable; note however that the POSIX standard forbids this (see Section 11.5). Instead Unix systems use *named semaphores* for cross-process synchronization which are not implemented in Ulix.

named semaphore

To summarize synchronization issues up to now—and as partially shown in Figure 11.6,

- in user mode, threads can call the user-mode library functions `pthread_mutex_lock`$_{373e}$ and `pthread_mutex_unlock`$_{373e}$ to acquire kernel level mutexes that can be used for inter-thread synchronization.
- in kernel mode, threads can also use the kernel functions `u_pthread_mutex_lock`$_{371a}$ and `u_pthread_mutex_unlock`$_{371a}$ for the same purposes,
- and these are (again) wrappers for the general kernel-internal synchronization functions `mutex_lock`$_{366a}$ and `mutex_unlock`$_{366c}$.
- Besides `mutex_lock`$_{366a}$ and `mutex_unlock`$_{366c}$ the Ulix kernel also supports synchronization with semaphores via the `wait_semaphore`$_{361c}$ and `signal_semaphore`$_{362}$ functions which are not accessible from user mode applications.

11.6.1.3 Sychronizing the Kernel

Kernel synchronization deals with cases (3) and (4) in Figure 11.5. The main task of kernel synchronization is to achieve mutual exclusion of concurrent activities in the kernel. This is a challenge because concurrent activities are natural in operating systems, let they be introduced through interrupts or through multiple CPUs.

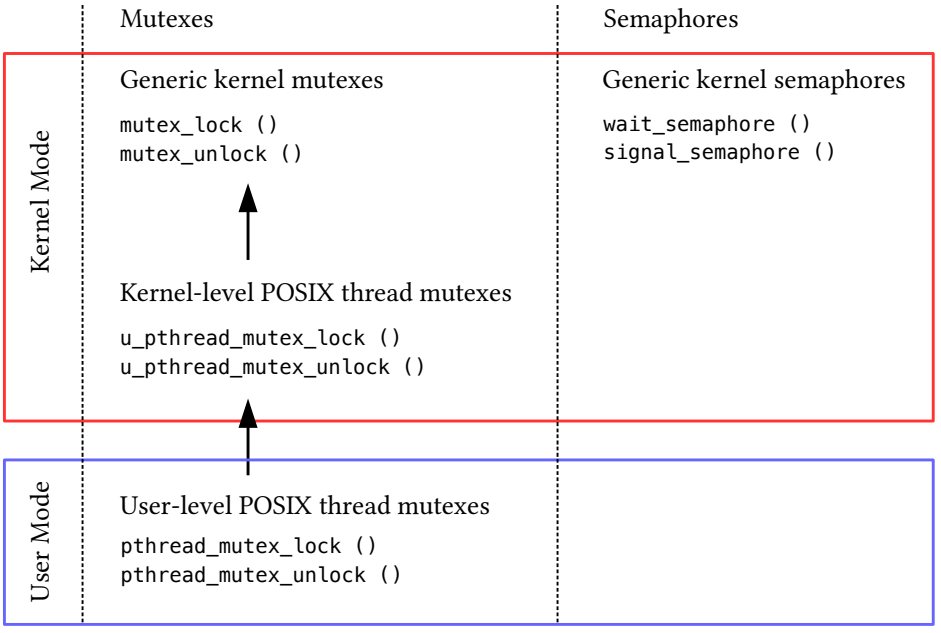


Figure 11.6: Mutex functionality is available in user mode and kernel mode; the kernel can also use semaphores.

The case of interrupts, case (3) in Figure 11.5, is probably the toughest issue to solve because it happens in all operating systems:

An interrupt handler may share data with a process which has made a system call. As we will show on the following pages, we cannot use the same mutex/locking approach as for inter-thread synchronization, instead we will need to revert to hardware-based mechanisms discussed in Section 11.2. In fact, Ulix uses the simplest approach to synchronize the kernel: It realizes a non-interruptible kernel. What this means will be discussed shortly.

The case of multiprocessing, case (4) in Figure 11.5, is equally tough. When more than one CPU (or CPU core) is used, the situation becomes more complicated because code will be executed truly simultaneously on those CPUs or cores. But as we will see, if case (3) has been solved conceptually, it is not too hard to extend this solution to case (4). Furthermore, since Ulix does not support more than one CPU or core, we do not need to work out this case in code anyway.

11.6.2 Minimizing the Size of Critical Sections: Interruptible Kernels

We now turn to the central problem of implementing critical sections in the kernel, i. e., achieving mutual exclusion at the lowest layer. Until now, we declared critical sections

using the markers ⟨*begin critical section in kernel* 380a⟩ and ⟨*end critical section in kernel* 380b⟩. A kernel built in this way is called an *interruptible kernel*. Such kernels allow concurrent activities within the kernel, but only in *some* parts. For a monoprocessor system this means that interrupts may be enabled during the execution of *some* kernel code. (It does not mean that interrupts are always on.) For multiprocessor systems, spin locks are needed in addition to disabling the interrupts (see Section 11.2).

interruptible kernel

Building correct interruptible kernels boils down to finding all critical sections and ensuring that the entry and exit protocols to these sections are implemented correctly. Both problems are non-trivial and have caused much misery in the history of operating systems:

- It is easy to spot some critical sections, but it is hard to overlook no critical section. Overlooking a critcal section (i. e., failing to mark it correctly) usually causes hard to diagnose system faults because bugs are usually the result of non-reproducible race conditions (so called *Heisenbugs*).
- It is easy to "play safe" and mark all possible candidates for critical sections as such, following the strategy: if in doubt, then it's a critcal section. But this results in rather large critical sections, and large critical sections cause their own problems (inefficiency being one). So the challenge is to declare "minimal" critical sections.

A typical approach is to let system call handlers be interruptible while disabling (all or some) interrupts during the execution of interrupt handlers.

These challenges have been debated by operating system designers for many decades. They effectively ask the question of the size of critical sections. The more code allows interrupts, the better the performance and responsiveness of the system can become, but at the same time complexity of synchronization increases.

Interestingly, there is a very simple synchronization strategy that works in most cases and allows you to start off with a system which is inefficient yet correctly synchronized: The most simple approach is to generally forbid interrupts while executing kernel code, thereby "maximizing" the size of critical sections in the kernel. We call such an implementation a *non-interruptible kernel*. For a single-CPU, single-core machine this means that all kernel code can expect to remain in control and keep the CPU until it either finishes or blocks (in a system call handler).

non-interruptible kernel

Only when the system runs in user mode (i. e., executes the application code of some thread), interrupts can occur. Note that this means that whenever a thread makes a system call (and thus transitions to kernel mode via the system call interface), interrupts will be disabled until the system call function completes its work and returns to user mode. In such systems, there will be no need to synchronize interrupt handlers and system call handlers.

Note however that system call handlers still have to consider synchronization because a thread executing such a system call handler may decide to block (for example, in order to wait for a disk operation to complete). In that case control will transfer to a different thread which might call some other or the same system call handler, thus possibly accessing the same kernel data structures. Therefore, the begin of a critical section might happen in thread A while the end of a critical section might happen in thread B (after a context switch). This needs to be considered.

Fortunately, the Intel CPU lets developers decide whether interrupts are automatically disabled upon entering some handler by providing both *interrupt gates* (which auto-disable interrupts) and *trap gates* (which do not), see also Section 6.4.1.

11.6.3 ULIX as a Non-Interruptible Kernel

We have decided to make the ULIX kernel non-interruptible which greatly reduces the demand for synchronization. For completeness, Section 11.6.4 will describe the issues arising with interruptible kernels and illustrate them with examples from the ULIX code.

As mentioned above, the Intel CPU lets developers decide whether interrupts are automatically disabled upon entering some handler by providing both *interrupt gates* and *trap gates*. We have used an interrupt gate in our system call implementation in Section 6.4.1. So we can safely assume that interrupts are off whenever we execute code in kernel mode. Since ULIX does not support multiple CPUs, there is nothing more to do.

Now, finally, we can show the entry and exit protocols for the critical sections in the kernel. Recall that we marked them using the code chunks ⟨*begin critical section in kernel* 380a⟩ and ⟨*end critical section in kernel* 380b⟩. Since we have a non-interruptible kernel and switch off interrupts using an interrupt gate, there is no further necessity for additional synchronization. Therefore, their implementations are empty.

[380a] ⟨*begin critical section in kernel* 380a⟩≡

(168d 169a 184–87 209c 216b 219c 221a 255a 260a 276d 361c 362 366 391 392 416b 521a 530 539c 540c 545b 548b 551 580c 581)

```
// do nothing
```

[380b] ⟨*end critical section in kernel* 380b⟩≡

(168d 169a 184–87 212 216b 219c 221a 255a 260a 276d 277a 280b 361c 362 366 391 392 416b 521b 531a 545b 580c 581)

```
// do nothing
```

The interested reader might ask: Why didn't we omit these markers from the beginning if they are empty anyway? There are two answers to this question:

1. Omitting these markers would have avoided the discussion (and identification) of critical sections, which would have avoided some nice intellectual challenges.
2. Keeping these markers allows for a future evolution of ULIX into an interruptible kernel.

In such a future ULIX version with an interruptible kernel these chunks would be implemented using ⟨*disable interrupts* 47a⟩ and ⟨*enable interrupts* 47b⟩, or with the nestable versions.

11.6.4 Illustrating Problems of Interruptible Kernels

In this section we describe some relevant problems that arise and discuss the additional care that needs to be taken when the kernel is interruptible. Note that the situation would become even more complex if several CPUs (or cores) were supported. Wherever possible, we use code of ULIX to illustrate the problems.

11.6.4.1 Example: Accessing a Keyboard Buffer

We begin with an introductory example for the synchronization problems that arise from using an interruptible kernel. Consider the way that reading from the keyboard works for a thread:

- A thread calls the `read`$_{429b}$`()` function (using file descriptor `STDIN_FILENO`$_{415b}$ = 0) which in turn makes a system call. This enters kernel mode and leads to the execution of `syscall_read`$_{426b}$`()` which in turn will make an other system call that activates the system call handler `syscall_readchar`$_{416b}$`()`.
- `syscall_readchar`$_{416b}$`()` determines the current terminal `term`, reads a character from its associated buffer `term->kbd[]` and updates the current position and number of characters in the buffer (`term->kbd_lastread` and `term->kbd_count`). Note that those two changes modify a global data structure.
- On the other hand, the keyboard interrupt handler, `keyboard_handler`$_{319d}$`()`, updates the same data structure: it determines the active terminal (the one currently displayed on the screen), enters the character's ASCII code in the corresponding buffer and also modifies the buffer's current position and number of contained characters.

If both accesses originated from threads, we could simply use a list of mutexes (one for each terminal) to lock these structures, but we cannot use the locking mechanism in the interrupt handler since that might put it to sleep if a thread had just acquired the lock right before the keyboard interrupt occurred. Thus, the obvious attempt of using a lock list `keyboard_buffer_lock[]` and implementing mutual exclusion via

⟨*first attempt for locking the keyboard buffer (in* `syscall_readchar`*)* 381a⟩≡ [381a]

```
void syscall_readchar (context_t *r) {
  char c;
  int t = thread_table[current_task].terminal;
  terminal_t *term = &terminals[t];

  // get character, return 0 if there is no new character in the buffer
  mutex_lock (keyboard_buffer_lock[t]);
    if (term->kbd_count > 0) {
      term->kbd_count--;
      term->kbd_lastread = (term->kbd_lastread+1) % SYSTEM_KBD_BUFLEN;
      c = term->kbd[term->kbd_lastread];
    }
  mutex_unlock (keyboard_buffer_lock[t]);
  // ...
```

and

⟨*first attempt for locking the keyboard buffer (in keyboard handler implementation)* 381b⟩≡ [381b]

```
terminal_t *term = &terminals[cur_vt];
mutex_lock (keyboard_buffer_lock[cur_vt]);
  if (term->kbd_count < SYSTEM_KBD_BUFLEN) {
    if (ctrl_pressed && c ≥ 'a' && c ≤ 'z') c -= 96;  // Ctrl
    term->kbd[term->kbd_pos] = c;
```

```
      term->kbd_pos = (term->kbd_pos + 1) % SYSTEM_KBD_BUFLEN;
      term->kbd_count++;
      if (scheduler_is_active) { ⟨keyboard handler: wake sleeping process 322a⟩ }
    }
  mutex_unlock (keyboard_buffer_lock[cur_vt]);
```

cannot work. However, note that the situation is not symmetrical: Whereas two threads will run alternately, this is not true for an interrupt handler which will interrupt the flow of control in a thread, but not vice versa. Thus it will be sufficient to prevent the interrupt handler from running at all while we access the data structures in a system call handler—we can simply disable interrupts before the critical section and re-enable them afterwards.

one-sided synchronization

This implements a synchronization model that is called *one-sided synchronization*. Since disabling interrupts disturbs the overall functioning of the operating system (e. g., by delaying the next scheduling decision) we should limit this to very short time spans. Figure 11.7 shows the difference between *two-sided* inter-thread synchronization and the synchronization of system call and interrupt handlers.

two-sided synchronization

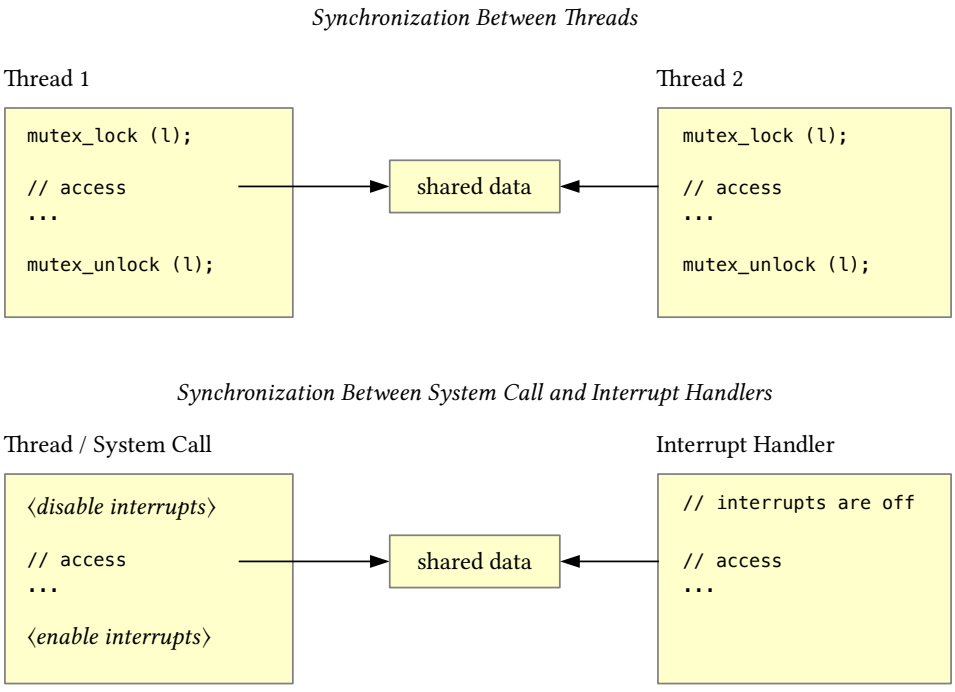


Figure 11.7: Mutex-based synchronization of threads (top) is an example of *two-sided synchronization*; data which is shared between system call and interrupt handlers is handled via a *one-sided synchronization* method (bottom; i. e., disabling interrupts in the system call handler).

A working implementation of `syscall_readchar`$_{416b}$`()` thus looks as follows:

⟨*second attempt for locking the keyboard buffer (in* `syscall_readchar`*)* 383a⟩≡ [383a]
```
void syscall_readchar (context_t *r) {
  char c;
  int t = thread_table[current_task].terminal;
  terminal_t *term = &terminals[t];

  // get character, return 0 if there is no new character in the buffer
  ⟨disable interrupts 47a⟩    // critical section starts here
    if (term->kbd_count > 0) {
      term->kbd_count--;
      term->kbd_lastread = (term->kbd_lastread+1) % SYSTEM_KBD_BUFLEN;
      c = term->kbd[term->kbd_lastread];
    }
  ⟨enable interrupts 47b⟩    // critical section ends here
  // ...
```

and that is also the way it could be implemented in Ulix (see page 416). The corresponding critical section in the interrupt handler would need no protection because there it would be impossible for the system call handler's code to interfere with the running interrupt handler.

11.6.4.2 Restoring Old Interrupt States

As discussed in Section 11.2.4, marking critical sections needs to be done with care since nested critical sections can cause premature enabling of interrupts. This can be avoided by using "nestable" entry and exit protocols for critical sections, as discussed. However, sometimes interrupts might have been turned off by other means (e. g., the interrupt gate) and so we might not even know whether interrupts were enabled before we wish to disable them. For example, $\texttt{deblock}_{186b}$ calls $\texttt{remove\_from\_blocked\_queue}_{186a}$ and $\texttt{add\_to\_ready\_queue}_{184b}$, and $\texttt{deblock}_{186b}$ itself is both called from the interrupt handler $\texttt{keyboard\_handler}_{319d}$ and from the syscall handler $\texttt{syscall\_kill}_{565c}$ via $\texttt{u\_kill}_{562b}$ and a further helper function (see Figure 11.8).

Thus, for a Ulix version with an interruptible kernel, we might enter $\texttt{deblock}_{186b}$ with interrupts on or off, depending on which function calls it.

In those situations we want to restore the original state afterwards. Thus, we need to save the current state of the interrupt flag (*IF*) which is bit 9 of the *EFLAGS* register before we disable interrupts. We can use a global variable

⟨*global variables* 92b⟩+≡ (44a) ◁365c 405b▷ [383b]
```
boolean if_state;   // state of the interrupt flag (IF)
```
Defines:
`if_state`, used in chunks 357 and 384.

for storing the state because interrupts will always be off after reading the state and until it is restored.

The following two new code chunks save and disable interrupts and restore them, respectively:

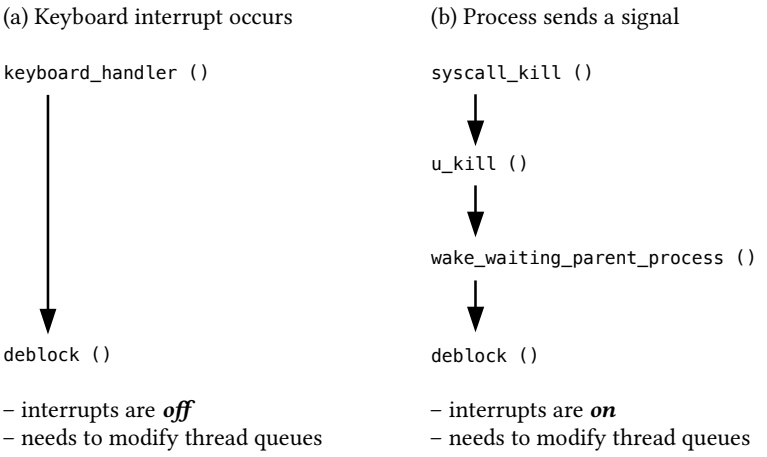


Figure 11.8: Two ways to enter the `deblock` functions with interrupts off or on.

[384a] ⟨*save and disable interrupts* 384a⟩≡

```
{  // create scope for scope-local variable eflags
  int eflags;
  asm volatile (
    "pushf                 \n"     // push EFLAGS
    "cli                   \n"     // disable interrupts
    "movl   (%%esp), %0    \n"     // copy to eflags variable
    "addl   $4,      %%esp \n"     // restore stack pointer
  : "=r"(eflags) );
  if_state = (eflags >> 9) & 1;    // bit 9 of EFLAGS is IF
}
```

Uses `if_state` 383b.

[384b] ⟨*restore interrupts* 384b⟩≡

```
if (if_state == 1) {
  ⟨enable interrupts 47b⟩
}
```

Uses `if_state` 383b.

The functions `add_to_blocked_queue`$_{185c}$ and `remove_from_blocked_queue`$_{186a}$ as well as `add_to_ready_queue`$_{184b}$ and `remove_from_ready_queue`$_{184c}$ which handle the blocked queues and the ready queue could use this feature.

11.6.4.3 Finding Critical Sections

As mentioned above, identifying critical sections is one of the major problems in interruptible kernels. It needs much experience to do this correctly, and doing it minimally is more an art than a science. A best-practice approach is to invest much discipline during development and for every new data structure investigate thoroughly whether this data

structure is critical and, if yes, who accesses it. These accesses must be declared as critical sections.

We will now discuss these points using Ulix code. For example, look at all the interrupt handlers of Ulix and search for data structures which are accessed elsewhere. For an interruptible kernel we would have to make sure that interrupts are disabled whenever such an access (outside interrupt handlers) occurs. Luckily, the number of handlers is small—there are only five interrupt handlers:

- `keyboard_handler`$_{319d}$ served as our initial example and has already been dealt with.
- `timer_handler`$_{342b}$ periodically activates the scheduler—we will discuss it below.

The next three handlers deal with filesystem activity; you will only fully understand the following discussion after reading Chapters 12 (Filesystems) and 13 (Disk I/O).

- `ide_handler`$_{532d}$ modifies the global `hd_buf`$_{530a}$ buffer and the `hd_direction`$_{530a}$ variable and deblocks a thread that has been waiting for the completion of a hard disk action. Other than from the IDE handler, any access to these data will originate from a system call handler initiated by a thread trying to read from or write to disk.

  Since the implementation of the filesystem code does not allow multiple parallel accesses to the disk (once a transfer has been initiated, all other threads block on a mutex `hd_lock`$_{530a}$), no thread will access the data before a current transaction has completed. Similarly, the IDE controller will only generate an interrupt after a (single) thread has made a system call that accesses the disk.
- `floppy_handler`$_{546b}$ calls `fdc_wakeup`$_{546a}$ which manipulates an entry in the thread list (like `ide_handler`$_{532d}$, it deblocks a thread that has been waiting for the floppy drive's action to complete).

  Floppy access is also serialized for threads using an `fdc_lock`$_{547b}$ variable (similar to `hd_lock`$_{530a}$). `fdc_timer`$_{547a}$ (which is called by the timer handler) checks the lock state of `fdc_lock`$_{547b}$ but does not change it.
- `serial_hard_disk_handler`$_{519d}$ also deblocks a thread which caused the recently finished action of the serial hard disk. Again, there is a `serial_disk_lock`$_{516b}$ variable which is used to serialize parallel accesses to the serial disk device.

For all hard disk, floppy disk and serial disk accesses the following order of execution is enforced:

1. A thread initiates disk access which leads to acquisition of one of the locks `hd_lock`$_{530a}$, `fdc_lock`$_{547b}$ or `serial_disk_lock`$_{516b}$. Eventually the thread will block (and wait for the disk operation to complete).
2. One or more interrupts are generated by the controller, thus the corresponding interrupt handler will be executed. When Ulix detects that the requested operation has completed, it will wake up the thread.
3. When the scheduler selects the thread, it will unlock the lock. Only then can the system start the next disk access (if other threads are waiting).

Thus, we do not need one-sided synchronization in the disk I/O code. Access to different devices (e. g., a hard disk and a floppy disk) can happen in parallel.

Note that the ready and blocked queues are manipulated by these interrupt handlers. These are critical regions because those queues are also modified via system calls (when a new process or thread is created or exits or when a process asks to wait for termination of a child process). Thus the functions

- `u_fork`$_{209c}$ (forks a process; called by `syscall_fork`$_{213d}$)
- `syscall_exit`$_{216b}$ (exits a process)
- `u_pthread_create`$_{255a}$ (creates a new thread in the current process)
- `syscall_waitpid`$_{219c}$ (makes a process wait for termination of a child)

(in an interruptible kernel) would have to disable interrupts while accessing the thread table in order to ensure that they cannot be interrupted by one of the interrupt handlers. (`syscall_pthread_exit`$_{260a}$ uses `syscall_exit`$_{216b}$ to make the thread terminate.)

The timer handler calls the scheduler which also modifies the thread list. Since it runs with deactivated interrupts it cannot conflict with the other functions that modify this list.

An earlier version of the Ulix code (up to release 0.12) used a `thread_list_lock` mutex for controlling access to the thread list (e. g., when creating a new thread or removing one from the list). However, the scheduler (which is sometimes started from a thread when it exits or yields, but typically runs on behalf of the timer interrupt handler) must not block and thus cannot use the mutex.

11.6.4.4 Dealing With Complex Handlers

In more complex operating systems handling an interrupt can become time-intensive. Since disabling interrupts should be limited to short time spans (see above), in those cases the approach of splitting the handler code into a "first-level interrupt handler" and a "second-level interrupt handler" can help.

1st-level handler (top half)
- The first-level interrupt handler (sometimes called top half) is registered as the regular handler and runs with other interrupts disabled. It performs only the most important tasks (e. g., it acknowledges the interrupt and saves volatile data from a device's internal buffer). As a last step it creates the lower half of the handler, re-enables interrupts and terminates.

2nd-level handler (bottom half)
- The second-level interrupt handler (sometimes: bottom half) is not part of the interrupt handler and runs while interrupts are enabled. It is somewhat similar to a kernel thread but without an address space of its own (it only uses the kernel's memory). The bottom half could be activated by the scheduler (in that case bottom halves would need to have a higher priority than regular threads) or some other mechanism could be used for making sure that the bottom halves run as early as possible—as long as no other top half has to be executed because new interrupts occurred.

tasklet

The Linux kernel uses this concept and uses the terms *top half* (for the first-level handler) and *bottom half* or *tasklet* (for the second-level handler) [Lov03, pp. 81–106]. Tasklets

in the Linux 2.6 kernel can have one of two priorities where the higher-priority tasklets are always executed before the lower-priority ones and both run before the next thread is scheduled. The top half of the interrupt handler registers a *tasklet handler*. Those tasklet handlers must not block (just like the top halves, they cannot be scheduled, so it is not possible for them to use locks or semaphores), and they must be able to cope with being interrupted.

11.6.4.5 Spurious Interrupts

A *spurious interrupt* is an interrupt whose occurrence is faulty and unexpected. Yet, on physical hardware it is a problem that needs to be dealt with. For example, the Intel 8259 PIC which Ulix uses allows the discovery of a spurious interrupt:

> "In both the edge and level triggered modes the IR inputs must remain high until after the falling edge of the first $\overline{\text{INTA}}$. If the IR input goes low before this time a DEFAULT IR7 will occur when the CPU acknowledges the interrupt. This can be a useful safeguard for detecting interrupts caused by spurious noise glitches on the IR inputs. To implement this feature the IR7 routine is used for 'clean up' simply executing a return instruction, thus ignoring the interrupt. If IR7 is needed for other purposes a default IR7 can still be detected by reading the ISR. A normal IR7 interrupt will set the corresponding ISR bit, a default IR7 won't." [Int88, p. 18]

(*ISR* is the In-Service Register.) Thus it is sufficient to modify handlers for interrupt numbers 7 (coming from the first PIC) and 15 (from the second PIC). In case of an interrupt from the first PIC, no action is required; especially it is not necessary to acknowledge the interrupt. However, if the interrupt comes from the second PIC, the first PIC (and only the first PIC) needs to be sent the acknowledgement.

Certain spurious interrupts can be detected by the system because interrupts usually provide services with prior demands. So if there is an interrupt which has no obvious demand, then it probably is spurious. Other types of spurious interrupts are harder to detect. Since Ulix uses neither interrupt 7 (first parallel port) nor 15 (secondary IDE controller), we fortunately need not deal with this situation.

11.6.4.6 Lost Wakeup

Blocking processes until an interrupt occurs introduces a problem that is called *lost wakeup*. Lost wakeups can be caused by bad programming or unfortunate circumstances.

For example, if the interrupt that should deblock a thread is received before the thread is actually blocked, the deblock operation is not triggered and the thread might never get deblocked.

Lost wakeup situations are hard to deal with in practice since it is difficult to distinguish the case where a wakeup is lost or merely very slow. Lost wakeups usually result in deadlocks or threads being blocked forever. So in principle, techniques to discover deadlocks or timeouts on the waiting times of threads can help resolve this issue.

The same problem can occur with signals (see Chapter 14) which might be sent to a process which is not yet waiting for them.

Lost wakeups could occur in a ULIX version with an interruptible kernel whenever a process or thread is blocked and the reason for blocking can disappear before the block operation completes. There are several such potential situations in the interruptible version's ULIX code, but they cannot occur in our non-interruptible version. We have, however, marked the critical sections in some places so that these cases are already dealt with (preparing again for the transition to an interruptible kernel).

As before, some of the following explanations will only make sense once you have read later chapters, so we suggest you return here after (for example) reading Chapter 13.5 which discusses the hard disk controller.

- `syscall_waitpid`$_{219c}$ blocks a process so that it can wait for a child process to terminate. In the interruptible version of ULIX, running this function with interrupts disabled would solve the possible situation of the child exiting after the parent entered `syscall_waitpid`$_{219c}$ but before it blocked.
- hard disk access (see Chapter 13.5): before sending a request to the disk controller, in the interruptible version of ULIX, the code would disable the interrupts. It would then check whether the request has already been completed and potentially block, re-enabling the interrupts after the thread has been moved to the blocked queue. This concerns the functions `writesector_hd`$_{530d}$ and `readsector_hd`$_{530c}$.
- floppy disk access (see Chapter 13.6): this is analogous to the hard disk case, but disabling and enabling interrupts would occur in different functions.

  The functions `fdc_read_sector`$_{549c}$ and `fdc_write_sector`$_{550b}$ call `fdc_command`$_{539c}$ which would disable interrupts and send a control sequence to the floppy controller.

  Then `fdc_command`$_{539c}$ calls `wait_fdc_interrupt`$_{547d}$ (in ⟨*fdc transfer* 540c⟩) which in turn calls `fdc_sleep`$_{545b}$ where interrupts would be re-enabled after blocking the thread.
- serial disk access (see Chapter 13.4): only read access can block, and this situation would be handled in the same way as hard disk access: before sending a read request to the serial disk, interrupts would be disabled.
- `syscall_readchar`$_{416b}$ (as discussed above) reads from a keyboard buffer and blocks if that is empty. Here, interrupts would be turned off between checking the buffer and calling `block`.
- `mutex_lock`$_{366a}$ blocks when the mutex is already held elsewhere. Since this function could be called in situations where interrupts may already be disabled, it saves the current state (interrupts on or off) and restores it before returning (when the lock has been acquired; see Section 11.6.4.2).

In all these cases, when a thread blocks, interrupts would be re-enabled after moving a thread to a blocked queue via `block`.

11.7 Further Reading

An excellent collection of synchronization problems (and solutions) is "The Little Book of Semaphores" by Allen B. Downey [Dow08] which is freely available on the publisher's website. It presents both classical and less well-known problems, and solutions are not simply given, but developed step by step—passing through several failed attempts and explaining why they failed.

For details on the POSIX synchronization functions, take a look at "Programming with POSIX Threads" by David R. Butenhof [But97]. Its chapter 3 (which deals with synchronization) is freely available online[1].

"The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit [HS12] is a very thorough introduction to the synchronization problems, especially those that occur on multiprocessor machines. As we mentioned earlier in Chapter 8.2, things get much more complicated when more than one CPU is used. It contains a lot of code that exemplifies the presented theory. That code is also available online.

A detailed discussion of top and bottom halves (tasklets) in the Linux kernel can be found in chapter 6 of Robert Love's kernel development book [Lov03].

11.8 Exercises

30. **Multiprocessor Kernel Synchronization**

    This exercise deals with the implementation of critical sections on multiprocessor systems, i. e., systems with multiple CPUs or CPUs with multiple cores. Assume you have a kernel in which critical sections are correctly labelled with `ENTER_MUTEX` and `EXIT_MUTEX`. For simplicity, assume that this holds for all interrupt handlers and all system calls. Assume further that two kernel level threads A and B are running on the system and that at least two CPUs execute these threads.

    For each of the following cases discuss whether one of the following two situations can arise:

    a) violation of mutual exclusion, i. e., threads A and B concurrently execute critical sections.
    b) deadlock, i. e., at least one thread will never be scheduled again.

    If one of these conditions can occur, construct a schedule of the system that ends in the condition being true. If the conditions can never occur, argue why it is impossible.

    - Case 1: single CPU, `ENTER_MUTEX` and `EXIT_MUTEX` are empty.
    - Case 2: single CPU, `ENTER_MUTEX` disables the interrupts on that CPU, `ENTER_MUTEX` enables the interrupts again.

[1] `http://ptgmedia.pearsoncmg.com/images/9780201633924/samplepages/0201633922.pdf`

- Case 3: single CPU, `ENTER_MUTEX` contains a spin lock on a global bit called `busy` and assigns 1 when the lock is taken. `EXIT_MUTEX` assigns 0 to `busy`.
- Case 4: single CPU, `ENTER_MUTEX` and `EXIT_MUTEX` are implemented as follows, using the definition of `Lock` from Section 11.2.2.2 (page 354):

[390a] ⟨*synchronization exercises, case 4* 390a⟩≡

```
Byte busy = false;

ENTER_MUTEX {
  ⟨disable interrupts 47a⟩
  while (Lock() == true);
}
EXIT_MUTEX {
  busy = false;
  ⟨enable interrupts 47b⟩
}
```

Uses `busy`, `ENTER_MUTEX`, and `EXIT_MUTEX`.

- Case 5: single CPU, `ENTER_MUTEX` and `EXIT_MUTEX` are implemented as follows:

[390b] ⟨*synchronization exercises, case 5* 390b⟩≡

```
Byte busy = false;

ENTER_MUTEX {
  ⟨disable interrupts 47a⟩
  while (Lock() == true);
  ⟨enable interrupts 47b⟩
}
EXIT_MUTEX {
  ⟨disable interrupts 47a⟩
  busy = false;
  ⟨enable interrupts 47b⟩
}
```

Uses `busy`, `ENTER_MUTEX`, and `EXIT_MUTEX`.

- Case 6: two CPUs, `ENTER_MUTEX` and `EXIT_MUTEX` are implemented as in case 1.
- Case 7: two CPUs, `ENTER_MUTEX` and `EXIT_MUTEX` are implemented as in case 2.
- Case 8: two CPUs, `ENTER_MUTEX` and `EXIT_MUTEX` are implemented as in case 3.
- Case 9: two CPUs, `ENTER_MUTEX` and `EXIT_MUTEX` are implemented as in case 4.
- Case 10: two CPUs, `ENTER_MUTEX` and `EXIT_MUTEX` are implemented as in case 5.

31. **Nestable Critical Sections on Multiprocessor Systems**

Does the implementation of nestable critical sections presented in chunks ⟨*nestable begin critical section* 357d⟩ and ⟨*nestable end critical section* 357e⟩ work on multiprocessor systems?

Does it work if disabling the interrupts is accompanied by a spin lock?

32. **Kernel Level Semaphores as Critical Sections**

Consider the implementation of `wait_semaphore`$_{361c}$ and `signal_semaphore`$_{362}$ in Section 11.3.4 and assume it were not declared as a critical section (i. e., interrupts remain on during execution of the code apart from within dispatcher operations). Is the implementation then still correct? Try to construct an example where two threads use a single semaphore, a context switch occurs and semaphore semantics are violated.

33. **Implementing Kernel Level Semaphores**

Consider the following implementation of `wait_semaphore`$_{361c}$ and `signal_semaphore`$_{362}$ that use atomic counter manipulations instead of non-atomic ones as in the actual implementation:

⟨*function implementations (inactive)* 391a⟩≡ 392a▷ [391a]

```
void wait_semaphore (kl_semaphore_id sid) {
  kl_semaphore sem = ⟨semaphore structure with identifier sid 363e⟩;
  ⟨begin critical section in kernel 380a⟩
  int count = __sync_sub_and_fetch (&sem.counter, 1);  // atomic "--sem.counter"
  if (count < 0) {
    block (&sem.bq, TSTATE_LOCKED);
    ⟨resign 221d⟩
  }
  ⟨end critical section in kernel 380b⟩
}

void signal_semaphore (kl_semaphore_id sid) {
  kl_semaphore sem = ⟨semaphore structure with identifier sid 363e⟩;
  ⟨begin critical section in kernel 380a⟩
  int count = __sync_add_and_fetch (&sem.counter, 1);  // atomic "++sem.counter"
  if (count < 1) {
    blocked_queue *bq = &(sem.bq);
    thread_id head = bq->next;
    if (head != 0) {
      deblock (head, bq);
    }
  }
  ⟨end critical section in kernel 380b⟩
}
```

Uses `__sync_add_and_fetch` 391b, `__sync_sub_and_fetch` 391b, `blocked_queue` 183a, `deblock` 186b, `kl_semaphore` 360a, `kl_semaphore_id` 360b, `signal_semaphore` 362, `thread_id` 178a, `TSTATE_LOCKED` 180a, and `wait_semaphore` 361c.

The atomic counter manipulation is done using the `gcc` compiler's built-in functions `__sync_sub_and_fetch`$_{391b}$ and `__sync_add_and_fetch`$_{391b}$ [Int01, section 7.4.1, p. 59] that require that we add the option `-march=i586` to the compiler flags (`CFLAGS`). The implementation of those functions is semantically equivalent to

⟨*compiler-internal functions* 354a⟩+≡ ◁354a [391b]

```
int __sync_sub_and_fetch (int *variable, int value) {
  *variable -= value;
```

```
  return *variable;
}

int __sync_add_and_fetch (int *variable, int value) {
  *variable += value;
  return *variable;
}
```

Defines:
__sync_add_and_fetch, used in chunk 391a.
__sync_sub_and_fetch, used in chunk 391a.

but performs all steps atomically.

The question is whether this implementation is in any aspect better than the one given in Section 11.3.4. Would it change anything if the body of the function were not protected as a critical section? (The latter question is an extension of exercise 32.)

34. **Locks as Critical Sections**

Consider the following variation of the implementation of mutex_lock$_{366a}$. The only difference is that the ⟨*begin critical section in kernel* 380a⟩ is moved to within the loop.

[392a] ⟨*function implementations (inactive)* 391a⟩+≡ ◁391a 392b▷

```
void mutex_lock (lock lockvar) {
  if (current_task == 0) { return; }        // no process
  while ( lockvar->l == 1 ) {
    ⟨begin critical section in kernel 380a⟩
    block (&(lockvar->bq), TSTATE_LOCKED);  // put process to sleep
    ⟨resign 221d⟩
  }
  lockvar->l = 1;
  ⟨end critical section in kernel 380b⟩
}
```

Is this implementation correct? Does it matter if the kernel were interruptible?

35. **Locks as Critical Sections (Variation)**

Here is another variation of the mutex_lock$_{366a}$ implementation. It uses the gcc-internal function __sync_lock_test_and_set$_{354a}$ discussed in Section 11.2.2.1 to guarantee that testing and setting are performed atomically.

[392b] ⟨*function implementations (inactive)* 391a⟩+≡ ◁392a

```
void mutex_lock (lock lockvar) {
  if (current_task == 0) { return; }        // no process
  while ( __sync_lock_test_and_set (&(lockvar->l), 1) != 0 ) {
    ⟨begin critical section in kernel 380a⟩
    block (&(lockvar->bq), TSTATE_LOCKED);  // put process to sleep
    ⟨resign 221d⟩
  }
  ⟨end critical section in kernel 380b⟩
}
```

Does this improve the correctness?

12
Filesystems

In this chapter we describe how operating systems store files on hard disks and floppy disks. The central concept for organizing directories and files is the filesystem: it is an abstract description of the required data structures.

In Chapter 13 we will look at what is needed to actually talk to a physical drive, but for now let's just assume that there is some mechanism which enables us to read and write "blocks": these are small chunks of disk storage into which we partition a disk—quite similar to the way that we've split memory into page frames.

We start with an overview of filesystem concepts in Section 12.1 that briefly discusses CP/M, FAT, NTFS and Unix filesystems. Section 12.2 explains the concept of mounting devices to mount points which is used on Unix operating systems. After this short theory block we jump right into the implementation details: First, Section 12.3 shows how the virtual filesystem (VFS) is organized in Ulix. It provides some abstractions which allow the OS to locate and use files on media which are formatted with various filesystems. In Section 12.4 we present the new system calls that can be used by user mode programs.

Then, in Sections 12.5 and 12.6 we introduce the Minix filesystem and show its implementation in Ulix.

Section 12.7 presents a second filesystem (for accessing device files in `/dev`), so you will see that the virtual filesystem layer is actually put to good use.

Finally, Section 12.8 gives a very short overview of the directory hierarchy that Ulix uses (which is modeled after other Unix systems).

12.1 Introduction to Filesystems

Every operating system needs to support one or more filesystems—at least one is required so that the OS can store and retrieve data, load programs from the disk and enable them to access files. Support for more than one filesystem makes sense when the OS wants to read media from other systems, e. g. FAT-formatted media which can be used on most of today's systems.

Why is there no single filesystem which all operating systems could agree on? Surely, this would make the cooperation of diverse systems much simpler.

If we want to understand why every OS has its own idea of how to store files (and possibly directories) on disk we have to look back to the beginnings of external storage.

Card Readers Early computers used punch cards (see Figure 12.1) to store jobs and the associated data: a card reader would be filled with a stack of such cards, the first set of cards contained the binary program to be run, and the following cards held the data. The system would read all these cards into memory and then run the job. After completion, the results of the computation would be punched on empty cards or sent to a connected printer. Data organization in such a card stack was strictly serial, so there was no concept of a filesystem: the first card(s) would describe how many program and data cards would follow and where to store their contents in RAM.

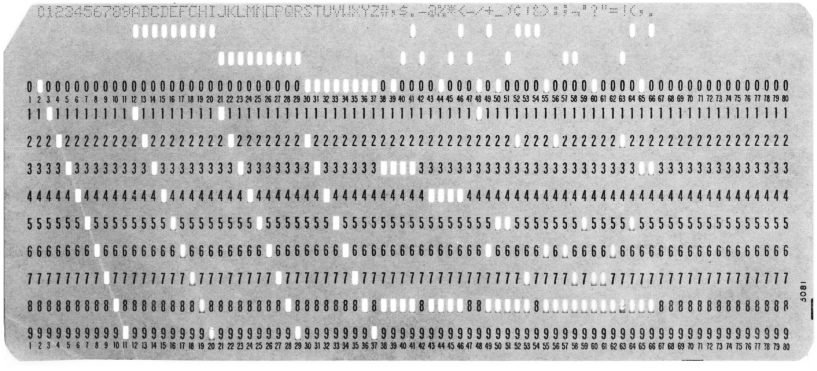


Figure 12.1: Punch cards were used to store program code and data.

Tape The next step was the introduction of magnetic tape drives. These live on until today (as backup media, or third-level storage), and they are also strictly serial: Typically tape drives can be sent a *rewind* command to move to the beginning of the tape and *read* and *write* commands to read or write the tape sequentially. While it is possible to store more than one file on a tape, accessing the third file requires skipping the first two ones—which can only be achieved by reading them first. Thus, tapes have no filesystem, either. Today, when people use a tape drive for backup, they typically

generate an archive file (which may or may not contain a central listing of the files) and write this single archive file onto the tape. The Unix `tar` program (`tar`: tape archive) writes a file header for each contained file and then the file's data; then the next file header and data follow. The created archive file is written raw to the tape. `tar`

Disk The seriality of storage was changed with the introduction of disks: it does not matter whether you think of floppy or hard disks, all of them allow *random access*, that means you can store several files on one disk and read any file (or part of a file) without looking at other data as long as you know where to find the file on the disk. Thus, disks need some kind of *directory* information so that the machine can look up files on a previously unseen disk. The question of how and where to store this directory information on the disk defines (most of) a filesystem, and many OS developers have had their own ideas about the organization of files on a disk. Early machines were not meant to be compatible with machines of other manufacturers, and this led to various incompatible filesystems. We'll look at some examples in the next section. random access, directory

12.1.1 Simple Filesystems

We assume throughout the rest of this chapter that a disk is divided into *blocks* of some fixed size (typically 512 or 1024 bytes), and these blocks can be read and written using some kind of `readblock` and `writeblock` commands which the disk drive controller supports. We're not going to delve into the details of how a hard disk is organized into platters, cylinders, sectors and tracks; instead we will assume that there is a logical ordering of blocks and that the controller allows to access these *logical blocks* directly via some mechanism that the OS can use. physical block, logical block

12.1.1.1 Contiguous Filesystems

As long as you want to write files to a disk only *once* (and then continue using it in read-only-mode), organizing files is rather simple: Assume you want to store 150 files on the disk. You can then reserve one or more blocks at the beginning of the disk for a central directory—just enough space to store filename, starting block and file length for each file (see Table 12.1). Then write each file to the disk, starting at the first unused block, and update the directory afterwards.

All files are stored *contiguously*, which means that you can later read them sequentially as long as you know where to start reading (see Figure 12.2). Opening and reading files then means just looking up the filename in the directory and starting to read at the start block number which is stored next to the filename.

The ISO-9660 filesystem [ECM87] which is used for compact discs works in a similar way, so this concept is still in use today. When we first implemented a disk driver for Ulix we used this approach as a quick hack to create a read-only filesystem. ISO-9660 (CD)

Why is there a problem with this kind of organization? Imagine you want to enable write-support for such a filesystem. As long as you only modify blocks inside a file, ev-

Filename	Size	First Block	Block Count
somefile.txt	3301	2	4
otherfile.txt	49152	6	48
test.txt	11147	54	11

Table 12.1: This is a simple directory of a contiguous filesystem. It stores both the number of blocks as well as the actual file size which might be less than a multiple of the block size (1024). The number of blocks could also be calculated from the file size.

T	**T**	A	A	A	A	B	B	B	B	B	B	B	B	B	B	00..15
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	16..31
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	32..47
B	B	B	B	B	B	C	C	C	C	C	C	C	C	C	C	48..63
C	(free)															64..79
(free)																80..95

Figure 12.2: Organization of disk blocks in a contiguous filesystem: The **T** blocks hold the table of contents, and A, B and C represent the three sample files.

erything is fine: In the same fashion used for reading a file, you can calculate the block number from the first block and the file offset and then change the right data block on the disk to update the file's contents. But what about appending to a file? Here's where we run into problems. Once you reach the end of the last block of a file, you cannot go on, since the next block on the disk belongs to a different file. You would first have to move all blocks of the following file to an unused region on the disk (and update the directory) before you can continue to write new blocks for the first file. But such a procedure will leave *holes* in the disk: unused regions which may be used for new files but more often than not will be too small to store a complete file inside. This is called *external fragmentation.*

external fragmentation

The solution to this problem is to not store just a starting block and the file length, but a collection of block numbers: with such a method any free block may be used to store file contents, and the order of the blocks is irrelevant. Also, files can then be spread all over the disk instead of being contiguous.

12.1.1.2 Non-contiguous Filesystems

Giving up the contiguousness makes file storage more flexible, but reading and writing become harder: once you start reading or writing to a file, you can only read/write until the end of the current block; to continue you need to first look up the block number of the next data block (which can be found in some kind of directory, see Table 12.2 and Figure 12.3). And again, this information is only good until the end of that block, when you need to look up the next block number.

Filename	Size	Block List
`somefile.txt`	3301	2, 3, 16, 44
`otherfile.txt`	49152	4, 5, ..., 15, 17, 18, ..., 43, 55, 56, ..., 63
`test.txt`	11147	45, 46, ..., 54, 64

Table 12.2: This is the directory of a non-contiguous filesystem. Instead of start block and block count it stores a block list for each entry.

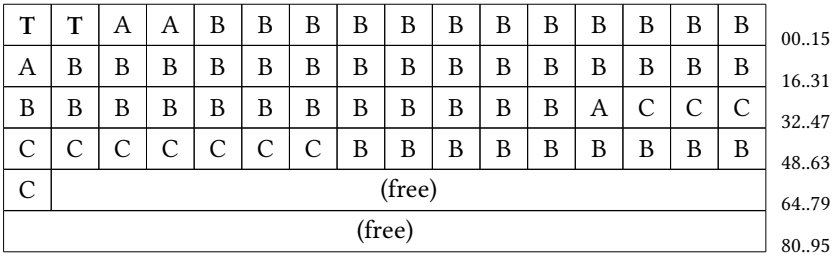


Figure 12.3: Organization of disk blocks in a non-contiguous filesystem: Files can use any set of available blocks.

Most modern filesystems work in this way, so the question remains where to store the block numbers.

The new flexibility comes at a cost: reading a disk sequentially is cheap because the read/write heads are always positioned properly for reading the next block. When you allow a file to spread over the disk, the disk must seek to the next block which costs time: it slows down reading. A disk that holds lots of files which are spread over the disk blocks in this fashion is called *fragmented*, and many operating systems provide defragmenting tools which reorganize the files so that all blocks of a file occur in-order on the disk. However, a freshly defragmented disk becomes fragmented again once you start deleting files and writing new ones.

fragmented disk

12.1.1.3 CP/M

An early example for non-contiguous filesystems is that of CP/M, an operating system which was popular in the 70s and early 80s. Actually, there was a variety of non-compatible CP/M filesystems, but they were at least all similar.

CP/M's filesystem was flat, i. e., it did not implement the concept of subdirectories. There was one central directory that held the information about all files on the disk. For bringing some order into a big list of files, CP/M introduced the concept of a user number: each file's metadata contained a number between 0 and 15 (standard: 0) and by changing the current user number to *n* the internal `DIR` command would only show files with that user number *n*. That did not protect a user's files from access by another user, but it allowed cooperative and trusted users to share a disk.

Block numbers for each file were stored in the directory entry (which was 32 bytes long, with 16 bytes reserved for block numbers). If a file used more data blocks than one directory entry could address, another directory entry for the same file was created (and called an *extent*). One directory entry could hold either 16 or eight block numbers, depending on the total size of the disk: early floppy disks stored ≈ 180 KByte, and the blocksize was 1 KByte, thus on those floppies one byte was enough to reference a block [JL83, pp. 19 ff.].

extent (CP/M)

CP/M files have three attributes which may be set or unset: 'read-only', 'system' and 'file changed'. These are stored in the highest bits of three of the filename bytes—filenames must be made of ASCII characters which use only seven bits of each byte. CP/M filenames follow the "8.3" convention which was later picked up by MS-DOS: the first eight characters named the file, the other three characters were used for the file extension which defined the filetype (e. g. `COM`, `C`, `PAS`). Lower-case letters were forbidden. The dot in a filename such as `TEST.COM` was not stored in the directory entry, and you could not create names like `X.ENDING`, since the filename and extension were treated separately (with their eight characters / three characters limits).

12.1.1.4 MS-DOS FAT

The *FAT* filesystem (*File Allocation Table*) of MS-DOS and Windows uses a different approach for storing the block numbers. Instead of blocks, disks are divided into *clusters*, and the size of a cluster depends on the filesystem size. For example, with FAT16 the cluster size is 512 bytes for filesystem sizes up to 32 MByte, it is 32 KByte for filesystems with a size between 1 GByte and 2 GByte [Mic00b].

cluster (FAT)

There are three variants of FAT named FAT12, FAT16 and FAT32. For example, a FAT16 directory entry maps a filename to the first data cluster of the file. Further clusters can be found via traversing a linked list, the file allocation table: it contains one 16-bit entry for each cluster, and such an entry has one of the following values:

- `0x0000`: Free cluster.
- `0x0002`–`0xFFEF`: points to the next cluster in the linked list.
- `0xFFF0`–`0xFFF6`: Reserved.
- `0xFFF7`: Bad cluster (cannot be used).
- `0xFFF8`–`0xFFFF`: This is the last cluster of the file. (Microsoft operating systems only use the `0xFFFF` value.)

(The size of the cluster number (16 bit) is what gives FAT16 its name. For FAT12 and FAT32 there are similar conventions, e. g. `0x0FFFFFFF` (28 1-bits) marks the end of the list on FAT32, and `0xFFF` (twelve 1-bits) on FAT12 media. As Microsoft's specification [Mic00b] notes, FAT32 is actually "FAT28" since only the lower 28 bits of a cluster number are interpreted.)

This means that it is impossible to quickly access the end of a large file, for example a file with FAT16's maximum size (which uses 65522 clusters) requires reading 65522 FAT entries in order to determine the last cluster number.

12.1.2 Advanced Filesystems

Leaving FAT and other classical filesystems behind, we now look at two more advanced specimens: the NTFS filesystem which was introduced with Microsoft Windows NT and the Unix way of storing files.

12.1.2.1 NTFS

The *New Technology Filesystem* (*NTFS*) is a successor to both FAT and the *High Performance Filesystem* (*HPFS*) that was developed for *OS/2* by Microsoft and IBM. When the two companies ended the cooperation on OS/2, IBM kept on using HPFS, and Microsoft developed Windows NT which introduced NTFS. The central data structure of an NTFS volume is the *Master File Table* (*MFT*) which contains entries for each file and each directory, including itself (since the MFT is also a file). Filename and data are attributes of a file. The filename is always part of the MFT entry, and the file data may also be if they are small enough. Otherwise, the data are external to the MFT entry. In that case, NTFS stores information about one or more *cluster runs*: A cluster run is a contiguous set of clusters, identified by a starting cluster and the number of clusters. Each such cluster run description is encoded so that is uses as few bytes as possible, it starts with a header byte in which the high half-byte gives the size (in bytes) of the following start cluster number, and the low half-byte tells how many bytes are used to describe the number of clusters (they follow behind the first encoded number).

HPFS, OS/2

MFT

cluster run

As an example, `32 EF CD AB 02 01` (all numbers are hexadecimal) describes a cluster run with the following properties:

- `32`: three bytes for the start cluster number
- `32`: two bytes for the cluster count number
- `EF CD AB`: little-endian encoding of the start cluster number, `0xabcdef`
- `02 01`: little-endian encoding of the cluster count number, `0x0102`

This means that the cluster starts at cluster `0xabcdef` and ends at cluster `0x0xabcef0` (= `0xabcdef` + `0x0102` − 1).

If this is the only cluster (i. e., the file is contiguous or non-fragmented), a further 0 byte ends the description, otherwise the encoded form of the next cluster run follows; in any case the last cluster run description is followed by a 0 byte, ending the entry. This may degenerate into a classical list of used clusters where the cluster count number is always 1, but that is not normally the case.

In comparison to FAT, access to an arbitrary cluster is much faster because all the information that is needed to find the nth cluster of a file is stored directly in the MFT entry, whereas on a FAT volume half the cluster chain must be inspected (on average).

NTFS allows a file to have several names; it can store more than one filename attribute in an MFT entry. For example, besides the regular name (as seen on Windows) files can also have a short filename for compatibility with old MS-DOS applications.

Directories on NTFS are also files (again with an MFT entry) and their file entries are organized as B-trees. Directory entries map a name to the MFT entry of the associated

file, but also (redundantly) store the file size and time stamps that are also available via each file's MFT entry [Cus94, p. 28].

The organization into directories that map filenames to MFT entries somewhat resembles the Unix filesystem structure where directories map filenames to inodes (see below).

data stream

NTFS provides several interesting features, for example it performs journaling and allows more than one standard *data stream*: The file contents are considered the default data stream, but there can be further, named data streams which can be accessed via a `filename:streamname` syntax. These extra streams could be used to implement versioning of files (keeping several old version of a file).

12.1.2.2 Unix Filesystems

Where CP/M and MS-DOS link all data to a filename as the significant identifier in a directory entry, Unix filesystems work differently: They assign numbers to files, not names. These numbers are called *inode numbers* and they point to entries in an inode table. Each

inode

inode (index node) stores metadata about a file, such as ownership, access permissions, file creation and modification dates and some kind of pointers to the file's data blocks.

Files get a name by writing that name and an inode number into a special directory file. As an inode may occur several times in such directory files, any file can have more than one name. Thus, filenames are not unique.

Like a CP/M directory entry, a Unix inode has a few fields which contain block numbers and lead to the first data blocks of a file. However, in order to support large files, a different mechanism is needed because the inode has a fixed (and small) size. We have already discussed in Section 3.2.4.9 (p. 81) that Unix filesystems work with several layers

indirection

of *indirection*—how many depends on the concrete filesystem. With single indirection one or more of the inode fields point to blocks which do not contain data but other block numbers. Double indirection introduces an extra layer of indirection, and triple indirection goes yet one step further (Figure 12.4).

We will not discuss the general Unix filesystem characteristics in this overview since you will see all the details in the upcoming sections which implement one of its variants (the Minix filesystem).

12.2 Mounting: the Unix Way to Access Many Volumes

drive letter

Some operating systems dedicate a "drive identifier" to each volume that is in use. CP/M was one of the earlier systems with a huge user base, and they used *drive letters* (A, B) to access more than one floppy drive. The idea was copied by MS-DOS and has been kept alive to this day, with Windows drive letters always starting at C (A and B are still reserved for floppy drives). Figure 12.5 shows how Windows accesses three volumes via three drive letters.

In the Unix world drive letters are unknown. Instead, Unix combines all volumes into one huge Unix directory tree (Figure 12.6). This means that parts of the tree can represent

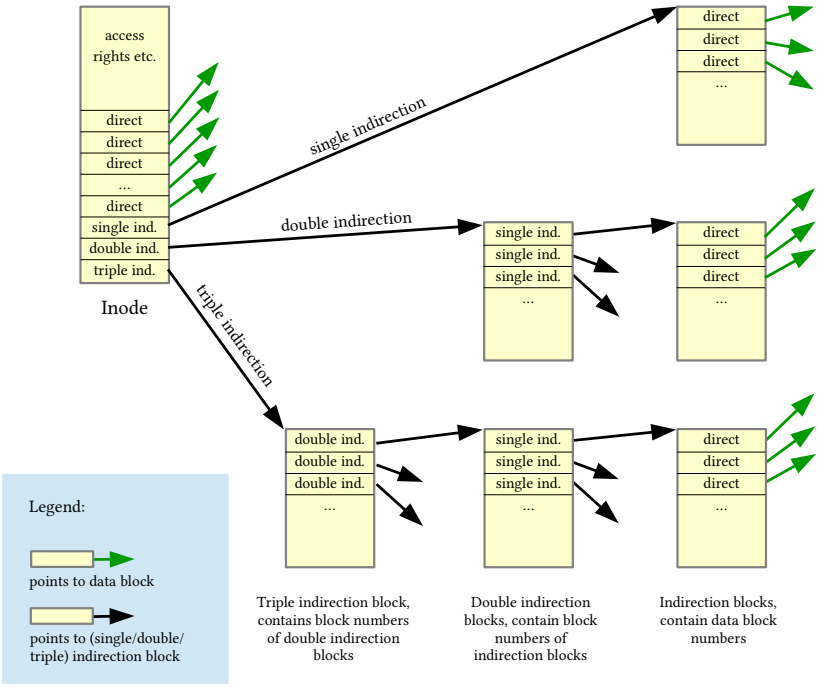


Figure 12.4: In Unix filesystems inodes store direct and single or mulitiple indirection block addresses.

the contents of several volumes. The process of adding a new volume (and thus enlarging the tree) is called *mounting*. The root of the newly included filesystem will not appear as the root of the overall tree, but as some node in the middle of the new tree. That node is called the *mount point*.

mounting

mount point

Originally, Unix used regular files as mount points. The idea was to look at the current directory tree (before the mount operation), then pick a leaf in this tree (regular files cannot have children in the tree) and attach the root of the new volume to this leaf [RT74].

Modern Unix-type systems use directories as mount points which can have the effect of hiding files if the directory chosen as a mount point is not empty (i. e., it is not a leaf of the tree).

A mount operation can be undone: the system can *unmount* a volume which removes the volume's sub-tree from the directory tree. This is only possible when there are no open files on the volume. Unix systems support mounting and unmounting via `mount` and `unmount` system calls (and library functions of the same names), whereas the corresponding command line tools are called `mount` and `umount` (without the "n" letter of `unmount`).

unmount

When mounting a volume, it is also possible to provide *mount options* which are specific to this volume, and special mount options may be available which depend on the filesystem. A classical mount option is *read only* which completely forbids write operations to

mount option

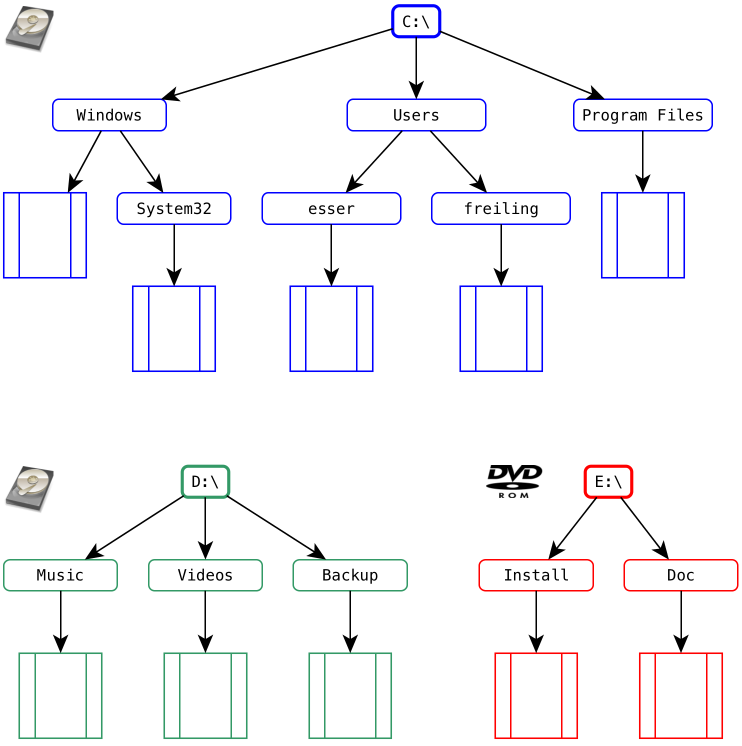


Figure 12.5: Windows uses separate drive letters for each volume.

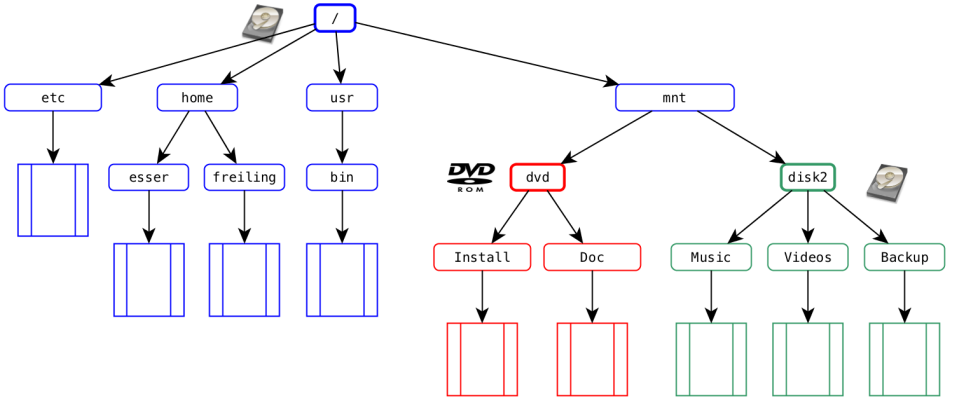


Figure 12.6: Unix systems integrate all volumes in one tree. In this example, `/`, `/mnt/dvd/` and `/mnt/disk2/` are mount points.

the volume. This is sometimes required by the device type (e. g. in case of a DVD), and at other times it is just so desired by the user; for example, computer forensic examiners need to mount media in read-only mode so that all data will remain in their original state.

12.3 The Ulix Virtual Filesystem

Ulix shall use a *virtual filesystem* (*VFS*). That means that several real filesystems might be used, and the VFS provides an abstraction so that generic functions such as `open`$_{429b}$, `read`$_{429b}$ and `write`$_{429b}$ may be used for accessing these. VFS

There are several layers, as you can see in Figure 12.7:

- Starting at the lowest level, we need code that can interact with the controllers to which floppy or hard disk drives are connected. They will work with blocks of data, i. e., they read or write a whole block (and not single bytes) each time. Chapter 13 will present the code necessary to talk to the devices and perform the data transfers between memory and disk.
- One level above, we provide generic block read/write functions that will work with any supported device. They take a device ID as a parameter but otherwise let the next level ignore specifics of the device in use.
- Yet higher in the driver hierarchy are the *logical filesystem drivers*. Ulix only provides one implementation of such a driver (we support the Minix filesystem), but we write the code in such a way that we (or you) could easily add additional drivers. The logical level is concerned with the organization of files, directories and *metadata* of a volume: this is what this chapter is mainly about. We will give an introduction to all the details of the *Minix* filesystem design. logical filesystem, metadata, Minix
- On the highest level there are the virtual file system functions. They work with pathnames for accessing files or directories, and such pathnames will be translated into (device, local pathname) pairs. If you split the mount point from an absolute path, the remainder is the local (absolute) path on the volume which is mounted on that mount point. This is also the code for which we provide a user mode interface via system calls.

Whenever a file is being accessed, we want Ulix to take the following steps:

1. Calculate the *absolute path* of the file. Note that a filename may already be given as an absolute path (e. g. `/usr/bin/ps`), but it may also be given as a relative path (e. g. `../ps`). In the latter case we construct the absolute path from the current working directory and the *relative path*. absolute path, relative path
2. Scan the mount table to find out on which filesystem the file is located. This will return two values: a pointer to the filesystem and a path which is local to this filesystem. In case of `/mnt/tmp/file.txt` this may lead to the number 1 (standing for filesystem number 1 which is mounted on `/mnt`) and the path (`/tmp/file.txt`) within that filesystem.

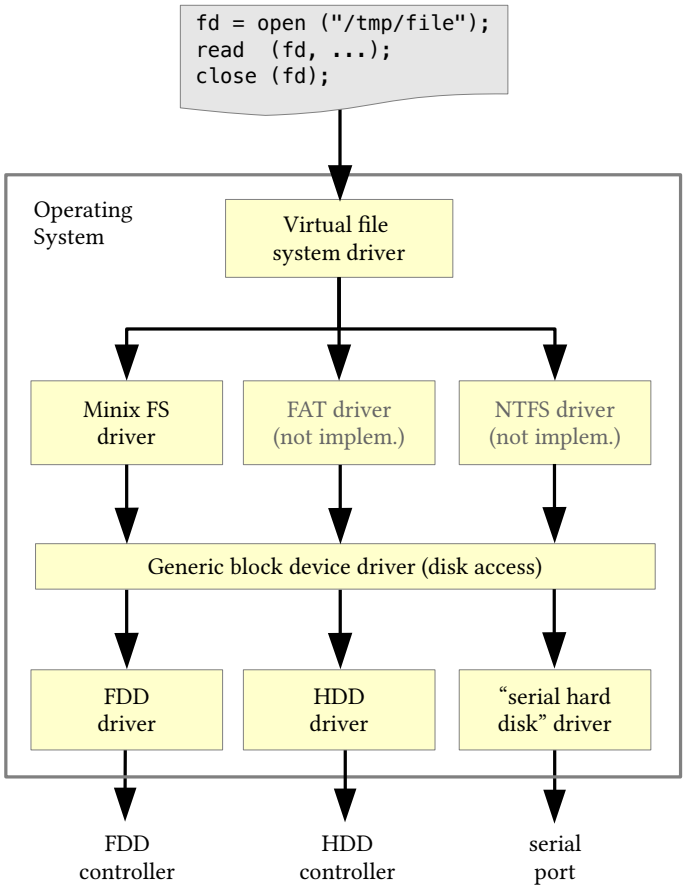


Figure 12.7: The Virtual Filesystem has a layered design. For accessing files, the kernel and processes use functions at the virtual filesystem level, and they will be translated through the layers until they are finally turned into floppy or hard disk access function calls.

3. Depending on the service function which was called (e. g. `open`), find a registered function that can talk to this kind of filesystem (e. g. `mx_open` for a Minix filesystem or `fat_open` for a DOS/FAT filesystem) and call it.

The filesystem-specific functions should assume that they can access the filesystem as a large, consecutive block of data. As already mentioned, ULIX will provide generic functions `readblock`$_{506b}$ and `writeblock`$_{507c}$ which can be used to access the raw data, be they on a disk partition, a floppy disk or inside RAM.

We will restrict the number of mounts to 16. A *mount table entry* looks like this: mount table

⟨ *type definitions* 91 ⟩+≡ (44a) ◁365a 440c▷ [405a]
```
typedef struct {
  char mountpoint[256];
  short fstype;           // filesystem type, e. g. Minix, device filesystem
  short device;           // e. g. DEV_FD0, DEV_HDA
  short mount_flags;      // always 0; we will not use mount flags
} mount_table_entry;
```
Defines:
mount_table_entry, used in chunk 405b.

We do not provide `mount` and `umount` functions in the kernel, but instead work with a fixed table. This could easily be remedied since those functions would just add or remove an entry to the following array and perform some checks:

⟨ *global variables* 92b ⟩+≡ (44a) ◁383b 410b▷ [405b]
```
mount_table_entry mount_table[16] = {
  { "/",     FS_MINIX, DEV_HDA,  0 },
  { "/mnt/", FS_MINIX, DEV_FD1,  0 },
  { "/tmp/", FS_MINIX, DEV_HDB,  0 },
  { "/dev/", FS_DEV,   DEV_NONE, 0 },
  { { 0 } }
};
short current_mounts = 4;   // how many FSs are mounted?
```
Defines:
current_mounts, used in chunks 406, 408c, and 492.
mount_table, used in chunks 406, 408, and 492.
Uses DEV_FD1 508a, DEV_HDA 508a, DEV_HDB 508a, DEV_NONE 508a, FS_DEV 410a, FS_MINIX 410a, and mount_table_entry 405a.

(The constants $\texttt{DEV\_HDA}_{508a}$, $\texttt{DEV\_HDB}_{508a}$ and $\texttt{DEV\_FD1}_{508a}$ identify the first two hard disks and the second floppy disk; we will define them in the next chapter.) The information in such an entry corresponds roughly to the data you can observe in `/etc/mtab` on a Linux system:

⟨ *Linux mtab entry* 405c ⟩≡ [405c]
```
/dev/sda2 / ext3 rw,errors=remount-ro 0 0
```

(This line shows the device filename, the mount point, the filesystem type, the mount options (in this case: read-write, remount as read-only in case of errors) and dump and filesystem check options which we will not deal with in ULIX.)

A mount table entry is unused if its `fstype` element is `0`. The mount flag value `0` calls for standard mount options. (In our case that means readable and writeable, though we do not provide alternatives such as read-only.) The following helper function

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁373b 408b▷ [405d]
```
void print_mount_table ();
```

will be called during system startup and show the mount table:

[406] ⟨*function implementations* 100b⟩+≡ (44a) ◁373c 408a▷

```
void print_mount_table () {
  int i, dev;
  for (i=0; i<current_mounts; i++) {
    dev = mount_table[i].device;
    if (dev != 0) {
      char *devname;
      switch (dev) {
        case DEV_HDA: devname = "hda"; break;
        case DEV_HDB: devname = "hdb"; break;
        case DEV_FD0: devname = "fd0"; break;
        case DEV_FD1: devname = "fd1"; break;
      }
      printf ("mount: dev[%02x:%02x] = /dev/%s on %-5s type %-5s (options %d)\n",
              devmajor (dev), devminor (dev), devname, mount_table[i].mountpoint,
              fs_names[mount_table[i].fstype], mount_table[i].mount_flags);
    } else
      printf ("mount: none                  on %-5s type %-5s (options %d)\n",
              mount_table[i].mountpoint,
              fs_names[mount_table[i].fstype], mount_table[i].mount_flags);
  }
}
```

Defines:
`print_mount_table`, used in chunks 45c and 405d.
Uses `current_mounts` 405b, `DEV_FD0` 508a, `DEV_FD1` 508a, `DEV_HDA` 508a, `DEV_HDB` 508a, `devmajor` 505b, `devminor` 505b, `fs_names` 410b, `mount_table` 405b, and `printf` 601a.

The `fs_names`$_{410b}$`[]` array contains strings describing the filesystems; its definition follows on page 410.

We will provide two concrete implementations of filesystems:

Minix
- a *Minix filesystem* implementation which describes how to (logically) read and write Minix-formatted media

/dev filesystem
- and a `/dev` filesystem, similar to Linux, which provides information about known devices (such as `/dev/fd0` for the first floppy drive).

The architecture will be such that it is possible to add support for other filesystems, for example FAT (from MS-DOS/Windows). Since ULIX belongs to the Unix family, we will provide abstract Unix filesystem features (such as symbolic and hard links, user and group information, classical Unix access permissions and timestamps) and have to map them to the data which are available in a concrete filesystem (on a disk).

The other layer is the hardware: It shall be possible to use all supported filesystems on any kind of device for which there are `blockread` and `blockwrite` functions. Thus, when ULIX tries to `open` a file and `read` from it, the system will start with executing the virtual `u_open`$_{412c}$ or `u_read`$_{414b}$ function, then call (for example) Minix-related `mx_open`$_{464b}$ or `mx_read`$_{470b}$ functions and finally end in calls to the hardware-specific `readblock`$_{506b}$ functions. The overall process of executing `fd = open`$_{429b}$`("/mnt/tmp/test");` with the second floppy drive mounted on `/mnt/` is shown in Figure 12.8.

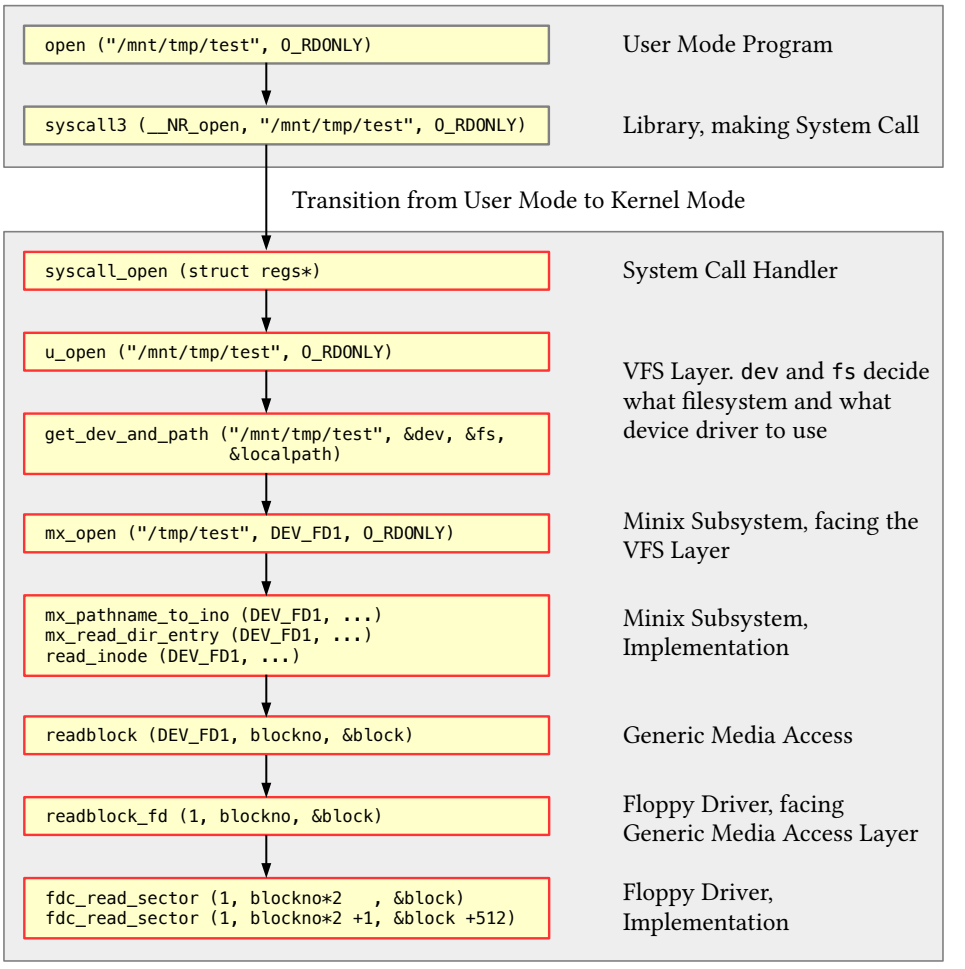


Figure 12.8: Opening a file via the user mode library function `open()`.

On the hardware side, ULIX will provide drivers for floppy disks and hard disks. Thus, there will be support for Minix-formatted floppy disks and hard disks.

12.3.1 Finding the Device and Local Path

Let's look at a possible scenario: Assume we have two floppy disks (`fda`, `fdb`) and two RAM disks (`ram0`, `ram1`) mounted like this:

```
/dev/fda  on  /             (minix)
/def/fdb  on  /home         (minix)
/dev/ram0 on  /mnt          (minix)
/dev/ram1 on  /home/ramtest (minix)
```

Since the path `/home/ramtest` does not exist before `fdb` has been mounted on `/home/`, the second RAM disk[1] must have been mounted *after* the second floppy disk. Thus, if we assume that we store the mount information in the order in which it was created by `mount`, we can search the mount table backwards, starting with the last entry, and compare each mount point to the leading chraracters of the absolute path name:

[408a] ⟨*function implementations* 100b⟩+≡ (44a) ◁406 409b▷
⟨*find device and local path* 408c⟩

[408b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁405d 409a▷
```
int get_dev_and_path (char *path, short *dev, short *fs, char *localpath);
```

[408c] ⟨*find device and local path* 408c⟩≡ (408a) 408d▷
```
int get_dev_and_path (char *path, short *dev, short *fs, char *localpath) {
  int i, mount_entry;
  for (i = current_mounts-1;  i ≥ 0;  i--) {
    // standard case: file; mount point (e.g. "/mnt/" is head of path)
    if (string_starts_with (path, mount_table[i].mountpoint)) {
      mount_entry = i;  break;
    }
    // second case: directory, path is mount point without /, e.g. "/mnt"
    if ( strlen (path) == strlen (mount_table[i].mountpoint)-1 &&
       string_starts_with (mount_table[i].mountpoint, path) ) {
      mount_entry = i;  break;
    }
  }
```
Defines:
`get_dev_and_path`, used in chunks 408b, 411e, 419, 588b, and 589a.
Uses `current_mounts` 405b, `g`, `mount_table` 405b, `string_starts_with` 409b, and `strlen` 594a.

Note that this loop cannot fail since the first mount entry always has the mount point `/`, and every syntactically correct absolute path begins with `/`. This only works because we search backwards: if we were searching forwards, we would always find the root filesystem and ignore all further mounts.

Once we have found the relevant entry we can split off the leading mount point and also know the device and filesystem type:

[408d] ⟨*find device and local path* 408c⟩+≡ (408a) ◁408c
```
  split_mountpoint (mount_table[mount_entry].mountpoint, path, localpath);
  if (strlen (localpath) == 0) {
    // empty string
    localpath[0] = '/';  localpath[1] = 0;
  }
  *dev = mount_table[mount_entry].device;
  *fs  = mount_table[mount_entry].fstype;
  return 0;
}
```
Uses `mount_table` 405b, `split_mountpoint` 409c, and `strlen` 594a.

[1] This version of ULIX does not implement RAM disks; if you want to see ULIX RAM disk support, you can read Liviu Beraru's bachelor thesis [Ber13].

That is really all there is to do. Every call of this function (with a syntactically correct absolute path) must be successful, however that does not mean that the path truly exists: Other functions must check whether the local part of the path (`localpath`) is available on the device—but we know where to look now.

We need to implement the two helper functions `string_starts_with`$_{409b}$ (which is similar to `strcmp`$_{594a}$) and `split_mountpoint`$_{409c}$:

⟨*function prototypes* 45a⟩+≡ (44a) ◁408b 411b▷ [409a]
```
int string_starts_with (char *str, char *start);
void split_mountpoint (char *mountpoint, char *path, char *localpath);
```

⟨*function implementations* 100b⟩+≡ (44a) ◁408a 409c▷ [409b]
```
int string_starts_with (char *str, char *prefix) {
  if (strlen (prefix) > strlen (str)) { return false; }  // cannot be a sub-string
  while (*prefix != '\0') {
    if (*prefix++ != *str++) { return false; }     // found different character
  };
  return true;                                     // parsed all of prefix; match!
}
```
Defines:
`string_starts_with`, used in chunks 408c and 409a.
Uses `strlen` 594a.

The function `split_mountpoint`$_{409c}$ expects that the `path` string does in fact start with `mountpoint`. It does not check this property but only removes as many characters as necessary:

⟨*function implementations* 100b⟩+≡ (44a) ◁409b 412b▷ [409c]
```
void split_mountpoint (char *mountpoint, char *path, char *localpath) {
  // input:  mountpoint, e.g. /home/
  //         path,       e.g. /home/user/file.txt
  // output: localpath,  e.g. /user/file.txt
  int len = strlen (mountpoint);
  strncpy (localpath, path+len-1, 256);
}
```
Defines:
`split_mountpoint`, used in chunk 408d.
Uses `g`, `strlen` 594a, and `strncpy` 594b.

12.3.2 Constants for Filesystems

We declare some constants for the filesystems which are (or might be) supported by ULIX: In most cases we will work with `FS_MINIX`$_{410a}$ since we provide a full implementation of the Minix filesystem. The device filesystem `FS_DEV`$_{410a}$ is also available, but `FS_ERROR`$_{410a}$ and als `FS_FAT`$_{410a}$ will only cause errors if they occur anywhere. We have included an `FS_FAT`$_{410a}$ constant because we use it occasionally for explaining how FAT support *could* be added to ULIX.

[410a] ⟨*constants* 112a⟩+≡ (44a) ◁365b 411c▷
```
#define FS_ERROR  0
#define FS_MINIX  1
#define FS_FAT    2
#define FS_DEV    3
```
Defines:
FS_DEV, used in chunks 405b, 412c, 414b, 415a, 418, 420–22, 425c, 588b, and 589a.
FS_ERROR, used in chunks 412c, 414b, 415a, 418, 420–22, 425c, 588b, and 589a.
FS_FAT, used in chunks 412c, 414b, 415a, 418, 420–22, 425c, 588b, and 589a.
FS_MINIX, used in chunks 405b, 412c, 414b, 415a, 418–22, 425c, 588b, and 589a.

[410b] ⟨*global variables* 92b⟩+≡ (44a) ◁405b 459c▷
```
char *fs_names[] = { "ERROR", "minix", "fat", "dev" };
```
Defines:
fs_names, used in chunks 406 and 492.

A definition of devices will follow in Chapter 13 on disk I/O.

12.3.3 Global File Descriptors

We will allow all filesystem drivers to manage their own sets of file descriptors since they may use them as an index into a private table. Thus, when we open several files on Minix-formatted media, file descriptors 0, 1, 2, 3, etc. will be in use. If another filesystem driver (e. g. one for FAT) exists, it may use the same numbers.

Obviously just passing those file descriptor numbers to the calling process (or kernel function) would create chaos with some numbers being used twice or more often. We avoid this problem by granting each filesystem a range of 256 numbers. For each filesystem `filesys` and a filesystem-*local file descriptor* `localfd` we calculate the *global file descriptor* via `fd = (filesys << 8) + localfd`.

Open Minix files will have file descriptors in the range 256–511, and FAT files would have descriptors in the range 512–767. When one of the functions `u_read`$_{414b}$, `u_write`$_{415a}$ etc. is called, it expects a global file descriptor as argument. By reversing the above calculation via `filesys = fd >> 8; localfd = fd & 0xff;` we can easily find out which filesystem function we need to call and which local file descriptor we have to provide it.

So, in order to clear the terminology, here is an overview of the three types of file descriptors we will use:

Global File Descriptor: A global file descriptor is used by the kernel to identify a unique open file—across all processes (and the kernel itself).

Local File Descriptor: Each subsystem (such as the Minix or `/dev` subsystem) uses its own set of file descriptors. These are also global in that they are not associated with any specific process, but no generic filesystem function is meant to use them.

Process File Descriptor: Each process keeps a list of its own file descriptors (in its thread table entry). Those descriptors are mapped to global file descriptors via those entries. They only make sense when seen from a process' point of view.

12.3.4 Opening a File

We start with an implementation of the u_open$_{412c}$ function which—in case of a file on a Minix filesystem—will call mx_open$_{464b}$. All filesystem related functions on the virtual filesystem layer will have a u_ prefix so that we can distinguish them from the user mode library functions with the same names (e. g. u_open$_{412c}$ and u_read$_{414b}$ inside the kernel, open$_{429b}$ and read$_{429b}$ in the user mode library). u_ prefix

The prototype for u_open$_{412c}$ almost follows the Unix standards. We do not allow the optional third argument of the POSIX standard,

⟨*POSIX prototypes* 411a⟩≡ [411a]
```
int open (const char *pathname, int flags);
int open (const char *pathname, int flags, mode_t mode);
```

but instead expect a third parameter open_link that we will use to decide whether u_open$_{412c}$ shall follow symbolic links or just open the link file itself. This option will not be available to the corresponding system call since we do not want processes to manually handle symbolic links.

When creating a file, ULIX always sets the standard access permissions 644_o; they can later be modified with u_chmod$_{589a}$. Thus the u_open$_{412c}$ prototype is:

⟨*function prototypes* 45a⟩+≡ (44a) ◁409a 412a▷ [411b]
```
int u_open (char *path, int oflag, int open_link);
```

⟨*constants* 112a⟩+≡ (44a) ◁410a 415c▷ [411c]
```
// u_open parameter  int open_link:
#define DONT_FOLLOW_LINK 1
#define FOLLOW_LINK 0
```
Uses DONT_FOLLOW_LINK, FOLLOW_LINK, and u_open 412c.

Before we start with the function implementation we note that there will be a recurring pattern in many of the following functions: Many of them start with converting a path argument into an absolute path via relpath_to_abspath$_{412b}$, and then they look up the device, the filesystem type and the local path via get_dev_and_path$_{408c}$. They will always use variables named localpath, abspath, device and fs. Thus, we create two code chunks for the variable declarations and the function calls:

⟨*VFS functions: declare default variables* 411d⟩≡ (412c 418b 421 422) [411d]
```
char  localpath[256], abspath[256];
short device, fs;
```

⟨*VFS functions: make absolute path, get device, fs and local path* 411e⟩≡ (412c 418b 421 422) [411e]
```
if (*path != '/')
  relpath_to_abspath (path, abspath);
else
  strncpy (abspath, path, 256);
get_dev_and_path (abspath, &device, &fs, (char*)&localpath);
```
Uses get_dev_and_path 408c, relpath_to_abspath 412b, and strncpy 594b.

[412a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁411b 414a▷
```
void relpath_to_abspath (const char *relpath, char *abspath);
```

[412b] ⟨*function implementations* 100b⟩+≡ (44a) ◁409c 412c▷
```
void relpath_to_abspath (const char *relpath, char *abspath) {
  if (strlen (thread_table[current_task].cwd) > 1) {
    // combine cwd and relpath, add "/" in the middle
    strncpy (abspath, thread_table[current_task].cwd, 256);
  } else {
    strncpy (abspath, "", 256);
  }
  strncpy (abspath + strlen (abspath) + 1, relpath, 256 - strlen (abspath) - 1);
  abspath[strlen (abspath)] = '/';
}
```
Defines:
relpath_to_abspath, used in chunks 411e, 412a, 419, 432e, 488a, 588b, and 589a.
Uses current_task 192c, cwd, strlen 594a, strncpy 594b, and thread_table 176b.

The implementation of u_open$_{412c}$ is rather short because the VFS layer forwards all the real work the the corresponding functions in some subsystem (for the Minix or device filesystem):

[412c] ⟨*function implementations* 100b⟩+≡ (44a) ◁412b 414b▷
```
int u_open (char *path, int oflag, int open_link) {
  ⟨VFS functions: declare default variables 411d⟩
  ⟨VFS functions: make absolute path, get device, fs and local path 411e⟩
  ⟨u_open: handle symlink 413b⟩
  if (scheduler_is_active) {
    ⟨u_open: check permissions 577c⟩              // see user/group chapter
  }

  int fd;
  switch (fs) {
    case FS_MINIX:
      fd = mx_open (device, localpath, oflag);
      if (fd == -1) return -1;                 // error (opening failed)
      else          return (fs << 8) + fd;
    case FS_FAT:    return -1;                 // not implemented
    case FS_DEV:
      fd = dev_open (localpath, oflag);
      if (fd == -1) return -1;                 // error (opening failed)
      else          return (fs << 8) + fd;
    case FS_ERROR:  return -1;                 // error (wrong FS)
    default:        return -1;                 // error (wrong FS)
  }
}
```
Defines:
u_open, used in chunks 190c, 229a, 293b, 411, 420, 426b, 488a, and 582a.
Uses dev_open 495c, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_open 464b, and scheduler_is_active 276e.

This is all there is to do: the function first looks into the mount table in order to determine on which device the file resides and what filesystem that device is formatted with. Since `get_dev_and_path`$_{408c}$ also calculates the local absolute path within the filesystem, `u_open`$_{412c}$ can immediately call `mx_open`$_{464b}$ or `dev_open`$_{495c}$ which do the real work. If we were to add support for the Linux Ext3 filesystem, we would modify this code by adding a case for `fs == FS_EXT3`:

⟨*adding Ext3 support* 413a⟩≡ [413a]
```
case FS_EXT3:
  fd = ext3_open (device, localpath, oflag);
  if (fd == -1) return -1;                // error (opening failed)
  else          return (fs << 8) + fd;
```

Note that we need not deal with special cases such as non-existing files or files which do not have the necessary access permissions in this code: This will be handled in the filesystem-specific functions, such as `mx_open`$_{464b}$. If these return an error code (instead of a file descriptor), the result is just forwarded to the caller of `u_open`$_{412c}$.

In case that the file to open is a *symbolic link*, we need to follow the link. We do this by reading the link file and using the path found there: symbolic link

⟨`u_open`*: handle symlink* 413b⟩≡ (412c) [413b]
```
struct stat st;
char link[256];
u_stat (abspath, &st);
if (((st.st_mode & S_IFLNK) == S_IFLNK) && (open_link == FOLLOW_LINK)) {
  // open (how?), read_, then u_open (symlink)
  int link_fd = u_open (abspath, O_RDONLY, DONT_FOLLOW_LINK);  // open link file
  u_read (link_fd, link, 256);
  u_close (link_fd);
  return u_open (link, oflag, FOLLOW_LINK);                    // recursion
}
```

This calls `u_open`$_{412c}$ recursively, and our simple function does not check the recursion level. Thus a simple sequence of `ln -s xyz xyz` and `cat xyz` in some directory will force `u_open`$_{412c}$ into an infinite recursion (and crash the system when the kernel stack exceeds its boundary).

To see this mechanism in operation, consider that there is a file `file` and we have two symbolic links, `a` and `b`, where `a` points to `b` and `b` points to `file`. Trying to open `a` will cause the following recursion:

⟨*example for recursive* `u_open` *calls* 413c⟩≡ [413c]
```
u_open ("/a", oflag, FOLLOW_LINK)             // open "a", called from somewhere
  st.st_mode == S_IFLNK                       // "a" is a symlink
  u_open ("/a", O_RDONLY, DONT_FOLLOW_LINK)   // open it read-only and...
  u_read () -> "/b"                           // read the contents: it's "b"
  u_close ("/a)                               // close "a"
  u_open ("/b", oflag, FOLLOW_LINK)           // open "b" (return its retval)
    st.st_mode == S_IFLNK                     // "b" is also a symlink
    u_open ("/b", O_RDONLY, DONT_FOLLOW_LINK) // open it RO and...
    u_read () -> "/file"                      // read the contents: it's "file"
```

```
u_close ("/b")                          // close b
u_open ("/file", oflag, FOLLOW_LINK)    // open "file" (return its retval)
  st.st_mode != S_IFLNK                 // no link; return file descriptor
```

12.3.5 Reading, Writing and Other Operations

The operations on open files are even simpler because we don't have to find out what filesystem to use: we get that information via the global file descriptor. For example, the $\texttt{u\_read}_{414b}$ function extracts the filesystem type and the local file descriptor from the global file descriptor (just like we have already described earlier); then it branches and calls one of the `*_read` functions.

[414a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁412a 421a▷

```
int u_read      (int fd, void *buf, int nbyte);
int u_write     (int fd, void *buf, int nbyte);
int u_close     (int fd);
int u_lseek     (int fd, int offset, int whence);
int u_unlink    (const char *path);
int u_link      (const char *path, const char *path2);
int u_symlink   (const char *path, const char *path2);
int u_truncate  (const char *path, int length);
int u_ftruncate (int fd, int length);
int u_readlink  (char *path, char *restrict buf, size_t bufsize);
```

$\texttt{u\_read}_{414b}$ simply requires the following code:

[414b] ⟨*function implementations* 100b⟩+≡ (44a) ◁412c 415a▷

```
int u_read (int fd, void *buf, int nbyte) {
  if (fd < -100) { ⟨read: standard I/O and pipes 416d⟩ }
  ⟨VFS functions: turn fd into (fs, localfd) pair or fail 414c⟩
  switch (fs) {
    case FS_MINIX: return mx_read  (localfd, buf, nbyte);
    case FS_FAT:   return -1;                                  // not implemented
    case FS_DEV:   return dev_read (localfd, buf, nbyte);
    case FS_ERROR: return -1;                                  // error
    default:       return -1;
  }
}
```

Defines:
  u_read, used in chunks 190c, 229a, 233b, 294, 413c, 420c, 426b, and 582a.
Uses dev_read 496d, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, and mx_read 470b.

with

[414c] ⟨*VFS functions: turn* fd *into* (fs, localfd) *pair or fail* 414c⟩≡ (414b 415a 418a 420b 425c)

```
if (fd < 0) return -1;   // file not open
int fs      = fd >> 8;
int localfd = fd & 0xff;
```

(Again we create a separate code chunk for this check and the calculation which turns a global file descriptor into a (filesystem, local file descriptor) pair since we will reuse it several times.)

Negative file descriptors with `fd < -100` deal with the special cases of *standard input/output* and *pipes* (though pipes have not been implemented in this version); we will describe that case soon. The code for writing is almost identical to that of `u_read`$_{414b}$:

standard I/O
pipe

⟨ *function implementations* 100b ⟩+≡ (44a) ◁414b 418a▷ [415a]

```
int u_write (int fd, void *buf, int nbyte) {
  if (fd < -100) { ⟨ write: standard I/O and pipes 417 ⟩ }
  ⟨ VFS functions: turn fd into (fs, localfd) pair or fail 414c ⟩
  switch (fs) {
    case FS_MINIX: return mx_write  (localfd, buf, nbyte);
    case FS_FAT:   return -1;  // not implemented
    case FS_DEV:   return dev_write (localfd, buf, nbyte);
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}
```

Defines:
`u_write`, used in chunks 293d, 414a, and 426b.
Uses `dev_write` 497, `FS_DEV` 410a, `FS_ERROR` 410a, `FS_FAT` 410a, `FS_MINIX` 410a, and `mx_write` 474c.

We define the file descriptors for `stdin`, `stdout` and `stderr` (as they are valid inside processes) and kernel-internal negative values for the three standard I/O streams:

`stdin`, `stdout`, `stderr`

⟨ *public constants* 46a ⟩+≡ (44a 48a) ◁328d 424b▷ [415b]

```
#define STDIN_FILENO  0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

Defines:
`STDIN_FILENO`, used in chunks 431 and 432b.
`STDOUT_FILENO`, used in chunks 431 and 598c.

⟨ *constants* 112a ⟩+≡ (44a) ◁411c 440a▷ [415c]

```
#define DEV_STDIN     (-101)
#define DEV_STDOUT    (-102)
#define DEV_STDERR    (-103)
```

Defines:
`DEV_STDERR`, used in chunks 190a, 416d, 417, and 421b.
`DEV_STDIN`, used in chunks 190a, 416d, 417, and 421b.
`DEV_STDOUT`, used in chunks 190a, 416d, 417, and 421b.

In order to read from standard input, we make a system call which in turn will execute `syscall_readchar`$_{416b}$, our function for reading from the keyboard:

⟨ *ulix system calls* 206e ⟩+≡ (205a) ◁372e 428b▷ [415d]

```
#define __NR_readchar  525
```

Defines:
`__NR_readchar`, used in chunk 416c.

[416a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁372c 426a▷
```
void syscall_readchar (context_t *r);
```
Uses context_t 142a and syscall_readchar 416b.

The implementation lets the current process block if no new character is available in the keyboard buffer:

[416b] ⟨*syscall functions* 174b⟩+≡ (202b) ◁372d 426b▷
```
void syscall_readchar (context_t *r) {
  char c;
  int t = thread_table[current_task].terminal;
  terminal_t *term = &terminals[t];

  // get character, return 0 if there is no new character in the buffer
  ⟨begin critical section in kernel 380a⟩  // access the thread table
  if (term->kbd_count > 0) {
    term->kbd_count--;
    term->kbd_lastread = (term->kbd_lastread+1) % SYSTEM_KBD_BUFLEN;
    c = term->kbd[term->kbd_lastread];
  } else {
    c = 0;
    if ((current_task > 1) && scheduler_is_active) {
      block (&keyboard_queue, TSTATE_WAITKEY);
      ⟨end critical section in kernel 380b⟩
      ⟨resign 221d⟩
    }
  };
  r->ebx = c;      // return value in ebx
  ⟨end critical section in kernel 380b⟩
};
```
Defines:
  syscall_readchar, used in chunk 416.
Uses context_t 142a, current_task 192c, keyboard_queue 323d, scheduler_is_active 276e, SYSTEM_KBD_BUFLEN 318a, terminal_t 318b, terminals 318c, thread_table 176b, and TSTATE_WAITKEY 180a.

As usual, we need to add the new system call handler to the table:

[416c] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁373a 428a▷
```
install_syscall_handler (__NR_readchar, syscall_readchar);
```
Uses __NR_readchar 415d, install_syscall_handler 201b, and syscall_readchar 416b.

Reading from standard output or standard error is not allowed and causes an error. Also there are no pipes in this Ulix version, but the idea is to let the kernel create an internal buffer for each pipe and associate two negative file descriptors with it.

[416d] ⟨*read: standard I/O and pipes* 416d⟩≡ (414b)
```
byte c = 0;
unsigned int u;
switch (fd) {
  case DEV_STDIN:
    for (int i = 0;  i < nbyte;  i++) {
      // read one character from the keyboard
```

```
        __asm__ ("\
          .intel_syntax noprefix; \
          mov eax, 525; \
          int 0x80; \
          mov %0, ebx; \
          .att_syntax"
          :
          "=r"(u)
        );
        c = (byte) u;
        ((byte*) buf)[i] = c;
      };
      break;

    case DEV_STDOUT:
    case DEV_STDERR:
      printf ("(ERROR: reading from stdout or stderr)\n");
      return (-1);          // error, cannot read from output

    default: return (-1);  // pipes not implemented yet
  }
```

Uses DEV_STDERR 415c, DEV_STDIN 415c, DEV_STDOUT 415c, and printf 601a.

Simlilarly, writing to standard output or standard error dumps data on the terminal using the kputch$_{335b}$ function, whereas writing to standard input (or pipes) is forbidden:

⟨*write: standard I/O and pipes* 417⟩≡ (415a) [417]

```
  byte c;
  switch (fd) {
    case DEV_STDIN:
      printf ("(ERROR: writing to stdin)\n");
      return (-1);  // error, cannot write to input

    case DEV_STDOUT:
    case DEV_STDERR:
      for (int i = 0;  i < nbyte;  i++) {
        c = ((char*)buf)[i];
        if (c > 31 || c == '\n' || c == 0x08) {
          kputch (c);                    // regular characters: 32..255, \n, \b
        } else {
          kputch ('^'); kputch (c+64);   // control characters: <32
        }
      }

    default: return (-1);                // pipes not implemented yet
  }
```

Uses DEV_STDERR 415c, DEV_STDIN 415c, DEV_STDOUT 415c, kputch 335b, and printf 601a.

Closing an open file or performing a seek operation work like reading, in that u_close$_{418a}$ and u_lseek$_{418a}$ simply call the corresponding mx_* or dev_* functions:

[418a] ⟨*function implementations* 100b⟩+≡ (44a) ◁415a 418b▷

```
int u_close (int fd) {
  ⟨VFS functions: turn fd into (fs, localfd) pair or fail 414c⟩
  switch (fs) {
    case FS_MINIX: return mx_close (localfd);
    case FS_FAT:   return -1;                          // not implemented
    case FS_DEV:   return dev_close (localfd);
    case FS_ERROR: return -1;                          // error
    default:       return -1;
  }
}

int u_lseek (int fd, int offset, int whence) {
  ⟨VFS functions: turn fd into (fs, localfd) pair or fail 414c⟩
  switch (fs) {
    case FS_MINIX: return mx_lseek (localfd, offset, whence);
    case FS_FAT:   return -1;                          // not implemented
    case FS_DEV:   return dev_lseek (localfd, offset, whence);
    case FS_ERROR: return -1;                          // error
    default:       return -1;
  }
}
```

Defines:
u_close, used in chunks 190c, 216b, 229a, 233b, 420, 426b, 488a, and 582a.
u_lseek, used in chunks 233b, 293, 294, and 426b.
Uses dev_close 496b, dev_lseek 498a, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_close 467b, and mx_lseek 469c.

The implementation of u_unlink$_{418b}$ is similar to that of u_open$_{412c}$ since in both cases we get a pathname as argument:

[418b] ⟨*function implementations* 100b⟩+≡ (44a) ◁418a 419a▷

```
int u_unlink (const char *path) {
  ⟨VFS functions: declare default variables 411d⟩
  ⟨VFS functions: make absolute path, get device, fs and local path 411e⟩
  switch (fs) {
    case FS_MINIX: return mx_unlink (device, localpath);
    case FS_FAT:   return -1;  // not implemented
    case FS_DEV:   return -1;  // no unlink support in device FS
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}
```

Defines:
u_unlink, used in chunks 426b and 488a.
Uses FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, and mx_unlink 480c.

hard link

On the other hand, creating a new *hard link* with u_link$_{419a}$ requires some extra work: The function takes two pathnames, one of which is for a name that does not yet exist. It must check that both reside on the same volume because hard links cannot cross volumes.

The rest is similar, but we work without the `switch` statement since hard links only exist on Unix filesystems (of which ULIX only supports Minix):

⟨*function implementations* 100b⟩+≡ (44a) ◁418b 419b▷ [419a]

```
int u_link (const char *path, const char *path2) {
  char localpath[256],  abspath[256];  short device,  fs;
  char localpath2[256], abspath2[256]; short device2, fs2;
  char dir2[256], base2[256], localdir2[256];

  if (*path != '/')  relpath_to_abspath (path, abspath);       // source
  else               strncpy (abspath, path, 256);
  get_dev_and_path (abspath, &device, &fs, (char*)&localpath);

  if (*path2 != '/') relpath_to_abspath (path2, abspath2);     // target
  else               strncpy (abspath2, path2, 256);
  splitpath (abspath2, dir2, base2);           // get dirname
  get_dev_and_path (dir2, &device2, &fs2, (char*)&localdir2);

  if (device != device2) return -1;            // error: link across volumes
  if (fs != FS_MINIX)    return -1;            // error: not Minix

  strncpy (localpath2, localdir2, 256);        // localpath2 = localdir2
  int len = strlen(localpath2);
  if (len == 1) len = 0;  // special case "/"
  localpath2[len] = '/';                       // localpath2 += "/"
  strncpy (localpath2 + len + 1, base2, 256);  // localpath2 += base2
  return mx_link (device, localpath, localpath2);
}
```

Defines:
`u_link`, used in chunk 426b.
Uses `dirname` 455b, `FS_MINIX` 410a, `get_dev_and_path` 408c, `mx_link` 480a, `relpath_to_abspath` 412b, `splitpath` 455a, `strlen` 594a, and `strncpy` 594b.

The `u_symlink`$_{419b}$ function only checks the target since it is allowed to write invalid paths (and paths to different volumes) into a symbolic link:

⟨*function implementations* 100b⟩+≡ (44a) ◁419a 420a▷ [419b]

```
int u_symlink (const char *path, const char *path2) {
  char localpath2[256], abspath2[256]; short device2, fs2;
  if (*path2 != '/') relpath_to_abspath (path2, abspath2);     // target
  else               strncpy (abspath2, path2, 256);
  get_dev_and_path (abspath2, &device2, &fs2, (char*)&localpath2);
  if (fs2 != FS_MINIX)  return -1;             // error: not Minix
  return mx_symlink (device2, (char*)path, localpath2);
}
```

Defines:
`u_symlink`, used in chunk 426b.
Uses `FS_MINIX` 410a, `get_dev_and_path` 408c, `mx_symlink` 484b, `relpath_to_abspath` 412b, and `strncpy` 594b.

For truncating a file we only need to think about the the `u_ftruncate`$_{420b}$ variant which works on an open file; the other function can just open the file and call `u_ftruncate`$_{420b}$:

[420a] ⟨*function implementations* 100b⟩+≡ (44a) ◁419b 420b▷
```
int u_truncate  (const char *path, int length) {
  int fd = u_open ((char*)path, O_WRONLY, FOLLOW_LINK);
  int retval = u_ftruncate (fd, length);
  u_close (fd);
  return retval;
}
```
Defines:
u_truncate, used in chunk 426b.
Uses FOLLOW_LINK, O_WRONLY 460b, u_close 418a, u_ftruncate 420b, and u_open 412c.

The u_ftruncate$_{420b}$ function looks like all the other u_* functions that have a file descriptor as their first argument, but we only allow the Minix filesystem:

[420b] ⟨*function implementations* 100b⟩+≡ (44a) ◁420a 421b▷
```
int u_ftruncate (int fd, int length) {
  ⟨VFS functions: turn fd into (fs, localfd) pair or fail 414c⟩
  switch (fs) {
    case FS_MINIX: return mx_ftruncate (localfd, length);
    case FS_FAT:   return -1;                              // not implemented
    case FS_DEV:   return -1;                              // forbidden
    case FS_ERROR: return -1;                              // error
    default:       return -1;
  }
}
```
Defines:
u_ftruncate, used in chunks 420a and 426b.
Uses FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, and mx_ftruncate 484e.

The function u_readlink$_{420c}$ reads a link file and retrieves the link target:

[420c] ⟨*minix filesystem implementation* 420c⟩≡ (440b) 443b▷
```
int u_readlink (char *path, char *restrict buf, size_t bufsize) {
  struct stat st;
  u_stat (path, &st);
  if ((st.st_mode & S_IFLNK) != S_IFLNK) {
    return -1;  // error: no symlink
  }
  int link_fd = u_open (path, O_RDONLY, DONT_FOLLOW_LINK);  // open the link file
  u_read (link_fd, buf, bufsize);
  u_close (link_fd);
  return 0;     // success
}
```
Defines:
u_readlink, used in chunk 426b.
Uses DONT_FOLLOW_LINK, O_RDONLY 460b, S_IFLNK 457c, size_t 46b, stat 429b 489b, u_close 418a, u_open 412c, u_read 414b, and u_stat 421d.

12.3.6 Detect Terminals

Sometimes a process needs to find out whether it is reading from or writing to a file or one of the standard I/O streams. For that purpose it can call the user mode function isatty$_{429b}$

which returns `true` if the file descriptor is connected to a terminal. In case of files or pipes it returns `false`. In order to implement the kernel function

⟨*function prototypes* 45a⟩+≡ (44a) ◁414a 421c▷ [421a]
```
boolean u_isatty (int fd);
```

we simply check whether the file descriptor is `DEV_STDIN`$_{415c}$, `DEV_STDOUT`$_{415c}$ or `DEV_STDERR`$_{415c}$:

⟨*function implementations* 100b⟩+≡ (44a) ◁420b 421d▷ [421b]
```
boolean u_isatty (int fd) {
  return ((fd == DEV_STDIN) || (fd == DEV_STDOUT) || (fd == DEV_STDERR));
}
```
Uses DEV_STDERR 415c, DEV_STDIN 415c, and DEV_STDOUT 415c.

12.3.7 Status

The `u_stat`$_{421d}$ function fills a `struct stat`$_{429b}$ entry with the status information about a file which can be queried with `mx_stat`$_{490a}$ or `dev_stat`$_{499d}$:

⟨*function prototypes* 45a⟩+≡ (44a) ◁421a 421e▷ [421c]
```
int u_stat (const char *path, struct stat *buf);
```

⟨*function implementations* 100b⟩+≡ (44a) ◁421b 422a▷ [421d]
```
int u_stat (const char *path, struct stat *buf) {
  ⟨VFS functions: declare default variables 411d⟩
  ⟨VFS functions: make absolute path, get device, fs and local path 411e⟩
  switch (fs) {
    case FS_MINIX: return mx_stat (device, localpath, buf);
    case FS_FAT:   return -1;  // not implemented
    case FS_DEV:   return dev_stat (localpath, buf);
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}
```
Defines:
u_stat, used in chunks 413b, 420c, 421c, 426b, 432e, 576, and 577.
Uses dev_stat 499d, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_stat 490a, and stat 429b 489b.

12.3.8 Directories

We also need to handle directories: it is possible to create and remove them via the

⟨*function prototypes* 45a⟩+≡ (44a) ◁421c 422b▷ [421e]
```
int u_mkdir (const char *path, int mode);
int u_rmdir (const char *path);
```

functions and to read their entries via `u_getdent`. The `u_mkdir`$_{422a}$ and `u_rmdir`$_{422a}$ implementations are just rewrites of `u_open`$_{412c}$ and `u_unlink`$_{418b}$:

[422a] ⟨*function implementations* 100b⟩+≡ (44a) ◁421d 422c▷

```
int u_mkdir (const char *path, int mode) {
  ⟨VFS functions: declare default variables 411d⟩
  ⟨VFS functions: make absolute path, get device, fs and local path 411e⟩
  switch (fs) {
    case FS_MINIX: return mx_mkdir (device, localpath, mode);
    case FS_FAT:   return -1;  // not implemented
    case FS_DEV:   return -1;  // not allowed
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}

int u_rmdir (const char *path) {
  ⟨VFS functions: declare default variables 411d⟩
  ⟨VFS functions: make absolute path, get device, fs and local path 411e⟩
  switch (fs) {
    case FS_MINIX: return mx_rmdir (device, abspath, localpath);  // two path args
    case FS_FAT:   return -1;  // not implemented
    case FS_DEV:   return -1;  // no rmdir support in device FS
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}
```

Defines:
u_mkdir, used in chunk 426b.
u_rmdir, used in chunks 421e and 426b.
Uses FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_mkdir 487a, and mx_rmdir 488a.

For reading a directory we provide the

[422b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁421e 424a▷

```
int u_getdent (const char *path, int index, struct dir_entry *buf);
```

function which reads single entries and writes them into a `struct dir_entry`$_{490b}$ buffer:

[422c] ⟨*function implementations* 100b⟩+≡ (44a) ◁422a 424d▷

```
int u_getdent (const char *path, int index, struct dir_entry *buf) {
  ⟨VFS functions: declare default variables 411d⟩
  ⟨VFS functions: make absolute path, get device, fs and local path 411e⟩
  switch (fs) {
    case FS_MINIX: return mx_getdent (device, localpath, index, buf);
    case FS_FAT:   return -1;  // not implemented
    case FS_DEV:   return dev_getdent (localpath, index, buf);
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}
```

Uses dev_getdent 500, dir_entry 490b, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_getdent 490d, and u_getdent.

12.4 System Calls for File Access

We have already discussed the three types of file descriptors which are used in ULIX. All the functions of the virtual filesystem layer which you have seen so far use global file descriptors which uniquely identify an open file across all processes. But this information should be hidden from individual processes—after all, operating systems are all about abstraction, and from a process' point of view only its own open files are relevant, thus a process can expect that its internal file descriptor numbers do not depend on the activities of other processes (or the kernel). Also, we want to support the Unix tradition of reserving file descriptor numbers 0, 1 and 2 for the standard I/O streams.

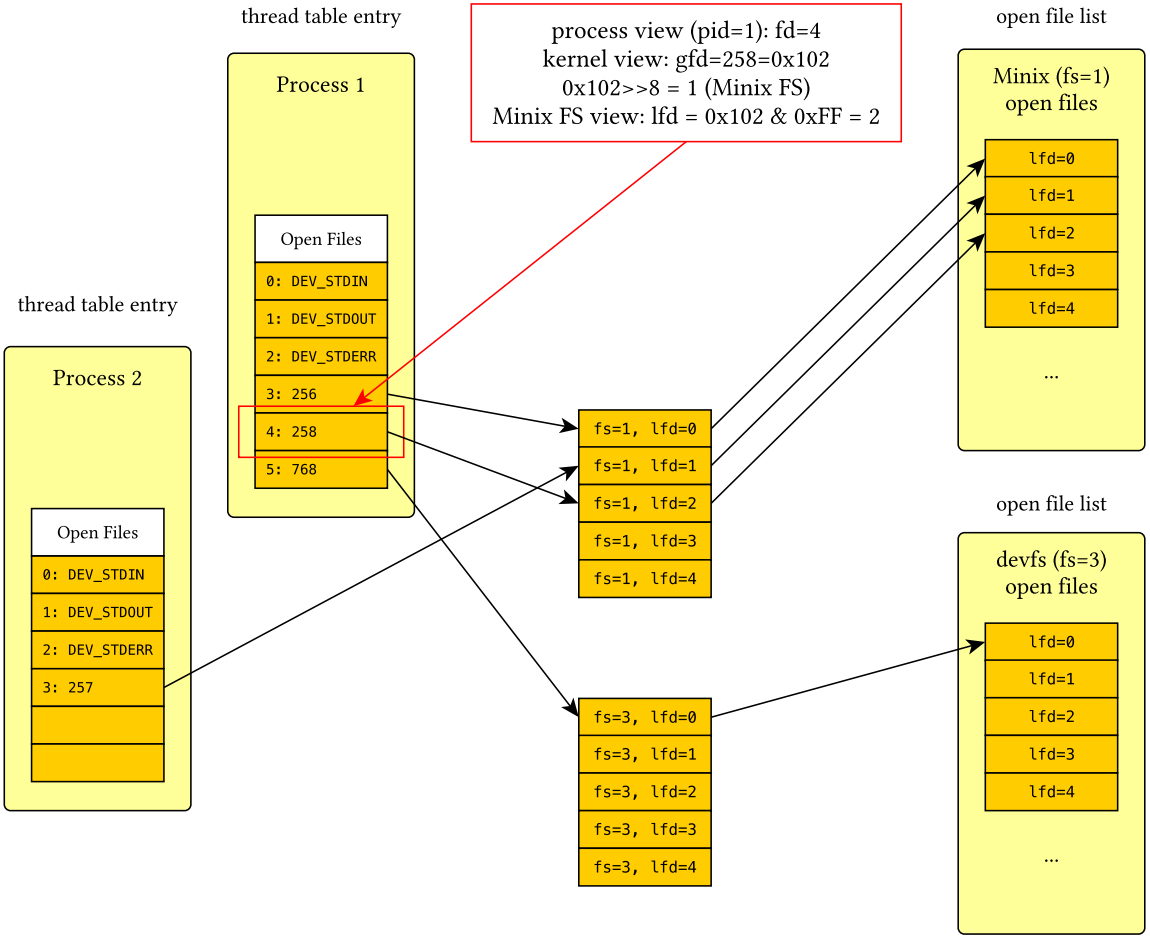


Figure 12.9: Relationship between the various file descriptors.

The `u_*` functions in the kernel use global file descriptors which do not recognize process affiliation. Since we do not want to export these numbers to the processes, we must convert them to process file descriptors.

We provide the following two functions `gfd2pfd` and `pfd2gfd` which convert global to process and process to global file descriptors. Figure 12.9 shows the relationship between process, local and global file descriptors.

[424a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁422b 425b▷

```
int gfd2pfd (int gfd);
int pfd2gfd (int pfd);
```

Processes may open up to 16 files; we add a new field to the TCB which stores that many file descriptors.

[424b] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁415b 457c▷

```
#define MAX_PFD 16   // up to 16 open files per process
```

Defines:
`MAX_PFD`, used in chunks 190a, 216b, and 424–26.

[424c] ⟨*more TCB entries* 158c⟩+≡ (175) ◁326d 432c▷

```
int files[MAX_PFD];
```

Uses `MAX_PFD` 424b.

If we start with a process file descriptor, the translation is simpler: we just look it up in the process' file descriptor table.

[424d] ⟨*function implementations* 100b⟩+≡ (44a) ◁422c 424e▷

```
int pfd2gfd (int pfd) {
  if (pfd == -1) return -1;
  thread_id pid = thread_table[current_task].pid;
  if (pfd ≥ 0 && pfd < MAX_PFD)
    return thread_table[pid].files[pfd];
  else return -1;
}
```

Uses `current_task` 192c, `MAX_PFD` 424b, `pfd2gfd`, `thread_id` 178a, and `thread_table` 176b.

(Note that $\texttt{thread\_table}_{176b}\texttt{[current\_task}_{192c}\texttt{].files[pfd]}$ may also be `-1` which is the standard value for an unused local file descriptor.)

Turning a global file descriptor into a process one is a bit more complicated: if there is no mapping (yet), we need to find a free place in the process' descriptor list `files`. But we start with searching for a mapping:

[424e] ⟨*function implementations* 100b⟩+≡ (44a) ◁424d 425c▷

```
int gfd2pfd (int gfd) {
  int pfd;
  if (gfd == -1) return -1;
  thread_id pid = thread_table[current_task].pid;
  for (pfd = 0;  pfd < MAX_PFD;  pfd++) {
    if (thread_table[pid].files[pfd] == gfd)
      return pfd;
  }
```

```
    // found none, create it
    for (pfd = 0;  pfd < MAX_PFD;  pfd++) {
      if (thread_table[pid].files[pfd] == -1) {
        thread_table[pid].files[pfd] = gfd;
        return pfd;
      }
    }
    // no free entry
    return -1;  // error
  }
```

Uses current_task 192c, gfd2pfd, MAX_PFD 424b, thread_id 178a, and thread_table 176b.

We need to copy file descriptors when we create a new process: The following chunk completes the TCB initialization in the u_fork$_{209c}$ function.

⟨u_fork: *copy the file descriptors* 425a⟩≡ (210b) [425a]

```
  if (t_new->pid != t_old->pid) {
    for (int pfd = 0;  pfd < MAX_PFD;  pfd++) {
      int gfd = t_old->files[pfd];
      if (gfd ≥ 0)
        t_new->files[pfd] = u_reopen (gfd);  // get new gfd
      else
        t_new->files[pfd] = gfd;             // use old gfd (stdio)
    }
  }
```

Uses MAX_PFD 424b, t_new 276c, t_old 276c, and u_reopen.

The function u_reopen creates a copy of a file descriptor. We cannot simply give the newly forked process access to the same (globally visible) file descriptor because the current read/write position in the file is associated with that descriptor. If one of the two processes changes that position, the modification must not be visible in the other process.

⟨*function prototypes* 45a⟩+≡ (44a) ◁424a 432d▷ [425b]

```
  int u_reopen (int fd);
```

⟨*function implementations* 100b⟩+≡ (44a) ◁424e 432e▷ [425c]

```
  int u_reopen (int fd) {
    ⟨VFS functions: turn fd into (fs, localfd) pair or fail 414c⟩
    switch (fs) {
      case FS_MINIX: return (fs << 8) + mx_reopen (localfd);
      case FS_FAT:   return -1;                  // not implemented
      case FS_DEV:   return -1;
      case FS_ERROR: return -1;                  // error
      default:       return -1;
    }
  }
```

Uses FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_reopen 468b, and u_reopen.

As usual, the real work is done my mx_reopen$_{468b}$. (There is no such function for the device filesystem.)

We can now implement the system calls:

[426a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁416a 433a▷

```
void syscall_open      (context_t *r);
void syscall_stat      (context_t *r);
void syscall_close     (context_t *r);
void syscall_read      (context_t *r);
void syscall_write     (context_t *r);
void syscall_lseek     (context_t *r);
void syscall_isatty    (context_t *r);
void syscall_mkdir     (context_t *r);
void syscall_rmdir     (context_t *r);
void syscall_getdent   (context_t *r);
void syscall_truncate  (context_t *r);
void syscall_ftruncate (context_t *r);
void syscall_link      (context_t *r);
void syscall_unlink    (context_t *r);
void syscall_symlink   (context_t *r);
void syscall_readlink  (context_t *r);
```

For `syscall_getdent` we will use the `__NR_readdir`$_{204c}$ syscall number but it should be noted that `readdir` accesses directory entries differently (and Ulix does not implement `readdir`).

[426b] ⟨*syscall functions* 174b⟩+≡ (202b) ◁416b 433b▷

```
void syscall_open (context_t *r) {
  eax_return ( gfd2pfd (u_open ((char*) r->ebx, r->ecx, 0) ) ); };

void syscall_stat (context_t *r) {
  eax_return ( u_stat ((char*) r->ebx, (struct stat*) r->ecx) ); };

void syscall_getdent (context_t *r) {
  // ebx: path, ecx: index, edx: dir_entry buffer
  eax_return ( u_getdent ((char*) r->ebx, r->ecx, (struct dir_entry*) r->edx) ); };

void syscall_close (context_t *r) {
  // ebx: fd
  int pfd = r->ebx;
  thread_id pid = thread_table[current_task].pid;
  r->eax = u_close (pfd2gfd (pfd));        // close (globally)
  if (pfd ≥ 0 && pfd < MAX_PFD)
    thread_table[pid].files[pfd] = -1; };  // close (locally)

void syscall_read (context_t *r) {
  // ebx: fd, ecx: *buf, edx: nbytes
  eax_return ( u_read (pfd2gfd (r->ebx), (byte*) r->ecx, r->edx) ); };

void syscall_write (context_t *r) {
  // ebx: fd, ecx: *buf, edx: nbytes
  eax_return ( u_write (pfd2gfd (r->ebx), (byte*) r->ecx, r->edx) ); };
```

```
void syscall_lseek (context_t *r) {
  // ebx: fd, ecx: offset, edx: whence
  eax_return ( u_lseek (pfd2gfd (r->ebx), r->ecx, r->edx) ); };

void syscall_isatty (context_t *r) {
  // ebx: file descriptor
  eax_return ( pfd2gfd (u_isatty (r->ebx)) ); }

void syscall_mkdir (context_t *r) {
  // ebx: name of new directory, ecx: mode
  eax_return ( u_mkdir ((char*)r->ebx, r->ecx) ); }

void syscall_rmdir (context_t *r) {
  // ebx: name of directory that we want to delete
  eax_return ( u_rmdir ((char*)r->ebx) ); }

void syscall_truncate  (context_t *r) {
  // ebx: filename, ecx: length
  eax_return ( u_truncate ((char*)r->ebx, r->ecx) ); }

void syscall_ftruncate (context_t *r) {
  // ebx: file descriptor, ecx: length
  eax_return ( u_ftruncate ( pfd2gfd (r->ebx), r->ecx) ); }

void syscall_link (context_t *r) {
  // ebx: original name, ecx: new name
  eax_return ( u_link ((char*)r->ebx, (char*)r->ecx) ); }

void syscall_unlink (context_t *r) {
  // ebx: pathname
  eax_return ( u_unlink ((char*)r->ebx) );
}

void syscall_symlink (context_t *r) {
  // ebx: target file name, ecx: symbolic link name
  eax_return ( u_symlink ((char*)r->ebx, (char*)r->ecx) ); }

void syscall_readlink (context_t *r) {
  // ebx: file name
  // ecx: buffer for link target
  // edx: buffer length
  eax_return ( u_readlink ((char*)r->ebx, (char*)r->ecx, r->edx) );
}
```

Defines:
syscall_close, used in chunk 428a.
syscall_ftruncate, used in chunk 428a.
syscall_isatty, used in chunk 428a.
syscall_link, used in chunk 428a.
syscall_lseek, used in chunk 428a.
syscall_mkdir, used in chunk 428a.

syscall_open, used in chunk 428a.
syscall_read, used in chunk 428a.
syscall_readlink, used in chunk 428a.
syscall_rmdir, used in chunk 428a.
syscall_symlink, used in chunk 428a.
syscall_truncate, used in chunk 428a.
syscall_unlink, used in chunk 428a.
syscall_write, used in chunks 426a and 428a.

Uses context_t 142a, current_task 192c, dir_entry 490b, eax_return 174a, gfd2pfd, MAX_PFD 424b, pfd2gfd, stat 429b 489b, syscall_getdent, syscall_stat, thread_id 178a, thread_table 176b, u_close 418a, u_ftruncate 420b, u_getdent, u_link 419a, u_lseek 418a, u_mkdir 422a, u_open 412c, u_read 414b, u_readlink 420c, u_rmdir 422a, u_stat 421d, u_symlink 419b, u_truncate 420a, u_unlink 418b, and u_write 415a.

As a last step we create syscall table entries for the new system call handlers:

[428a] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁416c 434a▷

```
install_syscall_handler (__NR_open,      syscall_open);
install_syscall_handler (__NR_stat,      syscall_stat);
install_syscall_handler (__NR_close,     syscall_close);
install_syscall_handler (__NR_read,      syscall_read);
install_syscall_handler (__NR_write,     syscall_write);
install_syscall_handler (__NR_lseek,     syscall_lseek);
install_syscall_handler (__NR_isatty,    syscall_isatty);
install_syscall_handler (__NR_mkdir,     syscall_mkdir);
install_syscall_handler (__NR_rmdir,     syscall_rmdir);
install_syscall_handler (__NR_readdir,   syscall_getdent);
install_syscall_handler (__NR_truncate,  syscall_truncate);
install_syscall_handler (__NR_ftruncate, syscall_ftruncate);
install_syscall_handler (__NR_link,      syscall_link);
install_syscall_handler (__NR_unlink,    syscall_unlink);
install_syscall_handler (__NR_symlink,   syscall_symlink);
install_syscall_handler (__NR_readlink,  syscall_readlink);
```

Uses __NR_close 204c, __NR_ftruncate, __NR_isatty 428b, __NR_link 204c, __NR_lseek 204c, __NR_mkdir 204c, __NR_open 204c, __NR_read 204c, __NR_readdir 204c, __NR_readlink 204c, __NR_rmdir 204c, __NR_stat 204c, __NR_symlink 204c, __NR_truncate 204c, __NR_unlink 204c, __NR_write 204c, install_syscall_handler 201b, syscall_close 426b, syscall_ftruncate 426b, syscall_getdent, syscall_isatty 426b, syscall_link 426b, syscall_lseek 426b, syscall_mkdir 426b, syscall_open 426b, syscall_read 426b, syscall_readlink 426b, syscall_rmdir 426b, syscall_stat, syscall_symlink 426b, syscall_truncate 426b, syscall_unlink 426b, and syscall_write 426b.

We need a system call number for $\texttt{syscall\_isatty}_{426b}$:

[428b] ⟨*ulix system calls* 206e⟩+≡ (205a) ◁415d 493c▷

```
#define __NR_isatty  521
```

Defines:
__NR_isatty, used in chunks 428a and 429b.

12.4.1 Library Functions

Here we define the user mode library functions for our collection of new system calls:

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁373d 430▷ [429a]

```
int open       (const char *path, int oflag, ...);
int stat       (const char *path, struct stat *buf);
int close      (int fildes);
int read       (int fildes, void *buf, size_t nbyte);
int write      (int fildes, const void *buf, size_t nbyte);
int lseek      (int fildes, int offset, int whence);
boolean isatty (int fd);
int mkdir      (const char *path, int mode);
int rmdir      (const char *path);
int getdent    (const char *path, int index, struct dir_entry *buf);
int ftruncate  (int fd, int length);
int truncate   (const char *path, int length);
int link       (const char *path1, const char *path2);
int unlink     (const char *path);
int symlink    (const char *path1, const char *path2);
int readlink   (char *path, char *buf, int bufsize);
```

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁373e 431▷ [429b]

```
int open (const char *path, int oflag, ...) {
  return syscall3 (__NR_open, (uint)path, oflag); }

int stat (const char *path, struct stat *buf) {
  return syscall3 (__NR_stat, (uint)path, (uint)buf); }

int close (int fildes) { return syscall2 (__NR_close, fildes); }

int read (int fd, void *buf, size_t nbyte) {
  return syscall4 (__NR_read, fd, (uint)buf, nbyte); }

int write (int fd, const void *buf, size_t nbyte) {
  return syscall4 (__NR_write, fd, (uint)buf, nbyte); }

int lseek (int fildes, int offset, int whence) {
  return syscall4 (__NR_lseek, fildes, offset, whence); }

boolean isatty (int fd) { return syscall2 (__NR_isatty, fd); }

int mkdir (const char *path, int mode) {
  return syscall3 (__NR_mkdir, (uint)path, mode); }

int rmdir (const char *path) {
  return syscall2 (__NR_rmdir, (uint)path); }

int getdent (const char *path, int index, struct dir_entry *buf) {
  return syscall4 (__NR_readdir, (uint)path, index, (uint)buf); }
```

```
int ftruncate (int fd, int length) {
  return syscall3 (__NR_ftruncate, fd, length); }

int truncate (const char *path, int length) {
  return syscall3 (__NR_truncate, (uint)path, length); }

int link (const char *path1, const char *path2) {
  return syscall3 (__NR_link, (uint) path1, (uint) path2); }

int unlink (const char *path) {
  return syscall2 (__NR_unlink, (unsigned int) path); }

int symlink (const char *path1, const char *path2) {
  return syscall3 (__NR_symlink, (uint) path1, (uint) path2); }

int readlink (char *path, char *buf, int bufsize) {
  return syscall4 (__NR_readlink, (uint)path, (uint)buf, bufsize); }
```

Defines:
close, used in chunks 467b and 585b.
lseek, used in chunk 498a.
mkdir, used in chunk 618.
open, used in chunks 411a, 414c, 467b, 475a, and 585b.
read, used in chunks 294, 431, 432b, 456, 475a, 477b, 490a, 503, 543a, 552c, and 585b.
stat, used in chunks 420c, 421d, 426b, 432e, 489, 490, 499, 576, 577c, and 608a.
write, used in chunks 35b, 123b, 170c, 199, 204, 213b, 429a, 431, 460b, 475a, 525a, 528a, 539c, 543a, 575, 598c, and 624.

Uses __NR_close 204c, __NR_ftruncate, __NR_isatty 428b, __NR_link 204c, __NR_lseek 204c, __NR_mkdir 204c, __NR_open 204c, __NR_read 204c, __NR_readdir 204c, __NR_readlink 204c, __NR_rmdir 204c, __NR_stat 204c, __NR_symlink 204c, __NR_truncate 204c, __NR_unlink 204c, __NR_write 204c, dir_entry 490b, size_t 46b, syscall2 203c, syscall3 203c, and syscall4 203b.

12.4.2 Reading from Standard Input

The functions

[430] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁429a 434b▷

```
int  ureadline (char *s, int maxlength, boolean echo);
byte ureadchar ();
```

use read$_{429b}$ with the standard input file descriptor STDIN_FILENO$_{415b}$ to read one or more characters. Writing to standard output will be handled by ulixlib_printchar$_{598c}$ which just write$_{429b}$s to the standard output (via file descriptor STDOUT_FILENO$_{415b}$) and is implemented where we discuss the printf$_{601a}$ function.

ureadline$_{431}$ takes three arguments: a buffer, a maximum length and an echo flag. If the length parameter is negative then pressing [Enter] to complete the input will not cause the newline character to be displayed. If echo is not set, output will be disabled completely which is useful for password queries: The /bin/login and /bin/su programs use that feature.

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁429b 432b▷ [431]

```
int ureadline (char *s, int maxlength, boolean echo) {
  // if maxlength is negative, dont print \n at the end
  char print_newline = 1;
  if (maxlength < 0) {
    print_newline = 0;
    maxlength = -maxlength;
  }
  int pos=0;
  for (;;) {
    startlabel:
    if (pos < 0) { printf ("ERROR: pos < 0\n"); return; }
    byte c = 0;
    int nbytes = read (STDIN_FILENO, &c, 1);          // read one char. from stdin
    if (nbytes == 0) return -1;

    if (c == 0 || c == 27 || c > 190)                 // Esc, cursor and other keys
      goto startlabel;

    if (c == 3) {                                     // Strg-C, kill command
      pos = 0; s[0] = 0;
      if (echo) printf ("\n");
      return 0;
    }

    if (c == 4 && pos == 0) {                         // Strg-D in first column
      strncpy (s, "ex" "it", 5);
      if (echo) printf ("ex" "it\n");
      return 0;
    }

    if ((c == 0x08) && (pos>0)) {                     // backspace
      pos--;
      if (echo) write (STDOUT_FILENO, "\010 \010", 3);
    } else if ( c == '\n' ) {                         // newline, end of input
      if ((print_newline == 1) && echo) write (STDOUT_FILENO, "\n", 1);
      s[pos] = '\0';
      return 0;
    } else if ( (c != 0x08) && (pos < maxlength) ) { // other character
      if (echo) write (STDOUT_FILENO, &c, 1);
      s[pos++] = c;
    };
  };
};
```

Defines:
`ureadline`, used in chunks 214, 430, 432a, and 586b.
Uses `kill` 568b, `print` 600, `printf` 601a, `read` 429b, `STDIN_FILENO` 415b, `STDOUT_FILENO` 415b, `strncpy` 594b, and `write` 429b.

gets

Since `gets`$_{432a}$ is a traditional Unix function for reading from standard input, we provide it as a macro that uses `ureadline`$_{431}$ with a maximum string length of 9999 characters. Note that using `gets`$_{432a}$ is deprecated since an application cannot control the length of the input which is likely to cause problems when the reserved buffer overflows due too overly long input.

[432a] ⟨*ulixlib macro definitions* 432a⟩≡ (48a)
```
#define gets(s) ((ureadline(s,9999,true)), s)
```
Uses ureadline 431.

The `ureadchar`$_{432b}$ function reads just one single character. It is used by the `/bin/vi`, `/bin/keys` and `/bin/hexdump` programs:

[432b] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁431 434c▷
```
byte ureadchar () {
  byte b;
  read (STDIN_FILENO, &b, 1);
  return b;
};
```
Uses read 429b and STDIN_FILENO 415b.

12.4.3 Working Directory, Relative Paths

We want processes to have a "current working directory", so we add an entry to the thread control block structure:

[432c] ⟨*more TCB entries* 158c⟩+≡ (175) ◁424c 560b▷
```
char cwd[256];
```

To query and set this value, we will need two functions `u_getcwd`$_{432e}$ and `u_chdir`$_{432e}$ which can also be accessed by user mode functions `getcwd`$_{434c}$ and `chdir`$_{434c}$ via system calls.

Now `u_getcwd`$_{432e}$ just copies a string, while `chdir`$_{434c}$ needs to check whether the argument is a valid directory:

[432d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁425b 443a▷
```
char *u_getcwd (char *buf, int size);
int u_chdir (const char *path);
```
Uses u_chdir 432e and u_getcwd 432e.

[432e] ⟨*function implementations* 100b⟩+≡ (44a) ◁425c 440b▷
```
char *u_getcwd (char *buf, int size) {
  strncpy (buf, thread_table[current_task].cwd, size);
  return buf;
}

int u_chdir (const char *path) {
  char abspath[256], dir[256], base[256], localpath[256];
  if (strequal (path, "..")) {        // special case ".."
    if (strequal (thread_table[current_task].cwd, "/"))
```

```
      return 0;  // already at root directory
    // change to ..
    strncpy (abspath, thread_table[current_task].cwd, 256);
    splitpath (abspath, dir, base);
    strncpy (thread_table[current_task].cwd, dir, 256);
    return 0;
  }

  // check relative/absolute path
  if (*path != '/')  relpath_to_abspath (path, abspath);
  else               strncpy (abspath, path, 256);

  // check if abspath is directory
  struct stat st;
  u_stat (abspath, &st);
  if ((st.st_mode & S_IFDIR) == S_IFDIR) {
    strncpy (thread_table[current_task].cwd, abspath, 256);
    return 0;
  } else {
    return -1;  // error
  }
}
```

Defines:
u_chdir, used in chunks 432d, 433b, 488a, and 582a.
u_getcwd, used in chunks 432d, 433b, and 488a.
Uses current_task 192c, cwd, relpath_to_abspath 412b, S_IFDIR 457c, splitpath 455a, stat 429b 489b, strequal 596a, strncpy 594b, thread_table 176b, and u_stat 421d.

As usual, we define and register system call functions ...

⟨*syscall prototypes* 173b⟩+≡ (202a) ◁426a 493a▷ [433a]

```
void syscall_getcwd (context_t *r);
void syscall_chdir (context_t *r);
```

⟨*syscall functions* 174b⟩+≡ (202b) ◁426b 493b▷ [433b]

```
void syscall_getcwd (context_t *r) {
  // ebx: buffer for directory
  // ecx: maximum length of path
  eax_return ( u_getcwd ((char*)r->ebx, r->ecx) );
}

void syscall_chdir (context_t *r) {
  // ebx: new directory
  eax_return ( u_chdir ((char*)r->ebx) );
}
```

Defines:
syscall_chdir, used in chunk 434a.
syscall_getcwd, used in chunks 433a and 434a.
Uses context_t 142a, eax_return 174a, u_chdir 432e, and u_getcwd 432e.

[434a] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁428a 493d▷
```
install_syscall_handler (__NR_getcwd, syscall_getcwd);
install_syscall_handler (__NR_chdir, syscall_chdir);
```
Uses `__NR_chdir` 204c, `__NR_getcwd` 204c, `install_syscall_handler` 201b, `syscall_chdir` 433b, and `syscall_getcwd` 433b.

...and provide user mode library functions:

[434b] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁430 493e▷
```
char *getcwd (char *buf, int size);
int chdir (const char *path);
```

[434c] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁432b 493f▷
```
char *getcwd (char *buf, int size) {
  syscall3 (__NR_getcwd, (unsigned int) buf, size);
}

int chdir (const char *path) {
  return syscall2 (__NR_chdir, (unsigned int) path);
}
```
Defines:
`getcwd`, used in chunk 434b.
Uses `__NR_chdir` 204c, `__NR_getcwd` 204c, `syscall2` 203c, and `syscall3` 203c.

12.5 The Minix Filesystem

We have chosen to use the Minix [Tan87] filesystem as the native filesystem for ULIX for two reasons:

- While Minix has all the properties of a Unix filesystem, it is very simple. That also means that explaining and implementing its function is fit for an introductory book. There is no redundancy (for example, Minix does not create backup copies of the superblock, like other Unix filesystems do), so the code does not become complex.
- Minix was the first filesystem that Linux used, and it is still supported. Thus, on a Linux machine the commands `mkfs.minix` and `fsck.minix` are available for creating and checking Minix volumes. The last tool is especially helpful because it allowed us to check the correctness of our implementation: Whenever earlier revisions of the ULIX code wrote wrong data to the volume, `fsck.minix` detected that.

Conceptually, the Minix filesystem uses data structures which can also be found in all other Unix filesystems:

Superblock: A *superblock* contains general information about the whole filesystem. For example it tells how large the volume is. In the Minix case it also lists the maximum numbers of data blocks and inodes as well as where the first data block starts (after the metadata).

Inode: For every file on the volume there is an *inode* (index node) that describes the file. You can find the file size, owner and group IDs, access permissions, timestamps and some pointer to the data blocks in an inode. How these data are organized and how exactly the data blocks can be found after inspecting the inode depends on the specific filesystem.

Directory: In order to have a hierarchic filesystem, *directories* are used. In all Unix filesystems a directory is a simple file that maps filenames to inode numbers. The version of the Minix filesystem that we will look at uses 32 bytes for each directory entry: 30 bytes for the file name and two bytes for a 16-bit inode number. Note again: Directories are files, too (if of a special kind), so when we create a new directory we also need a new inode that describes this directory (file). A freshly created filesystem already contains the root directory.

Bitmaps: Inodes are reserved on the volume when it is created. While it would be possible to scan the inode table for a free inode, this would take too long on larger filesystems. So there is also a *bitmap* that holds a bit for each inode that indicates if it is free or not. Similarly there is a second bitmap which describes the free/used status of the data blocks.

We will first look at the Minix filesystem by creating one on a Linux machine and trying to understand some of its properties. For that purpose we format a floppy image with the Minix filesystem. We create a 1440 KByte image file with `dd`:

[436a] ⟨*create 1.4 MB disk file* 436a⟩≡

```
$ dd if=/dev/zero of=minixfs.img bs=1k count=1440
1440+0 records in
1440+0 records out
1474560 bytes (1,5 MB) copied, 0.219079 s, 6.7 MB/s
```

and then format it with `mkfs.minix`:

[436b] ⟨*format the disk image with minix fs* 436b⟩≡

```
$ /sbin/mkfs.minix -2 minixfs.img
480 inodes
1440 blocks
Firstdatazone=34 (34)
Zonesize=1024
Maxsize=2147483647
```

The option `-2` creates a version 2 filesystem with "long" filenames (up to 30 characters; the option `-n 14` would enable "short" filenames that have only up to 14 characters). As a next step, `hexdump` will show that there is not much data on a freshly formatted Minix filesystem. (We removed lines that displayed only `0x00` bytes from the output.)

[436c] ⟨*look at the image with hexdump* 436c⟩≡

```
$ hexdump -C minixfs.img
00000400  e0 01 00 00 01 00 01 00  22 00 00 00 ff ff ff 7f  |........".......|
00000410  78 24 01 00 a0 05 00 00  00 00 00 00 00 00 00 00  |x$..............|
00000800  03 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000830  00 00 00 00 00 00 00 00  00 00 00 00 fe ff ff ff  |................|
00000840  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000c00  03 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000ca0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 80  |................|
00000cb0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00001000  ed 41 02 00 e8 03 e8 03  40 00 00 00 66 89 eb 53  |.A......@...f..S|
00001010  66 89 eb 53 66 89 eb 53  22 00 00 00 00 00 00 00  |f..Sf..S".......|
00008800  01 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00008820  01 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00008830  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00008840  00 00 2e 62 61 64 62 6c  6f 63 6b 73 00 00 00 00  |...badblocks....|
```

We cannot interpret the output without knowledge of the internal data structures; we will explain them in the next section where we present the implementation.

Next we ask `fsck.minix` to display as much information as it can. The `file` command also recognizes the file type:

[436d] ⟨*fsck on an empty minix filesystem* 436d⟩≡

```
$ /sbin/fsck.minix -sfv minixfs.img
Forcing filesystem check on minixfs.img
480 inodes
1440 blocks
Firstdatazone=34 (34)
Zonesize=1024
Maxsize=2147483647
Filesystem state=1
namelen=30
```

```
     1 inodes used (0%)
    35 zones used (2%)

     0 regular files
     1 directories
     0 character device files
     0 block device files
     0 links
     0 symbolic links
------
     1 files

$ file minixfs.img
minixfs.img: Minix filesystem, V2,
30 char names, 0 zones
```

Now we want to see what happens when we write a file onto that filesystem. For that purpose we mount the image and then create a file. We then print a new hexdump; the output only shows the changed or new lines:

⟨*write file to disk image* 437⟩≡ [437]

```
$ sudo mount -o loop minixfs.img /mnt
$ sudo echo "Hello World" > /mnt/hello.txt
$ sudo umount /mnt
$ hexdump -C minixfs.img      # only changed lines are shown
00000800  07 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000c00  07 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001000  ed 41 02 00 e8 03 e8 03  60 00 00 00 cf 8b eb 53  |.A......`......S|
00001010  d8 8b eb 53 d8 8b eb 53  22 00 00 00 00 00 00 00  |...S...S".......|
00001040  a4 81 01 00 e8 03 e8 03  0c 00 00 00 d8 8b eb 53  |...............S|
00001050  d8 8b eb 53 d8 8b eb 53  23 00 00 00 00 00 00 00  |...S...S#.......|
00008840  02 00 68 65 6c 6c 6f 2e  74 78 74 00 00 00 00 00  |..hello.txt.....|
00008c00  48 65 6c 6c 6f 20 57 6f  72 6c 64 0a 00 00 00 00  |Hello World.....|
$ /sbin/fsck.minix -sfv minixfs.img
[...]
     2 inodes used (0%)
    36 zones used (2%)

     1 regular files
     1 directories
[...]
```

So what happened here? First of all, `fsck.minix` tells us that one more inode and one more *zone* are used and that we have one regular file. Zones are blocks (of size 1 KByte); the original Minix filesystem implementation (on Minix) allows zones to have a larger size, but the Linux `mkfs.minix` tool cannot create such volumes. So from now on, whenever you read "zone", think "block". zone

As you can see, the filename `hello.txt` and the file contents `Hello World` show up in the new hexdump, and some of the other locations have new values, for example at addresses `0x800` and `0xc00` the bytes have changed from `0x03` to `0x07`. In binary these numbers are $\texttt{00000011}_b$ and $\texttt{00000111}_b$: One bit was flipped from 0 to 1 in both locations. We will soon see that these newly set bits refer to a new inode and a new zone—which makes sense since we created a new file which is so small that it fits in one block.

We perform another test (with a freshly `dd`'ed and `mkfs.minix`-formatted filesystem) and populate it with more than one file:

- We copy the file `testfile1.txt` (6144 bytes, hex.: 0x1800, exactly six blocks) onto the filesystem.
- With `sed -e 's/file1/file2' < testfile1.txt > testfile2.txt` we create a slightly modified copy (`testfile2.txt`),
- and then use `ln` to create a hard link (`Hardlink.txt`)
- and `ln -s` to create a symbolic link `Symlink.txt` (of `testfile1.txt`).

When listing the filesystem's root directory with `ls`, we get:

```
$ ls -il
2 -rw-r--r-- 2 root root 6144 2014-06-04 23:32 Hardlink.txt
4 lrwxrwxrwx 1 root root   14 2014-06-04 23:33 Symlink.txt -> testfile1.txt
2 -rw-r--r-- 2 root root 6144 2014-06-04 23:32 testfile1.txt
3 -rw-r--r-- 1 root root 6144 2014-06-04 23:32 testfile2.txt
$ ls -ild /mnt
1 drwxr-xr-x 2 root root  192 2012-06-04 23:33 /mnt
```

Every inode has an internal number. The output shows that the inodes with numbers 1–4 are in use (see first column of the `ls` output). Looking at the image again with $\texttt{hexdump}_{612c}$ would reveal the root directory's table of contents and the contents of the two files.

The following blocks are in use:

- The root directory `/` uses block 34.
- The file `testfile1.txt` uses blocks 35–40.
- The file `testfile2.txt` uses blocks 41–46.
- The symbolic link `Symlink.txt` uses block 47. (Symbolic links need data blocks as well!)
- No further blocks are in use, the hard link is just a further entry in the root directory.

inode bitmap

The *inode bitmap* starts at position `0x800` in the image file and has the following contents which we display with our `bindump` tool (see p. 630) that works like `hexdump` but displays bytes as binary numbers:

```
# bindump -r < minixfs.img
[...]
00000800  11111000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000808  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000810  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
[...]
```

zone bitmap

Those five ones represent inode numbers 0–4—however, there is no inode 0. This is what the *zone bitmap* (that starts at offset `0xc00` in the image) looks like:

```
# bindump -r < minixfs.img
[...]
00000c00  11111111 11111110 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000c08  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000c10  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
[...]
```

The 15 set bits refer to block numbers 33–47. Block 33 is used by the inodes and is not a data block! So this represents 14 used data blocks (which fits what we told you above: The 14 blocks 34–46 are in use.)

In comparison, when looking at a freshly created (empty) Minix filesystem (with no files and an empty root directory) the inode and zone bitmaps look like this:

```
# bindump -r < minix-empty.img
[...]
00000800  11000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000808  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
[...]
00000c00  11000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000c08  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
[...]
```

Two inodes (with numbers 0, 1) and two blocks (numbers 33, 34) are marked as used. Inode 0 does not exit, and inode 1 stores information about the root directory. A directory is never empty, since it always contains `.` and `..` entries.

It is now time to properly introduce the data structures that Minix filesystems use; they will shed light on the hexdump we've shown earlier.

12.6 The Ulix Implementation of the Minix Filesystem

There are five variants of the Minix filesystem which differ in the sizes of inodes and block numbers (leading to different maximum file sizes) and the maximum length of filenames (14, 30 or 60). For any Minix filesystem image you can find out its version. Our implementation allows access to version 2 of the filesystem with a filename length of up to 30 characters. The Linux tool `mkfs.minix` will create a Minix version 1 filesystem (with 30-character filenames and a theoretical maximum file size of 256 MByte) by default, but this can be changed by supplying the options `-v` or `-2` (for version 2 with an approximate maximum filesize of 2 GByte) or `-3` (for version 3). For version 1 and 2, the filename length can be set to 14 with the `-n 14` option which reduces the size of a directory entry from 32 bytes to 16 bytes, whereas version 3 only supports a filename length of 60 characters. This leads to the characteristics shown in Table 12.3.

The standard zone size of a Minix filesystem is 1 KByte (1024 bytes). It is possible to increase this size to 1024×2^n bytes (for some n), but not with the Linux tool `mkfs.minix`. The first block in the filesystem contains the boot sector (which we will ignore), the second block is the *superblock* which contains the setup information of a specific filesystem, including two magic bytes which tell the filesystem version number.

superblock

Minix version	inode number size	directory entry size	entries per block
Version 1 (filenames: 14)	2 bytes	16 bytes	64
Version 1 (filenames: 30)	2 bytes	32 bytes	32
Version 2 (filenames: 14)	2 bytes	16 bytes	64
Version 2 (filenames: 30)	2 bytes	32 bytes	32
Version 3 (filenames: 60)	4 bytes	64 bytes	16

Table 12.3: Characteristics of the Minix filesystem versions.

Since the Minix filesystem is the standard filesystem for ULIX, we will always work with KByte-sized blocks—even in functions that work on a lower level. We define:

[440a] ⟨*constants* 112a⟩+≡ (44a) ◁415c 444a▷
```
#define BLOCK_SIZE 1024
```
Defines:
BLOCK_SIZE, used in chunks 451a, 453b, 471–73, 475, 476b, 480c, 484e, 496d, 497, 508–10, 515a, 518b, 521a, and 582a.

12.6.1 The Minix Superblock

Every Unix filesystem has a superblock: that is a block which stores global information about a specific volume, and it must always be visited upon first interaction with a volume. It is created when the volume is formatted. In the case of Minix its contents are immutable; they remain the same over the lifetime of that volume.

We start our Minix filesystem implementation with a look at the superblock. All functions will appear inside the ⟨*minix filesystem implementation* 420c⟩ code chunk:

[440b] ⟨*function implementations* 100b⟩+≡ (44a) ◁432e 495c▷
⟨*minix filesystem implementation* 420c⟩

The superblock (i. e., the absolute bytes 1024–2047 of the image file) contains only the following few entries, with the rest of the block being ignored:

[440c] ⟨*type definitions* 91⟩+≡ (44a) ◁405a 442a▷
```
struct minix_superblock {
  uint16_t s_ninodes;        uint16_t s_nzones;
  uint16_t s_imap_blocks;    uint16_t s_zmap_blocks;
  uint16_t s_firstdatazone;  uint16_t s_log_zone_size;
  uint32_t s_max_size;       uint16_t s_magic;
  uint16_t s_state;          uint32_t s_zones;
};
```
Defines:
minix_superblock, used in chunks 443b, 448, and 492.

The uint16_t and uint32_t types are defined in /usr/include/stdint.h (on a Linux system). They are 16 bit and 32 bit wide unsigned integers, respectively. Thus, the superblock only uses 24 bytes.

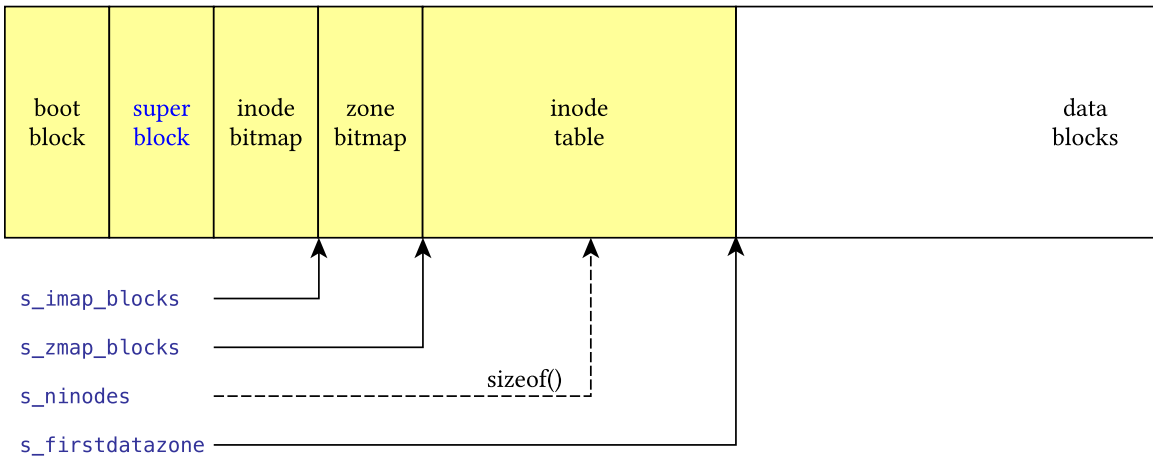


Figure 12.10: A Minix filesystem consists of boot block, superblock, inode and zone bitmaps, an inode table and the data blocks.

When you copy the superblock into a `struct minix_superblock`$_{440c}$ variable, you can check (or print) the values; an analysis of these values is a first step towards properly accessing the filesystem. The `s_magic` field tells what version of the Minix filesystem was used when the medium was formatted. There are five possible cases:

- Minix v1 (14 characters per filename): 0x137F
- Minix v1 (30 characters per filename): 0x138F
- Minix v2 (14 characters per filename): 0x2468
- Minix v2 (30 characters per filename): 0x2478
- Minix v3 (60 characters per filename): 0x4D5A (but stored elsewhere since the v3 superblock has a different layout)

Without checking the version it is impossible to access the filesystem since the versions differ in size and content of the inodes and the directory entries. A version 1 superblock stores the number of blocks in the `s_nzones` entry, whereas a version 2 superblock uses the `s_zones` entry. The unused value is set to 0.

zone

As mentioned before, instead of blocks, Minix uses "zones" as the smallest allocatable unit. Typically a zone contains just one block, but theoretically a zone may consist of a collection of blocks if `s_log_zone_size` is non-zero; the following formula expresses the relationship between zone size and block size:

$$\text{zone size} = \text{block size} \times 2^{\texttt{s\_log\_zone\_size}}$$

(With a default setting of `s_log_zone_size = 0` that formula reads: zone size = block size, since $2^0 = 1$.) In our implementation we only support the case where block size = zone

size = 1024, and so we will often use the terms *zone* and *block* interchangeably. Some filesystems use the name *cluster* to discuss smallest allocatable units, thus a zone is also a cluster.

cluster

The `s_imap_blocks` and `s_zmap_blocks` fields note how many blocks are reserved for the inode bitmap and the zone bitmap. Those bitmaps store a single bit for each inode or data block, respectively, and the bits show whether an inode/a data block is free (0) or occupied (1). `s_ninodes` is the number of inodes from which we can calculate the size of the inode table.

The inode bitmap and the zone bitmap follow directly after the superblock (see Figure 12.10). After that the filesystem contains the inode table and finally the data blocks.

The zone bitmap starts with a 1 bit (which does not represent any zone); the second bit (bit 1) refers to the first data zone that contains (the start of) the filesystem's root directory. Blocks before the data blocks area are not available for data storage, thus they are not represented in the zone bitmap.

An inode of a Minix (version 2) filesystem has the following form:

[442a] ⟨*type definitions* 91⟩+≡ (44a) ◁440c 452b▷

```
struct minix2_inode { ⟨external minix2 inode 442b⟩ };
```

Defines:
`minix2_inode`, used in chunks 451–53, 456, 457b, 461d, 466–68, 475c, 478–80, 484c, 487a, 488a, 490a, 589d, 607a, and 610d.

[442b] ⟨*external minix2 inode* 442b⟩≡ (442a 459a)

```
uint16_t i_mode;          uint16_t i_nlinks;
uint16_t i_uid;           uint16_t i_gid;
uint32_t i_size;          uint32_t i_atime;
uint32_t i_mtime;         uint32_t i_ctime;
uint32_t i_zone[10];
```

We're placing the $\texttt{minix2\_inode}_{442a}$ entries in a separate code chunk since we will later define another inode data structure for the in-memory management of open files; there we will also need these fields.

Thus, we can calculate the size of an inode: it uses $\texttt{sizeof(struct minix2\_inode}_{442a}\texttt{)} = 24 + 4 \times 10 = 64$ bytes, which lets $1024/64 = 16$ inodes fit inside one inode table block. When we look at a 1.44 MByte floppy disk that was formatted with the Minix version 2 filesystem (using `mkfs.minix -v`), we find that the superblock contains the following values:

[442c] ⟨*example minix super block* 442c⟩≡

```
s_ninodes:       480
s_nzones:        0
s_imap_blocks:   1
s_zmap_blocks:   1
s_firstdatazone: 34
s_log_zone_size: 0
s_max_size:      2147483647
s_magic:         9336
s_state:         1
s_zones:         1440
```

That tells us:

- There are 480 inodes. With 16 inodes per block the inode table requires $480/16 = 30$ blocks.
- The inode bitmap consists of only 480 bits (= 60 bytes) and fits in one block, with the rest of the block remaining unused. During filesystem creation the unused bits are filled with 1s.
- 1440 blocks require 1440 bits (= 180 bytes) for the zone bitmap which (again) fit in one block. This bitmap also fills the unused bits with 1s.

This leads to the layout shown in Table 12.4.

Blocks	Usage	Absolute Bytes	Absolute Bytes (hex)
0	unused (boot sector)	0–1023	`0x0000` – `0x03ff`
1	superblock	1024–2047	`0x0400` – `0x07ff`
2	inode bitmap	2048–3071	`0x0800` – `0x0bff`
3	zone bitmap	3072–4095	`0x0c00` – `0x0fff`
4–33	inode table (30 blocks)	4096–34815	`0x1000` – `0x87ff`
34–1439	data blocks (zones)	34816–…	`0x8800` –…

Table 12.4: The layout of a Minix version 2 floppy disk formatted with `mkfs.minix -v`.

From calculating the sizes of the bitmaps and the inode table we know that the first data block is block 34, but this information is also stored (redundantly) in the superblock's `s_firstdatazone` field.

We provide a function which extracts specific values from the superblock, e. g, the number of inodes. Since we do not want to write several similar functions we combine this in one function called `mx_query_superblock`$_{443b}$ that expects a device identifier and a constant which refers to some specific property:

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁ 432d 447b ▷ [443a]

```
int mx_query_superblock (int device, char index);
```

⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁ 420c 444b ▷ [443b]

```
int mx_query_superblock (int device, char index) {
  byte block[1024];
  struct minix_superblock *sblock;
  readblock (device, 1, (byte*)block);                    // superblock = block 1
  sblock = (struct minix_superblock*) &block;
  switch (index) {
    case MX_SB_NINODES:       return sblock->s_ninodes;
    case MX_SB_NZONES:        return sblock->s_nzones;
    case MX_SB_IMAP_BLOCKS:   return sblock->s_imap_blocks;
    case MX_SB_ZMAP_BLOCKS:   return sblock->s_zmap_blocks;
    case MX_SB_FIRSTDATAZONE: return sblock->s_firstdatazone;
    case MX_SB_LOG_ZONE_SIZE: return sblock->s_log_zone_size;
    case MX_SB_MAX_SIZE:      return sblock->s_max_size;
```

```
    case MX_SB_MAGIC:         return sblock->s_magic;
    case MX_SB_STATE:         return sblock->s_state;
    case MX_SB_ZONES:         return sblock->s_zones;
    default:                  return -1;  // error
  }
}
```

Defines:
mx_query_superblock, used in chunks 443a, 445, 451a, and 492.
Uses minix_superblock 440c and readblock 506b.

These constants can be declared as follows:

[444a] ⟨*constants* 112a⟩+≡ (44a) ◁440a 459b▷

```
enum { MX_SB_NINODES, MX_SB_NZONES, MX_SB_IMAP_BLOCKS, MX_SB_ZMAP_BLOCKS,
       MX_SB_FIRSTDATAZONE, MX_SB_LOG_ZONE_SIZE, MX_SB_MAX_SIZE,
       MX_SB_MAGIC, MX_SB_STATE, MX_SB_ZONES };
```

12.6.2 Zone and Inode Bitmaps

Going where? Now that we know how to access the superblock, our next task is to deal with the zone and inode bitmaps properly. This requires some fiddling to extract or modify single bits of a byte.

We need ways to access single bits in the inode and zone bitmaps. Reading is simple, because we only need to find the right byte and then perform a bit-shift operation, followed by a modulo operation to isolate a specific bit.

Let's start with the inode bitmap: One block stores 1024 bytes = 8192 bits, and the inode bitmap begins in block 2. Thus if `i` is the number of the bit we want to read, we must read in block `2 + i/8192`. Inside that block we need to read byte number `(i%8192) / 8`.

[444b] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁443b 445a▷

```
byte mx_get_imap_bit (int device, int i) {
  byte block[1024];
  byte thebyte;
  readblock (device, 2 + i/8192, (byte*)&block);
  thebyte = block[(i%8192)/8];
  return (thebyte >> (i%8)) % 2;
};
```

Defines:
mx_get_imap_bit, used in chunk 451a.
Uses readblock 506b.

For the zone map, we need to consider that the inode map (which is located just before it) may be larger than one block (if we have more than 8192 inodes). We can query the superblock to find out how many blocks are used.

Again, the zone map may be larger than one block, so we have to find out which block we need to read via a similar calculation.

⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁444b 445b▷ [445a]
```
byte mx_get_zmap_bit (int device, int i) {
  byte block[1024];
  byte thebyte;
  unsigned int zmap_start = 2 + mx_query_superblock (device, MX_SB_IMAP_BLOCKS);
  readblock (device, zmap_start + i/8192, (byte*)&block);
  thebyte = block[(i%8192)/8];
  return (thebyte >> (i%8)) % 2;
};
```
Uses mx_query_superblock 443b and readblock 506b.

In order to fetch the bit number i of an integer number n we use the formula `(n >> i) % 2` which performs a right shift (by i positions) and then cuts out the lowest bit with `% 2`.

We also need to set and clear individual bits. Since the code for accessing and changing a bit is almost identical for setting and clearing, we write two functions `mx_set_clear_*` which can both set and clear; they are called by the four `mx_set_*` and `mx_clear_*` functions with appropriate arguments:

⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁445a 446▷ [445b]
```
void mx_set_clear_imap_bit (int device, int i, int value) {
  byte block[1024];
  byte thebyte;
  readblock (device, 2 + i/8192, (byte*)&block);
  ⟨ set bit i from block block to value 445c ⟩
  writeblock (device, 2 + i/8192, (byte*)&block);
};

void mx_set_clear_zmap_bit (int device, int i, int value) {
  byte block[1024];
  byte thebyte;
  unsigned int zmap_start = 2 + mx_query_superblock (device, MX_SB_IMAP_BLOCKS);
  readblock (device, zmap_start + i/8192, (byte*)&block);
  ⟨ set bit i from block block to value 445c ⟩
  writeblock (device, zmap_start + i/8192, (byte*)&block);
};
```
Defines:
mx_set_clear_imap_bit, used in chunk 446.
mx_set_clear_zmap_bit, used in chunk 446.
Uses mx_query_superblock 443b, readblock 506b, and writeblock 507c.

where setting the bit from a block looks like this in both cases:

⟨ *set bit* i *from block* block *to* value 445c ⟩≡ (445b) [445c]
```
thebyte = block[(i%8192)/8];
if (value==0) {
  thebyte = thebyte & ~(1<<(i%8));          // Clear bit
} else {
  thebyte = thebyte | 1<<(i%8);             // Set bit
};
block[(i%8192)/8] = thebyte;
```

If the shift (<<), modulo (%), bitwise "and" (&), bitwise "or" (|) and bitwise negation (~) operations seem like magic to you, consider the following example calculations:

- For clearing bit 3 of 00101100_b:

```
byte                = 00101100
1<<3                = 00001000
~(1<<3)             = 11110111
00101100 & 11110111 = 00100100
```

- For setting bit 3 of 00100100_b:

```
byte                = 00100100
1<<3                = 00001000
00100100 | 00001000 = 00101100
```

This should explain setting and clearing a bit in a single byte well enough. The extra code which reads and writes `block[(i%8192)/8]` is necessary because we do not deal with a single byte but an array of such bytes.

Now the `mx_set_*` and `mx_clear_*` functions simply provide the right value (0 or 1) to the more general `mx_set_clear_*` function:

[446] ⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁445b 447c▷

```
void mx_set_imap_bit (int device, int i)   { mx_set_clear_imap_bit (device, i, 1); };
void mx_clear_imap_bit (int device, int i) { mx_set_clear_imap_bit (device, i, 0); };
void mx_set_zmap_bit (int device, int i)   { mx_set_clear_zmap_bit (device, i, 1); };
void mx_clear_zmap_bit (int device, int i) { mx_set_clear_zmap_bit (device, i, 0); };
```

Defines:
`mx_clear_imap_bit`, used in chunk 483.
`mx_clear_zmap_bit`, used in chunks 477, 481, 482, and 484e.
`mx_set_zmap_bit`, used in chunk 447a.
Uses `mx_set_clear_imap_bit` 445b and `mx_set_clear_zmap_bit` 445b.

Note: An optimized implementation will not do this calculation every single time, instead when mounting the filesystem, we should copy the superblock to memory and also memorize where the zone bitmap starts. Early versions of Ulix suffered from very slow disk access because reading blocks was not buffered—this led to actually reading the superblock from disk whenever we wanted to query the zone bitmap. With buffered read operations this is no longer a problem but still highly inefficient. It is, however, the simplest implementation and thus easy to grasp. Yet, you will see on the next pages that we did not stick with it because a slightly optimized version improved performance a lot.

Requesting a free inode or a free block means searching the corresponding bitmap for a zero bit. We start with the simple implementation of two $\texttt{mx\_request\_inode}_{448}\texttt{()}$ and $\texttt{mx\_request\_block}_{448}\texttt{()}$ functions which just loop over the whole bitmaps and check the bits with `mx_get_*_bit`. If all inodes or blocks are in use, these functions return `-1`.

Note that (as described above) the zone bitmap does not start with an entry for block 0, but with a fixed 1 entry, followed by the bit that describes the first data zone. In order to query the state of data zone `n` we need to call `mx_get_zmap_bit (device, n-s_firstdatazone +1)`. If this is unclear, go back to the example filesystem where `s_firstdatazone` is 34, then evaluate the expression for `n=34`.

⟨*old minix filesystem implementation* 447a⟩≡ [447a]
```
int mx_request_inode (int device) {
  int no_inodes = mx_query_superblock (device, MX_SB_NINODES);  // floppy: 480
  for (int i = 0;  i < no_inodes;  i++) {
    if (mx_get_imap_bit (device, i) == 0) {
      // found a free inode
      mx_set_imap_bit (device, i);   // mark as used
      return i;
    }
  }
  return -1;                         // found nothing
};

int mx_request_block (int device) {
  int no_zones   = mx_query_superblock (device, MX_SB_ZONES);   // floppy: 1440
  int first_data = mx_query_superblock (device, MX_SB_FIRSTDATAZONE);
  for (int i = 0;  i < no_zones - first_data - 2;  i++) {
    if (mx_get_zmap_bit (device, i) == 0) {
      mx_set_zmap_bit (device, i);   // mark as used
      return i + first_data - 1;     // floppy example: i+33
    }
  }
  return -1;                         // found nothing
};
```

While the above implementations of `mx_request_inode`$_{448}$ and `mx_request_block`$_{448}$ are correct, they are also highly inefficient. Thus, we will provide a second implementation which has a better performance. Normally, we do not focus on performance issues, but these functions are very slow which makes writing a new file unbearable.

We start with a helper function

⟨*function prototypes* 45a⟩+≡ (44a) ◁443a 450b▷ [447b]
```
int findZeroBitAndSet (byte *block, int maxindex);
```

which finds the first `0` bit in a block, changes it to `1` and returns its (bit) position

⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁446 448▷ [447c]
```
int findZeroBitAndSet (byte *block, int maxindex) {
  int i, j;
  byte b;
  for (i = 0;  i < 1024;  i++) {
    b = block[i];
    if (b != 0xFF) {
      // at least one bit in this byte is 0, find the first one
      for (j = 0;  j < 8;  j++) {
        if ( ((b >> j) % 2 == 0)       // bit is 0
             && (i*8 + j < maxindex) ) // bit position is ok
        {
          block[i] = b | (1 << j);     // set bit
          return i*8 + j;
```

```
        }
      }
    }
  }
  return -1;   // not found
}
```

Defines:
findZeroBitAndSet, used in chunks 447b and 448.

We need to provide a `maxindex` argument since the last block of the bitmap may not always be fully used. In the floppy example from above, only 480 bits of the inode bitmap and only less than 1440 bits of the inode bitmap have to be considered.

The implementations of $\texttt{mx\_request\_inode}_{448}$ and $\texttt{mx\_request\_block}_{448}$ that we actually use start with manually reading the superblock (since they need to access several of its entries). The optimization is reached via checking a whole block for a `0` bit with the helper function:

[448] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁447c 451a▷

```
int mx_request_inode (int device) {
  byte block[1024];
  struct minix_superblock *sblock;
  readblock (device, 1, (byte*)block);              // superblock = block 1
  sblock = (struct minix_superblock*) &block;

  int no_inodes  = sblock->s_ninodes;               // floppy: 480
  int imap_start = 2;

  int i, index;
  for (i = 0;  i < sblock->s_imap_blocks;  i++) {   // all IMAP blocks
    readblock (device, imap_start + i, (byte*)&block);
    index = findZeroBitAndSet ((byte*)&block, no_inodes);
    if (index != -1) {                              // found one!
      writeblock (device, imap_start + i, (byte*)&block);
      return i*8192 + index;
    }
  }
  return -1;                                        // found nothing
};

int mx_request_block (int device) {
  byte block[1024];
  struct minix_superblock *sblock;
  readblock (device, 1, (byte*)block);              // superblock = block 1
  sblock = (struct minix_superblock*) &block;

  int no_zones    = sblock->s_zones;                // floppy: 1440
  int zmap_start  = 2 + sblock->s_imap_blocks;
  int zmap_blocks = sblock->s_zmap_blocks;
  int data_start  = sblock->s_firstdatazone;
```

```
    int i, index;
    for (i = 0;  i < zmap_blocks;  i++) {              // all ZMAP blocks
      readblock (device, zmap_start + i, (byte*)&block);
      index = findZeroBitAndSet ((byte*)&block, no_zones);
      if (index != -1) {                               // found one!
        writeblock (device, zmap_start + i, (byte*)&block);
        return i*8192 + index + data_start - 1;        // convert to zone number
      }
    }
    return -1;                                         // found nothing
  };
```

Defines:
  mx_request_block, used in chunks 454a, 476, and 477.
  mx_request_inode, used in chunk 478b.
Uses findZeroBitAndSet 447c, minix_superblock 440c, readblock 506b, and writeblock 507c.

Note that mx_request_inode$_{448}$ simply returns the bit position of a free entry. On the other hand, mx_request_block$_{448}$ returns a block number which is *not* identical to the bit position since the zone bitmap holds no bits for the early blocks in the filesystem.

12.6.3 Reading and Writing Inodes

We're now able to query the superblock and read and write the two bitmaps. Our next goal is to create (empty) files.

Since empty files use no data blocks, this requires being able to read and write inodes.

Going where?

Creating a new (empty) file consists of the following steps:

1. Reserve an inode with mx_request_inode$_{448}$.
2. Write the inode.
3. Create an entry in the file's directory, i. e., create the (filename → inode number) mapping.

Before we start, remember how *pointer arithmetic* works; we will sometimes use memcpy$_{596c}$ to move data from a block around. If that block is declared as `char block[1024];` and you want to write a 32 byte chunk to position 512, then the code

pointer arithmetic

⟨*pointer arithmetic test 1* 449a⟩≡ [449a]

```
  offset = 512; size = 32;
  memcpy (&block + offset, &data, size);
```

will fail. On the other hand, if the block was declared via `char *block;` then the similar code

⟨*pointer arithmetic test 2* 449b⟩≡ [449b]

```
  offset = 512; size = 32;
  memcpy (block + offset, &data, size);
```

works as expected. The following example program `offset-test.c` shows the difference:

[450a] ⟨*offset-test.c* 450a⟩≡

```
#include <stdio.h>
int main () {
  char block[1024]; char *block2=(char*)&block; char data[]="Test";
  int size = sizeof (data); int offset = 512; long diff;

  printf ("&block:          %p \n", &block);
  printf ("&block + offset: %p \n", &block + offset);
  diff = (long)(&block+offset)-(long)&block;
  printf ("difference:      %ld \n", diff);

  printf ("block2:          %p \n", block2);
  printf ("block2 + offset: %p \n", block2 + offset);
  diff = (long)(block2+offset)-(long)block2;
  printf ("difference:      %ld \n", diff);
};
```

generates the following output:

```
$ ./offset-test
&block:          0x7ffff31802b0
&block + offset: 0x7ffff32002b0
difference:      524288                 // that is 512 x 1024 !
block2:          0x7ffff31802b0
block2 + offset: 0x7ffff31804b0
difference:      512                    // that's what we want
```

In the first attempt, `&block` creates a pointer to `block` that "knows" the size of `block` (which is a whole kilobyte). When adding the offset (512) the program actually adds that offset multiplied with the size (resulting in a 512 KByte offset). So in order to perform correct pointer arithmetic, it is necessary to first perform a cast to a `(char*)` pointer; otherwise you would access wrong memory areas (see also Appendix A.5).

We continue the implementation with two functions

[450b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁447b 450c▷

```
int mx_read_inode  (int device, int i, struct minix2_inode *inodeptr);
int mx_write_inode (int device, int i, struct minix2_inode *inodeptr);
```

which copy an inode from disk to memory or vice versa. They shall return 0 when an error occurs and the inode number `i` otherwise—this lets us write code of the form `if (!mx_read_inode`$_{451b}$`(...) { /* error */ }`. Trying to read an unused inode shall also generate an error.

Since reading and writing an inode are similar tasks we write a combined function

[450c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁450b 453a▷

```
int mx_read_write_inode (int device, int i, struct minix2_inode *inodeptr,
                         int wr_flag);
```

which can do both; the flag `wr_flag` decides about the direction.

⟨ *minix filesystem implementation* 420c⟩+≡ (440b) ◁448 451b▷ [451a]

```
int mx_read_write_inode (int device, int i, struct minix2_inode *inodeptr,
                         int wr_flag) {
  i--;          // first inode is No. 1, but has position 0 in table
  if ((i < 0) || (i ≥ mx_query_superblock (device, MX_SB_NINODES))) {
    return 0;  // illegal inode number
  }
  if (mx_get_imap_bit (device, i+1) == 0) {
    return 0;  // attempt to read unused inode; forbidden
  }
  const int inodesize      = sizeof (struct minix2_inode);
  const int inodesperblock = BLOCK_SIZE / inodesize;
  int       blockno        = i / inodesperblock + 2
                             + mx_query_superblock (device, MX_SB_IMAP_BLOCKS)
                             + mx_query_superblock (device, MX_SB_ZMAP_BLOCKS);
  int       blockoffset    = i % inodesperblock;
  // we need to read the block, even if this is a write operation
  byte block[1024];
  readblock (device, blockno, (byte*)&block);
  byte *addr = (byte*)&block;      // add offset, beware of pointer arithmetic
  addr += blockoffset * inodesize;
  if (!wr_flag) {                  // read or write_?
    memcpy (inodeptr, addr, inodesize);
  } else {
    memcpy (addr, inodeptr, inodesize);
    writeblock (device, blockno, (byte*)&block);  // write whole block to disk
  };
  return (i+1);                    // return original number
};
```

Defines:
`mx_read_write_inode`, used in chunks 450–52.
Uses `BLOCK_SIZE` 440a, `memcpy` 596c, `minix2_inode` 442a, `mx_get_imap_bit` 444b, `mx_query_superblock` 443b, `readblock` 506b, and `writeblock` 507c.

The only statement that needs some explanation is the calculation of `blockno` which queries the superblock twice to find out about the layout of the filesystem; the first block of the inode table is placed behind the zone bitmap. Block 2 is where the inode bitmap starts, and adding $\texttt{mx\_query\_superblock}_{443b}$ `(device, MX_SB_IMAP_BLOCKS)` and $\texttt{mx\_query\_superblock}_{443b}$ `(device, MX_SB_ZMAP_BLOCKS)` brings us right behind the zone bitmap. We need to add `i / inodesperblock` in order to pick the right block within the inode table.

As explained above, $\texttt{mx\_read\_inode}_{451b}$ and $\texttt{mx\_write\_inode}_{452a}$ simply call the function $\texttt{mx\_read\_write\_inode}_{451a}$ with `wr_flag` set to `0` or `1`:

⟨ *minix filesystem implementation* 420c⟩+≡ (440b) ◁451a 452a▷ [451b]

```
int mx_read_inode (int device, int i, struct minix2_inode *inodeptr) {
  return mx_read_write_inode (device, i, inodeptr, 0);  // 0 = false
}
```

Defines:
`mx_read_inode`, used in chunks 453b, 456, 457b, 466a, 479b, 480c, 484c, 487a, 490a, 589d, 607a, and 610d.
Uses `minix2_inode` 442a and `mx_read_write_inode` 451a.

[452a] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁451b 453b▷
```
int mx_write_inode (int device, int i, struct minix2_inode *inodeptr) {
  return mx_read_write_inode (device, i, inodeptr, 1);  // 1 = true
}
```
Defines:
`mx_write_inode`, used in chunks 450b, 454a, 456, 457b, 467b, 468c, 475c, 478b, 480c, 484c, 487a, and 589d.
Uses `minix2_inode` 442a and `mx_read_write_inode` 451a.

12.6.4 Directory Entries

Going where?

Reserving and writing inodes is only the first half of creating a new file; we also need to create directory entries (in the directory where we want to place a file). Our goal is to write a function $\texttt{mx\_write\_link}_{456}$ that takes an inode number and a pathname and creates the link between them.

That task consists of several smaller tasks: We need to be able to read and write single entries in the directory (file), find a free entry in the directory, split a pathname (such as `/home/user/dir/file`) into its *base name* (`file`) and its *directory name* (`/home/user/dir`). The directory name will lead us to the directory where we need to place the link.

base name, directory name

Let's look at the task step by step. In order to read the root directory of a volume, we need to know that Minix always uses inode number 1 for it, and it also starts counting inodes with 1. Thus, inode 1 is the first (not the second) inode, stored at position 0 of the inode table, *but* the corresponding bitmap entry is bit 1 (not 0), see further below.

Looking at the root directory's inode brings up a special case of file access (before reading the first regular file): Directories are a special kind of file which map filenames to inode numbers. We can read a directory block by block, and the first zone number is stored in `i_zone[0]`. We just need to know how to interpret the data: A directory file is an array of structures of type $\texttt{minix\_dir\_entry}_{452b}$:

[452b] ⟨*type definitions* 91⟩+≡ (44a) ◁442a 459a▷
```
struct minix_dir_entry {
  uint16_t inode;
  char name[30];
};
```
Defines:
`minix_dir_entry`, used in chunks 453b, 456, 461d, 480c, 487a, 488a, 490d, and 494b.

Each such entry has a size of $2 + 30 = 32$ bytes und starts with the 16-bit inode number, followed by the filename. Such filenames are normally null-terminated (as is standard for all strings on Unix systems), but when a filename uses the maximum allowed size of 30 characters, there is no space for the terminating `\0` character. Thus, simply copying an entry with $\texttt{strncpy}_{594b}$ will fail on filenames with maximum length. When dealing with Minix filenames internally, they should be stored in a `char[31]` string whose last byte is manually set to `\0`.

A block can hold $1024/32 = 32$ such directory entries. If there are more than 32 entries in a directory, an additional block is used (whose block number can be found in the next

entry, `i_zone[1]`. The total size of the directory is found by inspecting the inode entry `i_size`.

Now we start the work on the function `mx_write_link`$_{456}$`()` that can add a (filename $\mapsto$ inode number) mapping to a directory. As already mentioned, directories are special files and the root directory has the inode number 1. Each of the associated data blocks contains 32 directory entries of type `struct minix_dir_entry`$_{452b}$ (since $32 \cdot 32 = 1024$). An unused entry has the `inode` field set to 0. We provide two functions

⟨ *function prototypes* 45a⟩+≡ (44a) ◁450c 455c▷ [453a]

```
int mx_read_dir_entry  (int device, int inodenr, int entrynr,
                        struct minix_dir_entry *entry);
int mx_write_dir_entry (int device, int inodenr, int entrynr,
                        struct minix_dir_entry *entry);
```

for reading or writing individual entries of a directory.

We will use the same trick that we applied to reading and writing inodes by providing a common function `mx_read_write_dir_entry`$_{453b}$ that can do both and decides via an extra flag `wr_flag` whether it shall read or write. The other arguments are the device, the inode number of the directory (file) and a pointer to a `struct minix_dir_entry`$_{452b}$ variable.

Since only 32 entries fit in one block, we might have to reserve a new block and enter its location in the directory's inode; then we can go on writing entries in the new block. Similarly, when we later delete entries, we might want to remove additional blocks that are no longer needed.

In each inode's `i_zone` array only the first seven entries refer directly to data blocks, so using these we can work with up to $7 \cdot 32 = 224$ directory entries. After that an indirection block must be used, but we will restrict our Minix implementation to a maximum of 224 entries for a directory.

⟨ *minix filesystem implementation* 420c⟩+≡ (440b) ◁452a 456▷ [453b]

```
int mx_read_write_dir_entry (int device, int inodenr, int entrynr,
                             struct minix_dir_entry *entry, int wr_flag) {
  if (entrynr ≥ 32 * 7) {  // 7 direct blocks, 32 entries per block
    return false;
  }

  struct minix2_inode inode;
  mx_read_inode (device, inodenr, &inode);            // read directory inode
  int blockno;
  blockno = inode.i_zone[entrynr/32];    // number of block that holds the entry
  if (blockno == 0) {
    if (wr_flag) { ⟨ reserve a block and map it in the directory inode 454a⟩ }
    else         return false;
  }

  char block[1024];
  readblock (device, blockno, (byte*)&block);

  int offset = (32*entrynr) % BLOCK_SIZE;
```

```
    if (!wr_flag) {
      memcpy (entry, ((char*)&block)+offset, 32);       // reading
      return (entry->inode != 0);                       // true if entry non-empty
    } else {
      memcpy (((byte*)&block)+offset, entry, 32);       // writing
      writeblock (device, blockno, (byte*)&block);
      return true;
    };
  };

  int mx_read_dir_entry (int device, int inodenr, int entrynr,
                         struct minix_dir_entry *entry) {
    return mx_read_write_dir_entry (device, inodenr, entrynr, entry, false);
  };

  int mx_write_dir_entry (int device, int inodenr, int entrynr,
                          struct minix_dir_entry *entry) {
    return mx_read_write_dir_entry (device, inodenr, entrynr, entry, true);
  };
```

Defines:
mx_read_dir_entry, used in chunks 456, 462a, 480c, and 490d.
mx_write_dir_entry, used in chunks 453a, 456, and 480c.
Uses BLOCK_SIZE 440a, memcpy 596c, minix2_inode 442a, minix_dir_entry 452b, mx_read_inode 451b, readblock 506b, and writeblock 507c.

If the block which should hold the directory entry does not yet exist (and we're trying to write to it), we create it and enter it in the directory inode:

[454a] ⟨*reserve a block and map it in the directory inode* 454a⟩≡ (453b)

```
  blockno = mx_request_block (device);
  char empty_block[1024] = { 0 };
  writeblock (device, blockno, (byte*)&empty_block);
  inode.i_zone[entrynr/32] = blockno;
  mx_write_inode (device, inodenr, &inode);        // update directory inode
```

Uses mx_request_block 448, mx_write_inode 452a, and writeblock 507c.

For dealing with pathnames we will sometimes need two helper functions: dirname$_{455b}$ and basename$_{455b}$ can be used to split a path into a directory (path) and a file or directory name, for example `dirname ("/usr/bin/vi") = "/usr/bin"` and `basename ("/usr/bin/vi") = "vi"`. This works similarly for relative paths, and the special case of a pathname `x` without any slashes is handled by `dirname ("x") = "."` and `basename ("x") = "x"`.

We will also recycle these functions in the user mode library, since it does not matter whether the kernel or a program wants to split a pathname. Instead of parsing the path in two separate functions, we write a combined function splitpath$_{455a}$ and call that one from basename$_{455b}$ and dirname$_{455b}$.

[454b] ⟨*public function prototypes* 454b⟩≡ (44a 48a) 593▷

```
  void splitpath (const char *path, char *dirname, char *basename);
  char *basename (char *path);
  char *dirname  (char *path);
```

⟨*public function implementations* 455a⟩≡ (44a 48b) 455b▷ [455a]

```
void splitpath (const char *path, char *dirname, char *basename) {
  if (strlen (path) == 1 && path[0] == '/') { // special case "/"
    strncpy (dirname, "/", 1);  strncpy (basename, "/", 1);  return;
  }
  char p[256]; strncpy (p, path, 256);        // work on copy
  int pos = strlen (p) - 1;
  if (p[pos] == '/') { p[pos] = 0; pos--; }   // strip trailing '/'

  for (;;) {                                  // search for / (from back to front)
    pos--;
    if (pos == -1) {                          // no single slash found
      strncpy (dirname, ".", 2);  strncpy (basename, p, 256);  return;
    }
    if (p[pos] == '/') {                      // slash found
      if (pos==0)
        strncpy (dirname, "/", 2);            // special case /
      else {
        memcpy (dirname, p, pos);
        dirname[pos] = 0;                     // remove trailing '/'
      }
      strncpy (basename, p + pos + 1, 30);
      return;
    }
  }
}
```

Defines:
splitpath, used in chunks 419a, 432e, 454–56, 480c, 487a, 488a, and 577.
Uses basename 455b, dirname 455b, memcpy 596c, strlen 594a, and strncpy 594b.

In the implementations of basename$_{455b}$ and dirname$_{455b}$ we declare bname and dname as static so that they are not stored on the stack; that way we can return a pointer.

⟨*public function implementations* 455a⟩+≡ (44a 48b) ◁455a 594a▷ [455b]

```
char *basename (char *path) {
  static char bname[30];  static char dname[256];
  splitpath (path, dname, bname);  return (char *)bname;
}

char *dirname (char *path) {
  static char bname[30];  static char dname[256];
  splitpath (path, dname, bname);  return (char *)dname;
}
```

Defines:
basename, used in chunks 455a and 577b.
dirname, used in chunks 419a, 455a, 456, and 577.
Uses splitpath 455a.

The implementation of the

⟨*function prototypes* 45a⟩+≡ (44a) ◁453a 457a▷ [455c]

```
void mx_write_link (int device, int inodenr, const char *filename);
```

function is complex:

- First, we check whether the directory already contains an entry for the filename—it is not possible to have the same filename twice in a directory.
- Then we split the path into the directory name and the base filemame.
- We locate the inode that belongs to the directory file using `mx_pathname_to_ino`$_{461d}$ (a function that we still have to implement; this follows a few pages later).
- Then we read all the directory entries (using `mx_read_dir_entry`$_{453b}$ from above) until we find a free entry. If we don't, we have to abort because the directory is full.
- Once we've found a free entry, we prepare a directory entry and write it to the free location.
- As a last thought, we must not forget to increase the link count of the inode: It counts how many links to the inode exist. For a freshly created file we could always set that value to 1, but we will also use this function when we create a hard link.
- Finally we update the size of the directory (it may have grown by 32 bytes unless we've found a free entry between other, used entries) and write back the modified directory inode.

[456] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁453b 457b▷

```
void mx_write_link (int device, int inodenr, const char *path) {
  if (mx_file_exists (device, path)) {   // check if filename already exists
      printf ("ERROR: filename %s exists!\n", path);  return;
  };
  struct minix_dir_entry dentry;  struct minix2_inode inode;
  char dirname[256];  char filename[30];
  splitpath (path, dirname, filename);
  int dir_inode_no = mx_pathname_to_ino (device, dirname);
  // find free location and enter it
  mx_read_inode (device, dir_inode_no, &inode);   // read directory inode
  for (int i = 0;  i < 32 * 7;  i++) {
    mx_read_dir_entry (device, dir_inode_no, i, &dentry);
    if (dentry.inode==0 || i * 32 ≥ inode.i_size) {
      dentry.inode = inodenr;                     // found an empty entry
      memcpy ((char*)dentry.name, filename, 30);
      mx_write_dir_entry (device, dir_inode_no, i, &dentry);
      mx_increase_link_count (device, inodenr);   // link count for file
      if (inode.i_size < 32*(i+1)) {              // modify dir. inode size
        mx_read_inode (device, dir_inode_no, &inode);   // must read again
        inode.i_size = 32*(i+1);
        mx_write_inode (device, dir_inode_no, &inode);
      };
      return;                                     // success
    };
  };
  printf ("ERROR: no free entry in directory\n"); // search failed
};
```

Defines:
mx_write_link, used in chunks 455c, 478b, and 480a.
Uses dirname 455b, memcpy 596c, minix2_inode 442a, minix_dir_entry 452b, mx_file_exists 479b, mx_increase_link_count 457b, mx_pathname_to_ino 461d, mx_read_dir_entry 453b, mx_read_inode 451b, mx_write_dir_entry 453b, mx_write_inode 452a, printf 601a, read 429b, and splitpath 455a.

In the last few lines we need to read the directory inode from disk again (even though we did so just nine lines ago, but calling mx_write_dir_entry$_{453b}$ may have modified it if a new block was added to the directory—we must not overwrite this change.

This function uses mx_increase_link_count$_{457b}$() which adds 1 to the number of links for a given inode:

⟨*function prototypes* 45a⟩+≡ (44a) ◁455c 461c▷ [457a]
```
int mx_increase_link_count (int device, int inodenr);
```

It simply reads an inode, increments the i_nlinks entry and writes it back:

⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁456 461d▷ [457b]
```
int mx_increase_link_count (int device, int inodenr) {
  struct minix2_inode inode;
  mx_read_inode (device, inodenr, &inode);
  inode.i_nlinks++;
  mx_write_inode (device, inodenr, &inode);
  return inode.i_nlinks;
};
```
Defines:
mx_increase_link_count, used in chunks 456 and 457a.
Uses minix2_inode 442a, mx_read_inode 451b, and mx_write_inode 452a.

12.6.5 The i_mode Entry of the Inode

Each inode contains an i_mode entry that describes the file type and the access permissions. Since we will need to query this information in some of the following functions, we provide a few standard constants that make it easier to check for a specific property.

⟨*public constants* 46a⟩+≡ (44a 48a) ◁424b 460b▷ [457c]
```
#define S_IRWXU  0000700    // RWX mask for owner
#define S_IRUSR  0000400    // R for owner
#define S_IWUSR  0000200    // W for owner
#define S_IXUSR  0000100    // X for owner

#define S_IRWXG  0000070    // RWX mask for group
#define S_IRGRP  0000040    // R for group
#define S_IWGRP  0000020    // W for group
#define S_IXGRP  0000010    // X for group

#define S_IRWXO  0000007    // RWX mask for other
#define S_IROTH  0000004    // R for other
#define S_IWOTH  0000002    // W for other
#define S_IXOTH  0000001    // X for other
```

```
#define S_ISUID  0004000    // suid bit (set user ID)
#define S_ISGID  0002000    // sgid bit (set group ID)
#define S_ISVTX  0001000    // save swapped text even after use

#define S_IFMT   0170000    // mask the file type part
#define S_IFIFO  0010000    // named pipe (fifo)
#define S_IFCHR  0020000    // character special
#define S_IFDIR  0040000    // directory
#define S_IFBLK  0060000    // block special
#define S_IFREG  0100000    // regular
#define S_IFLNK  0120000    // symbolic link
#define S_IFSOCK 0140000    // socket
```

Defines:
S_IFBLK, used in chunk 499d.
S_IFDIR, used in chunks 432e, 479b, 487a, and 499d.
S_IFLNK, used in chunks 420c and 484c.
S_IFREG, used in chunk 478b.

Some of these constants can be used for direct comparisons, for example, in order to check whether a file has the read access bit for the file owner set, you could check whether `(i_mode & S_IRUSR`$_{457c}$`) != 0`. We assume that you're aware of the standard access permissions on Unix systems—if not, here's a brief summary:

access permissions

The standard *access permissions* encompass nine bits grouped in three groups of three bits each. The first group describes the permissions granted to the file owner who may or may not read, write or execute the file. In the output of the `ls -l` command, these are represented by the characters in the second to fourth column and shown as `rwx` (or some of those letters replaced with – if a permission is not set. For example, `rw-` in that place says: owner can read and write, but not execute (those are the standard settings for a document file). The next group describes the corresponding permissions for the members of the group that the file belongs to, and the third group shows permissions for all other users.

rwx (owner)

rwx (group, other users)

Set User ID bit

There are two further interesting bits which can be set: The *Set User ID bit* (`suid`), when set on an executable file, changes the effective user ID of a program to the file owner, regardless of who started it. Similarly the *Set Group ID bit* (`sgid`) sets the effective group ID to the file's group. Owner and group are also stored in the inode (in the `i_uid` and `i_gid` fields).

Set Group ID bit

The last block of constants can be used for identifying the type of a file. This does not refer to properties like "Word document" and "C source file", but to whether a file is a "classical" file or something else, like a directory, a block or character device file (only the block variety is implemented in Ulix, see Section 12.7), a symbolic link or a socket (not available, either). Since a file cannot be in more than one of these categories, you can check for a specific file type with an expression like `(i_mode & S_IFMT`$_{457c}$`) == S_IFDIR`$_{457c}$. This is an example for using a *mask* (to mask out the irrelevant bits of the `i_mode` field): We used `S_IFMT`$_{457c}$ to remove the access permissions. Similarly, `S_IRWXU`$_{457c}$, `S_IRWXG`$_{457c}$ and `S_IRWXO`$_{457c}$ are masks for the *owner*, *group* and *others* parts of the permissions.

mask

We will explain this in more detail in Chapter 15 where we discuss users and groups and also add permission checking code to the filesystem functions from this chapter.

12.6.6 Opening and Closing Files

Going where?

We're getting closer to actually opening files. First, we need to introduce some new data structures, including an internal inode representation and a file status structure. Next up is a function that gets the inode number when given a path. Then we can start work on the implementation of the mx_open$_{464b}$ function.

The goal is to provide the kernel functions which will be called when a process uses the standard filesystem functions open$_{429b}$, read$_{429b}$, write$_{429b}$ lseek$_{429b}$ and close$_{429b}$. In the Minix subsystem we need the functions mx_open$_{464b}$, mx_read$_{470b}$, mx_write$_{474c}$, mx_lseek$_{469c}$ and mx_close$_{467b}$ (and they are called by the corresponding u_∗ functions).

We have to introduce some new data structures that will help the kernel stay aware of the states of open files:

Internal Inode: This will be an enhanced copy of the inode (as it is stored on the filesystem), but with extra elements, e. g. a reference counter refcount that takes notice of how often the associated file is opened. Whenever we open a file we reserve such an internal inode. All changes to the inode will first be made in the internal copy only; when closing the file we will write the information back to disk. (For immediate writing there will be an mx_sync$_{468c}$ function.) One of the new fields is clean: It is set to 1 as long as no changes were made to the internal inode. A change resets it, and calling my_sync will set it again (after saving the changes to disk).

⟨ *type definitions* 91⟩+≡ (44a) ◁452b 460a▷ [459a]

```
struct int_minix2_inode {
  ⟨external minix2 inode 442b⟩  // fields from the external inode
  int ino;                      // inode number
  unsigned int    refcount;     // how many users?
  unsigned short  clean;        // 0: changed; 1: unchanged (as on disk)
  short           device;       // file resides on which device?
};
```

Defines:
int_minix2_inode, used in chunks 459c, 460a, 464d, 467–71, 473a, 475a, 476b, 484e, and 607b.

We create an internal inode table that can store up to 256 records on open files:

⟨ *constants* 112a⟩+≡ (44a) ◁444a 461a▷ [459b]

```
#define MAX_INT_INODES 256
```

Uses MAX_INT_INODES 459c.

⟨ *global variables* 92b⟩+≡ (44a) ◁410b 461b▷ [459c]

```
struct int_minix2_inode mx_inodes[MAX_INT_INODES] = {{ 0 }};
```

Defines:
MAX_INT_INODES, used in chunks 459b, 462c, and 464d.
mx_inodes, used in chunks 462c, 464d, and 466a.
Uses int_minix2_inode 459a.

Local File Descriptor: The LFD is a non-negative integer returned by mx_open$_{464b}$ which the other mx_∗ functions use for accessing an open file. It has the same function as

the process file descriptor in user mode programs, however the local file descriptor is valid in the whole Minix subsystem, whereas each process counts file descriptors separately. When we later allow processes to work with files (via system calls), we will map process-local file descriptors to global file descriptors which are more general than the local ones. For a reminder of how global, local and process file descriptors are connected, go back to Section 12.3.3 (p. 410).

File Status: This is a structure that points to an internal inode. (If there's a null pointer, then this specific file status structure is not in use.) Additionally, it stores the current read/write position and the access mode (see below).

[460a] ⟨*type definitions* 91⟩+≡ (44a) ◁459a 494d▷

```
struct mx_filestat {
  struct int_minix2_inode *int_inode;
  int pos;
  short mode;
};
```

Defines:
`mx_filestat`, used in chunks 461b, 467–70, 475a, and 484e.
Uses `int_minix2_inode` 459a.

Thus, if a file is opened twice, there will be *one* internal inode, referenced by the (Minix-subsystem-)local file descriptor, and *two* $\texttt{mx\_filestat}_{460a}$ structures, since access mode and read/write position may be different.

We support the following modes for opening:

[460b] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁457c 469b▷

```
#define O_RDONLY        0x0000    // read only
#define O_WRONLY        0x0001    // write only
#define O_RDWR          0x0002    // read and write
#define O_APPEND        0x0008    // append mode
#define O_CREAT         0x0200    // create file
```

Defines:
`O_APPEND`, used in chunks 465, 469c, and 470c.
`O_CREAT`, used in chunks 464c, 484b, 487a, 495c, and 576d.
`O_RDONLY`, used in chunks 190c, 420c, 475a, 488a, 579c, 582a, and 585b.
`O_RDWR`, used in chunks 293b and 579c.
`O_WRONLY`, used in chunks 420a, 470c, 484b, 487a, and 579c.

- $\texttt{O\_RDONLY}_{460b}$ and $\texttt{O\_WRONLY}_{460b}$ are used when the file shall be used exclusively for reading or writing, respectively.
- Using the $\texttt{O\_RDWR}_{460b}$ mode allows read and write access.
- The mode $\texttt{O\_APPEND}_{460b}$ can be supplied in addition to $\texttt{O\_WRONLY}_{460b}$ (by calculating the mode as $\texttt{O\_WRONLY}_{460b}$ | $\texttt{O\_APPEND}_{460b}$). In that case all write operations append to the file, and $\texttt{lseek}_{429b}$ calls are ignored.
- $\texttt{O\_CREAT}_{460b}$ allows the creation of new files. Trying to open a non-existing file without $\texttt{O\_CREAT}_{460b}$ will fail. On the other hand, using $\texttt{O\_CREAT}_{460b}$ with an already existing file does not change anything (specifically: it does not truncate the file).

Status List: This is an array that holds 256 file status entries, so we allow the Minix subsystem to open up to 256 files simultaneously—the same limit holds for all subsystems since we defined the global file descriptors in a way that allows the local component of the number to lie between 0 and 255.

⟨*constants* 112a⟩+≡ (44a) ◁459b 472▷ [461a]
```
#define MX_MAX_FILES 256
```
Defines:
MX_MAX_FILES, used in chunks 461b, 463a, 467–70, 475a, 484e, and 607b.

⟨*global variables* 92b⟩+≡ (44a) ◁459c 464a▷ [461b]
```
struct mx_filestat mx_status[MX_MAX_FILES] = { { 0 } };
```
Defines:
mx_status, used in chunks 463a, 465, 467–70, 475a, 484e, and 607b.
Uses mx_filestat 460a and MX_MAX_FILES 461a.

We need a function that looks up a filename in the directory and returns the inode number. Assume we want to look up the path `/etc/passwd`: We can assume that this is an absolute path (because the virtual filesystem layer already took care of that). Then we scan the path until we reach the next `/` (or the string terminator `\0`) which gives us a directory to search for. We can then look up its inode and continue the search with the next directory element. Eventually we reach the last part of the pathname and return its inode number.

The search begins in the volume's root directory that always has inode number 1 on a Minix filesystem.

⟨*function prototypes* 45a⟩+≡ (44a) ◁457a 462b▷ [461c]
```
int mx_pathname_to_ino (int device, const char *path);
```

⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁457b 462c▷ [461d]
```
int mx_pathname_to_ino (int device, const char *path) {
  struct minix2_inode dirinode, inode;
  struct minix_dir_entry dentry;
  char subpath[31];                          // maximum name length: 30
  char searchbuf[256];
  char *search = (char*)searchbuf;
  strncpy (search, path, 256);               // do not modify original path
  int dirinode_no = 1;                       // inode number of / directory
  int next_dirinode_no;
  short final = 0;                           // final = 1 if looking at final part

  search++;
  if (*search == '\0') { return 1; }         // searching for / : inode 1
  while (*search != '\0') {                  // work until end of path reached
    ⟨mx_pathname_to_ino: search loop 462a⟩
  }
  return next_dirinode_no;
};
```
Defines:
mx_pathname_to_ino, used in chunks 456, 461c, 464c, 479, 480, 484c, 487a, 490, and 589d.
Uses minix2_inode 442a, minix_dir_entry 452b, and strncpy 594b.

Inside the loop we pick the next sub-path and perform the inode lookup. We know that we're done when `search` points to the `'\0'` character (the end of the pathname):

[462a] ⟨mx_pathname_to_ino: *search loop* 462a⟩≡ (461d)

```
int i = 0;
while (*search != '\0' && *search != '/') {
  subpath[i] = *search;
  search++; i++;
}
subpath[i] = '\0';                    // terminate subpath string

if (*search == '\0') final = 1;       // looking at final part of path

next_dirinode_no = -1;                // look up subpath
for (i = 0;  i < 32*7;  i++) {        // max. 32 * 7 entries
  mx_read_dir_entry (device, dirinode_no, i, &dentry);
  if (dentry.inode != 0) {
    if (strequal (dentry.name, subpath)) {
      next_dirinode_no = dentry.inode;  // found it!
      break;                            // leave for loop
    }
  }
}

// now next_dirinode_no is either -1 (not found) or points to next step
if (next_dirinode_no == -1) { return -1; }   // not found!

dirinode_no = next_dirinode_no;
if (*search != '\0')  search++;
else                  break;          // finished, leave while loop
```

Uses `mx_read_dir_entry` 453b and `strequal` 596a.

We also need two helper functions that give us the index of a free $\texttt{mx\_inodes}_{459c}[]$ and a free $\texttt{mx\_status}_{461b}[]$ entry. The code is similar: We loop over the respective array and check whether an entry is free. $\texttt{mx\_inodes}_{459c}[\texttt{i}]$ is free if its `refcount` element is 0; the file status entry $\texttt{mx\_status}_{461b}[\texttt{i}]$ is free if its `int_inode` element is a $\texttt{NULL}_{46a}$ pointer.

[462b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁461c 463b▷

```
int mx_get_free_inodes_entry ();
int mx_get_free_status_entry ();
```

[462c] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁461d 463a▷

```
int mx_get_free_inodes_entry () {
  for (int i = 0;  i < MAX_INT_INODES;  i++) {      // returns internal inode no.
    if (mx_inodes[i].refcount == 0)  return i;
  }
  return -1;
}
```

Defines:
`mx_get_free_inodes_entry`, used in chunk 464d.
Uses `MAX_INT_INODES` 459c and `mx_inodes` 459c.

⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁462c 464b▷ [463a]

```
int mx_get_free_status_entry () {
  for (int i = 0;  i < MX_MAX_FILES;  i++) {        // returns an MFD
    if (mx_status[i].int_inode == NULL)  return i;
  }
  return -1;
}
```

Defines:
`mx_get_free_status_entry`, used in chunks 462b, 464d, and 468b.
Uses `MX_MAX_FILES` 461a, `mx_status` 461b, and `NULL` 46a.

We're about to show the implementation of the `mx_open`$_{464b}$ function

⟨*function prototypes* 45a⟩+≡ (44a) ◁462b 467a▷ [463b]

```
int mx_open (int device, const char *path, int oflag);
```

that will use many of the functions we've already discussed (and also some new ones). Figure 12.11 shows the function call graph for `mx_open`$_{464b}$, so you can see that opening a file is a rather complex task.

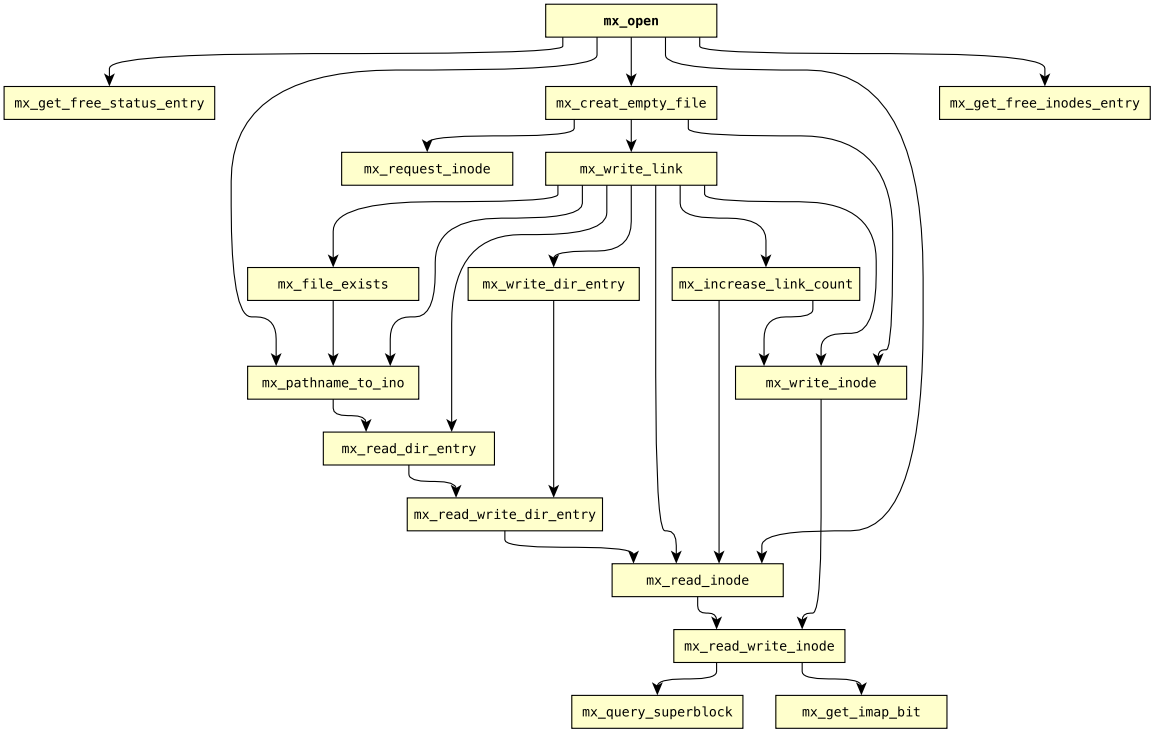


Figure 12.11: Minix subsystem functions called my `mx_open`.

12.6.6.1 mx_open

We define two variables

[464a] ⟨*global variables* 92b⟩+≡ (44a) ◁461b 494b▷
```
int count_open_files = 0;  // number of open files
int count_int_inodes = 0;  // number of internal inodes in use
```
Defines:
count_int_inodes, used in chunks 466c and 467b.
count_open_files, used in chunks 464d, 466c, and 467b.

to keep track of the open files and the used internal inodes and we dedicate a separate code chunk ⟨mx_open 464c⟩ to the implementation:

[464b] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁463a 467b▷
```
int mx_open (int device, const char *path, int oflag) {
  ⟨mx_open 464c⟩
}
```
Defines:
mx_open, used in chunks 412c, 463b, 484b, and 487a.

We start with checking whether the file exists—if it does not, but the O_CREAT$_{460b}$ flag was used, we will call mx_creat_empty_file$_{478b}$ to make a new file. In both cases ext_ino is set to the number of the external inode.

[464c] ⟨mx_open 464c⟩≡ (464b) 464d▷
```
int ext_ino = mx_pathname_to_ino (device, path);
if (ext_ino == -1) {
  // file not found
  if ((oflag & O_CREAT) != 0) {
    ext_ino = mx_creat_empty_file (device, path, 0644);
  }
  else {
    return (-1);  // file not found and no O_CREAT
  }
}
```
Uses mx_creat_empty_file 478b, mx_pathname_to_ino 461d, and O_CREAT 460b.

In int_ino we will store the index into the internal inode table. The file may already be open, because another process (or even the same one) opened it earlier. In that case a valid internal inode is in place and can be recycled; otherwise we create a fresh one. We find out if the file is open by checking the ino and device fields of all our mx_inodes$_{459c}$ array entries.

[464d] ⟨mx_open 464c⟩+≡ (464b) ◁464c 465▷
```
short file_already_open = false;
int mfd = mx_get_free_status_entry ();

int int_ino = -1;   // number of internal inode for this file
int i;
if (count_open_files == 0) {
  int_ino = 0;      // first file to be opened
```

```
  } else {
    for (i = 0; i < MAX_INT_INODES; i++) {
      if (mx_inodes[i].ino == ext_ino && mx_inodes[i].device == device) {
        // same inode number and same device: this is the same file!
        file_already_open = true;
        int_ino = i;
        break;
      }
    }
    // reached end of the loop: file is not open
    if (int_ino == -1) int_ino = mx_get_free_inodes_entry ();
  }

  if (int_ino == -1) {
    return -1;      // error: no free internal inode available
  }

  struct int_minix2_inode *inode = &(mx_inodes[int_ino]);
```

Uses count_open_files 464a, int_minix2_inode 459a, MAX_INT_INODES 459c, mx_get_free_inodes_entry 462c, mx_get_free_status_entry 463a, and mx_inodes 459c.

Now `int_ino` is either set to 0 (we're just opening the first file), to an index of an already existing internal inode or to the index of a fresh internal inode (provided by $\texttt{mx\_get\_free\_inodes\_entry}_{462c}$) and `inode` points to that entry.

`mfd` is the local file descriptor (an index into the $\texttt{mx\_status}_{461b}$ file status table. We can start filling that entry:

⟨mx_open 464c⟩+≡ (464b) ◁464d 466c▷ [465]

```
  mx_status[mfd].int_inode = inode;
  mx_status[mfd].pos       = 0;
  mx_status[mfd].mode      = oflag;

  if (file_already_open) { ⟨mx_open case: file already open 466b⟩ }
  else                   { ⟨mx_open case: file not open 466a⟩ }

  if ((oflag & O_APPEND) != 0)
    mx_status[mfd].pos = inode->i_size;   // append: set pos to end of file
```

Uses mx_status 461b and O_APPEND 460b.

$\texttt{mx\_status}_{461b}\texttt{[mfd].pos}$ is set to the current read/write position—normally that is 0 when freshly opening a file, but if the $\texttt{O\_APPEND}_{460b}$ flag was given, we set it to the the file size so that writing will begin at the end of the file. In $\texttt{mx\_status}_{461b}\texttt{[mfd].mode}$ we remember the `oflag` argument of the $\texttt{mx\_open}_{464b}$ call. This will later be used to determine whether it is acceptable to read from or write to the file. The current read/write position and the `mode` field are our reasons for having separate $\texttt{mx\_status}_{461b}\texttt{[]}$ entries, since both can differ for several opening operations on the file.

Now there are two cases that we need to treat differently. If the file is not yet open, we copy the inode from disk to memory. We can simply use the $\texttt{mx\_read\_inode}_{451b}$ function because we declared the two inode types (on-disk inode: $\texttt{struct minix2\_inode}_{442a}$, internal

inode: `struct int_minix2_inode`$_{459a}$) so that they both start with the same fields—that lets us cast the pointer to the internal inode to a normal inode pointer; here are both types for a quick comparison:

```
struct minix2_inode {
  uint16_t i_mode;
  uint16_t i_nlinks;
  uint16_t i_uid;
  uint16_t i_gid;
  uint32_t i_size;
  uint32_t i_atime;
  uint32_t i_mtime;
  uint32_t i_ctime;
  uint32_t i_zone[10];
};
```

```
struct int_minix2_inode {
  uint16_t i_mode;
  uint16_t i_nlinks;
  uint16_t i_uid;
  uint16_t i_gid;
  uint32_t i_size;
  uint32_t i_atime;
  uint32_t i_mtime;
  uint32_t i_ctime;
  uint32_t i_zone[10];
  int      ino;
  uint32_t refcount;
  uint16_t clean;
  short    device;
};
```

[466a] ⟨mx_open *case: file not open* 466a⟩≡ (465)

```
// copy diskinode[ext_ino] to mx_inodes[int_ino]
mx_read_inode (device, ext_ino, (struct minix2_inode*) inode);
inode->ino = ext_ino;    // number of external inode
inode->device = device;  // what device is the file on?
inode->refcount = 1;     // one user
inode->clean = true;     // inode is clean (just copied from disk)
```

Uses `minix2_inode` 442a, `mx_inodes` 459c, and `mx_read_inode` 451b.

If the file is already open, we have less work: We simply increase the inode's `refcount` field, because the inode is gaining an additional user:

[466b] ⟨mx_open *case: file already open* 466b⟩≡ (465)

```
inode->refcount++;       // file is opened once more
```

We set `clean` to `true` and `refcount` to 1, and we take note of the external inode and the device. Note that it is important to remember what device the file resides on because the inode number alone is not enough to identify a file when more than one filesystem is mounted.

When opening fails we return -1 (earlier in the code). Otherwise we increment the counters for open files and (possibly) used inodes and return the new local file descriptor. (The variable `mfd` that we use in this function is short for "Minix file descriptor".)

[466c] ⟨mx_open 464c⟩+≡ (464b) ◁465

```
count_open_files++;
if (!file_already_open) count_int_inodes++;
return mfd;
```

Uses `count_int_inodes` 464a and `count_open_files` 464a.

Note that we do not check access permissions—this is handled one layer further up in the virtual filesystem.

12.6.6.2 mx_close

Closing an open file with

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁463b 468a▷ [467a]

```
int mx_close (int mfd);
```

is a comparatively simple task. We set the `int_inode` pointer in the file status entry to `NULL`$_{46a}$ and decrement `count_open_files`$_{464a}$. We also check if this was the last user of the internal inode—if that is true and the internal inode is not clean, we write it back to disk. We don't have to explicitly mark the inode as unused—setting its `refcount` to 0 does the job since that property is what we use to find a free one.

⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁464b 468b▷ [467b]

```
int mx_close (int mfd) {
  if (mfd < 0 || mfd ≥ MX_MAX_FILES)  return -1;   // wrong mfd number
  struct mx_filestat      *st    = &mx_status[mfd];
  struct int_minix2_inode *inode = st->int_inode;
  if (inode == NULL)  return -1;                   // no open file
  short device = inode->device;

  // close file
  inode->refcount--;
  st->int_inode = NULL;

  if (inode->refcount == 0) {  // usage count down to 0? Then synchronize inode
    if (inode->clean == 0) {
      int ext_ino = inode->ino;
      mx_write_inode (device, ext_ino, (struct minix2_inode*) inode);
    }
    count_int_inodes--;
  }

  count_open_files--;
  return 0;
}
```

Defines:
`mx_close`, used in chunks 418a, 467a, 484b, and 487a.
Uses `close` 429b, `count_int_inodes` 464a, `count_open_files` 464a, `int_minix2_inode` 459a, `minix2_inode` 442a, `mx_filestat` 460a, `MX_MAX_FILES` 461a, `mx_status` 461b, `mx_write_inode` 452a, `NULL` 46a, and `open` 429b.

12.6.6.3 Helpers: mx_reopen and mx_sync

We provide an `mx_reopen`$_{468b}$ function that is used when file descriptors are duplicated by `fork`$_{213g}$, it is then called by `u_reopen`:

[468a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁467a 469a▷
```
int mx_reopen (int mfd);
```

It makes a copy of the `mx_status`$_{461b}$`[]` entry so that the original process and its child can work with different values for the read/write position—if we simply let the child process use the same `mx_status`$_{461b}$`[]` entry, every read or write operation would also update the position for the other process. `u_reopen` also increments the usage counter of the file; when one of the two processes closes (its copy of) the file, the counter is reset to the original value.

[468b] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁467b 468c▷
```
int mx_reopen (int mfd) {
  if (mfd < 0 || mfd ≥ MX_MAX_FILES)  return -1;   // wrong mfd number
  struct mx_filestat      *st    = &mx_status[mfd];
  struct int_minix2_inode *inode = st->int_inode;
  inode->refcount++;                               // increase reference count

  int new_mfd = mx_get_free_status_entry ();
  memcpy (&mx_status[new_mfd], &mx_status[mfd], sizeof (struct mx_filestat));

  return new_mfd;
}
```
Defines:
`mx_reopen`, used in chunks 425c and 468a.
Uses `int_minix2_inode` 459a, `memcpy` 596c, `mx_filestat` 460a, `mx_get_free_status_entry` 463a, `MX_MAX_FILES` 461a, and `mx_status` 461b.

The `mx_sync`$_{468c}$ function saves changes to the internal inode by writing it back to disk; after that it sets the `clean` flag.

[468c] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁468b 469c▷
```
int mx_sync (int device, int mfd) {
  if (mfd < 0 || mfd ≥ MX_MAX_FILES)  return -1;    // wrong mfd number
  struct mx_filestat      *st    = &mx_status[mfd];
  struct int_minix2_inode *inode = st->int_inode;
  if (inode == NULL)  return -1;                    // no open file

  if (inode->clean == 0) {
    int ext_ino = inode->ino;
    mx_write_inode (device, ext_ino, (struct minix2_inode*) inode);
    inode->clean = 1;   // now it is clean
  }
  return 0;
}
```
Uses `int_minix2_inode` 459a, `minix2_inode` 442a, `mx_filestat` 460a, `MX_MAX_FILES` 461a, `mx_status` 461b, `mx_write_inode` 452a, and `NULL` 46a.

12.6.6.4 `mx_lseek`

Seeking is a very simple operation: Since the file is open we know its size and the current read/write position; so

⟨*function prototypes* 45a⟩+≡ (44a) ◁468a 470a▷ [469a]
```
int mx_lseek (int mfd, int offset, int whence);
```

will only check if the request makes sense and then update the internal inode. As usual, we support the following three `SEEK_*` constants for the `whence` parameter which decide how the offset is to be interpreted:

⟨*public constants* 46a⟩+≡ (44a 48a) ◁460b 561a▷ [469b]
```
#define SEEK_SET 0    // absolute offset
#define SEEK_CUR 1    // relative offset
#define SEEK_END 2    // EOF plus offset
```
Defines:
`SEEK_CUR`, used in chunks 469c and 498a.
`SEEK_END`, used in chunks 293b, 469c, and 498a.
`SEEK_SET`, used in chunks 233b, 293, 294, 469c, and 498a.

When the file is opened in append mode, we must not change the position; otherwise we either set the position, add the offset to the current location or add it to the end of file position. For the last two cases negative values are OK.

⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁468c 470b▷ [469c]
```
int mx_lseek (int mfd, int offset, int whence) {
  if (mfd < 0 || mfd ≥ MX_MAX_FILES)  return -1;      // wrong mfd number
  struct mx_filestat *st = &mx_status[mfd];
  struct int_minix2_inode *inode = st->int_inode;
  if (inode == NULL)              return -1;          // no open file
  if (whence < 0 || whence > 2)   return -1;          // wrong lseek option
  if ((st->mode & O_APPEND) != 0) return st->pos;     // append mode, ignore lseek

  switch (whence) {
    case SEEK_SET: st->pos = offset;  break;          // set absolute
    case SEEK_CUR: st->pos += offset; break;          // relative to current loc.
    case SEEK_END: st->pos = inode->i_size + offset;  // relative to EOF
  };

  if (st->pos < 0)  st->pos = 0;                      // sanity check
  return st->pos;
}
```
Defines:
`mx_lseek`, used in chunks 418a and 469a.
Uses `int_minix2_inode` 459a, `mx_filestat` 460a, `MX_MAX_FILES` 461a, `mx_status` 461b, `NULL` 46a, `O_APPEND` 460b, `SEEK_CUR` 469b, `SEEK_END` 469b, and `SEEK_SET` 469b.

12.6.7 Reading and Writing

With all these preparations we can now approach the read and write operations which work on open files.

12.6.7.1 mx_read

We start with the function

[470a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁469a 471b▷
```
int mx_read (int mfd, void *buf, int nbyte);
```

that reads `nbyte` bytes from an open file identified by `mfd` into a buffer `buf`.

Since `mx_read`$_{470b}$ is a bit longer, we use a code chunk ⟨`mx_read` 470c⟩ for displaying the code:

[470b] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁469c 473a▷
```
int mx_read (int mfd, void *buf, int nbyte) {
  ⟨mx_read 470c⟩
}
```
Defines:
`mx_read`, used in chunks 414b and 470a.

We start with the usual variable initialization so that we have access to both the internal inode and the file status entry. If `mfd` has an invalid value or we attempt to read a file in write-only or append mode, we return −1 at once.

[470c] ⟨`mx_read` 470c⟩≡ (470b) 470d▷
```
if (mfd < 0 || mfd ≥ MX_MAX_FILES)  return -1;    // wrong mfd number

struct mx_filestat *st = &mx_status[mfd];
struct int_minix2_inode *inode = st->int_inode;
short device = inode->device;

if (inode == NULL)  return -1;                    // no open file
if (st->mode == O_WRONLY || st->mode == O_APPEND)
  return -1;                                      // reading is forbbiden
```
Uses `int_minix2_inode` 459a, `mx_filestat` 460a, `MX_MAX_FILES` 461a, `mx_status` 461b, `NULL` 46a, `O_APPEND` 460b, and `O_WRONLY` 460b.

Next we look at the current read/write position and determine which logical blocks of the file must be read—even if we want just a single byte from the file, we must read a whole block since that's the only way that we can access the hardware. (With "logical block" we mean the enumeration of the file's blocks. A file always starts with logical block 0 (unless it is empty).

Note that as a worst case, even reading two bytes can result in reading two blocks, if those bytes are placed precisely on a block boundary.

[470d] ⟨`mx_read` 470c⟩+≡ (470b) ◁470c 471a▷
```
int startbyte = st->pos;
if (startbyte ≥ inode->i_size) { return 0; }   // nothing to read
int endbyte = st->pos + nbyte - 1;
if (endbyte ≥ inode->i_size) {
  nbyte -= (endbyte - inode->i_size + 1);
  endbyte = inode->i_size - 1;
}
```

With `startbyte` and `endbyte` set, we can easily calculate the logical blocks:

⟨mx_read 470c⟩+≡ (470b) ◁470d 471c▷ [471a]
```
  int startblock = startbyte / BLOCK_SIZE;
  int endblock   = endbyte / BLOCK_SIZE;
  int curblock = startblock;
```
Uses BLOCK_SIZE 440a.

We need to loop over all the logical blocks and read them. In order to read a logical block curblock from the file we must find out where it is located on the device (i. e., find the physical block where it is stored); we put the lookup of that block number into a separate function

⟨*function prototypes* 45a⟩+≡ (44a) ◁470a 474b▷ [471b]
```
  int fileblocktozone (int device, int blockno, struct int_minix2_inode *inode);
```

that we will implement afterwards.

⟨mx_read 470c⟩+≡ (470b) ◁471a [471c]
```
  int read_bytes = 0;
  while (nbyte > 0) {
    int zone = fileblocktozone (device, curblock, inode);  // where is the block?
    if (zone == -1) {
      printf ("ERROR, fileblocktozone() = -1\n");
      return -1;
    };

    byte block[BLOCK_SIZE];  readblock (device, zone, (byte*) block);
    int offset, length;
    if (curblock == startblock) {
      offset = startbyte % BLOCK_SIZE;
      length = MIN (nbyte, BLOCK_SIZE - offset);
    } else {
      offset = 0;
      length = MIN (nbyte, BLOCK_SIZE);
    }
    memcpy (buf, block+offset, length);

    nbyte -= length;        buf   += length;
    read_bytes += length;   curblock++;
    st->pos += length;       // update current location in inode
  }

  return read_bytes;         // return the read bytes, might be != nbyte
```
Uses BLOCK_SIZE 440a, fileblocktozone 473a, memcpy 596c, MIN 471d, printf 601a, and readblock 506b.

This code uses the MIN$_{471d}$ macro that we have not defined yet:

⟨*macro definitions* 35a⟩+≡ (44a) ◁340a 597a▷ [471d]
```
  #define MIN(a,b) ((a) ≤ (b) ? (a) : (b))
  #define MAX(a,b) ((a) ≥ (b) ? (a) : (b))
```
Defines:
MIN, used in chunks 471c, 475c, 496d, and 497.

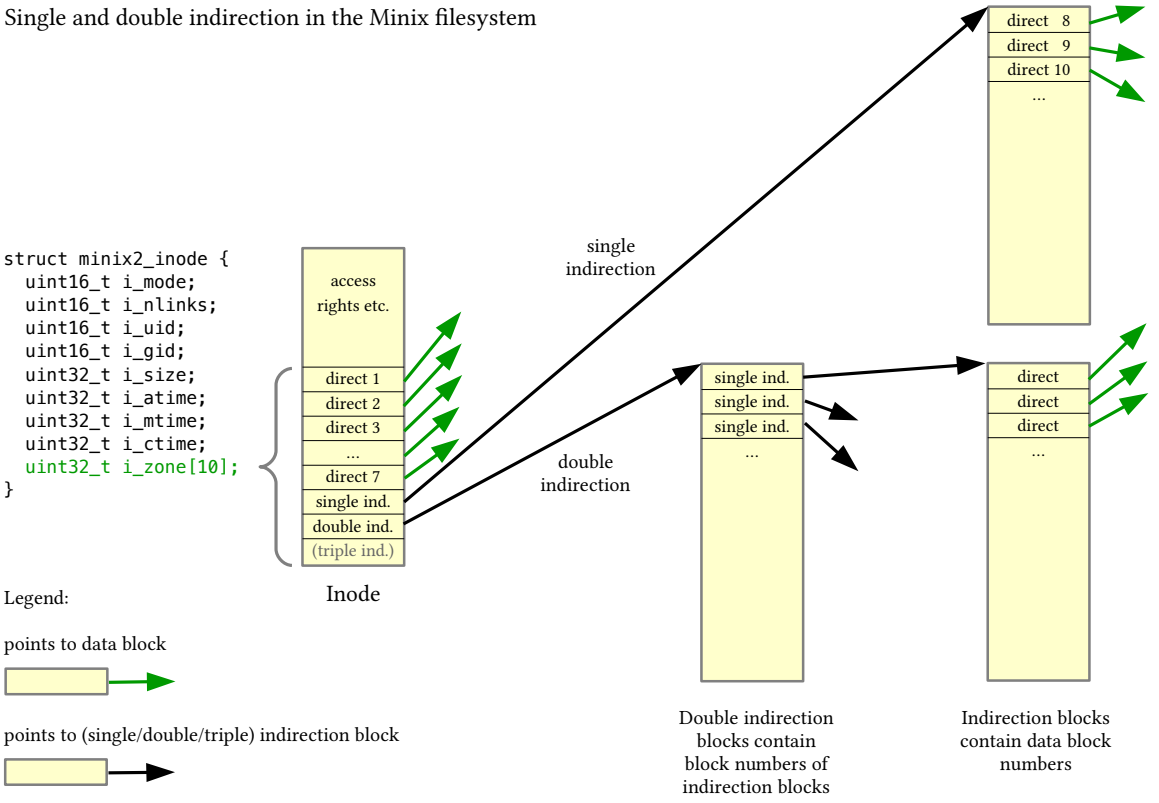


Figure 12.12: A Minix inode stores seven direct block numbers and two block numbers for single and double indirection blocks; `i_zone[9]` is unused.

indirection

Now we need to show how to translate a logical block number to a physical block number (or zone number). This may require looking at single and double indirection blocks. The Minix V2 filesystem uses four byte long integers as zone addresses, so one indirection block has space for `BLOCK_SIZE`$_{440a}$ `/ 4` = 256 such addresses. We'll define this number as a constant:

[472] ⟨*constants* 112a⟩+≡ (44a) ◁461a 494a▷

```
#define BLOCKADDRESSES_PER_BLOCK (BLOCK_SIZE / 4)
```

Defines:
`BLOCKADDRESSES_PER_BLOCK`, used in chunks 473a, 474a, 476b, and 477c.
Uses `BLOCK_SIZE` 440a.

Writing `fileblocktozone`$_{473a}$ is straightforward if we know how single and double indirection blocks are organized. Figure 12.12 once more shows how Minix uses indirect blocks. You have already seen a bigger version of that figure on page 82, here we only show the single and double indirection that we implement for Ulix; triple indirection was

not part of the original implementation in Minix either, but the inode entry `i_zone[9]` is reserved for triple indirection.

Thus, the physical block numbers of the first seven logical blocks (number 0–6) can be found directly in the inode. For the 256 blocks with numbers 7–262 we must load the indirection block (whose address is in `i_zone[7]`, and any block number beyond 262 requires us to first load the double indirection block (via `i_zone[8]`) and then search for the address of the right single indirection block, so locating such a block always requires reading two indirection blocks which is one of the reasons why it is so helpful to use a caching mechanism: When reading consecutive blocks beyond block number 262, the indirection blocks will remain in the cache once the first of those blocks has been accessed. caching

⟨ *minix filesystem implementation* 420c⟩+≡ (440b) ◁470b 474c▷ [473a]

```
int fileblocktozone (int device, int blockno, struct int_minix2_inode *inode) {
  int zone;  int *zone_ptr;  byte indirect_block[BLOCK_SIZE];
  if (blockno < 7) {
    // the first 7 zone numbers (0..6) are in the inode:
    zone = inode->i_zone[blockno];
  } else if (blockno ≥ 7 && blockno < 7+BLOCKADDRESSES_PER_BLOCK) {
    // inode holds the address of an indirection block
    ⟨fileblocktozone: single indirect 473b⟩
  } else {
    // inode holds the address of a double indirection block
    ⟨fileblocktozone: double indirect 474a⟩
  }
  return zone;
}
```

Defines:
`fileblocktozone`, used in chunks 471c, 475c, and 484e.
Uses `BLOCK_SIZE` 440a, `BLOCKADDRESSES_PER_BLOCK` 472, and `int_minix2_inode` 459a.

For the two indirection cases (singly indirect, doubly indirect) we provide two code chunks which are increasingly complex. Single indirection works like this:

⟨`fileblocktozone`: *single indirect* 473b⟩≡ (473a) [473b]

```
int indirect_zone = inode->i_zone[7];
if (indirect_zone == 0) {
  return -2;   // no indirection block found
}
readblock (device, indirect_zone, (byte *) indirect_block);
zone_ptr = (int *) indirect_block;
zone_ptr += (blockno - 7);
zone = *zone_ptr;
```

Uses `readblock` 506b.

Here we set `zone_ptr` to the start address of the loaded indirection block. Then we need to add an offset to find the right block number inside that block. We don't add `blockno` but `blockno - 7` because the first seven block addresses are already found in the inode and not repeated in the indirection block which starts with the block number of block 7.

Resolving a double indirection works similarly, but consists of two steps.

- First, `(blockno - 7 - BLOCKADDRESSES_PER_BLOCK`$_{472}$`) / BLOCKADDRESSES_PER_BLOCK`$_{472}$ is the index into the first indirection block—instead of `blockno - 7` (as in the single indirection case) we must also subtract `BLOCKADDRESSES_PER_BLOCK`$_{472}$ since the first `7 + BLOCKADDRESSES_PER_BLOCK`$_{472}$ blocks can be found via the direct addresses and the single indirection block. Then we also need to divide by `BLOCKADDRESSES_PER_BLOCK`$_{472}$ as each address in the first indirection block points to a whole block of addresses.
- In the second step, we take `(blockno - 7) % BLOCKADDRESSES_PER_BLOCK`$_{472}$ as an index into the second indirection block: Note that the equation

```
(blockno - 7 - BLOCKADDRESSES_PER_BLOCK) % BLOCKADDRESSES_PER_BLOCK
                                    == (blockno - 7) % BLOCKADDRESSES_PER_BLOCK
```

  holds; the left side is the original formula, but `n % n == 0` for all `n`.

[474a] ⟨`fileblocktozone`: *double indirect* 474a⟩≡ (473a)

```
int double_indirect_zone = inode->i_zone[8];
if (double_indirect_zone == 0) {
  return -2;  // no double indirection block found
}
readblock (device, double_indirect_zone, (byte *) indirect_block);
zone_ptr = (int *) indirect_block;
zone_ptr += (blockno - 7 - BLOCKADDRESSES_PER_BLOCK) / BLOCKADDRESSES_PER_BLOCK;
int indirect_zone = *zone_ptr;

readblock (device, indirect_zone, (byte *) indirect_block);
zone_ptr = (int *) indirect_block;
zone_ptr += (blockno - 7) % BLOCKADDRESSES_PER_BLOCK;
zone = *zone_ptr;
```

Uses `BLOCKADDRESSES_PER_BLOCK` 472 and `readblock` 506b.

12.6.7.2 `mx_write`

The `mx_write`$_{474c}$ function works similar to `mx_read`$_{470b}$, but it must also read blocks which are only partly modified so that writing the block does not erase the parts which are not modified.

[474b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁471b 476a▷

```
int mx_write (int mfd, void *buf, int nbyte);
```

The structure is the same as in `mx_read`$_{470b}$, and again we use a separate code chunk ⟨`mx_write` 475a⟩.

[474c] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁473a 476b▷

```
int mx_write (int mfd, void *buf, int nbyte) {
  ⟨mx_write 475a⟩
}
```

Defines:
`mx_write`, used in chunks 415a, 474b, 484b, and 487a.

We start with some checks:

⟨mx_write 475a⟩≡ (474c) 475b▷ [475a]
```
  if (mfd < 0 || mfd ≥ MX_MAX_FILES)   return -1;  // wrong mfd number
  struct mx_filestat      *st    = &mx_status[mfd];
  struct int_minix2_inode *inode = st->int_inode;
  short device = inode->device;
  if (inode == NULL)         return -1;            // no open file
  if (st->mode == O_RDONLY)  return -1;            // cannot write to read-only file
```
Uses int_minix2_inode 459a, mx_filestat 460a, MX_MAX_FILES 461a, mx_status 461b, NULL 46a, O_RDONLY 460b, open 429b, read 429b, and write 429b.

The calculation of start and end positions (and blocks) is the same as well, but without the checks: writing does not require the data to be available.

⟨mx_write 475a⟩+≡ (474c) ◁475a 475c▷ [475b]
```
  int startbyte  = st->pos;
  int endbyte    = st->pos + nbyte - 1;
  int startblock = startbyte / BLOCK_SIZE;
  int endblock   = endbyte / BLOCK_SIZE;
  int curblock   = startblock;
```
Uses BLOCK_SIZE 440a.

The code for actually writing the blocks is just a little more complex than that of the mx_read$_{470b}$ function:

⟨mx_write 475a⟩+≡ (474c) ◁475b [475c]
```
  byte block[BLOCK_SIZE];
  int offset, length;
  int written_bytes = 0;
  while (nbyte > 0) {
    int zone = fileblocktozone (device, curblock, inode);
    if (zone == -2 || zone == 0) {
      zone = mx_create_new_zone (device, curblock, inode); // block doesn't yet exist
    };
    if (zone == -1)  return -1;

    if (curblock == startblock) {
      offset = startbyte % BLOCK_SIZE;
      length = MIN (nbyte, BLOCK_SIZE - offset);
    } else {
      offset = 0;
      length = MIN (nbyte, BLOCK_SIZE);
    }

    if (offset != 0 || length != BLOCK_SIZE) {
      // writing a partial block -- read first!
      readblock (device, zone, (byte*) block);
    }
    memcpy (block+offset, buf, length);
    writeblock (device, zone, (byte*) block);
```

```
    nbyte -= length;          buf   += length;
    written_bytes += length;  curblock++;

    st->pos += length;
    if (st->pos > inode->i_size)  inode->i_size = st->pos;  // update size
  }

  inode->i_mtime = system_time;                              // update mtime
  mx_write_inode (device, inode->ino, (struct minix2_inode*) inode);
  return written_bytes;
```

Uses BLOCK_SIZE 440a, fileblocktozone 473a, memcpy 596c, MIN 471d, minix2_inode 442a, mx_create_new_zone 476b, mx_write_inode 452a, readblock 506b, system_time 338a, and writeblock 507c.

The function uses

[476a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁474b 478a▷

```
int mx_create_new_zone (int device, int blockno, struct int_minix2_inode *inode);
```

which requests a new block and inserts it in the inode's block list: The argument `blockno` is the logical block number (as seen from the file). In the function, `zone` is the physical block (zone) number that is assigned to the logical block.

[476b] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁474c 478b▷

```
int mx_create_new_zone (int device, int blockno, struct int_minix2_inode *inode) {
  int zone = mx_request_block (device);  // new data block
  if (zone == -1) {
    printf ("ERROR: cannot reserve data block; disk full\n");
    return -1;
  }
  int indirect_zone, double_indirect_zone;
  int *zone_ptr;
  byte indirect_block[BLOCK_SIZE] = { 0 };
  byte double_indirect_block[BLOCK_SIZE] = { 0 };
  if (blockno < 7) {
    ⟨create new zone: direct 477a⟩
  } else if (blockno ≥ 7 && blockno < 7+BLOCKADDRESSES_PER_BLOCK) {
    ⟨create new zone: single indirect 477b⟩
  } else {
    ⟨create new zone: double indirect 477c⟩
  }
  return zone;
}
```

Defines:
mx_create_new_zone, used in chunks 475c and 476a.
Uses BLOCK_SIZE 440a, BLOCKADDRESSES_PER_BLOCK 472, int_minix2_inode 459a, mx_request_block 448, and printf 601a.

with the following three cases for direct, indirect and double indirect blocks—they are similar to the three cases in the fileblocktozone[473a] function. If we deal with a direct zone, we can directly enter the zone number in the inode:

⟨*create new zone: direct* 477a⟩≡ (476b) [477a]
```
// the first 7 zone numbers (0..6) are in the inode:
inode->i_zone[blockno] = zone;
```

In case of single indirection, we may have to create the indirection block which requires another call of `mx_request_block`$_{448}$.

⟨*create new zone: single indirect* 477b⟩≡ (476b) [477b]
```
indirect_zone = inode->i_zone[7];

// if there is no indirection block yet, create it
if (indirect_zone == 0) {
  // no indirection block found
  indirect_zone = mx_request_block (device);  // data block for indirections
  if (indirect_zone == -1) {
    mx_clear_zmap_bit (device, zone);         // undo reservation of data block
    return -1;
  }
  inode->i_zone[7] = indirect_zone;
} else {
  // indirection block exists: read it
  readblock (device, indirect_zone, (byte *) indirect_block);
}

zone_ptr = (int *) indirect_block;
zone_ptr += (blockno - 7);
*zone_ptr = zone;   // write information about new data block
writeblock (device, indirect_zone, (byte *) indirect_block);
```
Uses `mx_clear_zmap_bit` 446, `mx_request_block` 448, `read` 429b, `readblock` 506b, and `writeblock` 507c.

Finally, in the case of double indirection, the worst that can happen is that we need to create both the first and the second indirection block. In both cases (single and double indirection) we use the same offset calculations as in `fileblocktozone`$_{473a}$`()`.

⟨*create new zone: double indirect* 477c⟩≡ (476b) [477c]
```
double_indirect_zone = inode->i_zone[8];

// if there is no double indirection block yet, create it
if (double_indirect_zone == 0) {
  // no double indirection block found
  double_indirect_zone = mx_request_block (device);  // data block for 2x indir.
  if (double_indirect_zone == -1) {
    mx_clear_zmap_bit (device, zone);    // undo reservation of data block
    return -1;
  }
  inode->i_zone[8] = double_indirect_zone;
} else {
  // indirection block exists: read it
  readblock (device, double_indirect_zone, (byte *) double_indirect_block);
}
```

```
  zone_ptr = (int *) double_indirect_block;
  zone_ptr += (blockno - 7 - BLOCKADDRESSES_PER_BLOCK) / BLOCKADDRESSES_PER_BLOCK;
  indirect_zone = *zone_ptr;

  // if there is no indirection block yet, create it
  if (indirect_zone == 0) {
    // no indirection block found
    indirect_zone = mx_request_block (device);  // data block for indirections
    if (indirect_zone == -1) {
      mx_clear_zmap_bit (device, zone);   // undo reservation of data block
      return -1;
    }

    // write to first level indirection block
    *zone_ptr = indirect_zone;
    writeblock (device, double_indirect_zone, (byte *) double_indirect_block);
  } else {
    // indirection block exists: read it
    readblock (device, indirect_zone, (byte *) indirect_block);
  }

  zone_ptr = (int *) indirect_block;
  zone_ptr += (blockno - 7) % BLOCKADDRESSES_PER_BLOCK;

  *zone_ptr = zone;   // write information about new data block
  writeblock (device, indirect_zone, (byte *) indirect_block);
```

Uses BLOCKADDRESSES_PER_BLOCK 472, mx_clear_zmap_bit 446, mx_request_block 448, readblock 506b, and writeblock 507c.

What's still missing is a way to create a new (empty) file. The function

[478a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁476a 479a▷

```
int mx_creat_empty_file (int device, const char *path, int mode);
```

requests a new inode, fills it with the appropriate data and writes it to disk. Then it writes a link into the directory that will hold the file.

[478b] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁476b 479b▷

```
int mx_creat_empty_file (int device, const char *path, int mode) {
  int inodenr = mx_request_inode (device);
  struct minix2_inode inode = { 0 };
  inode.i_size = 0;
  inode.i_atime = inode.i_ctime = inode.i_mtime = system_time;
  inode.i_uid = thread_table[current_task].uid;
  inode.i_gid = thread_table[current_task].gid;
  inode.i_nlinks = 0;
  inode.i_mode = S_IFREG | mode;
  mx_write_inode (device, inodenr, &inode);
  mx_write_link (device, inodenr, path);  // create directory entry
  return inodenr;
}
```

Defines:
mx_creat_empty_file, used in chunks 464c and 478a.
Uses current_task 192c, gid 573a, minix2_inode 442a, mx_request_inode 448, mx_write_inode 452a, mx_write_link 456, S_IFREG 457c, system_time 338a, thread_table 176b, and uid 573a.

12.6.8 Linking and Unlinking

Unix systems have no delete or erase system calls for files—instead there is an unlink system call which removes a directory entry (it deletes the link from a filename to an inode in that directory). Only if the last name was deleted, unlink will also delete the file, which means freeing all data blocks and the inode. file deletion

The opposite operation is creating a hardlink: This creates a new name (a new link from a filename to an inode in some directory).

Both operations modify an inode's *link count*: That is where the filesystem keeps track of how many names were given to a file. link count

We will start by showing two helper functions which can check whether a file exists and whether it is a directory, then we implement the link operation since it is the simpler one (of link and unlink).

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁478a 480b▷ [479a]
```
boolean mx_file_exists (int device, const char *path);
boolean mx_file_is_directory (int device, const char *path);
int mx_link (int device, const char *path1, const char *path2);
```

⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁478b 480a▷ [479b]
```
boolean mx_file_exists (int device, const char *path) {
  if (mx_pathname_to_ino (device, path) == -1)  return false;
  return true;
}

boolean mx_file_is_directory (int device, const char *path) {
  int ino = mx_pathname_to_ino (device, path);
  if (ino == -1)  return false;                    // does not exist
  struct minix2_inode inode;
  mx_read_inode (device, ino, &inode);
  if ((inode.i_mode & S_IFDIR) == 0)  return false;  // no directory
  return true;
}
```
Defines:
mx_file_exists, used in chunks 456 and 480.
mx_file_is_directory, used in chunk 480a.
Uses minix2_inode 442a, mx_pathname_to_ino 461d, mx_read_inode 451b, and S_IFDIR 457c.

12.6.8.1 mx_link

The mx_link$_{480a}$ function checks both paths and writes the link. Note that this implementation lets users create hard links of directories which is normally forbidden. We do check

the condition but only print a warning because it is interesting to "play" with hard-linked directories.

[480a] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁479b 480c▷

```
int mx_link (int device, const char *path1, const char *path2) {
  // check path1 exists
  if (!mx_file_exists (device, path1))  return -1;  // does not exist

  // check path1 is not a directory
  if (mx_file_is_directory (device, path1)) {
    printf ("ln: warning: %s is a directory. This option will be removed.\n");
  }

  // check path2 does NOT exist
  if (mx_file_exists (device, path2)) {
    return -1;   // path2 already exists; no forced linking
  }

  // everything ok now
  int ino = mx_pathname_to_ino (device, path1);
  mx_write_link (device, ino, path2);   // updates link count
  return 0;
}
```

Defines:
`mx_link`, used in chunks 419a and 479a.
Uses `mx_file_exists` 479b, `mx_file_is_directory` 479b, `mx_pathname_to_ino` 461d, `mx_write_link` 456, and `printf` 601a.

12.6.8.2 `mx_unlink`

Unlinking is similar as long as at least one filename (one link) remains. If none remains, the data blocks of the file and the inode must be freed.

[480b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁479a 484a▷

```
int mx_unlink (int device, const char *path);
```

[480c] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁480a 484b▷

```
int mx_unlink (int device, const char *path) {
  char ind_block[BLOCK_SIZE], double_ind_block[BLOCK_SIZE];
  struct minix_dir_entry dentry;

  // check if path exists
  if (!mx_file_exists (device, path)) {
    printf ("rm: file does not exist\n");
    return -1;                          // error: path does not exist
  }

  // get inodes of file and directory
  int inodenr = mx_pathname_to_ino (device, path);
  struct minix2_inode inode;
```

```
    mx_read_inode (device, inodenr, &inode);
    char dir[256], base[30];
    splitpath (path, dir, base);            // split path into dir and base
    int dir_inodenr = mx_pathname_to_ino (device, dir);

    // delete entry in directory
    boolean found = false;
    for (int i = 0;  i < 32*7;  i++) {
      mx_read_dir_entry (device, dir_inodenr, i, &dentry);
      if ( dentry.inode==inodenr && strequal (dentry.name, base) ) {
        dentry.inode = 0;
        memset (dentry.name, 0, 30);
        found = true;
        mx_write_dir_entry (device, dir_inodenr, i, &dentry);
        break;  // search finished
      }
    }
    if (found==false) { return -1; }  // error: not found in directory

    inode.i_nlinks--;   // one name less
    if (inode.i_nlinks == 0) { ⟨free this inode 481a⟩ }
    mx_write_inode (device, inodenr, &inode);
    return 0;
  }
```

Defines:
mx_unlink, used in chunks 418b and 480b.
Uses BLOCK_SIZE 440a, memset 596c, minix2_inode 442a, minix_dir_entry 452b, mx_file_exists 479b, mx_pathname_to_ino 461d, mx_read_dir_entry 453b, mx_read_inode 451b, mx_write_dir_entry 453b, mx_write_inode 452a, printf 601a, splitpath 455a, and strequal 596a.

We must take care of the case when the last link has been removed—then we have an inode with a reference count of 0, and that means, the file is truly to be deleted: We need to mark its data blocks as free (including an indirection block, if it exists) and also mark the inode as free. We show this action in four separate steps:

⟨*free this inode* 481a⟩≡ (480c) [481a]
⟨*free this inode: (1) direct blocks* 481b⟩
⟨*free this inode: (2) single indirect blocks* 482a⟩
⟨*free this inode: (3) double indirect blocks* 482b⟩
⟨*free this inode: (4) inode itself* 483⟩

⟨*free this inode: (1) direct blocks* 481b⟩≡ (481a) [481b]

```
  for (int i = 0;  i < 7;  i++) {
    if (inode.i_zone[i] != 0) {
      mx_clear_zmap_bit (device, inode.i_zone[i] - 33);  // mark data block as free
      inode.i_zone[i] = 0;
    }
  }
```

Uses mx_clear_zmap_bit 446.

For single indirection, we clear both the zone map entries for the indirection block itself and all the blocks it points to.

[482a] ⟨*free this inode: (2) single indirect blocks* 482a⟩≡ (481a)

```
if (inode.i_zone[7] != 0) {
  readblock (device, inode.i_zone[7], ind_block);
  unsigned int *zoneno;
  zoneno = (unsigned int*)ind_block;      // cast to uint*
  int count = 0;
  while (*zoneno != 0 && count < 256) {
    mx_clear_zmap_bit (device, *zoneno - 33);        // mark data block as free
    zoneno++;
    count++;
  }
  mx_clear_zmap_bit (device, inode.i_zone[7] - 33);  // mark indir. block as free
  inode.i_zone[7] = 0;
}
```

Uses `mx_clear_zmap_bit` 446 and `readblock` 506b.

And in case of double indirection, there are even more blocks to mark as free:

[482b] ⟨*free this inode: (3) double indirect blocks* 482b⟩≡ (481a)

```
if (inode.i_zone[8] != 0) {
  readblock (device, inode.i_zone[8], double_ind_block);

  unsigned int *ind_zoneno;
  ind_zoneno = (unsigned int*)double_ind_block;     // cast to uint*

  int count1 = 0;
  int count2;
  while (*ind_zoneno != 0 && count1 < 256) {
    readblock (device, *ind_zoneno, ind_block);
    unsigned int *zoneno;
    zoneno = (unsigned int*)ind_block;              // cast to uint*
    count2 = 0;
    while (*zoneno != 0 && count2 < 256) {
      mx_clear_zmap_bit (device, *zoneno - 33);     // mark data block as free
      zoneno++;
      count2++;
    }
    mx_clear_zmap_bit (device, *ind_zoneno - 33);   // mark indir. block as free
    ind_zoneno++;
    count1++;
  }

  mx_clear_zmap_bit (device, inode.i_zone[8] - 33); // mark double ind. block free
  inode.i_zone[8] = 0;
}
```

Uses `mx_clear_zmap_bit` 446 and `readblock` 506b.

Last, we free the inode:

⟨*free this inode: (4) inode itself* 483⟩≡ (481a) [483]
```
mx_clear_imap_bit (device, inodenr);
```
Uses mx_clear_imap_bit 446.

12.6.8.3 mx_symlink

Unix systems also know a second type of link, the *symbolic* or *soft link* (short: *symlink*). While this name reminds of the (hard) links for which we have just provided the implementation, and even the same Unix tool (`ln`) handles both hard and soft links, these two link types have nothing in common. symbolic link

A symbolic link is a special file which contains a path name as data. When you disable the treatment of symbolic links on a Unix system and try to read such a file, all you get is the stored path name: Figure 12.13 shows how Ulix displayed the content of a symbolic link before symlinks were implemented: `ulix.symlink`$_{429b}$ was created by executing

```
ln -s ulix.h ulix.symlink
```

on a Linux system which had loop-mounted the Minix filesystem: As you can see from the letter `l` at the start of the file entry, Ulix already recognized the file type but did not know better than to output the contents of the file's first data block.

```
Ulix-i386 0.08                              Build: Mon Jul 15 22:38:00 CEST 2013  -
Physical RAM (64 MB) mapped to 0xD0000000-0xD3FFFFFF.
Initialized ten terminals (press [Alt-1] to [Alt-0])
FDC: fda is 1.44M, fdb is 1.44M
Starting five shells on tty0..tty4. Type exit to quit.

Ulix Usermode Shell. Commands: help, ps, fork, ls, cat, head, cp, diff, sh,
hexdump, kill, loop, test, brk, cd, ln, rm, pwd, touch, exit
Press [Shift+Esc] to launch kernel mode shell (reboot to get back here)
esser@ulix[2]:/$ ls
directory / is in inode 1
 1 drwxr-xr-x  3 1000 1000    192 .
 1 drwxr-xr-x  3 1000 1000    192 ..
 2 -rwxr-xr-x  1 1000 1000  16384 sh
 3 -rw-r--r--  1 1000 1000   3794 ulix.h
 4 drwxr-xr-x  2 1000 1000    128 sub
 6 lrwxrwxrwx  1 1000 1000      6 ulix.symlink
esser@ulix[2]:/$ hexdump ulix.symlink
absolute path: /ulix.symlink
0000  75 6c 69 78 2e 68                                 ulix.h
esser@ulix[2]:/$ _



Ulix-i386 0.08                                  tty0  FF=3b70  AS=0001  00:00:21
```

Figure 12.13: Ulix version 0.08 displays the contents of a symbolic link—in that version symbolic links were not yet implemented. (The image was inverted for better readability.)

As you can see, creating a symlink is easy: We just write the link target into a data block and mark the file as symlink:

[484a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁480b 484d▷
```
int mx_symlink (int device, char *path1, char *path2);
```

[484b] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁480c 484c▷
```
int mx_symlink (int device, char *path1, char *path2) {
  int fd = mx_open (device, path2, O_WRONLY | O_CREAT);
  if (fd == -1)  return -1;    // error: cannot create file
  mx_write (fd, path1, strlen (path1));
  mx_close (fd);
```
Defines:
`mx_symlink`, used in chunks 419b and 484a.
Uses `mx_close` 467b, `mx_open` 464b, `mx_write` 474c, `O_CREAT` 460b, `O_WRONLY` 460b, and `strlen` 594a.

(Note that we do *not* write a terminating `\0` character: The link target need not be terminated, the length of the filename is simply the symlink's file size.)

We're not finished yet—now we have a regular file with the link target in the first data block. We need to turn it into a symlink:

[484c] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁484b 484e▷
```
  int inode_nr = mx_pathname_to_ino (device, path2);
  struct minix2_inode inode;
  mx_read_inode (device, inode_nr, &inode);
  inode.i_mode = S_IFLNK | 0777;
  mx_write_inode (device, inode_nr, &inode);
  return 0;  // OK.
}
```
Uses `minix2_inode` 442a, `mx_pathname_to_ino` 461d, `mx_read_inode` 451b, `mx_write_inode` 452a, and `S_IFLNK` 457c.

We've set the symlink's access rights to 0777 (`rwxrwxrwx`) which is the default value on Linux machines. The rights do not matter much anyway since reading, writing or executing the linked file requires the target to grant the needed access permissions.

12.6.9 Truncating Files

Sometimes is is necessary to truncate a file, i. e., to reduce its file size by cutting off everything after a given offset. A special case is deleting all the content (setting the file size to 0). For emptying the file, we could simply delete and recreate it, but that might give the new version a different inode number, and also that is impossible with an open file.

On other Unix systems, the truncate functions can also grow a file (by supplying a `length` argument that is larger than the current size); we do not support this feature.

[484d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁484a 486▷
```
int mx_ftruncate (int mfd, int length);
```

[484e] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁484c 487a▷
```
int mx_ftruncate (int mfd, int length) {
  if (mfd < 0 || mfd ≥ MX_MAX_FILES)  return -1;    // wrong mfd number
  struct mx_filestat      *st    = &mx_status[mfd];
  struct int_minix2_inode *inode = st->int_inode;
  if (inode == NULL)  return -1;                    // no open file
```

```
  short device = inode->device;

  if (inode->i_size ≤ length)  return -1;          // attempt to grow the file

  // calculate blocks to delete
  int last_kept_byte = length - 1;
  int firstblock;
  if (length == 0)  firstblock = 0;
  else              firstblock = last_kept_byte / BLOCK_SIZE + 1;
  int lastblock  = inode->i_size / BLOCK_SIZE - 1;

  if (lastblock ≥ firstblock) {    // any blocks to delete?
    for (int i = firstblock;  i ≤ lastblock; i++) {
      // delete block
      int zone = fileblocktozone (device, i, inode);
      mx_clear_zmap_bit (device, zone);
    }
  }

  // check indirection blocks
  if (lastblock > 6   && inode->i_zone[7] != 0) {
    ⟨ mx_ftruncate: free single indirection block ⟩
  }
  if (lastblock > 262 && inode->i_zone[8] != 0) {
    ⟨ mx_ftruncate: free double indirection block ⟩
  }

  // reset size and write changed inode
  inode->i_size = length;
  inode->clean  = false;    // inode was changed
  return 0;
}
```

Defines:
mx_ftruncate, used in chunks 420b and 484d.
Uses BLOCK_SIZE 440a, fileblocktozone 473a, int_minix2_inode 459a, mx_clear_zmap_bit 446, mx_filestat 460a, MX_MAX_FILES 461a, mx_status 461b, and NULL 46a.

We do not implement the ⟨ mx_ftruncate: *free single indirection block* ⟩ and ⟨ mx_ftruncate: *free double indirection block* ⟩ code chunks since they are basically a rewrite of corresponding chunks in fileblocktozone$_{473a}$. Instead of looking up a zone number, it must be set to 0. Thus, when we truncate a file, single and double indirection blocks remain in use (and linked by the inode); they will however be destroyed when the file is finally deleted, and they will also be reused when the file grows again.

12.6.10 Making and Removing Directories

What's left is code for creating and deleting directories. Similar to symlinks, directories are just a special type of file, and we already know how to modify existing directories.

Deleting a directory means to free the data blocks that had been used by it and to remove its entry in the super-directory (the directory which is one step closer to the filesystem root and contains it)—this task is already handled by the `mx_unlink`$_{480c}$ function, so we need no further code in the kernel. Actually, we could provide an `mx_rmdir`$_{488a}$ function which is simpler than `mx_unlink`$_{480c}$ since directories must not be hard-linked: if we remove a directory, we always remove its inode.

For making a directory, we could do this as the logical three-step-procedure that is involded:

- create the (empty) directory,
- within the new directory, create a hard link `.` to itself,
- and also create a hard link `..`—either to the super-directory or to `/` in case of the root directory `/`. But the kernel will never create a root directory (that's handled by the user mode tool `mkfs.minix`)

Also, all new directories look identical except for the two hard links `.` and `..`, so we will keep our task simple by defining what the contents of a new directory look like (data-bock-wise) and how to update the inode numbers in the `.` and `..` entries.

The "`.`" character (dot) has the ASCII value 46, or 0x2E in hexadecimal code. A hexdump of the data area of a Minix filesystem shows the following contents for an empty directory:

```
$ hexdump -C /tmp/minixdata.img
[...]
00013000  08 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00013010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00013020  01 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00013030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
[...]
```

This shows a directory on the top level with inode number 8. The first two lines contain the directory entry for `.` (pointing to itself: `08 00`) and the last two lines hold the directory entry for `..` (pointing to the root directory which has inode number 1 (`01 00`).

The size of such an entry is 64:

```
$ ls -ld /mnt/minix/empty
drwxr-xr-x 2 esser esser 64 Jul 17 16:13 /mnt/minix/empty
```

The code for creating a directory with

[486] ⟨*function prototypes* 45a⟩+≡ (44a) ◁484d 487b▷

```
int mx_mkdir (int device, const char *path, int mode);
```

is as follows:

⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁484e 488a▷ [487a]

```
int mx_mkdir (int device, const char *path, int mode) {
  struct minix_dir_entry entry = { 0 };
  char dir[256]; char base[256];
  int dir_inodenr, new_inodenr;
  struct minix2_inode inode;

  splitpath (path, dir, base);                    // split path into dir and base
  dir_inodenr = mx_pathname_to_ino (device, dir);

  // create new file
  int fd = mx_open (device, path, O_CREAT | O_WRONLY); mx_close (fd);
  new_inodenr = mx_pathname_to_ino (device, path);

  // enter "." and ".." inode numbers
  fd = mx_open (device, path, O_WRONLY);
  memset (&entry, 0, 32); memcpy (entry.name, ".", 2);  entry.inode = new_inodenr;
  mx_write (fd, &entry, 32);
  memset (&entry, 0, 32); memcpy (entry.name, "..", 3); entry.inode = dir_inodenr;
  mx_write (fd, &entry, 32);
  mx_close (fd);

  // fix inode (make it type directory, set nlinks to 2)
  mx_read_inode (device, new_inodenr, &inode);
  inode.i_mode = S_IFDIR | (mode & 0777);          // set mode
  inode.i_uid = thread_table[current_task].euid;   // set UID
  inode.i_gid = thread_table[current_task].egid;   // set GID
  inode.i_nlinks = 2;
  mx_write_inode (device, new_inodenr, &inode);

  // update link count of directory above
  mx_read_inode (device, dir_inodenr, &inode);
  inode.i_nlinks++;
  mx_write_inode (device, dir_inodenr, &inode);
  return 0;
}
```

Defines:
mx_mkdir, used in chunks 422a and 486.
Uses current_task 192c, egid 573a, euid 573a, memcpy 596c, memset 596c, minix2_inode 442a, minix_dir_entry 452b, mx_close 467b, mx_open 464b, mx_pathname_to_ino 461d, mx_read_inode 451b, mx_write 474c, mx_write_inode 452a, O_CREAT 460b, O_WRONLY 460b, S_IFDIR 457c, splitpath 455a, and thread_table 176b.

Deleting a directory via

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁486 488b▷ [487b]

```
int mx_rmdir (int device, const char *fullpath, const char *path);
```

is only allowed if `.` and `..` are its only entries. If so, we first unlink those two and then remove the directory (file). Note that the function expects two path arguments, the full

(global) path and the device-local path. Our implementation does not use the device-local path, but new implementations of similar `*_rmdir` functions for other filesystems might do so.

[488a] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁487a 489a▷

```
int mx_rmdir (int device, const char *fullpath, const char *path) {
  struct minix_dir_entry entry;
  char abspath[256]; char dir[256]; char base[256]; char cwd[256];
  int dir_inodenr, new_inodenr;
  struct minix2_inode inode;

  // check relative/absolute path
  if (*path != '/')  relpath_to_abspath (fullpath, abspath);
  else               strncpy (abspath, fullpath, 256);

  // check if directory exists
  int fd = u_open ((char*)fullpath, O_RDONLY, 0);
  if (fd == -1) { return -1; }
  u_close (fd);

  // split path into dir and base
  splitpath (abspath, dir, base);

  // save current working directory
  u_getcwd ((char*)cwd, 256);

  if (mx_directory_is_empty (device, path)) {
    if (u_chdir (dir) == -1) { return -1; }
    if (u_unlink (".")      == -1) { u_chdir (cwd); return -1; }
    if (u_unlink ("..")     == -1) { u_chdir (cwd); return -1; }
    if (u_unlink (fullpath) == -1) { u_chdir (cwd); return -1; }

    u_chdir (cwd);   // restore old current working directory
    return 0;
  }
  printf ("Directory not empty\n");
  return -1;   // not empty
}
```

Defines:
`mx_rmdir`, used in chunks 422a and 487b.
Uses `cwd`, `minix2_inode` 442a, `minix_dir_entry` 452b, `mx_directory_is_empty` 489a, `O_RDONLY` 460b, `printf` 601a, `relpath_to_abspath` 412b, `splitpath` 455a, `strncpy` 594b, `u_chdir` 432e, `u_close` 418a, `u_getcwd` 432e, `u_open` 412c, and `u_unlink` 418b.

We still need the function `mx_directory_is_empty`$_{489a}$ which could check the file size of a directory: if it is 64 then the directory should contain only the `.` and `..` entries. Our

[488b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁487b 489c▷

```
boolean mx_directory_is_empty (int device, const char *path);
```

function expects that the provided `path` argument is already a (device-local) absolute path, i. e., it starts with a slash. It does not use the size check but queries individual directory entries instead because we don't always update the directory size when we delete files.

⟨ *minix filesystem implementation* 420c ⟩+≡ (440b) ◁488a 490a▷ [489a]

```
boolean mx_directory_is_empty (int device, const char *path) {
  int count = 0;  // number of entries
  struct dir_entry d = { 0 };
  for (int i = 0;  i < 32 * 7;  i++) {
    if (mx_getdent (device, path, i, &d) == 0 && d.inode != 0) count++;
  }
  return (count == 2);
}
```

Defines:
`mx_directory_is_empty`, used in chunk 488.
Uses `dir_entry` 490b and `mx_getdent` 490d.

The function uses `mx_getdent`$_{490d}$`()` which we present in the following section.

12.6.11 Listing a Directory

In order to retrieve the information stored in the inode of a file, all Unix systems offer a `stat`$_{429b}$ function which fills a status structure with content. What that structure looks like depends on the flavor of Unix; for ULIX we use the `struct stat`$_{429b}$ definition as it is shown in the Linux man page `stat(2)`, but without the `st_blksize` and `st_blocks` entries:

⟨ *public type definitions* 142a ⟩+≡ (44a 48a) ◁369a 490b▷ [489b]

```
struct stat {
  unsigned int   st_dev;     // ID of device containing file
  unsigned short st_ino;     // inode number
  unsigned short st_mode;    // protection
  unsigned short st_nlink;   // number of hard links
  unsigned short st_uid;     // user ID of owner
  unsigned short st_gid;     // group ID of owner
  unsigned short st_rdev;    // device ID (if special file)
  unsigned int   st_size;    // total size, in bytes
  unsigned int   st_atime;   // time of last access
  unsigned int   st_mtime;   // time of last modification
  unsigned int   st_ctime;   // time of last status change
};
```

Defines:
`stat`, used in chunks 420c, 421d, 426b, 432e, 489, 490, 499, 576, 577c, and 608a.

We have not prefixed the type name with `mx_` because we use it for all supported filesystems. However, there are several `*_stat` functions that retrieve the data—one for each supported filesystem. As part of the Minix filesystem implementation, we provide the

⟨ *function prototypes* 45a ⟩+≡ (44a) ◁488b 490c▷ [489c]

```
int mx_stat (int device, const char *path, struct stat *buf);
```

function which simply locates the inode and copies the data into a `struct stat`$_{429b}$ variable:

[490a] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁489a 490d▷

```
int mx_stat (int device, const char *path, struct stat *buf) {
  struct minix2_inode inode;
  int ino = mx_pathname_to_ino (device, path);
  if (ino == -1) return -1;                              // error
  buf->st_dev   = device;            buf->st_rdev  = 0;
  buf->st_ino   = ino;
  mx_read_inode (device, ino, &inode);                   // read the inode
  buf->st_mode  = inode.i_mode;      buf->st_nlink = inode.i_nlinks;
  buf->st_uid   = inode.i_uid;       buf->st_gid   = inode.i_gid;
  buf->st_size  = inode.i_size;      buf->st_atime = inode.i_atime;
  buf->st_ctime = inode.i_ctime;     buf->st_mtime = inode.i_mtime;
  return 0;
}
```

Defines:
`mx_stat`, used in chunks 421d and 490d.
Uses `minix2_inode` 442a, `mx_pathname_to_ino` 461d, `mx_read_inode` 451b, `read` 429b, and `stat` 429b 489b.

We define the generic directory entry structure `struct dir_entry`$_{490b}$ that is similar to the Minix structure `struct minix_dir_entry`$_{452b}$ but allows longer filenames.

[490b] ⟨*public type definitions* 142a⟩+≡ (44a 48a) ◁489b 491a▷

```
struct dir_entry {
  word inode;              // inode number
  byte filename[64];       // filename
};
```

Defines:
`dir_entry`, used in chunks 422c, 426b, 429b, 489a, 490d, and 500.

The function

[490c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁489c 491b▷

```
int mx_getdent (int device, const char *path, int index, struct dir_entry *buf);
```

fills a `struct dir_entry`$_{490b}$ buffer with the data found in the Minix inode on disk via the `mx_read_dir_entry`$_{453b}$ function.

[490d] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁490a 491c▷

```
int mx_getdent (int device, const char *path, int index, struct dir_entry *buf) {
  struct minix_dir_entry d;
  struct stat s;

  int ret = mx_stat (device, path, &s);
  if (ret == -1) return -1;             // error does not exist

  if (index*32 ≥ s.st_size) return -1; //  index out of bounds

  int ino = mx_pathname_to_ino (device, path);
  if (ino == -1) return -1;             // error: not a directory

  ret = mx_read_dir_entry (device, ino, index, &d);
  if (ret == -1) return -1;             // error: no such entry in directory
```

```
    buf->inode = d.inode;
    strncpy ((char*)(buf->filename), d.name, 30);
    d.name[30] = 0;                          // terminate string
    return 0;                                // success
  }
```

Defines:
mx_getdent, used in chunks 422c, 489a, and 490c.
Uses dir_entry 490b, minix_dir_entry 452b, mx_pathname_to_ino 461d, mx_read_dir_entry 453b, mx_stat 490a, stat 429b 489b, and strncpy 594b.

12.6.12 Filesystem Information: `df`

In order to implement a `df` (disk free) application we need a method to query the number of free blocks on a filesystem. As we will only support this for the Minix filesystem, we do not provide a generic layer (as part of the virtual filesystem) but directly write a Minix-specific function. It will fill `struct diskfree_query`$_{491a}$ structures:

⟨*public type definitions* 142a⟩+≡ (44a 48a) ◁490b [491a]

```
  struct diskfree_query {
    int device;          // device ID (is set before calling mx_diskfree)
    int size;            // size of filesystem, in blocks
    int used;            // number of used blocks
    int free;            // number of free blocks (redundant; == size-used)
    char name[10];       // name (such as "/dev/hda")
    char mount[256];     // mount point
    char fstype[10];     // filesystem name, e.g. "minix"
  };
```

Defines:
diskfree_query, used in chunks 491–93.

The goal is to implement

⟨*function prototypes* 45a⟩+≡ (44a) ◁490c 494c▷ [491b]

```
  void mx_diskfree (struct diskfree_query *query);
```

Uses diskfree_query 491a and mx_diskfree 492.

which will use the helper function

⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁490d 492▷ [491c]

```
  int count_zeros (byte *block, int maxcount) {
    int count = 0;
    for (int i = 0;  i < (maxcount+7)/8;  i++) {
      if (block[i] == 0) { count += 8; }
      else {
        for (int j = 0;  j < 8;  j++) {
          if (i*8 + j < maxcount && (block[i] >> j) % 2 == 0) count++;
        }
      }
    }
    return count;
  }
```

Defines:
count_zeros, used in chunk 492.

which counts the number of zero bits in a given block, but only up to the bit position specified by the maxcount parameter. (We can slightly optimize the counting by checking whether a byte is 0, in that case we can add 8 to the counter; this assumes that maxcount is always a multiple of 8.)

mx_diskfree$_{492}$ takes the device element of the structure as an argument, reads all zone map blocks of that device and counts the contained zeros. That gives us the number of free blocks. The other values are taken from the mount table or the superblock (or we calculate them).

[492] ⟨*minix filesystem implementation* 420c⟩+≡ (440b) ◁491c

```
void mx_diskfree (struct diskfree_query *query) {
  int device = query->device;
  struct minix_superblock sblock;
  char block[1024];
  query->size = mx_query_superblock (device, MX_SB_ZONES);
  unsigned int nblocks = mx_query_superblock (device, MX_SB_ZONES);
  unsigned int zmap_start = 2 + mx_query_superblock (device, MX_SB_IMAP_BLOCKS);
  unsigned int free_blocks = 0;
  for (int i = 0; i < mx_query_superblock (device, MX_SB_ZMAP_BLOCKS); i++) {
    readblock (device, zmap_start + i, (byte*)&block);
    if ((i+1)*8192 < query->size)
      free_blocks += count_zeros ((byte*)&block, 8192);
    else
      free_blocks += count_zeros ((byte*)&block, query->size - i*8192);
  }
  query->free = free_blocks;
  query->used = query->size - free_blocks;

  // find device name
  switch (device) {
    case DEV_HDA: strncpy (query->name, "/dev/hda", 10); break;
    case DEV_HDB: strncpy (query->name, "/dev/hdb", 10); break;
    case DEV_FD0: strncpy (query->name, "/dev/fd0", 10); break;
    case DEV_FD1: strncpy (query->name, "/dev/fd1", 10); break;
    default:      strncpy (query->name, "unknown",  10); break;
  }

  // find mount point
  boolean mounted = false;
  for (int i=0; i<current_mounts; i++) {
    if (mount_table[i].device == device) {
      strncpy (query->fstype, fs_names[mount_table[i].fstype], 10);
      strncpy (query->mount,  mount_table[i].mountpoint, 255);
      mounted = true;
      break;
    }
```

```
    }
    if (!mounted) {
      strncpy (query->fstype, "none", 10);
      strncpy (query->mount,  "none", 10);
    }
  }
```

Defines:
mx_diskfree, used in chunks 491 and 493b.
Uses count_zeros 491c, current_mounts 405b, DEV_FD0 508a, DEV_FD1 508a, DEV_HDA 508a, DEV_HDB 508a, diskfree_query 491a, fs_names 410b, minix_superblock 440c, mount_table 405b, mx_query_superblock 443b, readblock 506b, and strncpy 594b.

We provide a system call handler

⟨*syscall prototypes* 173b⟩+≡ (202a) ◁433a 512d▷ [493a]
```
void syscall_diskfree (context_t *r);
```

⟨*syscall functions* 174b⟩+≡ (202b) ◁433b 513a▷ [493b]
```
void syscall_diskfree (context_t *r) {
  // ebx: address of diskfree query structure
  mx_diskfree ((struct diskfree_query*)r->ebx);
}
```

Defines:
syscall_diskfree, used in chunk 493.
Uses context_t 142a, diskfree 493f, diskfree_query 491a, and mx_diskfree 492.

and register it:

⟨*ulix system calls* 206e⟩+≡ (205a) ◁428b [493c]
```
#define __NR_diskfree  522
```

Defines:
__NR_diskfree, used in chunk 493.

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁434a 513b▷ [493d]
```
install_syscall_handler (__NR_diskfree, syscall_diskfree);
```

Uses __NR_diskfree 493c, install_syscall_handler 201b, and syscall_diskfree 493b.

User mode applications can then ask for the information by writing a value into the device field of a diskfree_query$_{491a}$ structure and calling

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁434b 513c▷ [493e]
```
void diskfree (struct diskfree_query *query);
```

which fills the other fields.

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁434c 513d▷ [493f]
```
void diskfree (struct diskfree_query *query) {
  syscall2 (__NR_diskfree, (unsigned int)query);
}
```

Defines:
diskfree, used in chunk 493b.
Uses __NR_diskfree 493c, diskfree_query 491a, and syscall2 203c.

12.7 The /dev Filesystem

This section provides an interface to block devices (via `/dev/fd0`, `/dev/fd1`, `/dev/hda` and `/dev/hdb`) and the physical memory (via `/dev/kmem`). We have not designed the code in a way that would easily allow extensions to other device classes (however, a third or fourth hard disk would be easy to add).

The `/dev` filesystem must be mounted on `/dev/`, so the filesystem itself has only the root directory and the following five files which reside inside:

[494a] ⟨*constants* 112a⟩+≡

```
#define DEV_HDA_NAME   "hda"
#define DEV_HDB_NAME   "hdb"
#define DEV_FD0_NAME   "fd0"
#define DEV_FD1_NAME   "fd1"
#define DEV_KMEM_NAME  "kmem"
#define DEV_FD0_INODE  3
#define DEV_FD1_INODE  4
#define DEV_HDA_INODE  5
#define DEV_HDB_INODE  6
#define DEV_KMEM_INODE 7
```

Defines: (44a) ◁472 495a▷
DEV_FD0_INODE, used in chunk 495c.
DEV_FD0_NAME, used in chunk 494b.
DEV_FD1_INODE, used in chunk 495c.
DEV_FD1_NAME, used in chunk 494b.
DEV_HDA_INODE, used in chunk 495c.
DEV_HDA_NAME, used in chunk 494b.
DEV_HDB_INODE, used in chunk 495c.
DEV_HDB_NAME, used in chunk 494b.
DEV_KMEM_INODE, used in chunk 495c.
DEV_KMEM_NAME, used in chunk 494b.

We will simulate behavior of the Minix filesystem for our `/dev` filesystem so that the function which inspects a directory works with the `/dev/` directory as well.

[494b] ⟨*global variables* 92b⟩+≡ (44a) ◁464a 495b▷

```
struct minix_dir_entry dev_directory[7] = {
  { 1, "." },          { 2, ".." },         { 3, DEV_FD0_NAME },
  { 4, DEV_FD1_NAME }, { 5, DEV_HDA_NAME }, { 6, DEV_HDB_NAME },
  { 7, DEV_KMEM_NAME } };
```

Defines:
dev_directory, used in chunks 495c, 499d, and 500.
Uses DEV_FD0_NAME 494a, DEV_FD1_NAME 494a, DEV_HDA_NAME 494a, DEV_HDB_NAME 494a, DEV_KMEM_NAME 494a, and minix_dir_entry 452b.

Opening a file works similar to the way we have implemented it for the Minix filesystem, it is just a bit simpler:

[494c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁491b 496a▷

```
int dev_open (const char *path, int oflag);
```

Note that, different from `mx_open`$_{464b}$, it does not take a `device` argument.

We will keep a list of open file descriptors:

[494d] ⟨*type definitions* 91⟩+≡ (44a) ◁460a 508c▷

```
struct dev_filestat {
  short dev;
  int pos;
  short mode;
};
```

Defines:
dev_filestat, used in chunk 495b.

and allow up to 32 simultaneously open files:

⟨*constants* 112a⟩+≡ (44a) ◁494a 506a▷ [495a]
```
#define MAX_DEV_FILES 32
```
Defines:
MAX_DEV_FILES, used in chunks 495, 496b, and 499b.

So, similar to the mx_status$_{461b}$ array, we declare a dev_status$_{495b}$ array that holds the same kind of information:

⟨*global variables* 92b⟩+≡ (44a) ◁494b 509a▷ [495b]
```
struct dev_filestat dev_status[MAX_DEV_FILES] = { { 0 } };
```
Defines:
dev_status, used in chunks 495–99.
Uses dev_filestat 494d and MAX_DEV_FILES 495a.

The dev_open$_{495c}$ function is much simpler than mx_open$_{464b}$ because we know exactly which files can be opened. We identify the inode number with the index into the table dev_directory$_{494b}$ (plus 1, as we start counting inodes at number 1).

⟨*function implementations* 100b⟩+≡ (44a) ◁440b 496b▷ [495c]
```
int dev_open (const char *path, int oflag) {
  if ((oflag & O_CREAT) != 0)  return -1;   // cannot create
  int i, dev_inode = -1;

  // get the inode number
  for (i = 0;  i < 7;  i++) {
    if (strequal (path+1, dev_directory[i].name)) {
      // found!
      dev_inode = dev_directory[i].inode;  // which is always i...
      break;
    }
  }
  if (dev_inode == -1) return -1;   // not found

  // find free file descriptor
  int fd = -1;
  for (i = 0;  i < MAX_DEV_FILES;  i++) {
    if (dev_status[i].dev == 0) { fd = i; break; }
  }
  if (fd == -1) return -1;          // no free file descriptor

  switch (dev_inode) {
    case DEV_FD0_INODE : dev_status[fd].dev = DEV_FD0;  break;
    case DEV_FD1_INODE : dev_status[fd].dev = DEV_FD1;  break;
    case DEV_HDA_INODE : dev_status[fd].dev = DEV_HDA;  break;
    case DEV_HDB_INODE : dev_status[fd].dev = DEV_HDB;  break;
    case DEV_KMEM_INODE: dev_status[fd].dev = DEV_KMEM; break;
    default: dev_status[fd].dev = -1;
  }
  dev_status[fd].pos  = 0;
```

```
    dev_status[fd].mode = oflag;
    return fd;
  }
```

Defines:
dev_open, used in chunks 412c and 494c.
Uses dev_directory 494b, DEV_FD0 508a, DEV_FD0_INODE 494a, DEV_FD1 508a, DEV_FD1_INODE 494a, DEV_HDA 508a, DEV_HDA_INODE 494a, DEV_HDB 508a, DEV_HDB_INODE 494a, DEV_KMEM 508a, DEV_KMEM_INODE 494a, dev_status 495b, MAX_DEV_FILES 495a, O_CREAT 460b, and strequal 596a.

Closing a device file via

[496a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁494c 496c▷

```
int dev_close (int fd);
```

is much simpler:

[496b] ⟨*function implementations* 100b⟩+≡ (44a) ◁495c 496d▷

```
int dev_close (int fd) {
  if (fd ≥ 0 && fd < MAX_DEV_FILES && dev_status[fd].dev != 0) {
    dev_status[fd].dev = 0;
    return 0;  // success
  } else {
    return -1; // fail
  }
}
```

Defines:
dev_close, used in chunks 418a and 496a.
Uses dev_status 495b and MAX_DEV_FILES 495a.

Finally we need $\texttt{dev\_read}_{496d}$, $\texttt{dev\_write}_{497}$ and $\texttt{dev\_lseek}_{498a}$ functions:

[496c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁496a 498b▷

```
int dev_read (int fd, char *buf, int nbyte);
int dev_write (int fd, char *buf, int nbyte);
int dev_lseek (int fd, int offset, int whence);
```

The implementation of $\texttt{dev\_read}_{496d}$ and $\texttt{dev\_write}_{497}$ is similar to the one of $\texttt{mx\_read}_{470b}$ and $\texttt{mx\_write}_{474c}$ (which you have seen earlier), but it is simpler: the new functions need not access an inode in order to retrieve block numbers.

There is a special case for reading from memory via /dev/kmem which does not require any block read/write operations at all: For memory access we can simple call $\texttt{memcpy}_{596c}$.

[496d] ⟨*function implementations* 100b⟩+≡ (44a) ◁496b 497▷

```
int dev_read (int fd, char *buf, int nbyte) {
  ⟨dev filesystem: check if fd is a proper file descriptor 499b⟩
  int startbyte = dev_status[fd].pos;
  int devsize = dev_size (dev_status[fd].dev);
  if (startbyte ≥ devsize) { return 0; }  // nothing to read
  int endbyte = dev_status[fd].pos + nbyte - 1;
  if (endbyte ≥ devsize) {
    nbyte -= (endbyte - devsize + 1);  endbyte = devsize - 1;
  }

  // special case /dev/kmem: direct memcpy()
```

```
    if (dev_status[fd].dev == DEV_KMEM) {
      memcpy (buf, (char*)(PHYSICAL(startbyte)), nbyte);
      dev_status[fd].pos += nbyte;
      return nbyte;
    }

    int readbytes = 0;  int offset, length;
    int startblock = startbyte / BLOCK_SIZE;  int curblock = startblock;
    while (nbyte > 0) {
      byte block[BLOCK_SIZE];
      readblock (dev_status[fd].dev, curblock, (byte*) block);
      if (curblock == startblock) {
        offset = startbyte % BLOCK_SIZE;  length = MIN (nbyte, BLOCK_SIZE - offset);
      } else {
        offset = 0;                       length = MIN (nbyte, BLOCK_SIZE);
      }
      memcpy (buf, block + offset, length);

      nbyte -= length;      buf   += length;
      readbytes += length;  curblock++;
      dev_status[fd].pos += length;
    }
    return readbytes;
  }
```

Defines:
dev_read, used in chunk 414b.
Uses BLOCK_SIZE 440a, DEV_KMEM 508a, dev_size 499a, dev_status 495b, memcpy 596c, MIN 471d, PHYSICAL 116a, and readblock 506b.

Writing is only slightly more complex because the first and last block must be read before being written:

⟨*function implementations* 100b⟩+≡ (44a) ◁496d 498a▷ [497]

```
  int dev_write (int fd, char *buf, int nbyte) {
    ⟨dev filesystem: check if fd is a proper file descriptor 499b⟩
    int startbyte = dev_status[fd].pos;
    int devsize = dev_size (dev_status[fd].dev);
    if (startbyte ≥ devsize) { return 0; }  // nothing to write
    int endbyte = dev_status[fd].pos + nbyte - 1;
    if (endbyte ≥ devsize) {
      nbyte -= (endbyte - devsize + 1);  endbyte = devsize - 1;
    }

    // special case /dev/kmem: direct memcpy()
    if (dev_status[fd].dev == DEV_KMEM) {
      memcpy ((char*)(PHYSICAL(startbyte)), buf, nbyte);
      dev_status[fd].pos += nbyte;
      return nbyte;
    }
```

```
  int written_bytes = 0;  int offset, length;
  int startblock = startbyte / BLOCK_SIZE;  int curblock = startblock;
  while (nbyte > 0) {
    byte block[BLOCK_SIZE];
    if (curblock == startblock) {
      offset = startbyte % BLOCK_SIZE;  length = MIN (nbyte, BLOCK_SIZE - offset);
    } else {
      offset = 0;                       length = MIN (nbyte, BLOCK_SIZE);
    }

    if (offset != 0 || length != BLOCK_SIZE) {
      // writing a partial block -- read it first!
      readblock (dev_status[fd].dev, curblock, (byte*) block);
    }
    memcpy (block + offset, buf, length);
    writeblock (dev_status[fd].dev, curblock, (byte*) block);

    nbyte -= length;          buf   += length;
    written_bytes += length;  curblock++;
    dev_status[fd].pos += length;
  }
  return written_bytes;
}
```

Defines:
dev_write, used in chunks 415a and 496c.
Uses BLOCK_SIZE 440a, DEV_KMEM 508a, dev_size 499a, dev_status 495b, memcpy 596c, MIN 471d, PHYSICAL 116a, readblock 506b, and writeblock 507c.

Seeking is also simple:

[498a] ⟨*function implementations* 100b⟩+≡ (44a) ◁497 499a▷

```
int dev_lseek (int fd, int offset, int whence) {
  ⟨dev filesystem: check if fd is a proper file descriptor 499b⟩
  if (whence < 0 || whence > 2)
    return -1;                          // wrong lseek option
  if (whence == SEEK_END && offset > 0)
    return -1;                          // cannot seek beyond end of device
  switch (whence) {
    case SEEK_SET: dev_status[fd].pos = offset; break;
    case SEEK_CUR: dev_status[fd].pos += offset; break;
    case SEEK_END: dev_status[fd].pos = dev_size (dev_status[fd].dev) + offset;
  };
  return dev_status[fd].pos;
}
```

Defines:
dev_lseek, used in chunk 418a.
Uses dev_size 499a, dev_status 495b, lseek 429b, SEEK_CUR 469b, SEEK_END 469b, and SEEK_SET 469b.

We have used a function

[498b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁496c 499c▷

```
long dev_size (int dev);
```

which returns the size of the drive; here is its implementation:

⟨*function implementations* 100b⟩+≡ (44a) ◁498a 499d▷ [499a]
```
long dev_size (int dev) {
  switch (dev) {
    case DEV_FD0  : return fdd_type[fdd[0].type].total_sectors * 512;  // fd0
    case DEV_FD1  : return fdd_type[fdd[1].type].total_sectors * 512;  // fd1
    case DEV_HDA  : return hd_size[0] * 512;                           // hda
    case DEV_HDB  : return hd_size[1] * 512;                           // hdb
    case DEV_KMEM : return MEM_SIZE;                                   // kmem
    default       : return -1;                                         // error
  }
}
```
Defines:
dev_size, used in chunks 496–99.
Uses DEV_FD0 508a, DEV_FD1 508a, DEV_HDA 508a, DEV_HDB 508a, DEV_KMEM 508a, fdd 541c, fdd_type 541c, hd_size 534a, and MEM_SIZE 111c.

In both functions for reading and writing we check whether a valid file descriptor was supplied and return -1 if not:

⟨*dev filesystem: check if* fd *is a proper file descriptor* 499b⟩≡ (496–98) [499b]
```
if (fd < 0 || fd ≥ MAX_DEV_FILES)  return -1;  // bad file descriptor
if (dev_status[fd].dev == 0)       return -1;  // file not open
```
Uses dev_status 495b and MAX_DEV_FILES 495a.

Last, we supply functions for querying a file and reading a directory entry which are called from u_stat$_{421d}$ and u_getdent in the virtual filesystem layer.

⟨*function prototypes* 45a⟩+≡ (44a) ◁498b 504▷ [499c]
```
int dev_stat (const char *path, struct stat *buf);
int dev_getdent (const char *path, int index, struct dir_entry *buf);
```

dev_stat$_{499d}$ compares the local path (which is expected to be /fd0, /fd1, /hda, /hdb or /kmem) against the list of known device names and fills the struct stat$_{429b}$ buffer:

⟨*function implementations* 100b⟩+≡ (44a) ◁499a 500▷ [499d]
```
int dev_stat (const char *path, struct stat *buf) {
  int devices[] = { -1, 0, 0, DEV_FD0, DEV_FD1, DEV_HDA, DEV_HDB, DEV_KMEM };
  int dev_inode;

  for (int i = 0;  i < 7;  i++) {                    // get the inode number
    if (strequal (path+1, dev_directory[i].name)) {
      // found!
      dev_inode = dev_directory[i].inode;            // which is always i...
      break;
    }
  }
  if (dev_inode == -1) return -1;                    // not found

  // buf->st_dev   = 0;  // no device, /dev is a virtual FS
  buf->st_dev   = devices[dev_inode];
```

```
    buf->st_rdev  = 0;
    buf->st_ino   = dev_inode;
    if (dev_inode > 2)
      buf->st_mode  = S_IFBLK | 0600;   // block device; we have no char. devices
    else
      buf->st_mode  = S_IFDIR | 0600;   // directory
    buf->st_nlink = 1;    buf->st_uid   = 0;
    buf->st_gid   = 0;    buf->st_size  = dev_size (devices[dev_inode]);
    buf->st_atime = 0;    buf->st_ctime = 0;
    buf->st_mtime = 0;
    return 0;
  }
```

Defines:
dev_stat, used in chunk 421d.
Uses dev_directory 494b, DEV_FD0 508a, DEV_FD1 508a, DEV_HDA 508a, DEV_HDB 508a, DEV_KMEM 508a, dev_size 499a, S_IFBLK 457c, S_IFDIR 457c, stat 429b 489b, and strequal 596a.

And dev_getdent$_{500}$ copies an entry in the dev_directory$_{494b}$ table into the buffer (of type struct dir_entry$_{490b}$).

[500] ⟨*function implementations* 100b⟩+≡ (44a) ◁499d 505b▷

```
int dev_getdent (const char *path, int index, struct dir_entry *buf) {
  if (index < 0 || index > 6) return -1;   // no such entry

  buf->inode = dev_directory[index].inode;
  strncpy (buf->filename, dev_directory[index].name, 5);
  return 0;
}
```

Defines:
dev_getdent, used in chunk 422c.
Uses dev_directory 494b, dir_entry 490b, and strncpy 594b.

12.8 Default Contents of the Filesystem

In the last two sections we describe the directories and files that you can find on the ULIX system disks and suggest two books which discuss other filesystems in depth.

Figure 12.14 shows the general tree structure of the virtual filesystem which is organized in a similar way as most Unix filesystems. The /bin directory contains executable programs, /etc is reserved for configuration files (currently the only one is /etc/passwd), /dev is the mount point for the device filesystem, and /home contains the *home directories* of system users, it is already populated with two home directories that "belong" to us (the authors of this book). They correspond to identical user names in the /etc/passwd file (with the passwords set to "xyz"). The administrator root has the home directory /root; it is standard practice to place it directly in the root directory (/) and not below /home.

home directory

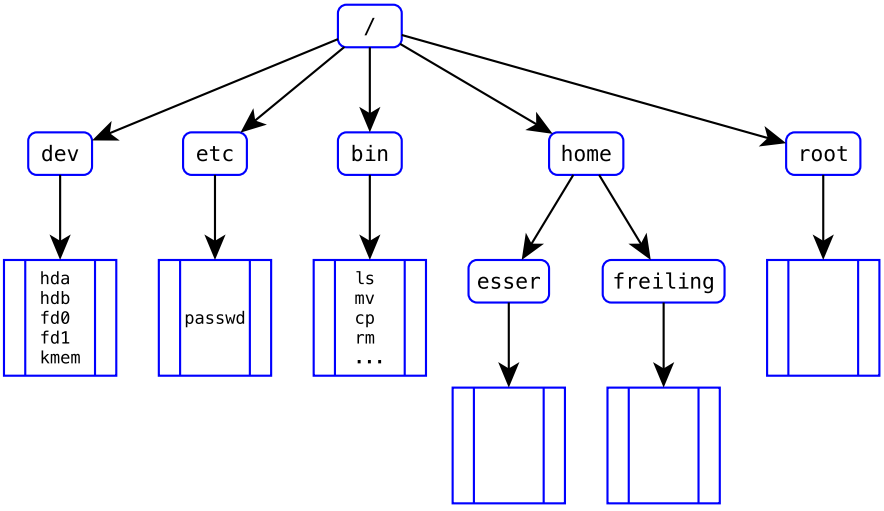


Figure 12.14: These are the contents of the ULIX disk.

12.9 Further Reading

If you are interested in further details about Unix filesystem implementations, we suggest you take a look at the following books:

- Steve D. Pate: "UNIX Filesystems: Evolution, Design, and Implementation", 2003, ISBN: 978-0-471-16483-8 [Pat03]
- William von Hagen: "Linux Filesystems", 2002, ISBN: 978-0672322723 [vH02]

12.10 Exercises

36. **Sparse Files**

    Sparse files are "files with holes": They have large areas which only contain `0x00` bytes. Storing all those zeros on disk is a waste of disk space (and also of time for initially writing them). Modern filesystems support a special treatment of sparse files where information about these holes is stored in the inode or in a separate block that is linked from the inode. The Minix filesystem does not support this kind of treatment, but you can modify it so that it does.

    Modify the file access functions so that a block address of -1 (`0xFFFF`) is interpreted as a reference to a sparse area, i. e., an area completely filled with zeros. No blocks need to be allocated for such areas. If a direct block address is -1, the whole (logical) block is considered to consist of zeros. If an indirect block address is -1, it means that there is no indirection block, but the space that could be addressed via the indirection

block (up to 256 blocks = 256 KByte) are assumed to contain zeros.

Every read access to a sparse area shall return a block of zeros, but when writing to a sparse area you need to allocate a block and change the block number from −1 to the new zone number. If data is written in the middle of a sparse indirection block, you need to allocate another block that then serves as indirection block with one zone number pointing to the new non-zero data block (and all other zone numbers set to −1).

Change the `mx_write`$_{474c}$ function so that it recognizes whether a whole sparse block is being written (or whether the result of the current write operation is a block full of zeros)—if so, add a new sparse area and release the block that is no longer needed.

You will need to make some changes to the system calls and user mode library functions so that you can test the behavior.

37. **Completion of the `mx_ftruncate` Function**

    Our implementation of `mx_ftruncate`$_{484e}$ is incomplete: It does not provide code for the ⟨ `mx_ftruncate`: *free single indirection block* ⟩ and ⟨ `mx_ftruncate`: *free double indirection block* ⟩ code chunks. Add those chunks (or modify the function otherwise) and try to do that in an optimized way that uses as few individual inode or block write operations as possible.

13
Disk I/O

In the last chapter you saw the implementations of both the Minix and the FAT filesystem—but what we have not discussed so far is how to actually talk to the hardware: we used functions `readblock`$_{506b}$ and `writeblock`$_{507c}$ to read or write kilobyte-sized chunks of data from the disk, and in this chapter you will see how to implement this.

13.1 Block and Character Devices

block device
character device

Classically, devices are split into two categories: *block devices* and *character devices*. The difference lies in the amounts of data which are transferred with every single request. A typical character device is the keyboard: each key-press generates an interrupt and the amount of data transferred is (typically) two bytes. On the other hand, disks (both floppy and hard disks) transfer whole chunks of data (512 bytes or larger quantities). The controllers for floppy and hard disks do not provide the functionality to read/write single bytes from/to the disk, but can only handle those larger chunks. That has consequences for code which wants to change a single byte in a disk file: The chunk containing this byte must first be read into memory, then modifed and finally rewritten to disk.

Block devices can be accessed in two ways: in the classical approach drivers used the processor's `in` and `out` instructions for every single byte that was to be transferred. Reading a 512-byte-sized chunk of data from the disk would basically look like this:

⟨*classical disk access example* 503⟩≡ [503]

```
out ioport1, sector         ; request data from the disk
out ioport2, READ_CMD
mov reg1, memory            ; set up target address, length of data
mov reg2, ioport3
mov reg3, length            ; for "rep"
rep insl                    ; read data (loop)
```

The `rep` prefix in the final statement uses the `length` argument stored in some register to repeat the `insl` instruction `length` times and auto-increment the memory address so that all the bytes coming from the disk will be stored in consecutive memory positions.

This approach requires the CPU to do a lot of work since it has to deal will every single byte that is to be transferred. It is better to use *direct memory access* (*DMA*) which allows the disk controller to store the data in memory without bothering the CPU.

DMA

In this chapter we will provide implementations of three device drivers:

- We start with a driver for a device that doesn't exist but can easily be simulated: the "serial disk". This driver assumes that a disk is connected to the serial port of the machine. The drive accepts read/write requests and sends or receives single bytes of such a request via the serial port. Every byte sent by the disk will generate an interrupt (and we need to provide an interrupt handler which will then read the newly-arrived byte via an `in` instruction), every byte we want to send to the disk must be sent explicitly via an `out` instruction.
- The second driver uses the classical (non-DMA) approach for accessing hard disks. It is easy to implement; a request for a 512-bytes-chunk is sent to the controller, the controller reads the data from the disk and then generates *one* interrupt. The interrupt handler must then read all 512 bytes from the controller. We use this to access the hard disks on the machine.
- The third driver uses DMA to talk to a floppy drive: Reading a 512-bytes-chunk also starts with requesting it from the controller, but the transfer happens in the background. When it's finished, the controller generates an interrupt, and the interrupt handler only needs to acknowledge it and tell the (suspended) process that its data have arrived.

This collection of drivers thus introduces three very different approaches for talking to mass media controllers.

13.2 Device Selection

We will provide two generic functions

[504] ⟨*function prototypes* 45a⟩+≡ (44a) ◁499c 505a▷

```
void readblock  (int device, int blockno, char *buffer);
void writeblock (int device, int blockno, char *buffer);
```

which read kilobyte-sized blocks from all the devices we support. As a naming convention for devices we use the Unix concept of *major* and *minor device IDs*—this lets us break down device IDs into a device class (the major device number) and the specific device of a class (the minor device number). We use the same numbers as Linux (for floppies and hard disks):

major/minor number

- Floppy drives have the major number 2, we support two drives `/dev/fd0` and `/dev/fd1` with minor numbers 0 and 1.

- Hard disks have the major number 3, we support two drives `/dev/hda` and `/dev/hdb` with minor numbers 0 and 64 (the numbers 1–63 and 65–127 are reserved for partitions of the first and second hard disk, respectively, but we do not implement partition support for hard disks).
- The serial disk has major number 42. There is only one of this kind: `/dev/sdisk` has minor number 0.

We combine major and minor numbers in one 16 bit wide device number via `device = major << 8 + minor`. That is: the upper eight bits of a device number contain the major number, and the lower eight bits contain the minor number.

With that formula we can also calculate major and minor numbers from a given device number: `major = device >> 8` and `minor = device & 0xff`.

This leads to the major, minor and device numbers shown in Table 13.1. Note that we do not provide devices for the serial ports or the keyboard even though they are also used by Ulix—they would be examples for the class of character devices.

We provide the following three functions to do the calculations:

⟨*function prototypes* 45a⟩+≡ (44a) ◁504 509c▷ [505a]
```
word makedev  (byte major, byte minor);
byte devmajor (word device);
byte devminor (word device);
```

They just use the formulas which we have already described above:

⟨*function implementations* 100b⟩+≡ (44a) ◁500 506b▷ [505b]
```
word makedev  (byte major, byte minor) { return ((major << 8) + minor); }
byte devmajor (word device)            { return (device >> 8); }
byte devminor (word device)            { return (device & 0xff); }
```
Defines:
`devmajor`, used in chunks 406, 506b, and 507b.
`devminor`, used in chunks 406, 506b, and 507b.
`makedev`, used in chunk 505a.

Defining some constants makes our life easier in the following implementations:

device file	major	minor	device
`/dev/fd0`	2	0	`0x0200` = 512
`/dev/fd1`	2	1	`0x0201` = 513
`/dev/hda`	3	0	`0x0300` = 768
`/dev/hdb`	3	64	`0x0340` = 832
`/dev/kmem`	4	0	`0x0400` = 1024
`/dev/sdisk` *)	42	0	`0x2a00` = 10752

Table 13.1: Ulix supports these devices. *) The device file `/dev/sdisk` is not available.

[506a] ⟨*constants* 112a⟩+≡ (44a) ◁495a 507a▷

```
#define MAJOR_FD      2
#define MAJOR_HD      3
#define MAJOR_KMEM    4
#define MAJOR_SERIAL 42
```

Defines:
MAJOR_FD, used in chunks 506b and 507b.
MAJOR_HD, used in chunks 506b and 507b.
MAJOR_SERIAL, used in chunks 506b and 507b.

The generic readblock$_{506b}$ and writeblock$_{507c}$ functions calculate the major and minor numbers from the device number and then call the appropriate reading or writing function for the correct device class:

[506b] ⟨*function implementations* 100b⟩+≡ (44a) ◁505b 507b▷

```
void readblock (int device, int blockno, char *buffer) {
  // check buffer
  if (buffer_read (device, blockno, buffer) == 0) { return; }

  // read from disk
  byte major = devmajor (device);
  byte minor = devminor (device);
  switch (major) {
    case MAJOR_HD:     readblock_hd (minor/64, blockno, buffer); break;
    case MAJOR_FD:     readblock_fd (minor,    blockno, buffer); break;
    case MAJOR_SERIAL: readblock_serial (      blockno, buffer); break;
    default: return;
  }

  // update buffer
  buffer_write (device, blockno, buffer, BUFFER_CLEAN);
}
```

Defines:
readblock, used in chunks 443–45, 448, 451a, 453b, 471c, 473–75, 477, 482, 492, 496d, and 497.
Uses BUFFER_CLEAN 510a, buffer_read 509d, buffer_write 510b, devmajor 505b, devminor 505b, MAJOR_FD 506a, MAJOR_HD 506a, MAJOR_SERIAL 506a, readblock_fd 550d, readblock_hd 531b, and readblock_serial 522e.

The case selection is straightforward: depending on the major number, readblock$_{506b}$ calls either readblock_hd$_{531b}$ (for hard disk access), readblock_fd$_{550d}$ (for the floppy disks), or readblock_serial$_{522e}$ (for the serial disk), and we will provide implementations of those functions in the following chapters.

The readblock$_{506b}$ function also calls buffer_read$_{509d}$ and buffer_write$_{510b}$ which we have not discussed yet—these two functions provide access to a system-wide disk cache which stores the contents of disk blocks so that they need not be read again when they are requested a second time. We will introduce the buffer mechanism in the next section. The short explanation for the above code is this: readblock$_{506b}$ first checks whether the requested block is already in the cache. If so, no disk access is necessary, and the function can return immediately. Otherwise one of the readblock_* functions takes care of the block transfer from disk to memory, and after that the freshly-read block is stored in the

cache. (The BUFFER_CLEAN$_{510a}$ argument states that the buffer's copy of the block is identical to the disk's version.) Note that the memory device (DEV_KMEM$_{508a}$) is not supported by readblock$_{506b}$ or writeblock$_{507c}$: it is no block device; the u_read$_{414b}$ and u_write$_{415a}$ functions use memcpy$_{596c}$ to access the memory.

Writing a block comes in two variations: We first show the writeblock_raw$_{507b}$ function which is the direct counterpart to readblock$_{506b}$: it has the same case selection and forwards the writing task to the writeblock_hd$_{531b}$ (for hard disks), writeblock_fd$_{550d}$ (for floppies) and writeblock_serial$_{522e}$ (serial disk) functions. When that is done, it also updates the buffer's copy of the block (if it is already cached).

⟨ *constants* 112a ⟩+≡ (44a) ◁506a 508a▷ [507a]

```
#define UPDATE_BUF      1
#define DONT_UPDATE_BUF 0
```

Defines:
DONT_UPDATE_BUF, used in chunk 512b.
UPDATE_BUF, used in chunk 507b.

⟨ *function implementations* 100b ⟩+≡ (44a) ◁506b 507c▷ [507b]

```
void writeblock_raw (int device, int blockno, char *buffer, char flag) {
  byte major = devmajor (device);
  byte minor = devminor (device);
  switch (major) {
    case MAJOR_HD:     writeblock_hd (minor/64, blockno, buffer); break;
    case MAJOR_FD:     writeblock_fd (minor,    blockno, buffer); break;
    case MAJOR_SERIAL: writeblock_serial (      blockno, buffer); break;
    default: break;
  }

  // update buffer cache (if it is in the cache)
  if ( (flag == UPDATE_BUF) && (buffer_contains (device, blockno)) )
    buffer_write (device, blockno, buffer, BUFFER_CLEAN);
}
```

Defines:
writeblock_raw, used in chunk 512b.
Uses BUFFER_CLEAN 510a, buffer_contains 512c, buffer_write 510b, devmajor 505b, devminor 505b, MAJOR_FD 506a, MAJOR_HD 506a, MAJOR_SERIAL 506a, UPDATE_BUF 507a, writeblock_fd 550d, writeblock_hd 531b, and writeblock_serial 522e.

However, in order to increase the disk performance, ULIX will not write blocks to disk immediately. Instead, we will always call the following function (writeblock$_{507c}$) which simply copies the data into the cache and marks it as dirty. At regular intervals the kernel will check whether there are dirty blocks and write them to disk.

⟨ *function implementations* 100b ⟩+≡ (44a) ◁507b 509d▷ [507c]

```
void writeblock (int device, int blockno, char *buffer) {
  buffer_write (device, blockno, buffer, BUFFER_DIRTY);
}
```

Defines:
writeblock, used in chunks 445b, 448, 451a, 453b, 454a, 475c, 477, 497, and 504.
Uses BUFFER_DIRTY 510a and buffer_write 510b.

Note that when calling `readblock_hd`$_{531b}$ or `writeblock_hd`$_{531b}$, we pass `minor/64` as argument which turns the only supported values for `minor` (0 and 64) into 0 and 1 for the two hard disks.

If you want to add a driver for a different kind of media (e. g. CD-ROM or DVD-ROM drives) all you need to do is develop `readblock_XX` and `writeblock_XX` functions for this new category, define a new `MAJOR_XX` constant and add the new case to the implementations of `readblock`$_{506b}$ and `writeblock_raw`$_{507b}$.

We define device constants for the five devices we plan to use regularly and another one that is used for error checking:

[508a] ⟨*constants* 112a⟩+≡ (44a) ◁507a 508b▷

```
#define DEV_HDA  0x300  //  disk   /dev/hda
#define DEV_HDB  0x340  //  disk   /dev/hdb
#define DEV_FD0  0x200  //  floppy /dev/fd0
#define DEV_FD1  0x201  //  floppy /dev/fd1
#define DEV_KMEM 0x400  //  memory /dev/kmem
#define DEV_NONE 0      //  no device
```

Defines:
`DEV_FD0`, used in chunks 406, 492, 495c, and 499.
`DEV_FD1`, used in chunks 405b, 406, 492, 495c, and 499.
`DEV_HDA`, used in chunks 405b, 406, 492, 495c, 499, 607a, and 610d.
`DEV_HDB`, used in chunks 405b, 406, 492, 495c, and 499.
`DEV_KMEM`, used in chunks 495–97 and 499.
`DEV_NONE`, used in chunk 405b.

13.3 A Simple Buffer Cache

In early versions of Ulix, the `readblock`$_{506b}$ and `writeblock`$_{507c}$ functions directly accessed the drive controllers which made even simple things such as displaying the contents of the root directory very slow, since many blocks were read over and over again.

Using a buffer cache can dramatically speed up disk access (to blocks which have already been read) by buffering them in memory. For our simple system it does not take much, we provide a buffer that can store 512 blocks:

[508b] ⟨*constants* 112a⟩+≡ (44a) ◁508a 510a▷

```
#define BUFFER_CACHE_SIZE 512
```

Defines:
`BUFFER_CACHE_SIZE`, used in chunks 509–12.

Buffer entries store the buffer and some additional information: the device and block numbers (in order to identify which block is cached), an access counter and a dirty flag:

[508c] ⟨*type definitions* 91⟩+≡ (44a) ◁494d 515a▷

```
struct buffer_entry {
  char buf[BLOCK_SIZE];
  int dev;      // from what device?              (-1 if free)
  int blockno;  // block number of buffered block (-1 if free)
  byte count;   // how often was it read_?
  byte dirty;   // true if not written to disk
};
```

Defines:
buffer_entry, used in chunk 509a.
Uses BLOCK_SIZE 440a.

The cache is just an array of buffer entries:

⟨*global variables* 92b⟩+≡ (44a) ◁495b 516a▷ [509a]
```
struct buffer_entry buffer_cache[BUFFER_CACHE_SIZE];
lock buffer_lock;
```
Defines:
buffer_cache, used in chunks 509–12.
buffer_lock, used in chunks 509, 510b, 512b, and 606.
Uses BUFFER_CACHE_SIZE 508b, buffer_entry 508c, and lock 365a.

and the kernel lock buffer_lock$_{509a}$ protects against parallel access attempts.

Here's how we initialize the buffer cache at system start:

⟨*initialize system* 45b⟩+≡ (44b) ◁326c 522b▷ [509b]
```
memset (buffer_cache, 0, sizeof (buffer_cache));
for (int i = 0;  i < BUFFER_CACHE_SIZE;  i++) {
  buffer_cache[i].blockno =
  buffer_cache[i].dev     = -1;
  buffer_cache[i].dirty   = 0;
}
buffer_lock = get_new_lock ("disk buffer");
```
Uses buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_lock 509a, get_new_lock 367b, and memset 596c.

Next we need code for entering data into and extracting it from the buffer cache; we write three functions

⟨*function prototypes* 45a⟩+≡ (44a) ◁505a 512a▷ [509c]
```
int buffer_write (int dev, int blockno, char *block, char dirtyflag);
int buffer_read  (int dev, int blockno, char *block);
boolean buffer_contains (int dev, int blockno);
```

The functions for reading and writing buffer entries have the same signatures as the readblock$_{506b}$ and writeblock_raw$_{507b}$ functions.

Reading is the simpler task, so we start with that:

⟨*function implementations* 100b⟩+≡ (44a) ◁507c 510b▷ [509d]
```
int buffer_read (int dev, int blockno, char *block) {
  // don't use the buffer before the scheduler is up
  if (!scheduler_is_active) { return -1; }  // -1 signals: must be read from disk
  mutex_lock (buffer_lock);

    // check if buffer cache holds the requested block
    int pos = -1;  // position in the cache
    for (int i = 0;  i < BUFFER_CACHE_SIZE;  i++) {
      if ((buffer_cache[i].dev == dev) && (buffer_cache[i].blockno == blockno)) {
        // found it!
        pos = i;
        break;
      }
    }
```

```
    if (pos == -1) { mutex_unlock (buffer_lock); return -1; }  // not found

    // we found it: copy the contents, update the counter
    memcpy (block, buffer_cache[pos].buf, BLOCK_SIZE);
    if ((int)buffer_cache[pos].count < 254) { buffer_cache[pos].count++; }
  mutex_unlock (buffer_lock); return 0;    // success
}
```

Defines:
buffer_read, used in chunk 506b.
Uses BLOCK_SIZE 440a, buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_lock 509a, memcpy 596c, mutex_lock 366a, mutex_unlock 366c, and scheduler_is_active 276e.

Writing to the buffer is a little more complicated—if there is no entry for the block we want to write. Otherwise it's pretty much the same:

[510a] ⟨*constants* 112a⟩+≡ (44a) ◁508b 515b▷

```
#define BUFFER_CLEAN 0
#define BUFFER_DIRTY 1
```

Defines:
BUFFER_CLEAN, used in chunks 506b and 507b.
BUFFER_DIRTY, used in chunk 507c.

[510b] ⟨*function implementations* 100b⟩+≡ (44a) ◁509d 512b▷

```
int buffer_write (int dev, int blockno, char *block, char dirtyflag) {
  // don't use the buffer before the scheduler is up
  if (!scheduler_is_active) { return 0; }
  mutex_lock (buffer_lock);
    // check if buffer cache already holds the requested block
    int pos = -1;              // position in the cache
    for (int i = 0;  i < BUFFER_CACHE_SIZE;  i++) {
      if ((buffer_cache[i].dev == dev) && (buffer_cache[i].blockno == blockno)) {
        pos = i; break;        // found it!
      }
    }

    // if not found, create it
    if (pos == -1) { ⟨buffer cache: find or create free entry; sets pos 511a⟩ }

    // copy the contents, update the counter
    if ((pos ≥ 0) && (pos < BUFFER_CACHE_SIZE)) {
      memcpy (buffer_cache[pos].buf, block, BLOCK_SIZE);
      if ((int)buffer_cache[pos].count < 254)
        buffer_cache[pos].count++;
        buffer_cache[pos].dirty = dirtyflag;
    }
  mutex_unlock (buffer_lock);
  return 0;    // success
}
```

Defines:
buffer_write, used in chunks 506, 507, and 509c.
Uses BLOCK_SIZE 440a, buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_lock 509a, memcpy 596c, mutex_lock 366a, mutex_unlock 366c, and scheduler_is_active 276e.

The obvious difference is that writing to the buffer cache always succeeds because we either update an existing entry or create a new entry. Creating a new one is not a problem as long as there remain free entries:

⟨*buffer cache: find or create free entry; sets* pos 511a⟩≡ (510b) [511a]
```
pos = -1;  // new search
for (int i = 0;  i < BUFFER_CACHE_SIZE;  i++) {
  if (buffer_cache[i].dev == -1) {
    pos = i;  break;     // this one is free
  }
}

if (pos == -1) {         // we found no free entry
  ⟨buffer cache: free an entry; sets pos 511b⟩
}

buffer_cache[pos].dev = dev;
buffer_cache[pos].blockno = blockno;
buffer_cache[pos].count = 0;
```
Uses buffer_cache 509a and BUFFER_CACHE_SIZE 508b.

This code prepares the buffer cache entry by setting its dev and blockno members. The memset596c command above would also zero out the buffer's contents, but this is not needed since it will be overwritten immediately.

Finally we need to say how to find an entry when all entries are in use. This asks for a replacement strategy and we'll implement a simple "least often used" strategy.

⟨*buffer cache: free an entry; sets* pos 511b⟩≡ (511a) [511b]
```
begin_buffer_search:             // find first clean entry
pos = -1;
for (int i = 0;  i < BUFFER_CACHE_SIZE;  i++) {
  if (buffer_cache[i].dirty == false) {
    pos = i;  break;             // end loop
  }
}
if (pos == -1) {
  buffer_sync (0);               // all buffers are dirty
  goto begin_buffer_search;
}

int least_used_val = buffer_cache[pos].count;

for (int i = pos+1;  i < BUFFER_CACHE_SIZE;  i++) {
  if (buffer_cache[i].count < least_used_val && buffer_cache[i].dirty == false) {
    // this entry is clean and was accessed less often
    least_used_val = buffer_cache[i].count;
    pos = i;  // update candidate
  }
}
```
Uses buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_sync 512b, and least_used_val.

When we want to force a synchronization of the buffer (i. e., writing dirty entries to disk and thereby making them clean), we call the `buffer_sync`$_{512b}$ function

[512a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁509c 518c▷
```
void buffer_sync (boolean lock_buffer);
```

which takes one argument indicating whether the `buffer_lock`$_{509a}$ needs to be acquired:

[512b] ⟨*function implementations* 100b⟩+≡ (44a) ◁510b 512c▷
```
void buffer_sync (boolean lock_buffer) {
  _set_statusline ("[B]", 34);
  if (lock_buffer) mutex_lock (buffer_lock);
    for (int i = 0;  i < BUFFER_CACHE_SIZE;  i++) {
      if (buffer_cache[i].dirty == true) {
        writeblock_raw (buffer_cache[i].dev, buffer_cache[i].blockno,
                        (char*)buffer_cache[i].buf, DONT_UPDATE_BUF);
        buffer_cache[i].dirty = false;
      }
    }
  if (lock_buffer) mutex_unlock (buffer_lock);
  _set_statusline ("[ ]", 34);
}
```
Defines:
`buffer_sync`, used in chunks 511–13.
Uses `_set_statusline` 337b, `buffer_cache` 509a, `BUFFER_CACHE_SIZE` 508b, `buffer_lock` 509a, `DONT_UPDATE_BUF` 507a, `mutex_lock` 366a, `mutex_unlock` 366c, and `writeblock_raw` 507b.

We also add a function `buffer_contains`$_{512c}$ which lets us query whether a specific block is currently buffered:

[512c] ⟨*function implementations* 100b⟩+≡ (44a) ◁512b 516d▷
```
boolean buffer_contains (int dev, int blockno) {
  // don't use the buffer before the scheduler is up
  if (!scheduler_is_active) { return false; }

  // check if buffer cache holds this block
  for (int i = 0;  i < BUFFER_CACHE_SIZE;  i++) {
    if ((buffer_cache[i].dev == dev) && (buffer_cache[i].blockno == blockno)) {
      return true;  // found it!
    }
  }
  return false;
}
```
Defines:
`buffer_contains`, used in chunk 507b.
Uses `buffer_cache` 509a, `BUFFER_CACHE_SIZE` 508b, and `scheduler_is_active` 276e.

In order to let the user synchronize the buffer cache (before shutting down the Ulix machine), we provide a `sync`$_{513d}$ system call:

[512d] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁493a 565b▷
```
void syscall_sync (context_t *r);
```

⟨*syscall functions* 174b⟩+≡ (202b) ◁493b 565c▷ [513a]
```
void syscall_sync (context_t *r) {
  // this syscall takes no arguments
  buffer_sync (1);   // with lock
}
```
Defines:
  syscall_sync, used in chunks 512d and 513b.
Uses buffer_sync 512b and context_t 142a.

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁493d 565a▷ [513b]
```
install_syscall_handler (__NR_sync, syscall_sync);
```
Uses __NR_sync 204c, install_syscall_handler 201b, and syscall_sync 513a.

The system call will be available via the user mode library function

⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁493e 568a▷ [513c]
```
void sync ();
```

that we implement here: It provides no arguments, so we use syscall1$_{203c}$.

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁493f 568b▷ [513d]
```
void sync () { syscall1 (__NR_sync); }
```
Defines:
  sync, used in chunk 513c.
Uses __NR_sync 204c and syscall1 203c.

Syncing will be done by the following swapper process which needs to be started during system initialization:

⟨*lib-build/tools/swapper.c* 311b⟩+≡ ◁311b [513e]
```
#include "../ulixlib.h"
int main () {
  int pid = getpid ();
  if (pid != 2) { printf ("swapper: don't start_ me manually.\n"); exit (1); }
  setterm (9); setpsname ("[swapper]");
  int init_frames = get_free_frames ();
  int last_free_frames;
  int free_frames = init_frames;
  unsigned int counter = 0;
  #define THRESHOLD (init_frames - 500)
  for (;;) {
    last_free_frames = free_frames;
    free_frames = get_free_frames ();
    if (free_frames != last_free_frames) {
      printf ("[%d.%d] swapper: %d free frames. threshold = %d.",
        pid, counter++, free_frames, THRESHOLD);
      if (free_frames < THRESHOLD) {
        printf ("calling free_a_frame (%d < %d)\n", free_frames, THRESHOLD);
        free_a_frame ();
      } else {
        printf ("\n");
      }
```

```
        }
      }
    }
```

Uses exit 218a, free_a_frame 310e, free_frames 112b, get_free_frames 310e, getpid 223b, main 44b, printf 601a, setterm 328g, and THRESHOLD.

We launch that program from the `init` process which guarantees that it will always have process ID 2; we also prevent that process against being killed.

13.4 Serial Hard Disk

Talking to device controllers requires knowledge of the protocols which those controllers understand. In later sections you will see how this is done for floppy and hard disk controllers, but we start with an example that is easier to understand, though it introduces a "device" that does not exist in real life: the serial hard disk. We provide support for a hypothetical disk which is connected to a serial port, and we can only make it work in an emulated machine (or with a second PC which takes the part of the serial disk).

Testing this code requires that you

- run Ulix in a virtual machine which supports two serial ports and that you
- add an external program which connects to the (virtual) second serial port, accepting commands and sending data back and forth.

It takes a little more than that, though, since we want to emulate the "normal" behavior of a disk controller. In real life, transfers use DMA (direct memory access). At the lowest level, the disk driver creates a `DMA_READ` or `DMA_WRITE` message and sends it to the controller. By itself, neither of these is a blocking action, since the disk controller will handle the transfer of data from the hard disk to memory (reading) or from memory to the disk (writing) independently of the CPU which continues executing. However, the process which initiated the transfer must be blocked anyway, since reading from the disk will take a while (and writing might not be safe if it continued and possibly changed the data which are currently written). After completion the disk controller creates an interrupt, the corresponding interrupt handler starts and puts the process back into the ready queue.

The actual DMA transfers work with physical memory addresses, so code using DMA must always know where data is or will be stored in physical memory.

in, out

Our serial hard disk works differently, it uses `in` and `out` commands to read or write single bytes through the serial port, and it can use virtual memory. In a simple implementation of this method the process would never block, it would just take a while to send or receive the data, and the scheduler would switch back and forth between this process and others.

In order to emulate "proper" disk controller behavior we take the following steps:

- Each time that a process starts a disk read/write operation, we create a special buffer for this transfer (which knows what data to send in what direction) and put it in a

disk queue; we then block the process. We limit the queue size so that no more than 100 processes may create an entry at the same time (the 101st process would fail and exit).

- Sending data via the serial port can happen immediately, while receiving depends on the other side (our external process). When the other side sends a byte, it causes an interrupt for the serial port, and inside the Ulix interrupt handler we fill a different buffer with that byte. So when "reading" in the timer handler, we don't actually talk to the serial port, but instead just copy data from one buffer to another.
- We do not allow a read and a write operation at the same time, since this would overcomplicate matters. Instead at each moment, we either read a whole sector, write one or do not access the serial disk at all.
- We add extra functions for *non-blocking* data transfer, because when we let the kernel (not processes) access the disk, we have nothing that we can block. Since this is the easier type of transfer, we start with it. non-blocking I/O

13.4.1 Kernel Code for the Serial Disk

We start with defining the buffer (which we create as a ring buffer):

⟨ *type definitions* 91 ⟩+≡ (44a) ◁508c 541b▷ [515a]

```
struct serial_disk_buffer_entry {
  int pid;                // process ID; -1 if kernel
  short status;           // New, Transfer, Finished, see BUF_STAT_*
  short direction;        // 100 = read_, 101 = write
  unsigned int secno;     // sector number
  memaddress address;     // memory address (in process' address space)
  byte sector[BLOCK_SIZE]; // 1024 bytes
};
```

Defines:
`serial_disk_buffer_entry`, used in chunks 516, 517c, and 520c.
Uses `BLOCK_SIZE` 440a and `memaddress` 46c.

⟨ *constants* 112a ⟩+≡ (44a) ◁510a 519b▷ [515b]

```
#define BUF_STAT_NEW      0
#define BUF_STAT_TRANSFER 1
#define BUF_STAT_FINISHED 2
#define BUF_READ          100
#define BUF_WRITE         101
#define SER_BUF_SIZE      100
```

Defines:
`BUF_READ`, used in chunks 517c, 518d, 520c, and 522e.
`BUF_STAT_FINISHED`, used in chunks 518 and 521a.
`BUF_STAT_NEW`, used in chunk 516d.
`BUF_WRITE`, used in chunks 517c, 518d, 520c, and 522e.
`SER_BUF_SIZE`, used in chunks 516, 518, and 521a.

The buffer is just an array with $\texttt{SER\_BUF\_SIZE}_{515b}$ buffer entries, and we mark its current use with two integers which remember its current start and end:

[516a] ⟨*global variables* 92b⟩+≡ (44a) ◁509a 516b▷
```
struct serial_disk_buffer_entry serial_disk_buffer[SER_BUF_SIZE];
int serial_disk_buffer_start = 0;   // initialize start and end of buffer usage
int serial_disk_buffer_end   = 0;   // interval in use is [start_, end[,
                                    // [0,0[ is empty
```
Defines:
serial_disk_buffer, used in chunks 516d, 517c, 519d, and 520c.
serial_disk_buffer_end, used in chunks 516d, 517c, and 520c.
serial_disk_buffer_start, used in chunks 516–21.
Uses SER_BUF_SIZE 515b and serial_disk_buffer_entry 515a.

This way we can always check whether the buffer is empty by testing if the two variables $\texttt{serial\_disk\_buffer\_start}_{516a}$ and $\texttt{serial\_disk\_buffer\_end}_{516a}$ are equal.

The buffer shall be protected by a lock:

[516b] ⟨*global variables* 92b⟩+≡ (44a) ◁516a 517a▷
```
lock serial_disk_lock;
```
Defines:
serial_disk_lock, used in chunks 516, 517c, and 520c.
Uses lock 365a.

[516c] ⟨*initialize kernel global variables* 184d⟩+≡ (44b) ◁363d
```
serial_disk_lock = get_new_lock ("serial disk");
```
Uses get_new_lock 367b and serial_disk_lock 516b.

Next we provide a function with which we can enter a new entry in the buffer:

[516d] ⟨*function implementations* 100b⟩+≡ (44a) ◁512c 517b▷
```
int serial_disk_enter (int pid, short direction, uint secno, uint address) {
  mutex_lock (serial_disk_lock);
    // check if buffer is full
    if ( (serial_disk_buffer_end+1) % SER_BUF_SIZE == serial_disk_buffer_start ) {
      mutex_unlock (serial_disk_lock);
      return -1;   // fail
    }
    struct serial_disk_buffer_entry *entry;
    entry = &serial_disk_buffer[serial_disk_buffer_end];
    entry->status = BUF_STAT_NEW;  entry->pid = pid;
    entry->direction = direction;  entry->secno = secno;
    entry->address = address;
    short tmp = serial_disk_buffer_end;
    serial_disk_buffer_end = (serial_disk_buffer_end+1) % SER_BUF_SIZE;
  mutex_unlock (serial_disk_lock);
  return tmp;   // tell the caller what entry number we used
}
```
Defines:
serial_disk_enter, used in chunks 518d and 522e.
Uses BUF_STAT_NEW 515b, mutex_lock 366a, mutex_unlock 366c, SER_BUF_SIZE 515b, serial_disk_buffer 516a, serial_disk_buffer_end 516a, serial_disk_buffer_entry 515a, serial_disk_buffer_start 516a, and serial_disk_lock 516b.

13.4.1.1 Non-Blocking Read/Write Operations

Now it is time to provide the non-blocking functions for reading and writing. We combine them in one function which does the appropriate thing, based on the buffer entry's `direction` field. The function takes no arguments since it finds all the necessary information in the buffer entry.

⟨*global variables* 92b⟩+≡ (44a) ◁516b 519c▷ [517a]
```
volatile int serial_disk_reader = 0;   // are we currently reading?
```
Defines:
`serial_disk_reader`, used in chunks 518b, 519d, and 521a.

When we want to send a sector number (as part of a request) we have to split it into bytes; a sector number is a 32 bit wide integer, so four bytes are needed:

⟨*function implementations* 100b⟩+≡ (44a) ◁516d 517c▷ [517b]
```
void serial_disk_send_sector_number (uint secno) {
  /* send... */  uart2putc ((byte)(secno % 256));  // lowest byte
  secno /= 256;  uart2putc ((byte)(secno % 256));  // 2nd lowest byte
  secno /= 256;  uart2putc ((byte)(secno % 256));  // 3rd lowest byte
  secno /= 256;  uart2putc ((byte)(secno % 256));  // highest byte
}
```
Defines:
`serial_disk_send_sector_number`, used in chunks 518 and 521a.
Uses `uart2putc` 345c.

The next function reads or writes a buffer.

⟨*function implementations* 100b⟩+≡ (44a) ◁517b 518d▷ [517c]
```
int serial_disk_non_blocking_rw () {
  mutex_lock (serial_disk_lock);
    serial_hard_disk_blocks = false;                // we don't block
    if (serial_disk_buffer_start == serial_disk_buffer_end) {
      mutex_unlock (serial_disk_lock);  return -1;   // buffer is empty
    }
    struct serial_disk_buffer_entry *entry;
    entry = &serial_disk_buffer[serial_disk_buffer_start];
    switch (entry->direction) {
      case BUF_WRITE: ⟨serial disk: write a buffer 518a⟩; break;
      case BUF_READ:  ⟨serial disk: read a buffer 518b⟩;  break;
      default:        mutex_unlock (serial_disk_lock); return -1;
    }
  mutex_unlock (serial_disk_lock);
  return 0;
}
```
Defines:
`serial_disk_non_blocking_rw`, used in chunk 518d.
Uses `BUF_READ` 515b, `BUF_WRITE` 515b, `mutex_lock` 366a, `mutex_unlock` 366c, `serial_disk_buffer` 516a, `serial_disk_buffer_end` 516a, `serial_disk_buffer_entry` 515a, `serial_disk_buffer_start` 516a, `serial_disk_lock` 516b, and `serial_hard_disk_blocks` 519c.

Writing is the simpler task: we only send the write command and the data via the serial port; we need not wait for a response since the serial port controller will not send one.

[518a] ⟨*serial disk: write a buffer* 518a⟩≡ (517c 520c)
```
uart2putc (CMD_PUT);  serial_disk_send_sector_number (entry->secno);
byte *addressptr = (byte*)(entry->address);
for (int i = 0;  i < 1024;  i++) {
  uart2putc (*addressptr); addressptr++;
}
entry->status = BUF_STAT_FINISHED;
serial_disk_buffer_start++;
serial_disk_buffer_start %= SER_BUF_SIZE;
```
Uses BUF_STAT_FINISHED 515b, CMD_PUT 519a, SER_BUF_SIZE 515b, serial_disk_buffer_start 516a, serial_disk_send_sector_number 517b, and uart2putc 345c.

CMD_PUT$_{519a}$ will be defined soon; along with CMD_GET$_{519a}$ it is used to tell the serial disk whether we initiate a write or read operation.

Reading is more complicated and requires the help of an interrupt handler; in the non-blocking implementation we do not put processes to sleep while a read operation is active. Instead we simply wait for its completion by repeatedly using the CPU instruction hlt.

[518b] ⟨*serial disk: read a buffer* 518b⟩≡ (517c)
```
uart2putc (CMD_GET);  serial_disk_send_sector_number (entry->secno);
serial_disk_reader = 1;                          // we're in read mode,
   // this value will be changed in the IRQ handler
while (serial_disk_reader == 1)  asm ("hlt");   // wait for data
entry->status = BUF_STAT_FINISHED;
serial_disk_buffer_start++;
serial_disk_buffer_start %= SER_BUF_SIZE;
// copy buffer to target memory location
memcpy ((char*)(entry->address), (char*)&(entry->sector), BLOCK_SIZE);
```
Uses BLOCK_SIZE 440a, BUF_STAT_FINISHED 515b, CMD_GET 519a, memcpy 596c, SER_BUF_SIZE 515b, serial_disk_buffer_start 516a, serial_disk_reader 517a, serial_disk_send_sector_number 517b, and uart2putc 345c.

Next we combine our functions to provide non-blocking read and write functions for the kernel (nb is short for "non-blocking"):

[518c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁512a 520b▷
```
void readblock_nb_serial  (int secno, char *buf);
void writeblock_nb_serial (int secno, char *buf);
```

[518d] ⟨*function implementations* 100b⟩+≡ (44a) ◁517c 519d▷
```
void readblock_nb_serial (int secno, char *buf) {
  int pid; if (scheduler_is_active) pid = current_task; else pid = -1;
  serial_disk_enter (pid, BUF_READ, secno, (uint)buf);
  serial_disk_non_blocking_rw ();
}

void writeblock_nb_serial (int secno, char *buf) {
  int pid; if (scheduler_is_active) pid = current_task; else pid = -1;
  serial_disk_enter (pid, BUF_WRITE, secno, (uint)buf);
  serial_disk_non_blocking_rw ();
}
```

Defines:
`writeblock_nb_serial`, used in chunk 518c.
Uses `BUF_READ` 515b, `BUF_WRITE` 515b, `current_task` 192c, `scheduler_is_active` 276e, `serial_disk_enter` 516d, and `serial_disk_non_blocking_rw` 517c.

These are the commands which we can send to the external controller process:

⟨*serial-hd/serial-hd-controller.h* 519a⟩≡ (519b) [519a]
```
#define CMD_STAT   1   // status query
#define CMD_GET    2   // GET a block (1024 bytes)
#define CMD_PUT    3   // PUT a block (1024 bytes)
#define CMD_TERM   99  // terminate controller
```
Defines:
`CMD_GET`, used in chunks 518b and 521a.
`CMD_PUT`, used in chunk 518a.

We use them both in the Ulix code as well as in the controller program.

⟨*constants* 112a⟩+≡ (44a) ◁515b 521c▷ [519b]
⟨*serial-hd/serial-hd-controller.h* 519a⟩

13.4.1.2 The Interrupt Handler

The interrupt handler `serial_hard_disk_handler`$_{519d}$ copies a byte from the serial port into the buffer, and if the buffer is full, it resets the `serial_disk_reader`$_{517a}$ variable to indicate that a whole block (of 1024 bytes) has been transferred.

⟨*global variables* 92b⟩+≡ (44a) ◁517a 522a▷ [519c]
```
char    serial_hard_disk_buffer[1024];
int     serial_hard_disk_pos    = 0;
boolean serial_hard_disk_blocks = false;
```
Defines:
`serial_hard_disk_blocks`, used in chunks 517c, 519d, and 520c.
`serial_hard_disk_buffer`, used in chunk 519d.
`serial_hard_disk_pos`, used in chunk 519d.

We will also have to read from the second serial port, so we provide a `uart2getc` function which reads a single character from that port. There is no corresponding `uartgetc` function for the first port, but it would look identical, except for using `uart`$_{344b}$`[0]` and `IO_COM1`$_{344a}$ instead of `uart`$_{344b}$`[1]` and `IO_COM2`$_{344a}$:

⟨*function implementations* 100b⟩+≡ (44a) ◁518d 520c▷ [519d]
```
static int uart2getc () {
  if (!uart[1]) { return -1; }
  if (!(inportb (IO_COM2+5) & 0x01)) { return -1; }
  return inportb (IO_COM2+0);
}

void serial_hard_disk_handler (context_t *r) {
  char c = uart2getc ();
  serial_hard_disk_buffer[serial_hard_disk_pos++] = c;
  if (serial_hard_disk_pos == 1024) {
    serial_hard_disk_pos = 0;
```

```
    // copy buffer to proper serial hard disk buffer
    memcpy ( &(serial_disk_buffer[serial_disk_buffer_start].sector),
             &serial_hard_disk_buffer, 1024 );
    serial_disk_reader = 0;  // reading a sector is finished
    if (serial_hard_disk_blocks) { ⟨serial hard disk: wake process 522c⟩ }
  }
}
```

Defines:
serial_hard_disk_handler, used in chunk 520a.
Uses context_t 142a, inportb 133b, IO_COM2 344a, memcpy 596c, serial_disk_buffer 516a, serial_disk_buffer_start 516a, serial_disk_reader 517a, serial_hard_disk_blocks 519c, serial_hard_disk_buffer 519c, serial_hard_disk_pos 519c, and uart 344b.

Finally we enter this interrupt handler in the handler list and enable the interrupt.

[520a] ⟨*setup serial hard disk* 345d⟩+≡ (45c) ◁345d

```
install_interrupt_handler (IRQ_COM2, serial_hard_disk_handler);
enable_interrupt (IRQ_COM2);
```

Uses enable_interrupt 140b, install_interrupt_handler 146c, IRQ_COM2 132, and serial_hard_disk_handler 519d.

Note that we're executing a code chunk ⟨*serial hard disk: wake process* 522c⟩ if we're currently working on a request for which blocking was enabled. We explain this in the next subsection.

13.4.1.3 Blocking Read/Write Operations

For a multitasking system it is inacceptable to work with blocking I/O operations, at least for processes. We will now implement the blocking read and write functions. For writing there is no difference (because the transfer commands to the serial port finish immediately), but for reading we will put the calling process to sleep until a whole block of data has been read. The function

[520b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁518c 522d▷

```
int serial_disk_blocking_rw ();
```

looks just like serial_disk_no_blocking_rw with two differences:

- It sets $\texttt{serial\_hard\_disk\_blocks}_{519c}$ to true (to indicate that we want to block),
- and in the switch expression it uses a fresh code chunk for reading.

[520c] ⟨*function implementations* 100b⟩+≡ (44a) ◁519d 522e▷

```
int serial_disk_blocking_rw () {
  mutex_lock (serial_disk_lock);
    serial_hard_disk_blocks = true;                  // we block
    if (serial_disk_buffer_start == serial_disk_buffer_end) {
      mutex_unlock (serial_disk_lock);  return -1;   // buffer is empty
    }
    struct serial_disk_buffer_entry *entry;
    entry = &serial_disk_buffer[serial_disk_buffer_start];
    switch (entry->direction) {
      case BUF_WRITE: ⟨serial disk: write a buffer 518a⟩; break;
```

```
      case BUF_READ:  ⟨serial disk: read a buffer and block 521a⟩;  break;
      default:        mutex_unlock (serial_disk_lock); return -1;
    }
  mutex_unlock (serial_disk_lock);
  return 0;
}
```

Defines:
serial_disk_blocking_rw, used in chunks 520b and 522e.
Uses BUF_READ 515b, BUF_WRITE 515b, mutex_lock 366a, mutex_unlock 366c, serial_disk_buffer 516a, serial_disk_buffer_end 516a, serial_disk_buffer_entry 515a, serial_disk_buffer_start 516a, serial_disk_lock 516b, and serial_hard_disk_blocks 519c.

The difference between the ⟨*serial disk: read a buffer* 518b⟩ and the following new code chunk is that we don't do busy waiting (as above) but the process to sleep. Only one line was changed (marked with [*]).

⟨*serial disk: read a buffer and block* 521a⟩≡ (520c) [521a]

```
uart2putc (CMD_GET);
⟨begin critical section in kernel 380a⟩
serial_disk_send_sector_number (entry->secno);
serial_disk_reader = 1;                          // we're in read mode,
   // this value will be changed in the IRQ handler
while (serial_disk_reader == 1)  { ⟨serial disk: put process to sleep 521b⟩ }  // [*]
entry->status = BUF_STAT_FINISHED;
serial_disk_buffer_start++;
serial_disk_buffer_start %= SER_BUF_SIZE;
// copy buffer to target memory location
memcpy ((char*)(entry->address), (char*)&(entry->sector), BLOCK_SIZE);
```

Uses BLOCK_SIZE 440a, BUF_STAT_FINISHED 515b, CMD_GET 519a, memcpy 596c, SER_BUF_SIZE 515b, serial_disk_buffer_start 516a, serial_disk_reader 517a, serial_disk_send_sector_number 517b, and uart2putc 345c.

In the non-blocking code we simply executed the assembler instruction `hlt` in the loop, we actively waited for the transfer to complete. Here we put the process to sleep:

⟨*serial disk: put process to sleep* 521b⟩≡ (521a) [521b]

```
if (scheduler_is_active) {
  // we access thread table; interrupts are off
  block (&serial_disk_queue, TSTATE_WAITSD);
  ⟨end critical section in kernel 380b⟩
  ⟨resign 221d⟩
} else {
  ⟨end critical section in kernel 380b⟩
}
```

Uses scheduler_is_active 276e, serial_disk_queue 522a, and TSTATE_WAITSD 521c.

We define the new state $\texttt{TSTATE\_WAITSD}_{521c}$ and the $\texttt{serial\_disk\_queue}_{522a}$ blocked queue:

⟨*constants* 112a⟩+≡ (44a) ◁519b 525a▷ [521c]

```
#define TSTATE_WAITSD 12
```

Defines:
TSTATE_WAITSD, used in chunks 521b and 564c.

[522a] ⟨*global variables* 92b⟩+≡ (44a) ◁519c 529a▷
```
blocked_queue serial_disk_queue;
```
Defines:
serial_disk_queue, used in chunks 521, 522, and 564c.
Uses blocked_queue 183a.

The queue must also be initialized:

[522b] ⟨*initialize system* 45b⟩+≡ (44b) ◁509b 529b▷
```
initialize_blocked_queue (&serial_disk_queue);
```
Uses initialize_blocked_queue 183c and serial_disk_queue 522a.

Since we need to wake the process up when the block was transmitted, we add the wake-up call to the interrupt handler. We've included the following code chunk in the serial_hard_disk_handler$_{519d}$ function:

[522c] ⟨*serial hard disk: wake process* 522c⟩≡ (519d)
```
if (scheduler_is_active) {
  int tid;
  if ((tid = serial_disk_queue.next) != 0)
    deblock (tid, &serial_disk_queue);
}
```
Uses deblock 186b, scheduler_is_active 276e, and serial_disk_queue 522a.

[522d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁520b 528b▷
```
void readblock_serial  (int secno, char *buf);
void writeblock_serial (int secno, char *buf);
```

[522e] ⟨*function implementations* 100b⟩+≡ (44a) ◁520c 528c▷
```
void readblock_serial (int secno, char *buf) {
  int pid; if (scheduler_is_active) pid = current_task; else pid = -1;
  serial_disk_enter (pid, BUF_READ, secno, (uint)buf);
  serial_disk_blocking_rw ();
}

void writeblock_serial (int secno, char *buf) {
  int pid; if (scheduler_is_active) pid = current_task; else pid = -1;
  serial_disk_enter (pid, BUF_WRITE, secno, (uint)buf);
  serial_disk_blocking_rw ();
}
```
Defines:
readblock_serial, used in chunk 506b.
writeblock_serial, used in chunks 507b and 522d.
Uses BUF_READ 515b, BUF_WRITE 515b, current_task 192c, scheduler_is_active 276e, serial_disk_blocking_rw 520c, and serial_disk_enter 516d.

13.4.2 The External Controller Process

The controller is a simple program that opens a TCP socket to talk to the serial port of the emulated PC that executes Ulix. The functions readsect and writesect transfer individual

blocks. The program only reacts to requests that come from the Ulix machine. In that way it simulates the behavior of a disk controller.

⟨*serial-hd/serial-hd-controller.c* 523⟩≡ [523]

```
#include <fcntl.h>        // open()
#include <sys/types.h>    // socket()
#include <sys/socket.h>   // socket()
#include <netinet/in.h>   // socket()
#include <unistd.h>       // close()
#include <string.h>       // bzero()
#include <stdio.h>
#include <unistd.h>       // lseek: SEEK_SET
#include "serial-hd-controller.h"

int socks;                // socket descriptor for ULIX connection
int fd = -1;              // file descriptor
int numsec = -1;          // number of sectors (1024 bytes) in disk image
byte sector[BLOCK_SIZE];

void readsocket (byte *buf, short len) {
  // We use this instead of recv(), since recv() does not always
  // read the expected number of bytes.
  int total = 0;
  while (total < len) {
    total += recv (socks, buf+total, len-total, 0);
  };
};

void openfile ()  { fd = open ("minix1.img", O_RDWR); numsec = 2880; };

void closefile () { close (fd); fd = -1; numsec = -1; }

void readsect (int i) {
  lseek (fd, i*BLOCK_SIZE, SEEK_SET);       // get sector from disk image
  int res = read (fd, &sector, BLOCK_SIZE);
  send (socks, &sector, BLOCK_SIZE, 0);     // send it to ULIX
};

void writesect (int i) {
  readsocket ((byte*) &sector, BLOCK_SIZE); // get sector from ULIX
  lseek (fd, i*BLOCK_SIZE, SEEK_SET);       // write it to disk image
  write (fd, &sector, BLOCK_SIZE);
};

int main () {
  openfile ();                              // open disk image
  socks = socket (AF_INET, SOCK_STREAM, 0); // connect to localhost:4444
  struct sockaddr_in serveraddr;
  bzero (&serveraddr, sizeof (serveraddr));
```

```
  inet_pton (AF_INET, "127.0.0.1", &(serveraddr.sin_addr));
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_port = htons (4444);

  // Create connection
  connect (socks, (struct sockaddr*) &serveraddr, sizeof (serveraddr));

  byte cmd_type;
  int sectornumber;
  setbuf (stdout, 0);

  while (1) {
    readsocket (&cmd_type, 1);
    switch (cmd_type) {
      case CMD_STAT: printf ("ULIX asked for status\n"); break;
      case CMD_GET:  readsocket ((byte*)&sectornumber, 4);
                     printf ("ULIX asked get %d\n", sectornumber);
                     readsect (sectornumber); break;
      case CMD_PUT:  readsocket ((byte*)&sectornumber, 4);
                     printf ("ULIX asked put %d\n", sectornumber);
                     writesect (sectornumber); break;
      case CMD_TERM: printf ("ULIX terminated connection. Quitting.\n");
                     goto finished;
      default:       printf ("ERROR in Command from ULIX\n");
    };
  }
  finished:                                  // Close connection
  close (socks); closefile ();
}
```

In order to connect the external process to `qemu` (running Ulix), we start `qemu` as follows:

[524] ⟨*qemu invocation* 524⟩≡

```
qemu -m 64 -fda ulix-fd0.img -d cpu_reset -s -serial mon:stdio \
-serial tcp::4444,server
```

Note that there are two `-serial` arguments; the first one connects COM1 with the terminal from which `qemu` was started; the second one connects COM2 with a TCP server on port 4444. That's the one our external program is going to connect to.

The final release of Ulix does not use the serial hard disk any more because its dependency on the external controller program made using the system uncomfortable. In the following two sections we present our hard disk and floppy disk drivers.

13.5 The Hard Disk Controller

As mentioned before, we will let our hard disk driver use non-DMA data transfer, called *PIO* (*programmed input/output*). The code in this section is based on the IDE driver code of the xv6 operating system [CKM12]. PIO

13.5.1 Sending Commands to the Controller

Communication with a device always needs to follow strict protocols, this holds for the hard disk controller, too. The following description is full of technical details about controller-internal registers and the ports used to access them, it also contains some assembler code. If you want to skip this, here's a summary: in this subsection we'll define two code chunks ⟨*ide: read sector* `sector` *on device* `hd` 527b⟩ and ⟨*ide: write sector* `sector` *on device* `hd` 527c⟩ which can be used for sending the controller the commands for *initiating* the transfer and for copying a sector from memory to the controller's internal memory. The other direction (from the controller's memory to RAM) will be dealt with inside the interrupt handler which we'll discuss in one of the following subsections.

We'll define some constants which will be used in the following code: The interrupt number for the (first) IDE controller is 14 ($\texttt{IRQ\_IDE}_{132}$). The controller accepts two commands for reading (`0x20`; $\texttt{IDE\_CMD\_READ}_{525a}$) and writing (`0x30`; $\texttt{IDE\_CMD\_WRITE}_{525a}$), and when we query the controller's status, there are four possible results which we'll be prepared to handle (busy, data ready, device fault and error):

⟨*constants* 112a⟩+≡ (44a) ◁521c 525b▷ [525a]

```
#define IDE_CMD_READ  0x20     // read from disk, with retries
#define IDE_CMD_WRITE 0x30     // write to disk, with retries
#define IDE_CMD_IDENT 0xec     // identify disk
#define IDE_BSY       0x80     // 0b10000000  (bit 7),  device busy
#define IDE_DRDY      0x40     // 0b01000000  (bit 6),  device ready
#define IDE_DF        0x20     // 0b00100000  (bit 5),  drive fault
#define IDE_ERR       0x01     // 0b00000001  (bit 0),  error
```

Defines:
`IDE_BSY`, used in chunk 533b.
`IDE_CMD_IDENT`, used in chunk 534b.
`IDE_CMD_READ`, used in chunk 527b.
`IDE_CMD_WRITE`, used in chunk 527c.
`IDE_DF`, used in chunk 533b.
`IDE_DRDY`, used in chunk 533b.
`IDE_ERR`, used in chunk 533b.
`IRQ_IDE`, used in chunk 534b.

In order to talk to the controller we use the ports `0x1f0` – `0x1f7` (the port numbers can be found in Seagate's ATA Interface Reference Manual [Sea93, p. 13]), we give them names to make things easier:

⟨*constants* 112a⟩+≡ (44a) ◁525a 529d▷ [525b]

```
// IDE output
#define IO_IDE_SEC_COUNT  0x1f2  // sector count register (read_/write_)
#define IO_IDE_SECTOR     0x1f3  // (32 bits in 0x1f3..0x1f6)
#define IO_IDE_DISKSEL    0x1f6  // disk select and upper 4 bits of sector no.
```

```
#define IO_IDE_COMMAND    0x1f7  // command register
#define IO_IDE_DEVCTRL    0x3f6  // device control register
// IDE input
#define IO_IDE_DATA       0x1f0  // data (read_/write_)
#define IO_IDE_STATUS     0x1f7  // status register (identical to command reg.)
```

Defines:
IO_IDE_COMMAND, used in chunks 527 and 534b.
IO_IDE_DATA, used in chunks 532 and 534b.
IO_IDE_DEVCTRL, used in chunk 526.
IO_IDE_DISKSEL, used in chunks 526 and 534b.
IO_IDE_SEC_COUNT, used in chunk 526.
IO_IDE_SECTOR, used in chunk 527a.
IO_IDE_STATUS, used in chunks 532–34.

Note that $\texttt{IO\_IDE\_COMMAND}_{525b}$ and $\texttt{IO\_IDE\_STATUS}_{525b}$ are the same port number (`0x1f7`), but depending on the type of access, they refer to different registers: When reading that port, we access the *status register*, and when writing, we access the *command register*.

status/command register

The read and write commands are specified in the "AT Attachment Interface for Disk Drives" document [Lam94, p. 40]; `0x20` and `0x30` are the read/write commands which trigger data transfer with retries (in case of errors); there are also further commands (`0x21`, `0x31`) which trigger corresponding reads or writes without retries. The kernel can send a command by writing the required value into the controller's command register via the $\texttt{IO\_IDE\_COMMAND}_{525b}$ port `0x1f7` (`outb IDE_CMD_READ`$_{525a}$`, 0x1f7`).

The status values represent the bit positions 7 (`0x80` $= 128 = 2^7$), 6 (`0x40` $= 64 = 2^6$), 5 (`0x20` $= 32 = 2^5$) and 0 (`0x01` $= 1 = 2^0$) of the status register [Lam94, p. 34], see Table 13.2.

The read and write commands can make the disk read/write several sectors with one command. To start such a read or write operation, we need to tell the controller three things:

- How many sectors shall be read/written? This information must be stored in the *sector count register* which is accessible via the $\texttt{IO\_IDE\_SEC\_COUNT}_{525b}$ port (`0x1f2`). We will always read or write just a single sector:

sector count register

[526] ⟨*ide: read/write sector* `sector` *on device* `hd` 526⟩≡ (527) 527a▷

```
idewait (0);
outportb (IO_IDE_DISKSEL,   0xe0 | (hd<<4));   // select disk
outportb (IO_IDE_DEVCTRL,   0);                // generate interrupt
outportb (IO_IDE_SEC_COUNT, 1);                // one sector
```

Uses idewait 533b, IO_IDE_DEVCTRL 525b, IO_IDE_DISKSEL 525b, IO_IDE_SEC_COUNT 525b, and outportb 133b.

7	6	5	4	3	2	1	0
BSY	DRDY	DWF	DSC	DRQ	CORR	IDX	ERR

Table 13.2: The status register of the IDE controller.

- Via the four ports `0x1f3` – `0x1f6`, we can specify the 28 bits of a sector number (four bits are used for selecting the drive, making the target register 32 bits wide):

⟨ *ide: read/write sector* `sector` *on device* `hd` 526⟩+≡ (527) ◁526 [527a]

```
outportb (IO_IDE_SECTOR,     sector        & 0xff);
outportb (IO_IDE_SECTOR+1,  (sector >> 8)  & 0xff);
outportb (IO_IDE_SECTOR+2,  (sector >> 16) & 0xff);
outportb (IO_IDE_SECTOR+3, ((sector >> 24) & 0x0f) | ((0xe + hd) << 4) );
```

Uses `IO_IDE_SECTOR` 525b and `outportb` 133b.

- Finally, we send the $\texttt{IDE\_CMD\_READ}_{525a}$ or $\texttt{IDE\_CMD\_WRITE}_{525a}$ command to the controller via the $\texttt{IO\_IDE\_COMMAND}_{525b}$ port `0x1f7`:

⟨ *ide: read sector* `sector` *on device* `hd` 527b⟩≡ (530c) [527b]

```
⟨ ide: read/write sector sector on device hd 526 ⟩
outportb (IO_IDE_COMMAND, IDE_CMD_READ);
```

Uses `IDE_CMD_READ` 525a, `IO_IDE_COMMAND` 525b, and `outportb` 133b.

⟨ *ide: write sector* `sector` *on device* `hd` 527c⟩≡ (530d) [527c]

```
⟨ ide: read/write sector sector on device hd 526 ⟩
outportb (IO_IDE_COMMAND, IDE_CMD_WRITE);
```

Uses `IDE_CMD_WRITE` 525a, `IO_IDE_COMMAND` 525b, and `outportb` 133b.

Using 28 bits for the sector number allows us to access $2^{28} = 268\,435\,456$ sectors, thus the maximum disk size is $2^{28} \times 512 = 137\,438\,953\,472$ bytes (128 GByte) which should be enough for most Ulix uses ... This is called *LBA28* (*Logical Block Addressing*, 28 bits). In 2002, ATA-6 [McL02], the sixth version of the ATA standard, introduced LBA48 which uses 48 bits to specify sector numbers and allows for much larger disks. LBA28

Our IDE driver will not use DMA transfer but instead copy the bytes with `in` and `out` operations, directly talking to the controller.

Now we're able to send read/write commands to the hard disk controller, and it will start servicing those requests immediately. But what happens when the disk has read the sector contents and copied them into the controller's internal memory? We need to fetch the data.

The controller is helpful in that it will tell us when it finished its work: it will raise the $\texttt{IRQ\_IDE}_{132}$ interrupt (number 14), and our interrupt handler must then copy the data from the controller to RAM.

We do this by directly reading the data via the controller's data register (i. e., by reading from $\texttt{IO\_IDE\_DATA}_{525b}$). The CPU instruction `insl` can read four bytes (a 32 bit value) in one go. For reading the contents of a whole sector we would need $512/4 = 128$ of these `insl` instructions, but the processor has a way of repeating this command automatically: if we add a `rep` prefix to `insl`, we get repeated executions of `insl` witout manually writing a loop. `insl` `rep insl`

The logic of `rep insl` requires us to fill some registers with proper values:

- The memory address (our buffer) goes into the *EDI* register (after each step, *EDI* will be incremented by 4),

- the number of repetitions must be stored in the *ECX* register (after each step, *ECX* will be decremented; the loop continues while $ECX \neq 0$),
- and the port number must be stored in the (16 bit) *DX* register.

Thus, in C-like pseudo code, `rep insl` does the following:

[528a] ⟨*pseudo code for* `rep insl` 528a⟩≡
```
while (%ecx != 0) {
  *(%edi) = inportl (%ecx);  // read 4 bytes, write them to *(%edi)
  %edi += 4;                 // update target memory
  %ecx--;                    // decrement counter
}
```

In the inline assembler language the registers *ECX*, *DX* and *EDI* can be accessed using `"c"`, `"d"` and `"D"`, respectively (see Appendix B for an introduction to `gcc` inline assembler). `rep` can either auto-increment or auto-decrement the target address with each step; in order to make it increment (like we want), we need to set the *direction flag* of the *EFLAGS* register to 0 using the machine instruction `cld` (*clear direction flag*).

direction flag
cld

The following function definition sets everything up properly and executes `rep insl`:

[528b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁522d 528d▷
```
static inline void repeat_inportsl (int port, void *addr, int cnt);
```

[528c] ⟨*function implementations* 100b⟩+≡ (44a) ◁522e 528e▷
```
static inline void repeat_inportsl (int port, void *addr, int cnt) {
  asm volatile ("cld \n"
                "rep insl" :
                "=D" (addr), "=c" (cnt) :
                "d" (port), "0" (addr), "1" (cnt) :
                "memory", "cc");
}
```
Defines:
`repeat_inportsl`, used in chunks 528b, 532d, and 534b.

With `"0"` and `"1"` we refer to the first two registers used, that is, `"D"` (*EDI*) and `"c"` (*ECX*); see also Appendix B.4.

For the other direction, we provide a `repeat_outportsl`$_{528e}$ function which looks almost identical:

[528d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁528b 529c▷
```
static inline void repeat_outportsl (int port, void *addr, int cnt);
```

[528e] ⟨*function implementations* 100b⟩+≡ (44a) ◁528c 530c▷
```
static inline void repeat_outportsl (int port, void *addr, int cnt) {
  asm volatile ("cld \n"
                "rep outsl" :
                "=S" (addr), "=c" (cnt) :
                "d" (port), "0" (addr), "1" (cnt) :
                "cc");
}
```
Defines:
`repeat_outportsl`, used in chunks 528d and 532b.

Instead of *EDI*, the `outsl` instruction expects the *ESI* register to contain the memory address, which is why we wrote `"=S"(addr)` instead of `"=D"(addr)`.

13.5.2 The Blocked Queue

We define a `harddisk_queue`$_{529a}$ which will contain processes waiting for a hard disk access operation to finish:

⟨*global variables* 92b⟩+≡ (44a) ◁522a 530a▷ [529a]
```
blocked_queue harddisk_queue;   // processes which wait for the hard disk
```
Defines:
`harddisk_queue`, used in chunks 529b, 531a, 532d, 564c, and 606.
Uses `blocked_queue` 183a.

and we have to initialize this queue:

⟨*initialize system* 45b⟩+≡ (44b) ◁522b 530b▷ [529b]
```
initialize_blocked_queue (&harddisk_queue);
```
Uses `harddisk_queue` 529a and `initialize_blocked_queue` 183c.

Processes that access a hard disk drive will wait on this queue, and the interrupt handler will wake up these processes when the operation was completed.

13.5.3 Reading and Writing

Now we use the code presented so far to create two functions

⟨*function prototypes* 45a⟩+≡ (44a) ◁528d 532c▷ [529c]
```
void readblock_hd  (int hd, int blockno, char *buffer);
void writeblock_hd (int hd, int blockno, char *buffer);
```

which read and write a complete block (1024 bytes). They will use a buffer `hd_buf`$_{530a}$ which can contain one sector (512 bytes) and must be protected by a lock. We also declare a global variable `hd_direction`$_{530a}$ which we set to `HD_OP_READ`$_{529d}$ or `HD_OP_WRITE`$_{529d}$ when we initialize a read or write operation.

⟨*constants* 112a⟩+≡ (44a) ◁525b 535▷ [529d]
```
#define HD_OP_READ     0
#define HD_OP_WRITE    1
#define HD_OP_NONE    -1

#define HD_SECSIZE   512
```
Defines:
`HD_OP_NONE`, used in chunks 530 and 532d.
`HD_OP_READ`, used in chunks 530c and 532d.
`HD_OP_WRITE`, used in chunks 530d and 532d.
`HD_SECSIZE`, used in chunks 530 and 532.

We also need some global variables: `hd_buf`$_{530a}$ is a buffer that can store one sector, `hd_lock`$_{530a}$ is used for locking disk access, and `hd_direction`$_{530a}$ will always be set to one of the `HD_OP_`$*$ constants to indicate the current transfer direction.

[530a] ⟨*global variables* 92b⟩+≡ (44a) ◁529a 534a▷
```
char hd_buf[HD_SECSIZE];
lock hd_lock;
char hd_direction;
```
Defines:
hd_buf, used in chunks 530 and 532.
hd_direction, used in chunks 530 and 532d.
hd_lock, used in chunk 530.
Uses HD_SECSIZE 529d and lock 365a.

We initialize the lock at system start-up:

[530b] ⟨*initialize system* 45b⟩+≡ (44b) ◁529b 544e▷
```
hd_lock = get_new_lock ("hard disk");
```
Uses get_new_lock 367b and hd_lock 530a.

The functions readblock_hd$_{531b}$ and writeblock_hd$_{531b}$ will transfer 1 KByte blocks of data. Since the hard disk controller defaults to transferring 512-bytes-sized sectors, we first provide functions for reading and writing such sectors.

[530c] ⟨*function implementations* 100b⟩+≡ (44a) ◁528e 530d▷
```
void readsector_hd (int hd, int sector, char *buffer) {
  mutex_lock (hd_lock);
    hd_direction = HD_OP_READ;
    ⟨begin critical section in kernel 380a⟩
    ⟨ide: read sector sector on device hd 527b⟩
    while (hd_direction == HD_OP_READ) { ⟨ide: put process to sleep 531a⟩ }
    ⟨ide: read data from the controller 532a⟩
    memcpy (buffer, hd_buf, HD_SECSIZE);
    hd_direction = HD_OP_NONE;
  mutex_unlock (hd_lock);
}
```
Uses hd_buf 530a, hd_direction 530a, hd_lock 530a, HD_OP_NONE 529d, HD_OP_READ 529d, HD_SECSIZE 529d, memcpy 596c, mutex_lock 366a, and mutex_unlock 366c.

For writing, the sequence of events is slightly different; we first transfer the data to the controller and then put the process to sleep, letting it wait for the transfer to finish:

[530d] ⟨*function implementations* 100b⟩+≡ (44a) ◁530c 531b▷
```
void writesector_hd (int hd, int sector, char *buffer) {
  mutex_lock (hd_lock);
    hd_direction = HD_OP_WRITE;
    memcpy (hd_buf, buffer, HD_SECSIZE);
    ⟨begin critical section in kernel 380a⟩
    ⟨ide: write sector sector on device hd 527c⟩
    ⟨ide: write data to the controller 532b⟩
    while (hd_direction == HD_OP_WRITE) { ⟨ide: put process to sleep 531a⟩ }
    hd_direction = HD_OP_NONE;
  mutex_unlock (hd_lock);
}
```
Defines:
writesector_hd, used in chunk 531b.
Uses hd_buf 530a, hd_direction 530a, hd_lock 530a, HD_OP_NONE 529d, HD_OP_WRITE 529d, HD_SECSIZE 529d, memcpy 596c, mutex_lock 366a, and mutex_unlock 366c.

Why do we put the process to sleep anyway? While the PIO transfer between system RAM and the controller's memory wastes some time (and a DMA would save that waste), the actual transfer between controller and disk takes a lot longer—this transfer is what the process must wait for to complete.

- When the process reads from the disk, we cannot get around waiting anyway: the data will not be available before the transfer completes. Putting the process to sleep guarantees that execution of the process only continues after the data have been read (and stored in the buffer that this process has set up for reading).
- In the case of writing, we could in principle let the process continue immediately after sending the data off to the controller. The process need not wait for the controller-to-disk transfer to finish. But if it issued another write request immediately, that would get in the way of the previous one. To make things simpler, we block the process until the write operation is done.

When we put the process to sleep we use the waiting state `TSTATE_WAITHD`$_{180a}$ defined on page 180.

⟨*ide: put process to sleep* 531a⟩≡ (530) [531a]
```
if (scheduler_is_active) {
  // interrupts are off; we access the thread table
  block (&harddisk_queue, TSTATE_WAITHD);
  ⟨end critical section in kernel 380b⟩
  ⟨resign 221d⟩
} else {
  ⟨end critical section in kernel 380b⟩
}
```
Uses harddisk_queue 529a, scheduler_is_active 276e, and TSTATE_WAITHD 180a.

The block read/write functions are now implemented as follows:

⟨*function implementations* 100b⟩+≡ (44a) ◁530d 532d▷ [531b]
```
void readblock_hd (int hd, int blockno, char *buffer) {
  readsector_hd (hd, blockno*2,   buffer);
  readsector_hd (hd, blockno*2+1, buffer + HD_SECSIZE);
}

void writeblock_hd (int hd, int blockno, char *buffer) {
  writesector_hd (hd, blockno*2,   buffer);
  writesector_hd (hd, blockno*2+1, buffer + HD_SECSIZE);
}
```
Defines:
readblock_hd, used in chunk 506b.
writeblock_hd, used in chunks 507b and 529c.

After a read operation has finished (and the controller has generated an interrupt) we can copy the read data from the controller's memory to the buffer. That transfer happens in the interrupt handler, here we only let the process wait for an interrupt.

[532a] ⟨*ide: read data from the controller* 532a⟩≡ (530c)
```
idewait (0);
inportb (IO_IDE_STATUS);  // read status, ack irq
```
Uses idewait 533b, inportb 133b, and IO_IDE_STATUS 525b.

Writing is similar:

[532b] ⟨*ide: write data to the controller* 532b⟩≡ (530d)
```
inportb          (IO_IDE_STATUS);                          // read status, ack irq
repeat_outportsl (IO_IDE_DATA, hd_buf, HD_SECSIZE / 4);
inportb          (IO_IDE_STATUS);                          // read status, ack irq
```
Uses hd_buf 530a, HD_SECSIZE 529d, inportb 133b, IO_IDE_DATA 525b, IO_IDE_STATUS 525b, and repeat_outportsl 528e.

Now the only missing bit is the interrupt handler which will only acknowledge the interrupt and possibly wake up a waiting process.

13.5.4 Interrupt Handler

The interrupt handler for the IDE controller

[532c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁529c 533a▷
```
void ide_handler (context_t *r);
```

will be executed whenever the controller finishes an operation and signals the CPU. What it has do then depends on the transfer direction:

- In case of a write operation, writesector_hd$_{530d}$ had filled the buffer, copied the data from there into the controller's memory and asked the controller to start the write operation onto the disk. So when that is finished, the whole operation is completed, and the interrupt handler only has to wake up the waiting process.
- The situation is different during a read operation: In that case the readsector_hd$_{530c}$ function had only asked the controller to read the data from disk and store them in the controller's memory. When the controller signals completion, the sector is waiting there (in the controller memory) to be retrieved. Thus, the interrupt handler must copy the data to system memory. Once it has finished that, it can also wake up the waiting process.

[532d] ⟨*function implementations* 100b⟩+≡ (44a) ◁531b 533b▷
```
void ide_handler (context_t *r) {
  switch (hd_direction) {
    case HD_OP_READ:  repeat_inportsl (IO_IDE_DATA, hd_buf, HD_SECSIZE / 4);
                      hd_direction = HD_OP_NONE;
                      break;

    case HD_OP_WRITE: hd_direction = HD_OP_NONE;
                      break;

    case HD_OP_NONE:  printf ("Funny IDE interrupt -- no request waiting\n");
                      return;
```

```
    }

    if (scheduler_is_active) {
      int tid;
      if ((tid = harddisk_queue.next) != 0)
        deblock (tid, &harddisk_queue);     // wake up process
    }
  }
```

Defines:
ide_handler, used in chunks 532c and 534b.
Uses context_t 142a, deblock 186b, harddisk_queue 529a, hd_buf 530a, hd_direction 530a, HD_OP_NONE 529d, HD_OP_READ 529d, HD_OP_WRITE 529d, HD_SECSIZE 529d, IO_IDE_DATA 525b, printf 601a, repeat_inportsl 528c, and scheduler_is_active 276e.

The last function we need to discuss in the context of reading form or writing to disk is

⟨*function prototypes* 45a⟩+≡ (44a) ◁532c 536a▷ [533a]

```
int idewait (int checkerr);
```

which waits if the IDE controller is not yet ready to receive the next command. It checks the IO_IDE_STATUS$_{525b}$ register's flags DRDY (device ready; bit 6) and BSY (busy; bit 7) and loops until the controller is ready and not busy.

⟨*function implementations* 100b⟩+≡ (44a) ◁532d 534b▷ [533b]

```
int idewait (int checkerr) {
  ⟨enable interrupts 47b⟩
  int r;
  for (;;) {
    r = inportb (IO_IDE_STATUS);
    if ((r & (IDE_BSY | IDE_DRDY)) == IDE_DRDY)  break;  // ready, not busy
  }
  if (checkerr && (r & (IDE_DF|IDE_ERR)) != 0) {
    ⟨disable interrupts 47a⟩
    return -1;
  } else {
    if (current_task > 1) { ⟨disable interrupts 47a⟩ }  // see comment
    return 0;
  }
}
```

Defines:
idewait, used in chunks 526, 532a, and 533a.
Uses busy, current_task 192c, IDE_BSY 525a, IDE_DF 525a, IDE_DRDY 525a, IDE_ERR 525a, inportb 133b, and IO_IDE_STATUS 525b.

(idewait$_{533b}$() must enable interrupts and disable them again after an IDE interrupt has occurred. With the check for current_task$_{192c}$ > 1 we treat the special case of disk access before scheduling has started, i. e., when we load the init program.)

13.5.5 Hard Disk Initialization

When the system boots we check what hard disks (if any) are available. We support up to two IDE disks and store their disk sizes in the

[534a] ⟨*global variables* 92b⟩+≡ (44a) ◁530a 536d▷
```
ulonglong hd_size[2] = {-1, -1};
```
Defines:
hd_size, used in chunks 499a and 534b.
Uses ulonglong 46b.

record. If a disk is not available we keep the −1 value.

disk identification

The ata_init$_{534b}$ function selects a disk by sending the encoded disk number to the IO_IDE_DISKSEL$_{525b}$ port and then asks for *identification* by sending the IDE_CMD_IDENT$_{525a}$ command to port IO_IDE_COMMAND$_{525b}$. The answer is 512 bytes long and copied into a buffer using the repeat_inportsl$_{528c}$ function. The disk size is then assembled from four bytes and written to the right hd_size$_{534a}$ array entry (and displayed in the boot messages).

We also use this function to install the interrupt handler.

[534b] ⟨*function implementations* 100b⟩+≡ (44a) ◁533b 536b▷
```
void ata_init () {
  // detect installed hard disks
  word buf[512]; short drivecount = 0;
  char *names[2] = { "hda", "hdb" };
  printf ("ATA: ");
  for (int disk = 0;  disk < 2;  disk++) {
    outportb (IO_IDE_DISKSEL, 0xe0 | (disk<<4));          // select disk
    for (int i = 0;  i < 1000;  i++) {
      if (inportb (IO_IDE_STATUS) != 0) {
        drivecount++;
        outportb (IO_IDE_COMMAND, IDE_CMD_IDENT);         // identify!
        repeat_inportsl (IO_IDE_DATA, buf, 256);          // 512 bytes = 256 words
        hd_size[disk] =   (ulonglong)buf[100]       + (((ulonglong)buf[101])<<16)
                        + (((ulonglong)buf[102])<<32) + (((ulonglong)buf[103])<<48);
        if (drivecount > 1) printf (", ");
        printf ("%s (%d KByte)", names[disk],
                hd_size[disk]/2);                         // 512-byte sectors!
        break;
      }
    }
  }
  printf ("\n");
  outportb (IO_IDE_DISKSEL, 0xe0 | (0<<4));               // select disk 0

  // install the interrupt handler
  install_interrupt_handler (IRQ_IDE, ide_handler);
  enable_interrupt (IRQ_IDE);
}
```
Defines:
ata_init, used in chunk 45c.
Uses enable_interrupt 140b, hd_size 534a, IDE_CMD_IDENT 525a, ide_handler 532d, inportb 133b, install_interrupt_handler 146c, IO_IDE_COMMAND 525b, IO_IDE_DATA 525b, IO_IDE_DISKSEL 525b, IO_IDE_STATUS 525b, IRQ_IDE 132 525a, outportb 133b, printf 601a, repeat_inportsl 528c, and ulonglong 46b.

13.6 The Floppy Controller

In the previous sections you have already seen two ways to talk to a device controller:

- We accessed the "serial hard disk" by sending individual bytes across the serial port. In one direction they contained controller commands and data (sectors to be written on the disk), in the other direction only data (sectors read from the disk). Every single received byte caused an interrupt, and so the sector had to be assembled byte by byte.
- For the IDE controller we used some kind of block transfer where we copied a sector between the PC's memory and the controller's internal memory. For that purpose we used a global buffer, though that was not strictly necessary; we could have used the memory location that the read/write functions use for storing the sector, i. e., memory that belongs to a process.

  The transfer between controller memory and the actual disk was performed by the controller itself, and we had to wait for that transfer to complete. This type of data transfer is called PIO transfer (Parallel I/O).

Now we show you a third way that uses *DMA* transfer (*Direct Memory Access*). Here we need to work with a global buffer and we also need to know the physical address of that buffer because the floppy controller will access it directly—it cannot use the MMU to translate a virtual address. Once the controller has been told what to do, the data transfer from or to that buffer happens automatically, no further activity by the CPU is required. That is possible because the controller can access the memory bus (just like the CPU does). This is most interesting in case of a read operation: Our code only has to tell the controller that it shall read a certain sector from the floppy, and the next time the controller generates an interrupt, the data will already be stored in the buffer—we need not call `repeat_inportsl`$_{528c}$ or an equivalent instruction, as in our hard disk driver. DMA

(Note that the IDE controller also supports DMA transfers. We have decided to let it work in PIO mode so that you can see both approaches at work. However, PIO transfers increase the load on the CPU, so if performance was your goal, you would have to replace the PIO code with DMA code.)

13.6.1 Talking to the Controller

The floppy controller has several ports that can be used to communicate with it; either for sending it a command or data or for reading data. These ports are the following:

⟨ *constants* 112a ⟩+≡ (44a) ◁529d 536c▷ [535]

```
#define IO_FLOPPY_OUTPUT      0x3f2   // digital output register (DOR)
#define IO_FLOPPY_STATUS      0x3f4   // main status register (MSR)
#define IO_FLOPPY_COMMAND     0x3f5   // command/data register
#define IO_FLOPPY_RATE        0x3f7   // configuration control register
```

Defines:
IO_FLOPPY_COMMAND, used in chunks 536b and 537a.
IO_FLOPPY_OUTPUT, used in chunks 544, 551a, and 552c.
IO_FLOPPY_RATE, used in chunks 542c and 552c.
IO_FLOPPY_STATUS, used in chunks 536b and 537a.

In most cases we will use the function `fdc_out`$_{536b}$ to send a command to the controller and read in the results with `fdc_getresults`$_{537a}$. These functions

[536a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁533a 538a▷
```
void fdc_out (byte data);
int fdc_getresults ();
```

work as follows:

[536b] ⟨*function implementations* 100b⟩+≡ (44a) ◁534b 537a▷
```
void fdc_out (byte data) {
  for (int i = 0;  i < 10000;  i++) {
    byte status = inb_delay (IO_FLOPPY_STATUS) & (FLOPPY_MASTER | FLOPPY_DIRECTION);
    if (status != FLOPPY_MASTER) continue;
    outb_delay (IO_FLOPPY_COMMAND, data);
    return;
  }
  fdc_need_reset = true;  printf ("FDC: can't send byte %w to controller\n", data);
}
```
Defines:
`fdc_out`, used in chunks 536a, 540, 542c, 548b, and 551.
Uses `fdc_need_reset` 536d, `FLOPPY_DIRECTION` 536c, `FLOPPY_MASTER` 536c, `inb_delay` 538b, `IO_FLOPPY_COMMAND` 535, `IO_FLOPPY_STATUS` 535, `outb_delay` 538b, and `printf` 601a.

Before we can write to the controller we need to check whether it is ready. We read the status from the status register via the `IO_FLOPPY_STATUS`$_{535}$ port. We are only interested in the highest two bits of the status register that tell us whether it is ready (bit 7) and whether it is prepared for a write operation (bit 6). So we mask the returned status value with (`FLOPPY_MASTER`$_{536c}$ | `FLOPPY_DIRECTION`$_{536c}$):

[536c] ⟨*constants* 112a⟩+≡ (44a) ◁535 537b▷
```
#define FLOPPY_DIRECTION     0b01000000  // bit 6 of status reg.
#define FLOPPY_MASTER        0b10000000  // bit 7 of status reg.
```
Defines:
`FLOPPY_DIRECTION`, used in chunks 536b and 537b.
`FLOPPY_MASTER`, used in chunks 536 and 537.

data/command register

If only bit 7 is set in the resulting value, then we know that the controller is ready and expects a write operation. Then we can send the byte to the *data/command register* via the `IO_FLOPPY_COMMAND`$_{535}$ port; otherwise we loop until the status changes to what we need.

Sometimes a single byte (that was sent to the controller) constitutes a complete command, but often we need to send a sequence. The controller can tell from the first byte how many more bytes follow. When the command is complete, the controller executes it and generates a result that may consist of a sequence of bytes as well.

If the loop completes without managing to send the byte, the controller needs to be reset. We store that information in the

[536d] ⟨*global variables* 92b⟩+≡ (44a) ◁534a 537d▷
```
static volatile int fdc_need_reset = 0;
```
Defines:
`fdc_need_reset`, used in chunks 536b, 537a, 539, 540, 546–48, and 551.

variable and return.

We read the result in the following function:

⟨*function implementations* 100b⟩+≡ (44a) ◁536b 538b▷ [537a]

```
int fdc_getresults () {
  int i, results = 0;
  if (fdc_need_reset) { printf ("exit_\n"); return 0; } ;

  for (i = 0; i < 30000; i++) {
    byte status = inb_delay (IO_FLOPPY_STATUS) & FLOPPY_NEW_BYTE;
    if (status == FLOPPY_MASTER)  return true;   // results are complete
    if (status != FLOPPY_NEW_BYTE) continue;
    if (results == MAX_FLOPPY_RESULTS) break;
    fdc_results[results++] = inb_delay (IO_FLOPPY_COMMAND);
  }

  fdc_need_reset = true;  printf ("FDC: reply error\n");
  return false;
}
```

Defines:
`fdc_getresults`, used in chunks 540d, 548b, and 551.
Uses `fdc_need_reset` 536d, `fdc_results` 537d, `FLOPPY_MASTER` 536c, `FLOPPY_NEW_BYTE` 537b, `inb_delay` 538b, `IO_FLOPPY_COMMAND` 535, `IO_FLOPPY_STATUS` 535, `MAX_FLOPPY_RESULTS` 537c, and `printf` 601a.

If the last byte has been read, the status changes to $\texttt{FLOPPY\_MASTER}_{536c}$ and our function can return. We check whether the status is $\texttt{FLOPPY\_NEW\_BYTE}_{537b}$ (i. e., the bits 4, 6 and 7 are set which indicates that the controller is busy, we're reading from the controller and it is ready to have us query it). If this is not yet the case, we repeat until the status changes to $\texttt{FLOPPY\_NEW\_BYTE}_{537b}$: Then we can read the new byte via the $\texttt{IO\_FLOPPY\_COMMAND}_{535}$ port.

If we exceed the maximum number of bytes that we expect the controller to send, we cancel the operation and set the $\texttt{fdc\_need\_reset}_{536d}$ flag.

⟨*constants* 112a⟩+≡ (44a) ◁536c 537c▷ [537b]

```
#define FLOPPY_CONTROLLER_BUSY    0b00010000  // bit 4 of status reg., busy
#define FLOPPY_NEW_BYTE  (FLOPPY_MASTER | FLOPPY_DIRECTION | FLOPPY_CONTROLLER_BUSY)
```

Defines:
`FLOPPY_NEW_BYTE`, used in chunk 537a.
Uses `busy`, `FLOPPY_DIRECTION` 536c, and `FLOPPY_MASTER` 536c.

When we read the results, we store them in the $\texttt{fdc\_results}_{537d}$ buffer:

⟨*constants* 112a⟩+≡ (44a) ◁537b 539a▷ [537c]

```
#define MAX_FLOPPY_RESULTS              0x07
```

Defines:
`MAX_FLOPPY_RESULTS`, used in chunk 537.

⟨*global variables* 92b⟩+≡ (44a) ◁536d 538c▷ [537d]

```
byte fdc_results[MAX_FLOPPY_RESULTS];
```

Defines:
`fdc_results`, used in chunks 537a, 540d, 548b, and 551b.
Uses `MAX_FLOPPY_RESULTS` 537c.

The functions

[538a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁536a 538d▷
```
void outb_delay (word __port, byte __value);
byte inb_delay (word __port);
```

do the same as `outportb`$_{133b}$`()` and `inportb`$_{133b}$`()`, but they execute an extra `outb` to an unused port (`0xE0`) in order to create a short delay. It does not matter which value is sent to the port, so they just send `al` (but could use any other value):

[538b] ⟨*function implementations* 100b⟩+≡ (44a) ◁537a 539c▷
```
/**** FROM proc/i386.h *********/
void outb_delay (word __port, byte __value) {
  asm volatile ("outb %0,%1; \
                 outb %%al,$0xE0"      :
                /* no output */        :
                "a" (__value),
                "dN" (__port)
                /* "eax","edx" */ );
}

byte inb_delay (word __port) {
  byte data;
  asm volatile ("inb %1,%0; \
                 outb %%al,$0xE0"      :
                "=a" (data)            :
                "dN" (__port)
                /* "eax","edx" */ );
  return data;
}
```
Defines:
`inb_delay`, used in chunks 536b, 537a, and 552c.
`outb_delay`, used in chunks 536b, 538a, 542–44, 551a, and 552c.

13.6.2 Setting Up the DMA Transfer

As we already mentioned, we will use a global buffer and declare its address here:

[538c] ⟨*global variables* 92b⟩+≡ (44a) ◁537d 539b▷
```
static char *fdc_buf = (char *)0x9a800;
```
Defines:
`fdc_buf`, used in chunks 543a, 549c, and 550b.

We have to pick the address manually because there are limitations on which memory areas can be used for DMA transfers.

The central function of the floppy driver which also handles the DMA setup is

[538d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁538a 542b▷
```
int fdc_command (int cmd, int drive, int track, int sector);
```

It takes four parameters: `cmd` is set to either `FLOPPY_READ`$_{539a}$ or `FLOPPY_WRITE`$_{539a}$ to indicate the transfer direction, and the last three parameters describe the sector in terms of the physical layout of a floppy disk.

⟨ *constants* 112a⟩+≡ (44a) ◁537c 540a▷ [539a]

```
#define FLOPPY_READ     0xe6
#define FLOPPY_WRITE    0xc5
```

Defines:
`FLOPPY_READ`, used in chunks 543a and 549c.
`FLOPPY_WRITE`, used in chunks 540d and 550b.

`fdc_command`$_{539c}$ first sets the three variables

⟨ *global variables* 92b⟩+≡ (44a) ◁538c 541c▷ [539b]

```
static int fdc_drive, fdc_track, fdc_head;
```

Defines:
`fdc_drive`, used in chunks 539c, 540c, 544, 547d, 548b, and 551b.
`fdc_head`, used in chunks 539, 540, and 548b.
`fdc_track`, used in chunks 539, 540, and 548b.

to the corresponding argument values; they will also be accessed by other functions of the floppy driver, so using global variables we can avoid passing these around as parameters.

Then the function resets the controller (if needed), starts the motor, lets the drive seek to the right track (we have to do that manually) and initiates the DMA transfer (see the ⟨ *fdc transfer* 540c⟩ chunk):

⟨ *function implementations* 100b⟩+≡ (44a) ◁538b 542c▷ [539c]

```
int fdc_command (int cmd, int drive, int track, int sector) {
  fdc_drive  = drive;
  fdc_track  = track;
  fdc_head   = sector / current_fdd_type->sectors;
  int fdc_sector = sector % current_fdd_type->sectors + 1;

  fdc_ticks_till_motor_stops = 3 * HZ;

  ⟨ begin critical section in kernel 380a⟩
  // will be re-enabled in fdc_read_/write_sector
  for (int err = 0;  err < MAX_FLOPPY_ERRORS;  err++) {
    if (fdc_need_reset)  fdc_reset ();
    ⟨ fdc start motor 544a⟩
    if (!fdc_seek ())  continue;
    ⟨ fdc transfer 540c⟩
    switch (transfer_status) {
      case -1: printf ("FDC: disk in drive %d is write protected\n", fdc_drive);
               return 0;
      case  0: continue;
      case  1: return 1;
    }
  }
  return 0;
}
```

Defines:
fdc_command, used in chunks 538d, 549c, and 550b.
Uses current_fdd_type 541c, fdc_drive 539b, fdc_head 539b, fdc_need_reset 536d, fdc_reset 551a, fdc_seek, fdc_ticks_till_motor_stops 546c, fdc_track 539b, HZ 540a, MAX_FLOPPY_ERRORS 540b, printf 601a, and write 429b.

[540a] ⟨*constants* 112a⟩+≡ (44a) ◁539a 540b▷
```
#define HZ 100    // frequency of the timer
```
Defines:
HZ, used in chunks 539c and 547a.

We allow up to eight floppy errors before we fail:

[540b] ⟨*constants* 112a⟩+≡ (44a) ◁540a 541a▷
```
#define MAX_FLOPPY_ERRORS               0x08
```
Defines:
MAX_FLOPPY_ERRORS, used in chunk 539c.

With all the information available we can initiate the transfer which means sending a longer sequence of bytes to the command/data register. We first tell the controller that we want to transfer in DMA mode (and not in PIO mode) which requires another command sequence shown in ⟨*fdc dma init* 543a⟩.

[540c] ⟨*fdc transfer* 540c⟩≡ (539c) 540d▷
```
int transfer_status = 0;  // will be set to 1 when successful
int sectors;              // number of transmitted sectors
⟨begin critical section in kernel 380a⟩

if (!fdc_need_reset && current_fdd->motor && current_fdd->calibrated) {
  ⟨fdc dma init 543a⟩
  fdc_mode ();
  fdc_out (cmd);   fdc_out (fdc_head << 2 | fdc_drive);
  fdc_out (fdc_track); fdc_out (fdc_head); fdc_out (fdc_sector);
  fdc_out (current_fdd_type->sectorsize);      // 2: 512 bytes/sector
  fdc_out (current_fdd_type->sectors);         // end of track
  fdc_out (current_fdd_type->gap);             // gap length
  fdc_out (FLOPPY_DTL);                        // data length
```
Uses current_fdd 541c, current_fdd_type 541c, fdc_drive 539b, fdc_head 539b, fdc_mode 542c, fdc_need_reset 536d, fdc_out 536b, fdc_track 539b, and FLOPPY_DTL 541a.

We need not terminate the sequence because the controller knows when it has received a complete command. So we can immediately continue by waiting for the answer (via wait_fdc_interrupt$_{547d}$) and call fdc_getresults$_{537a}$ to check whether the transfer was successful:

[540d] ⟨*fdc transfer* 540c⟩+≡ (539c) ◁540c
```
  if (!fdc_need_reset && !wait_fdc_interrupt () && fdc_getresults ()) {
    if (cmd == FLOPPY_WRITE && fdc_results[1] & WRITE_PROTECTED) {
      fdc_out (FLOPPY_SENSE);
      fdc_getresults ();
      transfer_status = -1;
    } else if ((fdc_results[0] & TEST_BITS) != TRANSFER_OK ||
```

```
        fdc_results[1] || fdc_results[2]) {
      current_fdd->calibrated = 0;
      transfer_status = 0;
    } else {
      sectors = (fdc_results[3] - fdc_track) * current_fdd_type->sectors * 2
              + (fdc_results[4] - fdc_head)  * current_fdd_type->sectors
              +  fdc_results[5] - fdc_sector;
      if (sectors == 1)  transfer_status = 1;   // success
    }
  }
}
```

Uses current_fdd 541c, current_fdd_type 541c, fdc_getresults 537a, fdc_head 539b, fdc_need_reset 536d, fdc_out 536b, fdc_results 537d, fdc_track 539b, FLOPPY_SENSE 549a, FLOPPY_WRITE 539a, TEST_BITS 548c, TRANSFER_OK 541a, wait_fdc_interrupt 547d, and WRITE_PROTECTED 541a.

⟨*constants* 112a⟩+≡ (44a) ◁540b 542a▷ [541a]

```
#define FLOPPY_DTL                  0xFF
#define TRANSFER_OK                 0x00
#define WRITE_PROTECTED             0x02
```

Defines:
FLOPPY_DTL, used in chunk 540c.
TRANSFER_OK, used in chunk 540d.
WRITE_PROTECTED, used in chunk 540d.

Both when sending the request and when checking whether the sector was successfully read we need the device information which declares the physical properties of the disk drive: In recent years only 3.5” drives with a formatted capacity of 1440 KByte have been built into PCs (if at all), but older machines used 5.25” drives with a 1200 KByte capacity. We store the information that we need to tell the controller in $\texttt{fdd\_type}_{541c}$:

⟨*type definitions* 91⟩+≡ (44a) ◁515a [541b]

```
typedef struct {
  int total_sectors, tracks, sectors, sectorsize, trackstep, rate, gap, spec1;
} struct_fdd_type;

typedef struct {
  int present, calibrated, motor, current_track, type;
} struct_fdd;
```

Defines:
struct_fdd, used in chunk 541c.
struct_fdd_type, used in chunk 541c.

⟨*global variables* 92b⟩+≡ (44a) ◁539b 544d▷ [541c]

```
char *fdd_drive_name[6] = {
  "not installed", "360 KByte (not supported)",
  "1200 KByte",    "720 KByte (not supported)",
  "1440 KByte",    "2880 KByte (not supported)" };

struct_fdd_type fdd_type[2] = {
  { 80*15*2, 80, 15, 2, 0, 0, 0x1B, 0xDF },          /* 1.2M   */
  { 80*18*2, 80, 18, 2, 0, 0, 0x1B, 0xCF }      };   /* 1.44M  */
```

```
struct_fdd_type *current_fdd_type;
struct_fdd fdd[2] = { { 0, 0, 0, INVALID_TRACK, 0 },  { 0, 0, 0, INVALID_TRACK, 0 } };
int fdds_in_use[2] = { 0, 0 };
struct_fdd *current_fdd;
```

Defines:
current_fdd, used in chunks 540, 544a, 548b, 549d, and 551b.
current_fdd_type, used in chunks 539, 540, 542c, 543a, 548b, and 549d.
fdd, used in chunks 499a, 544, 547a, 549d, 551a, and 552c.
fdd_drive_name, used in chunk 552c.
fdd_type, used in chunks 499a and 549d.
Uses INVALID_TRACK 542a, struct_fdd 541b, and struct_fdd_type 541b.

[542a] ⟨*constants* 112a⟩+≡ (44a) ◁541a 542d▷

```
#define INVALID_TRACK   -1
```

Defines:
INVALID_TRACK, used in chunks 541c and 551b.

When we initialize the system, we will detect the available floppies and write the information into $\texttt{fdd}_{541c}$, see Section 13.6.7.

configuration control register

We have been using an $\texttt{fdc\_mode}_{542c}$ function that tells the controller what kind of drive it has to access, but we have not shown its implementation yet. It uses $\texttt{fdc\_out}_{536b}$ but also writes to the *configuration control register* via the $\texttt{IO\_FLOPPY\_RATE}_{535}$ port:

[542b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁538d 545a▷

```
void fdc_mode ();
```

[542c] ⟨*function implementations* 100b⟩+≡ (44a) ◁539c 545b▷

```
void fdc_mode () {
  fdc_out (FLOPPY_SPECIFY);
  fdc_out (current_fdd_type->spec1);
  fdc_out (FLOPPY_SPEC2);
  outb_delay (IO_FLOPPY_RATE, current_fdd_type->rate & ~0x40);
}
```

Defines:
fdc_mode, used in chunks 540c and 542b.
Uses current_fdd_type 541c, fdc_out 536b, FLOPPY_SPEC2 542d, FLOPPY_SPECIFY 542d, IO_FLOPPY_RATE 535, and outb_delay 538b.

[542d] ⟨*constants* 112a⟩+≡ (44a) ◁542a 542e▷

```
#define FLOPPY_SPECIFY      0x03
#define FLOPPY_SPEC2        0x06
```

Defines:
FLOPPY_SPEC2, used in chunk 542c.
FLOPPY_SPECIFY, used in chunk 542c.

DMA controller

For setting up the DMA transfer we need to talk to the *DMA controller* which uses its own ports for configuring:

[542e] ⟨*constants* 112a⟩+≡ (44a) ◁542d 543b▷

```
#define IO_DMA0_INIT        0x0A   // single channel mask register
#define IO_DMA0_MODE        0x0B   // mode register
#define IO_DMA0_FLIPFLOP    0x0C   // flip-flop reset register
```

```
#define IO_DMA_PAGE_2       0x81   // page    register for DMA channel 2
#define IO_DMA_ADDR_2       0x04   // address register for DMA channel 2
#define IO_DMA_COUNT_2      0x05   // count   register for DMA channel 2

#define DMA_READ_MODE       0x44
#define DMA_WRITE_MODE      0x48
```

Defines:
DMA_READ_MODE, used in chunk 543a.
DMA_WRITE_MODE, used in chunk 543a.
IO_DMA0_FLIPFLOP, used in chunk 543a.
IO_DMA0_INIT, used in chunk 543a.
IO_DMA0_MODE, used in chunk 543a.
IO_DMA_ADDR_2, used in chunk 543a.
IO_DMA_COUNT_2, used in chunk 543a.
IO_DMA_PAGE_2, used in chunk 543a.

The important bit about the following code chunk is that we tell the DMA controller which chunk of memory it shall use as buffer for the DMA data transfer. The controller only accepts 24 bit wide physical addresses. We tell it our buffer address `fdc_buf`$_{538c}$ by sending the lowest 16 bits to `ID_DMA_ADDR_2` and the highest eight bits to `IO_DMA_PAGE_2`$_{542e}$. That stores the lower 16 bits in the controller's *address register* and the eight extra bits in the *page register*. The reason for this separate treatment is compatibility: Older DMA controllers only supported 16-bit addresses. The amount of bytes to read or write must be written to the *count register* via `IO_DMA_COUNT_2`$_{542e}$. It takes a 16-bit value which requires two `outb` commands.

address register
page register
count register

⟨*fdc dma init* 543a⟩≡ (540c) [543a]

```
int count = 1 << (current_fdd_type->sectorsize + 7);    // = 512
int mode;
if (cmd == FLOPPY_READ)
  mode = DMA_READ_MODE;                                 // prepare read operation
else
  mode = DMA_WRITE_MODE;                                // prepare write operation

outb_delay (IO_DMA0_INIT,     FLOPPY_CHANNEL | 4);      // disable DMA channel
outb_delay (IO_DMA0_FLIPFLOP, 0);                       // clear DMA ch. flipflop
outb_delay (IO_DMA0_MODE,     mode | FLOPPY_CHANNEL);   // set DMA ch. mode (r/w)
// set count, address and page registers
outb_delay (IO_DMA_COUNT_2,   (byte)(count-1));         // count
outb_delay (IO_DMA_COUNT_2,   (byte)((count-1) >> 8));
outb_delay (IO_DMA_ADDR_2,    (byte)(unsigned)fdc_buf); // address,    bits  0.. 7
outb_delay (IO_DMA_ADDR_2,    (byte)((unsigned)fdc_buf >> 8));      // bits  8..15
outb_delay (IO_DMA_PAGE_2,    (unsigned)fdc_buf >> 16); // page,       bits 16..23
outb_delay (IO_DMA0_INIT,     FLOPPY_CHANNEL);          // enable DMA channel
```

Uses current_fdd_type 541c, DMA_READ_MODE 542e, DMA_WRITE_MODE 542e, fdc_buf 538c, FLOPPY_CHANNEL 543b, FLOPPY_READ 539a, IO_DMA0_FLIPFLOP 542e, IO_DMA0_INIT 542e, IO_DMA0_MODE 542e, IO_DMA_ADDR_2 542e, IO_DMA_COUNT_2 542e, IO_DMA_PAGE_2 542e, outb_delay 538b, read 429b, and write 429b.

⟨*constants* 112a⟩+≡ (44a) ◁542e 544c▷ [543b]

```
#define FLOPPY_CHANNEL                  0x02
```

Defines:
FLOPPY_CHANNEL, used in chunk 543a.

13.6.3 Starting and Stopping the Motor

In comparison to the hard disk controller, the floppy controller needs a lot of help to get things right. For example, it is necessary to manually turn the floppy motor on and off. The code chunks ⟨*fdc start motor* 544a⟩ and ⟨*fdc stop motor* 544b⟩ send the right commands:

[544a] ⟨*fdc start motor* 544a⟩≡ (539c 551b)
```
if (!current_fdd->motor) {
  outb_delay (IO_FLOPPY_OUTPUT, FLOPPY_CONTROLLER_ENABLE | FLOPPY_DMAINT_ENABLE |
                                fdc_drive | (16 << fdc_drive));
  current_fdd->motor = 1;
  fdd[!fdc_drive].motor = 0;
  for (int i = 0;  i < 500000;  i++);  // delay
}
```
Uses current_fdd 541c, fdc_drive 539b, fdd 541c, FLOPPY_CONTROLLER_ENABLE 544c, FLOPPY_DMAINT_ENABLE 544c, IO_FLOPPY_OUTPUT 535, and outb_delay 538b.

[544b] ⟨*fdc stop motor* 544b⟩≡ (547a)
```
outb_delay (IO_FLOPPY_OUTPUT,
            FLOPPY_CONTROLLER_ENABLE | FLOPPY_DMAINT_ENABLE | fdc_drive);
fdd[0].motor = fdd[1].motor = 0;
```
Uses fdc_drive 539b, fdd 541c, FLOPPY_CONTROLLER_ENABLE 544c, FLOPPY_DMAINT_ENABLE 544c, IO_FLOPPY_OUTPUT 535, and outb_delay 538b.

[544c] ⟨*constants* 112a⟩+≡ (44a) ◁543b 548c▷
```
#define FLOPPY_CONTROLLER_ENABLE        0x04
#define FLOPPY_DMAINT_ENABLE            0x08
```
Defines:
FLOPPY_CONTROLLER_ENABLE, used in chunks 544 and 551a.
FLOPPY_DMAINT_ENABLE, used in chunks 544 and 551a.

13.6.4 Handling Floppy Interrupts

We define a floppy_queue$_{544d}$ which will contain processes waiting for a floppy access operation to finish:

[544d] ⟨*global variables* 92b⟩+≡ (44a) ◁541c 545c▷
```
blocked_queue floppy_queue;   // processes which wait for the floppy
```
Defines:
floppy_queue, used in chunks 544–46, 564c, and 606.
Uses blocked_queue 183a.

and we have to initialize this queue:

[544e] ⟨*initialize system* 45b⟩+≡ (44b) ◁530b
```
initialize_blocked_queue (&floppy_queue);
```
Uses floppy_queue 544d and initialize_blocked_queue 183c.

Processes that access a floppy disk drive will wait on this queue, and the interrupt handler will wake up these processes when the operation was completed.

For example, when reading from an open file, we will call `fdc_read_sector`$_{549c}$ which in turn calls `fdc_command`$_{539c}$.

The last function potentially calls `fdc_reset`$_{551a}$, and it will call further functions all of which call `wait_fdc_interrupt`$_{547d}$.

`wait_fdc_interrupt`$_{547d}$ actually puts the process to sleep via `fdc_sleep`$_{545b}$ until an interrupt occurs.

Here is the implementation of the

⟨*function prototypes* 45a⟩+≡ (44a) ◁542b 545d▷ [545a]
```
void fdc_sleep ();
```

function:

⟨*function implementations* 100b⟩+≡ (44a) ◁542c 546a▷ [545b]
```
void fdc_sleep () {
  if ((current_task > 1) && scheduler_is_active) {
    // block process
    fdc_is_busy = true;
    ⟨begin critical section in kernel 380a⟩ // access thread table
    block (&floppy_queue, TSTATE_WAITFLP);
    ⟨end critical section in kernel 380b⟩
    ⟨resign 221d⟩
  }
  fdc_is_busy = false;
};
```
Defines:
`fdc_sleep`, used in chunks 545a and 547d.
Uses `current_task` 192c, `fdc_is_busy` 545c, `floppy_queue` 544d, `scheduler_is_active` 276e, and `TSTATE_WAITFLP` 180a.

It uses the global

⟨*global variables* 92b⟩+≡ (44a) ◁544d 546c▷ [545c]
```
short int fdc_is_busy = false;
```
Defines:
`fdc_is_busy`, used in chunk 545b.

variable to keep track of whether the floppy is currently working on a request.

When we need to wake up a process that has been waiting for the floppy drive, we use the

⟨*function prototypes* 45a⟩+≡ (44a) ◁545a 546d▷ [545d]
```
void fdc_wakeup ();
```

function. At any given time there can only be one active floppy operation, because (as you will see in the implementation of `fdc_read_sector`$_{549c}$ and `fdc_write_sector`$_{550b}$), all read and write operations are critical sections, protected by the lock `fdc_lock`$_{547b}$. So for `fdc_wakeup`$_{546a}$ we just wake the first process in the queue as there can only be one.

[546a] ⟨*function implementations* 100b⟩+≡ (44a) ◁545b 546b▷
```
void fdc_wakeup () {
  thread_id tid = floppy_queue.next;
  if (tid != 0)  deblock (tid, &floppy_queue);
}
```
Defines:
fdc_wakeup, used in chunks 545–47.
Uses deblock 186b, floppy_queue 544d, and thread_id 178a.

The actual interrupt handler for the floppy drives is rather simple: it first checks whether interrupts were expected at all, and if so, it wakes the waiting process.

[546b] ⟨*function implementations* 100b⟩+≡ (44a) ◁546a 547a▷
```
void floppy_handler (context_t *r) {
  fdc_timeout = false;
  if (!fdc_waits_interrupt)
    fdc_need_reset = 1;  // unexpected floppy interrupt, reset controller
  fdc_waits_interrupt = false;
  fdc_wakeup ();
}
```
Defines:
floppy_handler, used in chunk 552c.
Uses context_t 142a, fdc_need_reset 536d, fdc_timeout 546c, fdc_waits_interrupt 546c, and fdc_wakeup 546a.

We use two counters $\texttt{fdc\_ticks}_{546c}$ and $\texttt{fdc\_ticks\_till\_motor\_stops}_{546c}$: The first one starts counting when we have started a read or write operation. If it does not finish within two seconds (200 ticks) we abort the current operation (and fail). The second counter makes sure that the motor is stopped three seconds after the last operation completed. We don't turn the motor off immediately because further requests might follow. The flags $\texttt{fdc\_timeout}_{546c}$ and $\texttt{fdc\_waits\_interrupt}_{546c}$ show whether our two seconds have run out and whether we're waiting for an interrupt.

[546c] ⟨*global variables* 92b⟩+≡ (44a) ◁545c 547b▷
```
int     fdc_ticks                 = 0;
int     fdc_ticks_till_motor_stops = 0;
boolean fdc_timeout               = false;
boolean fdc_waits_interrupt       = false;
```
Defines:
fdc_ticks, used in chunk 547.
fdc_ticks_till_motor_stops, used in chunks 539c and 547a.
fdc_timeout, used in chunks 546 and 547.
fdc_waits_interrupt, used in chunks 546 and 547.

The $\texttt{fdc\_timer}_{547a}$ function will be called from the timer handler, so we need to declare it here:

[546d] ⟨*function prototypes* 45a⟩+≡ (44a) ◁545d 547c▷
```
void fdc_timer ();
```

We add it to the timer tasks which we have defined on page 342:

[546e] ⟨*timer tasks* 306d⟩+≡ (342b) ◁343b
```
fdc_timer ();
```
Uses fdc_timer 547a.

The floppy timer has two objectives: It checks whether a timeout has occurred (and cancels the current operation) and it checks if the motor has been running too long:

⟨*function implementations* 100b⟩+≡ (44a) ◁546b 547d▷ [547a]

```
void fdc_timer () {
  if (fdc_waits_interrupt && ++fdc_ticks > HZ * 2) {
    fdc_waits_interrupt = false;
    fdc_timeout = true;
    fdc_wakeup ();
  } else if ((fdd[0].motor | fdd[1].motor) &&
             ~(fdc_lock->l) && !--fdc_ticks_till_motor_stops) {
    ⟨fdc stop motor 544b⟩
  }
}
```

Defines:
fdc_timer, used in chunk 546.
Uses fdc_lock 547b, fdc_ticks 546c, fdc_ticks_till_motor_stops 546c, fdc_timeout 546c, fdc_waits_interrupt 546c, fdc_wakeup 546a, fdd 541c, and HZ 540a.

It only stops the motor if the lock $\texttt{fdc\_lock}_{547b}$ is not held. We have not declared it yet, but already mentioned it twice:

⟨*global variables* 92b⟩+≡ (44a) ◁546c 604a▷ [547b]

```
lock fdc_lock;
```

Defines:
fdc_lock, used in chunks 547a, 549, and 552c.
Uses lock 365a.

We will initialize it in $\texttt{fdc\_init}_{552c}$ ().

We also provide the function

⟨*function prototypes* 45a⟩+≡ (44a) ◁546d 548a▷ [547c]

```
int wait_fdc_interrupt ();
```

that waits for an interrupt. It will return from the $\texttt{fdc\_sleep}_{545b}$ call if either an interrupt has occurred or it has timed out:

⟨*function implementations* 100b⟩+≡ (44a) ◁547a 548b▷ [547d]

```
int wait_fdc_interrupt () {
  fdc_ticks = 0;                  // reset the wait time
  fdc_waits_interrupt = true;     // yes, we wait
  fdc_sleep ();
  if (fdc_timeout) {              // a timeout occurred
    fdc_need_reset = 1;
    printf ("FDC: drive %d timeout\n", fdc_drive);
  }
  return fdc_timeout;
}
```

Defines:
wait_fdc_interrupt, used in chunks 540d, 547c, 548b, and 551.
Uses fdc_drive 539b, fdc_need_reset 536d, fdc_sleep 545b, fdc_ticks 546c, fdc_timeout 546c, fdc_waits_interrupt 546c, and printf 601a.

13.6.5 Reading and Writing

Reading and writing require that we first move the drive head to the right location. This is performed by the

[548a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁547c 549b▷
```
int fdc_seek ();
```

function which calculates the physical parameters and sends them to the controller, using the FLOPPY_SEEK$_{549a}$ command.

[548b] ⟨*function implementations* 100b⟩+≡ (44a) ◁547d 549c▷
```
int fdc_seek () {
  if (fdc_need_reset ||
      (!current_fdd->calibrated && !fdc_recalibrate ()))  return false;
  if (fdc_track == current_fdd->current_track)  return true;

  ⟨begin critical section in kernel 380a⟩
  if (!current_fdd->motor)  return false;

  fdc_out (FLOPPY_SEEK);
  fdc_out (fdc_head << 2 | fdc_drive);
  fdc_out (fdc_track);

  if (fdc_need_reset || wait_fdc_interrupt ())  return false;

  current_fdd->current_track = fdc_track;
  fdc_out (FLOPPY_SENSE);

  if (!fdc_getresults ())  return false;
  if ((fdc_results[0] & TEST_BITS) != SEEK_OK ||
      fdc_results[1] != fdc_track * (current_fdd_type->trackstep + 1))
    return false;
  return true;
}
```
Uses current_fdd 541c, current_fdd_type 541c, fdc_drive 539b, fdc_getresults 537a, fdc_head 539b, fdc_need_reset 536d, fdc_out 536b, fdc_recalibrate 551b, fdc_results 537d, fdc_seek, fdc_track 539b, FLOPPY_SEEK 549a, FLOPPY_SENSE 549a, SEEK_OK 548c, TEST_BITS 548c, and wait_fdc_interrupt 547d.

Via fdc_getresults$_{537a}$ we check whether the seek operation was successful: In that case the highest five bits of fdc_results$_{537d}$[0] will be 00100_b.

[548c] ⟨*constants* 112a⟩+≡ (44a) ◁544c 549a▷
```
#define TEST_BITS     0b11111000   //   0xf8
#define SEEK_OK       0b00100000   //   0x20
```
Defines:
SEEK_OK, used in chunks 548b and 551b.
TEST_BITS, used in chunks 540d, 548b, and 551b.

The commands FLOPPY_SEEK$_{549a}$ and FLOPPY_SENSE$_{549a}$ perform the seek operation and request status information from the floppy drive which is required after every command.

⟨*constants* 112a⟩+≡ (44a) ◁548c 550a▷ [549a]
```
#define FLOPPY_SEEK                     0x0f
#define FLOPPY_SENSE                    0x08
```
Defines:
FLOPPY_SEEK, used in chunk 548b.
FLOPPY_SENSE, used in chunks 540d, 548b, and 551.

With seeking completed, we're ready to read or write.

⟨*function prototypes* 45a⟩+≡ (44a) ◁548a 550c▷ [549b]
```
int fdc_read_sector  (int device, int block, char *buffer);
int fdc_write_sector (int device, int block, char *buffer);
```

These functions read and write 512 byte sized sectors:

⟨*function implementations* 100b⟩+≡ (44a) ◁548b 550b▷ [549c]
```
int fdc_read_sector (int device, int block, char *buffer) {
  ⟨fdc: prepare read/write sector 549d⟩
  result = fdc_command (FLOPPY_READ, device, ctrack, csector);  // will sleep
  if (result) {
    memcpy ((void *)buffer, PHYSICAL(fdc_buf), FD_SECSIZE);
  }
  ⟨fdc: finish read/write sector 549e⟩
}
```
Defines:
fdc_read_sector, used in chunk 550d.
Uses csector, ctrack, FD_SECSIZE 550a, fdc_buf 538c, fdc_command 539c, FLOPPY_READ 539a, memcpy 596c, and PHYSICAL 116a.

with

⟨*fdc: prepare read/write sector* 549d⟩≡ (549c 550b) [549d]
```
int spt;                            // sectors per track
int ctrack, csector;
int result;

mutex_lock (fdc_lock);
  current_fdd      = &fdd[device];
  current_fdd_type = &fdd_type[current_fdd->type];

  spt = current_fdd_type->sectors * 2;  // 36 ??
  ctrack   = block / spt;
  csector  = block % spt;
```
Uses csector, ctrack, current_fdd 541c, current_fdd_type 541c, fdc_lock 547b, fdd 541c, fdd_type 541c, mutex_lock 366a, and spt.

and

⟨*fdc: finish read/write sector* 549e⟩≡ (549c 550b) [549e]
```
mutex_unlock (fdc_lock);
if (result) return FD_SECSIZE;
else        return 0;
```
Uses FD_SECSIZE 550a, fdc_lock 547b, and mutex_unlock 366c.

[550a] ⟨*constants* 112a⟩+≡ (44a) ◁549a 552a▷
```
#define FD_SECSIZE 512
```
Defines:
FD_SECSIZE, used in chunks 549 and 550.

Writing is similar, but the order of calling fdc_command$_{539c}$ and copying the buffer contents is reversed; also fdc_command$_{539c}$ supplies the argument FLOPPY_WRITE$_{539a}$ instead of FLOPPY_READ$_{539a}$, and the memcpy$_{596c}$ operation works the other way round:

[550b] ⟨*function implementations* 100b⟩+≡ (44a) ◁549c 550d▷
```
int fdc_write_sector (int device, int block, char *buffer) {
  ⟨fdc: prepare read/write sector 549d⟩
  memcpy (PHYSICAL(fdc_buf), (void *)buffer, FD_SECSIZE);
  result = fdc_command (FLOPPY_WRITE, device, ctrack, csector);  // will sleep
  ⟨fdc: finish read/write sector 549e⟩
}
```
Defines:
fdc_write_sector, used in chunks 549b and 550d.
Uses csector, ctrack, FD_SECSIZE 550a, fdc_buf 538c, fdc_command 539c, FLOPPY_WRITE 539a, memcpy 596c, and PHYSICAL 116a.

Since we will always read or write whole blocks (1 KByte) we add readblock_fd$_{550d}$ and writeblock_fd$_{550d}$ functions

[550c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁549b 550e▷
```
void readblock_fd  (int device, int blockno, char *buffer);
void writeblock_fd (int device, int blockno, char *buffer);
```

which simply call the sector functions twice:

[550d] ⟨*function implementations* 100b⟩+≡ (44a) ◁550b 551a▷
```
void readblock_fd  (int device, int blockno, char *buffer) {
  fdc_read_sector  (device, blockno*2,     buffer);
  fdc_read_sector  (device, blockno*2 + 1, buffer + FD_SECSIZE);
};

void writeblock_fd (int device, int blockno, char *buffer) {
  fdc_write_sector (device, blockno*2,     buffer);
  fdc_write_sector (device, blockno*2 + 1, buffer + FD_SECSIZE);
};
```
Defines:
readblock_fd, used in chunk 506b.
writeblock_fd, used in chunks 507b and 550c.
Uses FD_SECSIZE 550a, fdc_read_sector 549c, and fdc_write_sector 550b.

13.6.6 Resetting and Recalibrating

Two further functions

[550e] ⟨*function prototypes* 45a⟩+≡ (44a) ◁550c 552b▷
```
void fdc_reset ();
int  fdc_recalibrate ();
```

are required for our floppy driver implementation. `fdc_reset`$_{551a}$ is called when too many errors have occurred; in that case it asks the controller to reset so that a new attempt can be started.

⟨*function implementations* 100b⟩+≡ (44a) ◁550d 551b▷ [551a]

```
void fdc_reset () {
  ⟨begin critical section in kernel 380a⟩
  outb_delay (IO_FLOPPY_OUTPUT, FLOPPY_DMAINT_ENABLE);
  for (int i = 0;  i < 10000;  i++)  asm ("nop");   // wait a bit
  outb_delay (IO_FLOPPY_OUTPUT, FLOPPY_CONTROLLER_ENABLE | FLOPPY_DMAINT_ENABLE);

  fdc_need_reset = 0;
  fdd[0].calibrated = fdd[1].calibrated = 0;
  fdd[0].motor      = fdd[1].motor      = 0;

  if (wait_fdc_interrupt ())  printf ("FDC: can't reset controller (timeout)\n");
  fdc_out (FLOPPY_SENSE);
  if (!fdc_getresults ())  printf ("FDC: can't reset controller\n");
}
```

Defines:
`fdc_reset`, used in chunks 539c and 550e.
Uses `fdc_getresults` 537a, `fdc_need_reset` 536d, `fdc_out` 536b, `fdd` 541c, `FLOPPY_CONTROLLER_ENABLE` 544c, `FLOPPY_DMAINT_ENABLE` 544c, `FLOPPY_SENSE` 549a, `IO_FLOPPY_OUTPUT` 535, `outb_delay` 538b, `printf` 601a, and `wait_fdc_interrupt` 547d.

The `fdc_recalibrate`$_{551b}$ function sends the `FLOPPY_RECALIBRATE`$_{552a}$ command to the controller which is necessary when the drive is not calibrated; this should happen only once (when the drive is accessed for the first time).

⟨*function implementations* 100b⟩+≡ (44a) ◁551a 552c▷ [551b]

```
int fdc_recalibrate () {
  if (fdc_need_reset)  return 0;
  ⟨begin critical section in kernel 380a⟩
  ⟨fdc start motor 544a⟩
  fdc_out (FLOPPY_RECALIBRATE);
  fdc_out (fdc_drive);
  if (fdc_need_reset || wait_fdc_interrupt ())  return 0;
  fdc_out (FLOPPY_SENSE);
 if (!fdc_getresults () || (fdc_results[0] & TEST_BITS) != SEEK_OK || fdc_results[1])
    goto bad_recalibration;

  current_fdd->current_track = INVALID_TRACK;
  return current_fdd->calibrated = 1;

  bad_recalibration:
  printf ("FDC: can't recalibrate\n");
  fdc_need_reset = 1;
  return 0;
}
```

Defines:
fdc_recalibrate, used in chunk 548b.
Uses current_fdd 541c, fdc_drive 539b, fdc_getresults 537a, fdc_need_reset 536d, fdc_out 536b, fdc_results 537d, FLOPPY_RECALIBRATE 552a, FLOPPY_SENSE 549a, INVALID_TRACK 542a, printf 601a, SEEK_OK 548c, TEST_BITS 548c, and wait_fdc_interrupt 547d.

[552a] ⟨*constants* 112a⟩+≡ (44a) ◁550a 579b▷
```
#define FLOPPY_RECALIBRATE              0x07
```
Defines:
FLOPPY_RECALIBRATE, used in chunk 551b.

13.6.7 Floppy Driver Initialization

As with the hard disk driver, we also need to initialize the floppy driver when the system boots. This happens in the

[552b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁550e 553a▷
```
void fdc_init ();
```

function which initializes the locks, gathers the information about available floppy drives from the CMOS, enters them in the data structures (and displays them on the screen) and installs the interrupt handler for the floppy interrupt. It is comparable to the ata_init$_{534b}$ function.

[552c] ⟨*function implementations* 100b⟩+≡ (44a) ◁551b 553b▷
```
void fdc_init () {
  fdc_lock = get_new_lock ("fdc");                  // initialize lock

  outb_delay (IO_CMOS_CMD, 0x10);                   // read floppy status from CMOS
  int fdd_type_byte = inb_delay (IO_CMOS_DATA);

  int type; printf ("FDC: ");                       // enter and display data
  for (int i = 0;  i < 2;  i++) {
    // check floppy drive i
    if (i == 0) type = fdd_type_byte >> 4;    // upper 4 bits
    else        type = fdd_type_byte & 0x0F;  // lower 4 bits
    if ((fdd[i].present = (type == 2 || type == 4 || type == 5 )))
      fdd[i].type = fdc_map_type (type);
    printf ("fd%d (%s)%s", i, fdd_drive_name[type], (i==0) ? ", " : "\n");
  }

  if (fdd[0].present || fdd[1].present) {           // enable floppy handler
    install_interrupt_handler (IRQ_FDC, floppy_handler);
    enable_interrupt (IRQ_FDC);
    outportb (IO_FLOPPY_RATE,    0); // FDC Reset
    outportb (IO_FLOPPY_OUTPUT, 12); // enable DMA, disable Reset
  }
}
```

Defines:
fdc_init, used in chunks 45c and 552b.
Uses enable_interrupt 140b, fdc_lock 547b, fdc_map_type 553b, fdd 541c, fdd_drive_name 541c, floppy_handler 546b, get_new_lock 367b, inb_delay 538b, install_interrupt_handler 146c, IO_CMOS_CMD 339b, IO_CMOS_DATA 339b, IO_FLOPPY_OUTPUT 535, IO_FLOPPY_RATE 535, IRQ_FDC 132, lock 365a, outb_delay 538b, outportb 133b, printf 601a, and read 429b.

It uses the helper function

⟨*function prototypes* 45a⟩+≡ (44a) ◁552b 561b▷ [553a]
```
int fdc_map_type (int t);
```

that converts the floppy drive type (as seen in the CMOS) into an index into the $\texttt{fdd\_type}_{541c}[]$ table which contains description of the drive characteristics.

⟨*function implementations* 100b⟩+≡ (44a) ◁552c 562b▷ [553b]
```
int fdc_map_type (int t) {
  switch (t) {
    case 2:  return 0;   // 1.2  MByte drive
    case 4:  return 1;   // 1.44 MByte drive
    default: return -1;
  }
}
```
Defines:
fdc_map_type, used in chunks 552c and 553a.

Credits

As a final note we want to give credit to Tudor Hulubei who wrote the Thix Operating System [Hul95] and published the source code under the GPL 2 license. The whole floppy code in Section 13.6 is based on his floppy driver implementation, though we have removed a lot of the original code. For example, Thix uses several disk buffers so that floppy requests can be queued.

14 Signals

Signals are a classical Unix mechanism which allows a simple kind of messaging: processes can send signals to other processes which makes them either terminate or call a registered *signal handler*. In many ways these signals are very similar to interrupts, but while an interrupt handler can only be set up in kernel mode (and serves the whole operating system), signal handlers can be set up in user mode and belong to just one process.

signal handler

The similarity between signals and interrupts goes even further: Interrupts exist in two varieties, *synchronous* (e. g. interrupts caused by accessing a bad memory address or trying to execute an unknown CPU instruction) and *asynchronous* (e. g. raised by a device, such as the timer or a floppy or hard disk controller), and the same holds for signals: a *synchronous signal* is caused by the process itself (again access to a bad memory address is an example—in that case it causes an interrupt first and the interrupt handler sends a corresponding signal to the process), but most signals are *asynchronous* (sent by a different process).

synchronous signal

asynchronous signal

Some functionality is available in both worlds (interrupts and signals), take for example a timer: the computer's timer chip generates regular timer interrupts which are asynchronous events and invoke the kernel's timer interrupt handler. Besides other things, this handler checks whether one of the processes has registered a (process-private) timer—and if so, generates an alarm signal. When the process is scheduled the next time, instead of continuing its normal execution it enters its alarm signal handler and treats the asynchronous signal.

An example for synchronous events in both worlds is bad memory access. When a process tries to access a virtual address which is not available (because the page tables do not map it to some physical address), a page fault is generated, so the CPU jumps into the page fault (interrupt) handler. That one checks the reason (the only acceptable reason being that the page was paged out to disk). If that is not the case, the process must be terminated. To achieve this goal, the page fault handler sends the process a `SIGSEGV`$_{562a}$ (*segmentation*

`SIGSEGV`

violation) signal. When the process is scheduled again, it will normally abort, though it may have registered a SIGSEGV$_{562a}$ signal handler to deal with such a situation.

For the memory example, consider the following program `segfault.c`:

[556] ⟨*segfault.c* 556⟩≡
```
int main () {
  char *adr = (char *)0;
  char c = *adr;
  putchar (c);
}
```

and its execution via the debugger `gdb`:

```
$ gcc -g segfault.c
$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
(gdb) run
Starting program: /tmp/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400508 in main () at segfault.c:3
3         char c = *adr;
```

14.1 Use Cases for Signals

What are signals good for? In this section we show you three examples which demonstrate the range of application of signals.

Program Error: In many cases programs may run into a problem when they try to perform an action that the CPU will not allow, for example when trying to access an invalid memory address. This is normally an indication that the program is faulty, and the best action will be to terminate the process. However, instead of checking every memory address in the program before it is accessed, a developer may decide to rely on an error handler that will be called if such behavior is detected. So a solution could be to write an error handler (and install it as the signal handler for the signal that will be generated) that resets the program to some initial state and starts over. Then, when an error occurs, the processor jumps into the fault handler which will send a signal to the process. When the process continues, it will execute the signal handler.

Inter Process Communication: Processes that work together somehow, often need to communicate. Unix systems provide special mechanisms for sending complex messages, but if only some kind of "ping" is required to tell another process that a certain condition has been reached, then a signal can serve that purpose.

Voluntary Abort: A process may decide to terminate itself when it recognizes an error condition. Instead of calling `exit`$_{218a}$ with an error return value, it can use `abort` to send itself a SIGABRT$_{562a}$ signal.

SIGABRT

14.2 Signals in Classical Unix Systems

In classical Unix systems, processes can use the `kill` system call to deliver a signal to an arbitrary process (as long as both have the same owner or the signal-sending process belongs to `root`) or the `raise` system call to send a signal to themselves.

Every Unix system knows a few standard signals, with signal numbers typically ranging from 0 to 31. While some signals have standard signal numbers (such as 9 and 15 for `SIGKILL`$_{562a}$ and `SIGTERM`$_{562a}$), the POSIX standard does not require signals to use standard values; it only asks for signal names to be defined[1]:

SIGABRT (default action: abort)

Process abort signal. The signal could be sent by the process itself (see above), by a different process or by the kernel.

SIGALRM (default action: abort)

Unix systems normally supply an alarm clock mechanism. By defining a timer, this signal will be sent when the requested timeout occurs.

SIGBUS (default action: abort)

The process tried to access an invalid address. This is similar to **SIGSEGV**, but the latter deals with forbidden access to a valid address (an address that requires kernel privileges).

SIGCHLD (default action: ignore)

When a child process terminates, stops (**SIGSTOP**) or continues (**SIGCONT**), its parent process is notified with this signal.

SIGCONT (default action: continue)

A process that was stopped (via **SIGSTOP**) continues execution.

SIGFPE (default action: abort)

Originally the acronym FPE stood for *Floating Point Exception* and the signal was raised when the process caused the FPU (Floating Point Unit) to perform a faulting calculation, such as a division by zero. Today **SIGFPU** is used for all kinds of arithmetic errors.

SIGHUP (default action: abort)

The Hangup signal tells a process that its controlling terminal is gone. On a modern Unix system this often refers to a closed terminal window, traditionally it occurred when the connection of a dumb terminal device to the machine (via a serial line or a dial-in connection) was lost.

SIGILL (default action: abort)

The process tried to execute an illegal instruction (e. g., one that requires a newer processor with an extended instruction set).

SIGINT (default action: abort)

Pressing [Ctrl-C] generates this signal. It is possible to write a signal handler which may decide to ignore [Ctrl-C]. Note that Ulix terminates

[1] see `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html`

processes when this key combination is pressed.

SIGKILL (default action: abort)

This is the aggressive KILL signal that cannot be intercepted by a signal handler. It terminates the process at once.

SIGPIPE (default action: abort)

The process wrote to a pipe that has no reader.

SIGQUIT (default action: abort)

This is similar to **SIGINT**. Some systems write a core dump, in addition to what is caused by **SIGINT**.

SIGSEGV (default action: abort)

The process tried to access a memory location for which it lacks the required privileges, see **SIGBUS**.

SIGSTOP (default action: stop)

The signal stops a process. It will remain blocked until it receives **SIGCONT**. The signal cannot be intercepted by a handler.

SIGTERM (default action: abort)

The process is asked to terminate. It can install a signal handler for this signal which allows it to write in-memory data to files or perform other final actions. The signal can also be ignored.

SIGTSTP (default action: stop)

This is a variant of **SIGSTOP** and allows the installation of a signal handler.

SIGTTIN (default action: stop)

The process has no terminal but tried to read from a non-redirected standard input.

SIGTTOU (default action: stop)

The process has no terminal but tried to write to a non-redirected standard output or standard error output.

SIGUSR1, SIGUSR2 (default: abort)

These signals can be used by application developers for their own purposes.

SIGPOLL (default action: ignore)

When new data appear on a process' standard input, this signal is raised. Network sockets can also generate this signal.

SIGPROF (default action: abort)

A special timer that counts CPU time which was spent in this process (or in the kernel, but for this process) generates this signal. It is similar to **SIGALRM** but that one uses real time. Also compare with **SIGVTALRM**.

SIGSYS (default action: abort)

The process tried to execute a system call with an unknown system call number.

SIGTRAP (default action: abort)

This signal is raised when a process is run in a debugger and a breakpoint was reached.

SIGURG (default action: ignore)

For systems that support networking, this signal indicates that data

have arrived on a socket which require urgent treatment.

SIGVTALRM (default action: abort)

A special timer that counts CPU time which was spent in this process (but not the kernel) generates this signal. It is similar to **SIGALRM** but that one uses real time. Also compare with **SIGPROF**.

SIGXCPU (default action: abort)

This can be used if a process is only granted a certain amount of CPU time before it is terminated. The signal is sent a bit earlier so that the process can finish its work.

SIGXFSZ (default action: abort)

The process has tried to create a file that is larger than allowed.

Table 14.1 shows how these signal names are mapped to signal numbers on some standard systems.

The information in the table was gathered from the following sources:

- `kill -l` on Linux (kernel 3.0.0) and OS X (Darwin kernel 10.8.0),
- Minix `signal.h` header file; "n/a" (not available) means: these system calls have not been implemented in Minix, but the numbers were assigned because the POSIX standard requires Unix implementations to define them; `http://faculty.qu.edu.qa/rriley/cmpt507/minix/signal_8h-source.html`.
- FreeBSD: `http://www.unix.com/man-page/FreeBSD/3/signal/`
- OpenSolaris 2009.06, `http://www.unix.com/man-page/opensolaris/3head/signal.h/`

Over these five operating systems, only the signals SIGHUP (1), SIGINT (2), SIGQUIT (3), SIGILL (4), SIGTRAP (5), SIGABRT (6), SIGFPE (8), SIGKILL (9), SIGSEGV (11), SIGPIPE (13), SIGARLM (14) and SIGTERM (15) have common numbers.

You can find further details about signals in section 2.4 (*Signal Concepts*) of the *System Interfaces* volume of the Single UNIX® Specification, Version 4, 2013 Edition [IT13].

14.3 Implementation of Signals in Ulix

In order to implement signals the following two sets of functionalities are normally required:

- Methods which let a process register *signal handlers* (via a `signal` system call) and decide which signals to block (by setting the *signal mask* via a `sigprocmask` system call). (Ulix does not support changing the signal mask.) signal mask
- Methods to deliver signals to processes and have the process react accordingly: for delivering, we need to implement the `kill` system call, and making the process execute (and return from) the signal handler requires changes to the scheduling code.

We will allow each process to define signal handlers for 32 signals (0–31), so we need space for 32 addresses, and we need to store 2×32 bits in each process for pending signals and blocked signals: A signal handler is stored as its address:

Signal	Linux	OS X	Minix	FreeBSD	Solaris
SIGABRT	6	6	6	6	6
SIGALRM	14	14	14	14	14
SIGBUS	7	10	7	10	10
SIGCHLD	17	20	17	20	18
SIGCONT	18	19	n/a, 18	19	25
SIGFPE	8	8	8	8	8
SIGHUP	1	1	1	1	1
SIGILL	4	4	4	4	4
SIGINT	2	2	2	2	2
SIGKILL	9	9	9	9	9
SIGPIPE	13	13	13	13	13
SIGQUIT	3	3	3	3	3
SIGSEGV	11	11	11	11	11
SIGSTOP	19	17	n/a, 19	17	23
SIGTERM	15	15	15	15	15
SIGTSTP	20	18	n/a, 20	18	24
SIGTTIN	21	21	n/a, 22	21	26
SIGTTOU	22	22	n/a, 23	22	27
SIGUSR1	10	30	10	30	16
SIGUSR2	12	31	12	31	17
SIGPOLL *)	29	23	–	23	22
SIGPROF	27	27	25	27	29
SIGSYS	31	12	–	12	12
SIGTRAP	5	5	5	5	5
SIGURG	23	16	–	16	21
SIGVTALRM	26	26	24	26	28
SIGXCPU	24	24	–	24	30
SIGXFSZ	25	25	–	25	31

Table 14.1: Standard signals on five Unix systems.

*) Linux, Mac OS and FreeBSD call the SIGPOLL signal SIGIO.

[560a] ⟨*public elementary type definitions* 45e⟩+≡ (44a 48a) ◁178a

```
typedef void (*sighandler_t)(int);
```

Defines:
`sighandler_t`, used in chunks 560b, 561a, 566, and 568b.

and 32 bits fit precisely in an `unsigned long` integer, so we can add two variables `sig_pending` and `sig_blocked` for storing those bits:

[560b] ⟨*more TCB entries* 158c⟩+≡ (175) ◁432c 573a▷

```
sighandler_t sighandlers[32];
unsigned long sig_pending;
unsigned long sig_blocked;
```

Uses `sighandler_t` 560a.

For every signal number `i` and each state of a process there are several possibilities:

- The signal is *blocked*; in that case the `i`-th bit in `sig_blocked` is 1, and the value in `sighandlers[i]` is irrelevant, because the signal will cause the `i`-th bit of `sig_pending` to be set.
- The signal is not blocked and the default action shall take place; in that case the corresponding bit in `sig_blocked` is 0 and `sighandlers[i]` is `SIG_DFL`$_{561a}$.
- The signal is not blocked, but the process ignores this signal; in that case the corresponding bit in `sig_blocked` is 0 and `sighandlers[i]` is `SIG_IGN`$_{561a}$.
- The signal is not blocked and a signal handler `handler()` has been installed; in that case the corresponding bit in `sig_blocked` is 0 and `sighandlers[i]` is `handler`.

We use the `SIG_DFL`$_{561a}$ and `SIG_IGN`$_{561a}$ values from a Linux system (on a 32 bit Linux system their definitions can be found in `/usr/include/asm-generic/signal-defs.h`):

⟨*public constants* 46a⟩+≡ (44a 48a) ◁469b 562a▷ [561a]

```
#define SIG_DFL ((sighandler_t)0)     // default signal handling
#define SIG_IGN ((sighandler_t)1)     // ignore signal
#define SIG_ERR ((sighandler_t)-1)    // error code
```

Defines:
`SIG_DFL`, used in chunks 562b and 567b.
`SIG_ERR`, used in chunk 566b.
`SIG_IGN`, used in chunks 562b and 567b.
Uses `sighandler_t` 560a and `signal` 568b.

This assumes that 0 and 1 can never be the addresses of a signal handler function.

We will not implement queues for signals; if the same process receives the same signal more than once before the scheduler activates it the next time, then the extra signal(s) will be lost (which is also what other Unix implementations do). Thus, our internal `u_kill`$_{562b}$ function is rather simple:

⟨*function prototypes* 45a⟩+≡ (44a) ◁553a 566a▷ [561b]

```
int u_kill (int pid, int signo);
```

will set `errno`$_{207b}$ (more precisely: the TCB element `error`) to one of the following error constants if something goes wrong:

⟨*error constants* 370a⟩+≡ (207a) ◁371b 577a▷ [561c]

```
#define EPERM   1  // not permitted
#define ESRCH   3  // no such process
#define EINVAL 22  // invalid argument
```

Defines:
`EINVAL`, used in chunks 562b and 565c.
`EPERM`, used in chunks 562b and 565c.
`ESRCH`, used in chunks 562b and 565c.

Now we present the implementation. We have taken the signal numbers from an OS X system (and renamed `SIGIO` to `SIGPOLL`$_{562a}$):

[562a] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁561a 584a▷

```
#define SIGHUP     1
#define SIGINT     2
#define SIGQUIT    3
#define SIGILL     4
#define SIGTRAP    5
#define SIGABRT    6
#define SIGFPE     8
#define SIGKILL    9
#define SIGBUS    10
#define SIGSEGV   11
#define SIGSYS    12
#define SIGPIPE   13
#define SIGALRM   14
#define SIGTERM   15
#define SIGURG    16
#define SIGSTOP   17
#define SIGTSTP   18
#define SIGCONT   19
#define SIGCHLD   20
#define SIGTTIN   21
#define SIGTTOU   22
#define SIGPOLL   23
#define SIGXCPU   24
#define SIGXFSZ   25
#define SIGVTALRM 26
#define SIGPROF   27
#define SIGUSR1   30
#define SIGUSR2   31
```

Defines:
SIGABRT, used in chunk 562b.
SIGALRM, used in chunk 562b.
SIGBUS, used in chunk 562b.
SIGCONT, used in chunks 563b and 566b.
SIGFPE, used in chunk 562b.
SIGHUP, used in chunk 562b.
SIGILL, used in chunk 562b.
SIGINT, used in chunk 562b.
SIGKILL, used in chunks 321a, 562b, 564a, and 566b.
SIGPIPE, used in chunk 562b.
SIGPROF, used in chunk 562b.
SIGSTOP, used in chunks 562b, 563a, and 566b.
SIGSYS, used in chunk 562b.
SIGTERM, used in chunk 562b.
SIGTRAP, used in chunk 562b.
SIGTSTP, used in chunk 562b.
SIGTTIN, used in chunk 562b.
SIGTTOU, used in chunk 562b.
SIGUSR1, used in chunk 562b.
SIGUSR2, used in chunk 562b.
SIGVTALRM, used in chunk 562b.
SIGXCPU, used in chunk 562b.
SIGXFSZ, used in chunk 562b.

The `u_kill`$_{562b}$ function first tests a few error conditions and leaves immediately if it finds one of them: Signal numbers must be between 0 and 31, the target process must exist and must neither have PID 1 (`init`/`idle`) or 2 (`swapper`) nor `login` as command name.

Then it checks whether `SIG_DFL`$_{561a}$ is set as signal handler; depending on the signal it changes the signal number to `SIGKILL`$_{562a}$ (for killing the process) or `SIGSTOP`$_{562a}$ (for stopping it). After that it treats the three special cases of the `SIGKILL`$_{562a}$, `SIGSTOP`$_{562a}$ and `SIGCONT`$_{562a}$ signals which cannot be blocked or ignored.

Finally it checks whether the signal shall be ignored; if the signal is neither ignored nor blocked, it sets the pending bit in the target TCB's `sig_pending` field and returns.

[562b] ⟨*function implementations* 100b⟩+≡ (44a) ◁553b 566b▷

```
int u_kill (int pid, int signo) {
  TCB *tcb = &thread_table[pid];
  if (signo < 0 || signo > 31) { set_errno (EINVAL); return -1; }
  if (!tcb->used)              { set_errno (ESRCH);  return -1; }
  if ( (pid < 3) || (strncmp (tcb->cmdline, "login", 5) == 0) )
                               { set_errno (EPERM);  return -1; }

  if (tcb->sighandlers[signo] == SIG_DFL) {          // default action
    if (signo == SIGABRT || signo == SIGALRM  || signo == SIGBUS  ||
        signo == SIGFPE  || signo == SIGHUP   || signo == SIGILL  ||
```

```
        signo == SIGINT  || signo == SIGPIPE  || signo == SIGTERM ||
        signo == SIGUSR1 || signo == SIGUSR2  || signo == SIGPROF ||
        signo == SIGSYS  || signo == SIGTRAP  || signo == SIGVTALRM ||
        signo == SIGXCPU || signo == SIGXFSZ) {
      // default: abort, send SIGKILL
      printf ("Replacing signal %d with kill signal (9)\n", signo);
      signo = SIGKILL;    // no handler? kill it
    } else if (signo == SIGTTIN || signo == SIGTTOU || signo == SIGTSTP) {
      // default: stop, send SIGSTOP
      printf ("Replacing signal %d with kill signal (9)\n", signo);
      signo = SIGSTOP;    // no handler? kill it
    }
  }

  switch (signo) { ⟨u_kill: special cases 563a⟩ };     // cannot ignore/block

  if (tcb->sighandlers[signo] == SIG_IGN) return 0;   // ignore signal
  int blocked = tcb->sig_blocked & (1<<signo);
  if (!blocked && signo ≥ 0 && signo < 32) {
    tcb->sig_pending |= (1<<signo);                   // set the pending bit
  }
  return 0;
}
```

Defines:
u_kill, used in chunks 321a, 561b, and 565c.
Uses EINVAL 561c, EPERM 561c, ESRCH 561c, kill 568b, login 584c, printf 601a, set_errno 206b, SIG_DFL 561a, SIG_IGN 561a, SIGABRT 562a, SIGALRM 562a, SIGBUS 562a, SIGFPE 562a, SIGHUP 562a, SIGILL 562a, SIGINT 562a, SIGKILL 562a, signal 568b, SIGPIPE 562a, SIGPROF 562a, SIGSTOP 562a, SIGSYS 562a, SIGTERM 562a, SIGTRAP 562a, SIGTSTP 562a, SIGTTIN 562a, SIGTTOU 562a, SIGUSR1 562a, SIGUSR2 562a, SIGVTALRM 562a, SIGXCPU 562a, SIGXFSZ 562a, strncmp 594a, TCB 175, and thread_table 176b.

We treat three special cases directly in the $\texttt{u\_kill}_{562b}$ function:

- The $\texttt{SIGSTOP}_{562a}$ signal:

⟨u_kill: *special cases* 563a⟩≡ (562b) 563b▷ [563a]

```
case SIGSTOP:   ⟨u_kill: remove thread from queue 564c⟩
                tcb->state = TSTATE_STOPPED;
                if (pid == current_task) {
                  ⟨resign 221d⟩  // enter scheduler
                }
                return 0;
```

Uses current_task 192c, SIGSTOP 562a, and TSTATE_STOPPED 180a.

- The $\texttt{SIGCONT}_{562a}$ signal:

⟨u_kill: *special cases* 563a⟩+≡ (562b) ◁563a 564a▷ [563b]

```
case SIGCONT:   if (tcb->state == TSTATE_STOPPED) {
                  add_to_ready_queue (pid);  // sets TSTATE_READY
                }  // else ignore
                return 0;
```

Uses add_to_ready_queue 184b, SIGCONT 562a, TSTATE_READY 180a, and TSTATE_STOPPED 180a.

- The SIGKILL$_{562a}$ signal:

[564a] ⟨u_kill: *special cases* 563a⟩+≡ (562b) ◁563b

```
case SIGKILL:   ⟨u_kill: remove thread from queue 564c⟩
                tcb->used = false;
                tcb->state = TSTATE_EXIT;
                ⟨u_kill: write kill message 564b⟩
                wake_waiting_parent_process (pid);
                ⟨enable scheduler 276a⟩
                if (pid == current_task) {
                  ⟨resign 221d⟩  // enter scheduler
                }
                return 0;
```

Uses current_task 192c, SIGKILL 562a, TSTATE_EXIT 180a, and wake_waiting_parent_process 217a.

We want to notify the user that the process was killed, so we write a "Killed" message onto the terminal that the task uses. For that purpose we need to temporarily change the value of thread_table$_{176b}$[current_task$_{192c}$].terminal (since printf$_{601a}$() uses that value to find its target) and restore it after writing:

[564b] ⟨u_kill: *write kill message* 564b⟩≡ (564a)

```
int tmp_term = thread_table[current_task].terminal;
thread_table[current_task].terminal = thread_table[pid].terminal;
printf ("\nKilled\n");
thread_table[current_task].terminal = tmp_term;
```

Uses current_task 192c, printf 601a, and thread_table 176b.

In order to remove the thread from a queue we can only guess what queue it might be on since we do not have a list of all available queues; for example, every lock has its separate queue, and locks can be generated on the fly. We check the standard blocked queues (waiting for a child process, a keyboard, floppy or hard disk event) and the ready queue:

[564c] ⟨u_kill: *remove thread from queue* 564c⟩≡ (563a 564a)

```
switch (tcb->state) {
  case TSTATE_READY:   remove_from_ready_queue (pid);                      break;
  case TSTATE_WAITFOR: remove_from_blocked_queue (pid, &waitpid_queue);    break;
  case TSTATE_WAITKEY: remove_from_blocked_queue (pid, &keyboard_queue);   break;
  case TSTATE_WAITFLP: remove_from_blocked_queue (pid, &floppy_queue);     break;
  case TSTATE_WAITHD:  remove_from_blocked_queue (pid, &harddisk_queue);   break;
  case TSTATE_WAITSD:  remove_from_blocked_queue (pid, &serial_disk_queue); break;
  default:             printf ("cannot remove process %d (state: %d) from blocked"
                               " queue, probably failure!\n", pid, tcb->state);
}
```

Uses floppy_queue 544d, harddisk_queue 529a, keyboard_queue 323d, printf 601a, remove_from_blocked_queue 186a, remove_from_ready_queue 184c, serial_disk_queue 522a, TSTATE_READY 180a, TSTATE_WAITFLP 180a, TSTATE_WAITFOR 180a, TSTATE_WAITHD 180a, TSTATE_WAITKEY 180a, TSTATE_WAITSD 521c, and waitpid_queue 218b.

Note that we need not (and do not) perform any checks in this function: `u_kill`$_{562b}$ can be called by the kernel itself (which may send any signal to any process), but it cannot be called directly by a process. Sending by a process requires using a system call, and the system call handler will check whether the process is allowed to send the signal to the target process before calling `u_kill`$_{562b}$.

It is also classical for a process to send a signal to itself; that is what the `raise`$_{568b}$ function does. We will not implement it specifically inside the kernel, but in the user mode library: `raise`$_{568b}$`(sig)` is the same as `kill`$_{568b}$`(getpid`$_{223b}$`(),sig)`.

Here's the code for the system call handler:

⟨ *initialize syscalls* 173d ⟩+≡ (44b) ◁513b 567a▷ [565a]
```
install_syscall_handler (__NR_kill, syscall_kill);
```
Uses `__NR_kill` 204c, `install_syscall_handler` 201b, and `syscall_kill` 565c.

⟨ *syscall prototypes* 173b ⟩+≡ (202a) ◁512d 566c▷ [565b]
```
void syscall_kill (context_t *r);
```

⟨ *syscall functions* 174b ⟩+≡ (202b) ◁513a 566d▷ [565c]
```
void syscall_kill (context_t *r) {
  // ebx: pid of child to send a s signal,  ecx: signal number
  int retval; int target_pid = r->ebx; int signo = r->ecx;

  if (!thread_table[target_pid].used) {      // check if target process exists
    // target process does not exist
    set_errno (ESRCH);
    retval = -1; goto end;
  }

  if (signo < 0 || signo > 31) {             // check if signal is in range 0..31
    set_errno (EINVAL);
    retval = -1; goto end;
  }

  // check if current process may send a signal
  if ((thread_table[current_task].euid == 0) ||
      (thread_table[target_pid].euid == thread_table[current_task].euid)) {
    retval = u_kill (target_pid, signo);
  } else {
    set_errno (EPERM);
    retval = -1;
  }
  end: r->eax = retval;

  // run scheduler if this was a raise operation
  if (current_task == target_pid) {  ⟨ resign 221d ⟩  }
};
```
Defines:
`syscall_kill`, used in chunk 565.
Uses `context_t` 142a, `current_task` 192c, `EINVAL` 561c, `EPERM` 561c, `ESRCH` 561c, `euid` 573a, `raise` 568b, `set_errno` 206b, `signal` 568b, `target_pid`, `thread_table` 176b, and `u_kill` 562b.

We only allow sending a signal if either the sender's owner has user ID 0 or if sender and recipient have the same owner.

If sender and receiver are the same process, we have a `raise`$_{568b}$ operation, and in that case we will jump into the scheduler: we do not want the current process to continue execution since it might have sent a `SIGKILL`$_{562a}$ signal to itself.

(Note that we cannot use the `eax_return`$_{174a}$ macro in this function because we may or may not call ⟨*resign* 221d⟩.)

How can a process declare a signal handler? It just defines a function `void *handler (int)` (of type `sighandler_t`$_{560a}$) and makes a `signal`$_{568b}$ syscall. The internal function for entering a system call is the following:

[566a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁561b 576a▷
```
sighandler_t u_signal (int sig, sighandler_t func);
```

[566b] ⟨*function implementations* 100b⟩+≡ (44a) ◁562b 576b▷
```
sighandler_t u_signal (int sig, sighandler_t func) {
  sighandler_t old_func;
  if (sig ≥ 0 && sig < 32 &&
      sig != SIGKILL && sig != SIGSTOP && sig != SIGCONT) {
    old_func = thread_table[current_task].sighandlers[sig];
    thread_table[current_task].sighandlers[sig] = func;
  } else {
    old_func = SIG_ERR;   // wrong signal number
  }
  return old_func;
}
```
Defines:
`u_signal`, used in chunk 566.
Uses `current_task` 192c, `SIG_ERR` 561a, `SIGCONT` 562a, `sighandler_t` 560a, `SIGKILL` 562a, `signal` 568b, `SIGSTOP` 562a, and `thread_table` 176b.

The function sets the new signal handler and returns the address of the old handler (or 0 if there was none); that way the process can keep a copy of the old handler address in order to restore it at a later point.

As usual, we need to provide a system call so that a process can access this function.

[566c] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁565b 582b▷
```
void syscall_signal (context_t *r);
```

It performs the already well-known transfers of register values to arguments and of the return value to the *EAX* register:

[566d] ⟨*syscall functions* 174b⟩+≡ (202b) ◁565c 583a▷
```
void syscall_signal (context_t *r) {
  // ebx: signal number
  // ecx: address of signal handler
  int signo         = r->ebx;
  sighandler_t func = (sighandler_t)r->ecx;
  func = u_signal (signo, func);
  eax_return (func);
};
```

Defines:
  syscall_signal, used in chunks 566c and 567a.
Uses context_t 142a, eax_return 174a, sighandler_t 560a, and u_signal 566b.

⟨ *initialize syscalls* 173d⟩+≡ (44b) ◁565a 583b▷ [567a]
```
install_syscall_handler (__NR_signal, syscall_signal);
```
Uses __NR_signal 204c, install_syscall_handler 201b, and syscall_signal 566d.

Finally, we need to add code to the scheduler: it needs to check for pending signals and—if there are any—launch the registered handler function or execute the standard action. Running a handler can be achieved by modifying the process' stack. We loop over the set of possible signal numbers (0–31) and check the bits in sig_pending. (Remember that $\texttt{t\_new}_{276c}$ points to the newly chosen process' TCB.)

Note that we only modify the stack (and the *EIP* value) if the process is currently running in user mode (i. e., $\texttt{t\_new}_{276c}$->regs.eip < 0xc0000000), because otherwise we would change the kernel stack (making the signal handler run with kernel privileges).

⟨ *scheduler: check pending signals* 567b⟩≡ (277b) [567b]
```
for (int signo = 0;  signo < 32;  signo++) {
  if ((t_new->sig_pending & (1<<signo)) != 0    // signal is pending
      && t_new->regs.eip < 0xc0000000) {        // and thread is in user mode
    if (t_new->sighandlers[signo] == SIG_DFL) {
      ;    // nop                               // default action, cannot happen
    } else if (t_new->sighandlers[signo] == SIG_IGN) {
      ;    // nop                               // ignored, should not happen
    } else {
      // handler exists
      ⟨ modify process to execute signal handler 567c⟩
    }
    t_new->sig_pending &= ~(1<<signo);          // remove bit
    break;                                      // only one handler at a time
  }
}
```
Uses SIG_DFL 561a, SIG_IGN 561a, and t_new 276c.

When a handler exists, we modify both the (user mode) stack and the *EIP* register.

⟨ *modify process to execute signal handler* 567c⟩≡ (567b) [567c]
```
// Note: t_new->regs has already been copied to r
memaddress oldeip = r->eip;
r->eip = (memaddress)t_new->sighandlers[signo];
// push signal number and oldeip on user mode stack
POKE_UINT (r->useresp, signo);   // overwrites old RET address
r->useresp -= 4;
POKE_UINT (r->useresp, oldeip);  // writes new RET address
```
Uses memaddress 46c, POKE_UINT 117, and t_new 276c.

14.3.1 Library Functions

Now we can provide the user mode library functions for the two new system calls; we also define a `raise`$_{568b}$ function which sends a signal to the own process:

[568a] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁513c 584b▷
```
int kill (int pid, int signo);
int raise (int signo);
sighandler_t signal (int sig, sighandler_t func);
```

[568b] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁513d 584c▷
```
int kill (int pid, int signo) {
  return syscall3 (__NR_kill, pid, signo);
}

int raise (signo) {
  return kill (getpid (), signo);
}

sighandler_t signal (int sig, sighandler_t func) {
  return (sighandler_t)syscall3 (__NR_signal, sig, (memaddress)func);
}
```
Defines:
`kill`, used in chunks 321a, 431, and 562b.
`raise`, used in chunk 565c.
`signal`, used in chunks 561a, 562b, 565c, 566b, and 568a.
Uses `__NR_kill` 204c, `__NR_signal` 204c, `getpid` 223b, `memaddress` 46c, `sighandler_t` 560a, and `syscall3` 203c.

14.3.2 Example Program

We end this chapter with an example program that you can also find on the Ulix disk image: `/bin/sigtest` forks, registers two signal handlers in the child and sends two signals twice from the parent process. Both processes otherwise print sequences of “p”s or “c”s to show that they are active. The expected behavior is that the output is interrupted with four messages from the two signal handlers.

[568c] ⟨*example for signal handlers* 568c⟩≡
```
#include "../ulixlib.h"

void handler1 (int sig);
void handler2 (int sig);
void waste_time ();

int main (int argc, char *argv[]) {
  int i; int pid = fork ();
  if (pid == 0) {                                    // child
    signal (5, handler1);   // register handler 1
    signal (6, handler2);   // register handler 2
    for (i = 0; i < 40; i++) { printf ("c"); waste_time (); }
    exit (0);
```

```
  } else {                                          // parent
    for (i = 0; i < 20; i++) { printf ("p"); waste_time (); }
    kill (pid, 5);
    kill (pid, 6);
    for (i = 0; i < 22; i++) { printf ("p"); waste_time (); }
    printf ("--done\n");
    exit (0);
  }
}

void handler1 (int sig) { printf ("\nHandler 1\n"); }
void handler2 (int sig) { printf ("\nHandler 2\n"); }

void waste_time () {
  long int i, z;
  for (i=0L; i<1000000L; i++)  z = i*i - (i+1)*(i+1);
}
```

15
Users and Groups

Unix and all Unix-derived operating systems are multi-user-capable. There are configuration files which contain the information about all known *users*, and there is also a list of groups that users can be members of. Each user has a *standard group* but may—additionally—have the membership of one or more further groups. The `id` command lists the user ID, a corresponding user name, the standard group ID and name and a list of all additional group memberships. The following is an example from a Linux installation:

users

standard group

```
$ id
uid=1000(esser) gid=1000(esser) groups=1000(esser),24(cdrom),25(floppy),29(audio),
30(dip),44(video),46(plugdev)
```

Internally, the systems use only the numerical IDs; tools which display user and group names will look them up using functions such as `getpwnam` or `getgrnam`. The inode of each file stores a user ID and a group ID, the first one expresses file ownership by the specific user who uses this user ID, whereas the second one establishes an additional group ownership. Members of a group may (or may not) have access rights to a file which has the corresponding group ID. The group is sometimes called the *owner group* or *group owner*, both variants mean the same thing.

owner group

Classical Unix systems can associate nine *access permissions* with every file and every directory: for files, these are the three basic rights to read (`r`), write (`w`) or execute (`x`) a file, for directories the interpretation is listing, modifying and searching/entering a directory (where "searching" means getting the inode number of a file in the directory and "entering" means setting the *current working directory* to a folder). Each of these three permissions can be granted or denied to the file owner, all members of one specific group of users and all other users. This leads to nine permissions typically represented as a nine-character string of the form `rwxrwxrwx` where missing rights are expressed by exchanging a letter

access permissions

with a minus character. For example, the permission string `rwxr-x---` for a file lets the file owner read, write and execute that file (`rwx`); it lets members of the file's owner group read and execute (but not write) it (`r-x`), and other users (those who neither are the owner nor belong to the owner group) cannot access the file at all (`---`).

There are a few more attributes which can be set for files and directories which have specific effects:

effective user ID

SUID: If the *Set User ID bit* (*SUID*) is set for an executable file, the *effective user ID* (EUID) is set to the ID of the file owner. For example, the `passwd`$_{584d}$ tool uses this feature: It may be called by any user (who wants to change his own password), but it needs administrator privileges to modify the password file which is non-writeable for ordinary users. In order to allow this, the `passwd`$_{584d}$ program is set to belong to the `root` user and has the SUID bit set. When a regular user starts `passwd`$_{584d}$, the effective user ID of the process is set to 0 (the user ID of `root`), and write access to the password file is granted.

In the directory listing of an executable file with a set SUID bit, the first `x` in the permissions string is replaced with an `s` to show this. It is also possible to set the SUID bit on a non-executable file which will show up with a capital `S`, however this is useless.

effective group ID

SGID: The *Set Group ID bit* (*SGID*) has a function that is similar to the SUID bit's, however it changes the *effective group ID* (EGID). It appears as a capital `S` in the group permissions block.

mandatory locking

On some Unix systems, a non-group-executable file which has the SGID bit set is marked for *mandatory locking* (for the Linux OS, see `https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt`).

Sticky Bit: The *sticky bit* appears as a `t` letter in the last position of the permissions string (replacing the `x` which shows the world-executable state). Similar to the difference between `s` and `S`, if a file is set to be sticky but not world-executable, it appears as a capital `T`.

The effect of a set sticky bit depends on the Unix variant. For example, the Linux man page for `chmod`$_{591b}$ says: "For directories, it prevents unprivileged users from removing or renaming a file in the directory unless they own the file or the directory; this is called the restricted deletion flag for the directory, and is commonly found on world-writable directories like `/tmp`."

On traditional Unix systems, a sticky bit on an executable file caused the system to keep the program code in memory after termination of a process, with the idea that it did not have to be reloaded when the same program was executed again. That is where the term "sticky" comes from.

access control list (ACL)

Many modern Unix-like systems provide further access mechanisms through *extended attributes* or *access control lists* (ACLs). This is a feature which was not available in classical Unix, and we will not implement it for Ulix, either.

15.1 Users and Groups in Ulix

In order to implement the user and group concepts in Ulix, each thread control block needs four new entries: a user ID ($\texttt{uid}_{573a}$), a group ID ($\texttt{gid}_{573a}$) and effective user and group IDs ($\texttt{euid}_{573a}$, $\texttt{egid}_{573a}$) plus a third set called real user ID and real group ID ($\texttt{ruid}_{573a}$, $\texttt{rgid}_{573a}$:

⟨ *more TCB entries* 158c ⟩+≡ (175) ◁560b [573a]

```
word uid;   // user ID
word gid;   // group ID
word euid;  // effective user ID
word egid;  // effective group ID
word ruid;  // real user ID
word rgid;  // real group ID
```

Defines:
egid, used in chunks 487a, 573b, 576–78, 580–83, and 587d.
euid, used in chunks 487a, 565c, 573b, 576–78, 580–83, 587d, and 588b.
gid, used in chunks 478b, 573b, 580–84, and 587d.
rgid, used in chunks 573b and 582a.
ruid, used in chunks 573b and 582a.
uid, used in chunks 478b, 573b, 580–85, and 587d.

The purpose of the *real user and group IDs* is to always remember which user started the process: it will (normally) not change over a process' lifetime whereas $\texttt{uid}_{573a}$, $\texttt{gid}_{573a}$ and the effective IDs can be changed by a running process using $\texttt{setuid}_{584c}$, $\texttt{setgid}_{584c}$, $\texttt{seteuid}_{584c}$ and $\texttt{setegid}_{584c}$ functions; we will explain soon why we need to keep track of so many IDs.

0 is a special ID, both for users and groups. It belongs to the root user and root group, respectively. The root user is the *system administrator* who can override all permission settings (e. g. open files for which no read permissions have been set at all). All other IDs have no special meaning, though it is standard on many systems to reserve IDs below 100 (or below 1000) for *system services* with regular user IDs starting at 100 (or 1000). system services

user/group ID 0

The default setting of Ulix processes is to run with root privileges. The function $\texttt{start\_program\_from\_disk}_{189}$ sets all four IDs to 0:

⟨ *start program from disk: set uid, gid, euid, egid* 573b ⟩≡ (189) [573b]

```
thread_table[tid].uid  = 0;  thread_table[tid].gid  = 0;
thread_table[tid].euid = 0;  thread_table[tid].egid = 0;
thread_table[tid].ruid = 0;  thread_table[tid].rgid = 0;
```

Uses egid 573a, euid 573a, gid 573a, rgid 573a, ruid 573a, thread_table 176b, and uid 573a.

The Linux man page for setreuid states:

> "POSIX.1 does not specify all of [the] possible ID changes that are permitted on Linux for an unprivileged process. For setreuid(), the effective user ID can be made the same as the real user ID or the save set-user-ID, and it is unspecified whether unprivileged processes may set the real user ID to the real user ID, the effective user ID or the saved set-user-ID. For setregid(), the real group ID can be changed to the value of the saved set-group-ID, and

> the effective group ID can be changed to the value of the real group ID or the saved set-group-ID. The precise details of what ID changes are permitted vary across implementations."

For Ulix we will take a simplified approach with the following rules:

- Processes can invoke the `setuid` and `setgid` system calls to change their user and group IDs ($\texttt{uid}_{573a}$, $\texttt{gid}_{573a}$), and this will also set the effective user/group ID to the same value. These functions shall only succeed if the process had an $\texttt{uid}_{573a}$ value of 0 or if the desired new ID is identical to either the current user ID or the current effective user ID.
- Processes can invoke the `seteuid` and `setegid` system calls to change their effective user and group IDs ($\texttt{euid}_{573a}$, $\texttt{egid}_{573a}$). This will *not* change the user/group ID ($\texttt{uid}_{573a}$, $\texttt{gid}_{573a}$). These functions shall only succeed if the process had an $\texttt{uid}_{573a}$ value of 0 or if the desired new effective ID is identical to either the current user ID or the current effective user ID.
- The `login` system call which expects a user ID and a password allows changing the real and effective user and group IDs as long as the correct password was supplied. `login` is also the only system call which sets the real user and group IDs, thus allowing the system to keep track of which user initially started a process.
- Whenever file access is checked, the system looks at the effective IDs to establish the permissions of the current process.

The consequence of these rules is that changing, for example, the user ID and effective user ID via $\texttt{setuid}_{584c}$ is a permanent change which cannot be undone, whereas a modification of only the effective user ID via $\texttt{seteuid}_{584c}$ can be followed up with another $\texttt{seteuid}_{584c}$ call that sets it back to the original value. Once $\texttt{uid}_{573a}$ and $\texttt{euid}_{573a}$ have the same non-zero value, there is only one way to go back to a different ID or effective ID: it has to make a `login` system call which reperforms authorization against the password database.

Whenever a process forks, the new process inherits all IDs from its parent.

15.1.1 Checking Permissions

Before we delve into the implementation of $\texttt{login}_{584c}$ and the `set*id` functions, let's look at how the $\texttt{open}_{429b}$ and $\texttt{execv}_{235e}$ functions use the effective IDs to test whether access can be granted or not.

Checking whether a user may access a file seems to be a simple task: The OS just needs to look up the file's inode and check the file owner, group and permissions stored in the corresponding fields. But this is only half of the work we need to do because there is also the issue of getting into the directory which holds the file—after all, if the directory does not provide sufficient read permissions, it is forbidden to find the inode number that belongs to a filename entry.

The access rules are also different for opening (existing) files and newly creating files.

- In order to *access* an existing file (either for reading or writing), the user must have read and execute permissions on all directories which are passed on the way to file, starting with the root directory `/`.
- In order to *create* a new file, the same permissions are needed, and the user must also have write permission for the last directory (in which the file will be created).

ULIX will first check whether the file already exists. It will then follow the path from either the existing file or from the target directory, all the way up to the root directory and test for each directory whether the needed permissions are available. We can do this in a loop which repeatedly uses `splitpath`$_{455a}$ to strip the last element of the current path.

Pseudo code for this loop looks like this:

⟨*pseudo code for checking permissions* 575⟩≡ [575]

```
curpath = abspath;
step = 0;
for (;;) {
  // check current path
  if (step == 0 && fileexists (abspath)) {  // can we access the (existing) file?
    ok = check (curpath, oflag);            // check requested mode
  } else
  if (step == 1 && !fileexists (abspath)) { // can we create the file, i.\,e. write
                                            // to the target directory?
    ok = check (curpath, "rwx");
  } else {
    ok = check (curpath, "rx");             // check some intermediate directory
  }
  if (!ok) return false;                    // access denied
  if (curpath == "/") return true;          // end of loop (/), access granted

  // move to upper directory
  lastpath = curpath;
  splitpath (lastpath, curpath, tmp);
  step++;
}
```

Note that it would be more efficient to directly implement the access checks in the loop inside the `mx_open`$_{464b}$ function of the Minix subsystem which traverses the path down from the root to the file, but we did not want to discuss access rights when we presented the code for opening a file. Also, our method is independent of the filesystem (e. g. Minix). But it does more than duplicate the efforts of walking down the path, so it would be unacceptable for production systems.

In each step both user, group and world access permissions need to be considered: for example, if user permissions allow access to the first directory, group permissions allow access to the second directory and world permissions allow access to the third directory, then that is an acceptable sequence. Only if none of the directory permissions allow access (somewhere in the path), access must be denied.

We start with the implementation of

[576a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁566a 576c▷
```
boolean fileexists (char *abspath);
```

for which we can use the u_stat$_{421d}$ function:

[576b] ⟨*function implementations* 100b⟩+≡ (44a) ◁566b 576d▷
```
boolean fileexists (char *abspath) {
  struct stat tmp;                        // stat info will not be used
  return (u_stat (abspath, &tmp) != -1);  // -1 means: does not exist
}
```
Defines:
fileexists, used in chunk 576a.
Uses stat 429b 489b and u_stat 421d.

For checking the permissions on any level, we write a function

[576c] ⟨*function prototypes* 45a⟩+≡ (44a) ◁576a 579a▷
```
boolean check_access (char *path, word euid, word egid, word mode);
```

which evaluates the owner, group and world access permissions, depending on the provided user and group IDs (euid$_{573a}$ and egid$_{573a}$):

[576d] ⟨*function implementations* 100b⟩+≡ (44a) ◁576b 579c▷
```
boolean check_access (char *path, word euid, word egid, word mode) {
  struct stat st;
  int res = u_stat (path, &st);   // get file permissions
  if (res == -1 && (mode & O_CREAT) == 0) {
    set_errno (ENOENT); // file not found
    return false;
  }

  if (res == -1 && (mode & O_CREAT) != 0) {
    ⟨check_access special case: create file 577b⟩
  }

  if (euid == st.st_uid) {
    // case 1: user owns the file
    res = check_perms (CHECK_USER,  mode, st.st_mode);
  } else if (egid == st.st_gid) {
    // case 2: group matches owner group
    res = check_perms (CHECK_GROUP, mode, st.st_mode);
  } else {
    // case 3: world access?
    res = check_perms (CHECK_WORLD, mode, st.st_mode);
  }
  if (!res) set_errno (EACCES);
  return res;
}
```
Defines:
check_access, used in chunk 577c.
Uses CHECK_GROUP 579b, check_perms 579c, CHECK_USER 579b, CHECK_WORLD 579b, EACCES 577a, egid 573a, ENOENT 577a, euid 573a, O_CREAT 460b, set_errno 206b, stat 429b 489b, and u_stat 421d.

(The function check_perms$_{579c}$ checks just one possible way of getting access, e. g. via the owner permissions. We will describe it soon.)

⟨*error constants* 370a⟩+≡ (207a) ◁561c [577a]

```
#define ENOENT          2               // No such file or directory
#define EACCES          13              // Permission denied
```

Defines:
EACCES, used in chunks 576 and 577.
ENOENT, used in chunks 576d and 577b.

There's also one special case we need to consider: when we create a new file, it does not yet exist, and so u_stat$_{421d}$ will return −1. For file creation we have to check the access permissions of the directory in which the new file is to be placed.

⟨check_access *special case: create file* 577b⟩≡ (576d) [577b]

```
char dirname[256], basename[256];
splitpath (path, dirname, basename);
res = u_stat (dirname, &st);   // get directory permissions
if (res == -1) {
  set_errno (ENOENT);          // directory not found
  return false;
}
if (euid == st.st_uid) {
  // case 1: user owns the directory
  res = (((st.st_mode >> CHECK_USER)  & 0x7) == 0x7);   // 7: rwx
} else if (egid == st.st_gid) {
  // case 2: group matches owner group
  res = (((st.st_mode >> CHECK_GROUP) & 0x7) == 0x7);
} else {
  // case 3: world access?
  res = (((st.st_mode >> CHECK_WORLD) & 0x7) == 0x7);
}
if (!res) set_errno (EACCES);
return res;
```

Uses basename 455b, CHECK_GROUP 579b, CHECK_USER 579b, CHECK_WORLD 579b, dirname 455b, EACCES 577a, egid 573a, ENOENT 577a, euid 573a, set_errno 206b, splitpath 455a, and u_stat 421d.

The u_open$_{412c}$ function calculates the absolute path of the file it shall open and stores it in abspath; the requested mode for opening is held in the function's oflag parameter. It can then call u_stat$_{421d}$ to read its access permissions as well as the permissions of the directories involved:

⟨u_open*: check permissions* 577c⟩≡ (412c) [577c]

```
boolean access_ok = false;
word euid = thread_table[current_task].euid;
word egid = thread_table[current_task].egid;
struct stat st;

if (euid == 0) {
  access_ok = true;   // user root can do anything
} else {
```

```
    // loop over the directories
    char old_dirname[256]; char dirname[256]; char rest[256];
    strncpy (dirname, abspath, 256);
    for (;;) {
      strncpy (old_dirname, dirname, 256);
      splitpath (old_dirname, dirname, rest);   // ignore rest
      u_stat (dirname, &st);
      access_ok = ⟨u_open: access to directory is ok 578⟩;
      if (!access_ok) {
        set_errno (EACCES);
        return -1;
      }
      if (strlen (dirname) == 1) break;         // reached root directory
    }
    // finally: check file permissions
    access_ok = check_access (abspath, euid, egid, oflag);
  }
  if (!access_ok) {
    return -1;  // wrong permissions
  }
```

Uses check_access 576d, current_task 192c, dirname 455b, EACCES 577a, egid 573a, euid 573a, set_errno 206b, splitpath 455a, stat 429b 489b, strlen 594a, strncpy 594b, thread_table 176b, and u_stat 421d.

A process may only access a directory if it can read and "execute" it, and that's possible if one of the following three conditions is fulfilled:

- the process' effective user (determined by the $\texttt{euid}_{573a}$ field) owns the file and the user read and execute bits ($\texttt{0500}_o$) are set in the access permissions,
- the process' effective user *does not* own the file, the process' effective group (determined by the $\texttt{egid}_{573a}$ field) is the file's owner group and the group read and execute bits ($\texttt{0050}_o$) are set in the access permissions,
- or neither the user and group fields of the file match the effective user or group, but the world permissions allow read and execute access ($\texttt{0005}_o$).

Thus, checking whether the process may access a directory (read and execute, $1+4=5$) can be done with the following code:

[578] ⟨u_open: *access to directory is ok* 578⟩≡ (577c)

```
  ( // user may have access,                          r-x------ ?
    (((euid == st.st_uid) &&                        (st.st_mode & 0500) == 0500)) ||
    // group may have access (if wrong user),        ---r-x--- ?
    (((euid != st.st_uid) && (egid == st.st_gid) && (st.st_mode & 0050) == 0050)) ||
    // others may have access (wrong user, group),   ------r-x ?
    (((euid != st.st_uid) && (egid != st.st_gid) && (st.st_mode & 0005) == 0005))  )
```

Uses egid 573a and euid 573a.

Now we need to provide the function

⟨*function prototypes* 45a⟩+≡ (44a) ◁576c 580b▷ [579a]
```
boolean check_perms (short what, word req_mode, word perms);
```

which accepts one of the three constants

⟨*constants* 112a⟩+≡ (44a) ◁552a 608a▷ [579b]
```
#define CHECK_USER  6
#define CHECK_GROUP 3
#define CHECK_WORLD 0
```
Defines:
CHECK_GROUP, used in chunks 576d and 577b.
CHECK_USER, used in chunks 576d and 577b.
CHECK_WORLD, used in chunks 576d and 577b.

and a requested mode `req_mode` and the file permissions `perms`. The lowest two bits of `req_mode` are either 00_b (in case of $\texttt{O\_RDONLY}_{460b}$), 01_b (in case of $\texttt{O\_WRONLY}_{460b}$) or 10_b (in case of $\texttt{O\_RDWR}_{460b}$). We can check them by looking at `req_mode & 0x3`.

File access permissions can be found in the lowest nine bits of `perms`.

- If we want to check world permissions (for "others"), we look at the lowest three bits, `perms & 0x7`.
- For the group permissions we can first right-shift `perms` so that we drop the lowest three bits, that is, we look at `(perms >> 3) & 0x7`.
- Finally, for the owner permissions, we need a right-shift of six bits, which gives us `(perms >> 6) & 0x7`

By setting $\texttt{CHECK\_USER}_{579b}$, $\texttt{CHECK\_GROUP}_{579b}$ and $\texttt{CHECK\_WORLD}_{579b}$ to the necessary amount of shifting (6, 3, 0), we can do this automatically:

⟨*function implementations* 100b⟩+≡ (44a) ◁576d 580c▷ [579c]
```
boolean check_perms (short what, word req_mode, word perms) {
  boolean req_read  = ((req_mode & 0x3) == O_RDONLY) | ((req_mode & 0x3) == O_RDWR);
  boolean req_write = ((req_mode & 0x3) == O_WRONLY) | ((req_mode & 0x3) == O_RDWR);
  word check = (perms >> what) & 0x7;
  if (req_read  && ((check & 0x4) != 0x4)) return false;  // read  perm. failure
  if (req_write && ((check & 0x2) != 0x2)) return false;  // write perm. failure
  set_errno (0);
  return true;  // both are ok
}
```
Defines:
check_perms, used in chunks 576d and 579a.
Uses O_RDONLY 460b, O_RDWR 460b, O_WRONLY 460b, and set_errno 206b.

Note that some of this behavior is not obvious: for example, consider a user with user ID 1000 and group ID 1000 who wants to open the following file in $\texttt{O\_RDWR}_{460b}$ mode:

```
-r--rw-r--   1000  1000  filename
```

The file belongs to him and also to his group, but the owner permissions do not contain the right to write to the file. Access will be denied in this case, even though the group permissions would allow writing. In this case it just makes no sense that the owner's

write access is not enabled in the permission string. The user would first have to fix this situation via chmod$_{591b}$.

The u_execv$_{228b}$ function must also check whether it may load an application: this requires that the read and executable flags are set for the owner, group or others. We're leaving this as an exercise to you.

[580a] ⟨u_execv: *check permissions* 580a⟩≡ (228b)
```
// TO DO, see "Exercises" section.
```

15.1.2 Changing User and Group IDs

Now we can deal with the login$_{584c}$ and set*id system calls. As usual we start with the kernel functions which do the real work:

[580b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁579a 588a▷
```
int u_setuid  (word uid);    // set user ID
int u_setgid  (word gid);    // set group ID
int u_seteuid (word euid);   // set effective user ID
int u_setegid (word egid);   // set effective group ID
int u_login (word uid, char *pass);
```

All of these functions shall return 0 if they were successful and −1 otherwise. The implementations are simple, we only have to follow the rules described earlier:

[580c] ⟨*function implementations* 100b⟩+≡ (44a) ◁579c 581▷
```
int u_setuid (word uid) {
  TCB *t = &thread_table[current_task];
  ⟨begin critical section in kernel 380a⟩  // access thread table
  if (t->uid == 0 || uid == t->uid || uid == t->euid) {
    t->uid = uid;                 // set UID
    t->euid = uid;                // and also EUID
    ⟨end critical section in kernel 380b⟩
    return 0;                     // success
  } else {
    ⟨end critical section in kernel 380b⟩
    return -1;                    // failure
  }
}

int u_setgid (word gid) {
  TCB *t = &thread_table[current_task];
  ⟨begin critical section in kernel 380a⟩  // access thread table
  if (t->uid == 0 || gid == t->gid || gid == t->egid) {
    t->gid = gid;                 // set GID
    t->egid = gid;                // and also EGID
    ⟨end critical section in kernel 380b⟩
    return 0;                     // success
  } else {
    ⟨end critical section in kernel 380b⟩
```

```
    return -1;                    // failure
  }
}
```

Defines:
u_setgid, used in chunk 583a.
u_setuid, used in chunk 583a.
Uses current_task 192c, egid 573a, euid 573a, gid 573a, TCB 175, thread_table 176b, and uid 573a.

The implementations of $\texttt{u\_seteuid}_{581}$ and $\texttt{u\_setegid}_{581}$ are almost identical to the above code, they just skip setting the $\texttt{uid}_{573a}$ or $\texttt{gid}_{573a}$ elements of the TCB:

⟨*function implementations* 100b⟩+≡ (44a) ◁580c 582a▷ [581]

```
int u_seteuid (word uid) {
  TCB *t = &thread_table[current_task];
  ⟨begin critical section in kernel 380a⟩  // access thread table
  if (t->uid == 0 || uid == t->uid || uid == t->euid) {
    t->euid = uid;                // set the EUID (only!)
    ⟨end critical section in kernel 380b⟩
    return 0;                     // success
  } else {
    ⟨end critical section in kernel 380b⟩
    return -1;                    // failure
  }
}

int u_setegid (word gid) {
  TCB *t = &thread_table[current_task];
  ⟨begin critical section in kernel 380a⟩  // access thread table
  if (t->uid == 0 || gid == t->gid || gid == t->egid) {
    t->egid = gid;                // set the EGID (only!)
    ⟨end critical section in kernel 380b⟩
    return 0;                     // success
  } else {
    ⟨end critical section in kernel 380b⟩
    return -1;                    // failure
  }
}
```

Defines:
u_setegid, used in chunk 583a.
u_seteuid, used in chunk 583a.
Uses current_task 192c, egid 573a, euid 573a, gid 573a, TCB 175, thread_table 176b, and uid 573a.

The $\texttt{u\_login}_{582a}$ function is just a little more complicated: In a real Unix system it would look up the *password hash* stored in `/etc/passwd`, `/etc/shadow` or some similar file, calculate the hash from the password that was provided as the second argument, compare the two hashes and then decide on setting all user and group IDs (including the real user and group IDs $\texttt{ruid}_{573a}$, `guid`). Since we don't want to include a hash function in the ULIX source code, we just store the plaintext password in the file. We also restrict the password file size to 1024 bytes since the ULIX kernel has no advanced functions for line reading; we read one block of data and parse it byte by byte.

password hash

[582a] ⟨*function implementations* 100b⟩+≡ (44a) ◁581 588b▷

```
int u_login (word uid, char *pass) {
  TCB *t = &thread_table[current_task];
  char passwords[BLOCK_SIZE];
  char *words[128];
  int fd = u_open ("/etc/passwd", O_RDONLY, 0);
  if (fd == -1) return -1;                          // fail: no password database
  int size = u_read (fd, &passwords, BLOCK_SIZE);
  u_close (fd);
  int pos;  int index = 0;                          // position in words array

  words[index++] = (char*)&passwords;               // split
  for (pos = 1;  pos < size;  pos++) {
    if (passwords[pos] == ':' || passwords[pos] == '\n') {
      passwords[pos] = 0;                           // terminate string
      words[index++] = ((char*)&passwords)+pos+1;
    }
  }

  for (int i = 0;  i < index/5;  i++) {             // search
    if ( (atoi (words[5*i+2]) == uid)               // found right entry
          && strequal (words[5*i+1], pass) ) {      // password matches
      int gid = atoi (words[5*i+3]);                // get group ID
      u_chdir (words[5*i+4]);                       // make home directory the cwd
      t->uid = t->euid = t->ruid = uid;
      t->gid = t->egid = t->rgid = gid;
      return 0;                                     // success
    }
  }

  return -1;                                        // fail
}
```

Defines:
u_login, used in chunk 583a.
Uses atoi 595, BLOCK_SIZE 440a, current_task 192c, cwd, egid 573a, euid 573a, gid 573a, O_RDONLY 460b, passwd 584d, passwords, rgid 573a, ruid 573a, strequal 596a, TCB 175, thread_table 176b, u_chdir 432e, u_close 418a, u_open 412c, u_read 414b, and uid 573a.

As usual we need to provide system calls for these functions:

[582b] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁566c 587c▷

```
void syscall_setuid  (context_t *r);
void syscall_setgid  (context_t *r);
void syscall_seteuid (context_t *r);
void syscall_setegid (context_t *r);
void syscall_login   (context_t *r);
```

⟨ *syscall functions* 174b⟩+≡ (202b) ◁566d 587d▷ [583a]

```
void syscall_setuid (context_t *r) {
  // ebx: uid
  eax_return ( u_setuid (r->ebx) );
}

void syscall_setgid (context_t *r) {
  // ebx: gid
  eax_return ( u_setgid (r->ebx) );
}

void syscall_seteuid (context_t *r) {
  // ebx: euid
  eax_return ( u_seteuid (r->ebx) );
}

void syscall_setegid (context_t *r) {
  // ebx: egid
  eax_return ( u_setegid (r->ebx) );
}

void syscall_login (context_t *r) {
  // ebx: uid, ecx: password
  eax_return ( u_login (r->ebx, (char*)r->ecx) );
}
```

Defines:
syscall_login, used in chunk 583b.
syscall_setegid, used in chunk 583b.
syscall_seteuid, used in chunk 583b.
syscall_setgid, used in chunk 583b.
syscall_setuid, used in chunks 582b and 583b.
Uses context_t 142a, eax_return 174a, egid 573a, euid 573a, gid 573a, u_login 582a, u_setegid 581, u_seteuid 581, u_setgid 580c, u_setuid 580c, and uid 573a.

and enter the new handler function in the system call table:

⟨ *initialize syscalls* 173d⟩+≡ (44b) ◁567a 587e▷ [583b]

```
install_syscall_handler (__NR_setuid32, syscall_setuid);
install_syscall_handler (__NR_setgid32, syscall_setgid);
install_syscall_handler (__NR_setreuid32, syscall_seteuid);
install_syscall_handler (__NR_setregid32, syscall_setegid);
install_syscall_handler (__NR_login, syscall_login);
```

Uses __NR_login 584a, __NR_setgid32 204c, __NR_setregid32 204c, __NR_setreuid32 204c, __NR_setuid32 204c, install_syscall_handler 201b, syscall_login 583a, syscall_setegid 583a, syscall_seteuid 583a, syscall_setgid 583a, and syscall_setuid 583a.

(Note that the syscall numbers $\texttt{__NR\_setreuid32}_{204c}$ and $\texttt{__NR\_setregid32}_{204c}$ do not really match the corresponding functions ($\texttt{seteuid}_{584c}$ and $\texttt{setegid}_{584c}$), but they are the closest candidates, so we chose them instead of reserving new numbers. We must, however, declare the system call number $\texttt{__NR\_login}_{584a}$:

[584a] ⟨*public constants* 46a⟩+≡ (44a 48a) ◁562a 587a▷
```
#define __NR_login 523
```
Defines:
__NR_login, used in chunks 583b and 584c.

The user mode library functions just make the system calls:

[584b] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁568a 584e▷
```
int setuid  (word uid);    // set user ID
int setgid  (word gid);    // set group ID
int seteuid (word euid);   // set effective user ID
int setegid (word egid);   // set effective group ID
int login   (word uid, char *pass);
```

[584c] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁568b 585a▷
```
int setuid (word uid)  { return syscall2 (__NR_setuid32, uid);   }
int setgid (word gid)  { return syscall2 (__NR_setgid32, gid);   }
int seteuid (word uid) { return syscall2 (__NR_setreuid32, uid); }
int setegid (word gid) { return syscall2 (__NR_setregid32, gid); }
int login (word uid, char *pass) {
                          return syscall3 (__NR_login, uid, (unsigned int) pass); }
```
Defines:
login, used in chunks 191a and 562b.
Uses __NR_login 584a, __NR_setgid32 204c, __NR_setregid32 204c, __NR_setreuid32 204c, __NR_setuid32 204c, gid 573a, syscall2 203c, syscall3 203c, and uid 573a.

In order to let user mode programs look up `/etc/passwd` entries, we implement some functions in the library which fill data structures of the following type:

[584d] ⟨*ulixlib type definitions* 584d⟩≡ (48a)
```
struct passwd {
  char pw_name[32];    // user name
  char pw_passwd[32];  // password
  word pw_uid;         // user ID
  word pw_gid;         // group ID
  char *pw_gecos;      // long name        (ULIX: unused)
  char pw_dir[32];     // home directory
  char *pw_shell;      // shell            (ULIX: unused)
};
```
Defines:
passwd, used in chunks 582a and 585.

The functions

[584e] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁584b 586c▷
```
int getpwnam_r (const char *name,   struct passwd *pwd,
                char *buffer, int bufsize, struct passwd **result);
int getpwuid_r (word uid, struct passwd *pwd,
                char *buffer, int bufsize, struct passwd **result);
```

have to read the password file, search it for the username (or user ID) and then fill the supplied data structure. Since both reading the file and storing the data in the structure are the same in both functions, we use two code chunks that deal with these tasks.

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁584c 587b▷ [585a]

```
int getpwnam_r (const char *name,   struct passwd *pwd,
                char *buffer, int bufsize, struct passwd **result) {
  ⟨get password entry: read password file into passwords and parse it 585b⟩
  int i; for (i = 0;  i < index/5;  i++) {
    if ( strequal (words[5*i], name) ) {        // found right entry
      ⟨get password entry: fill target buffers 586a⟩
      return 0;   // success
    }
  }
  return -1;      // fail
}

int getpwuid_r (word uid, struct passwd *pwd,
                char *buffer, int bufsize, struct passwd **result) {
  ⟨get password entry: read password file into passwords and parse it 585b⟩
  int i; for (i = 0;  i < index/5;  i++) {
    if (atoi (words[5*i+2]) == uid) {           // found right entry
      ⟨get password entry: fill target buffers 586a⟩
      return 0;   // success
    }
  }
  return -1;      // fail
}
```

Uses atoi 595, getpwnam_r, getpwuid_r, passwd 584d, strequal 596a, and uid 573a.

with

⟨*get password entry: read password file into* passwords *and parse it* 585b⟩≡ (585a) [585b]

```
#define PASSWD_SIZE 1024
char passwords[PASSWD_SIZE] = "passwords";  char *words[128];
int fd = open ("/etc/passwd", O_RDONLY);
if (fd == -1) {
  printf ("Cannot open /etc/passwd\n");
  return -1;    // fail: no password database
}
int size = read (fd, (char*)passwords, PASSWD_SIZE);
if (size == -1) {
  printf ("Cannot read /etc/passwd, fd = %d\n", fd);
  return -1;    // fail: cannot read from password database
}
close (fd);

int index = 0;   // position in words array
words[index++] = (char*)passwords;
int pos; for (pos = 1;  pos < size;  pos++) {
  if (passwords[pos] == ':' || passwords[pos] == '\n') {
    passwords[pos] = 0;   // terminate string
    words[index++] = ((char*)&passwords)+pos+1;
  }
```

```
}
```

Uses close 429b, O_RDONLY 460b, open 429b, passwd 584d, PASSWD_SIZE, passwords, printf 601a, and read 429b.

and

[586a] ⟨*get password entry: fill target buffers* 586a⟩≡ (585a)

```
strncpy (pwd->pw_name, words[5*i], 32);     // field 0: username
strncpy (pwd->pw_passwd, words[5*i+1], 32); // field 1: password
pwd->pw_uid = atoi (words[5*i+2]);          // field 2: user ID
pwd->pw_gid = atoi (words[5*i+3]);          // field 3: group ID
strncpy (pwd->pw_dir, words[5*i+4], 32);    // field 4: home directory
```

Uses atoi 595 and strncpy 594b.

Once more: With the way we store the plaintext passwords in /etc/passwd, any user can inspect that file or use the getpw*_r functions to fetch the password. If we got rid of this property, the system would provide the same security as other Unix implementations do because the decision whether a user will be logged in or may change the user or group ID happens in the kernel's u_login$_{582a}$ and u_set*id functions. A non-*root* user mode process which calls setuid$_{584c}$ (0) will be denied.

15.1.3 The su Program

Using the login$_{584c}$ library function we can create a simple su implementation which reads in the password and tries to log in as root. It launches a new shell that runs with root privileges. When that shell is left, control returns to the original shell.

[586b] ⟨*lib-build/tools/su.c* 586b⟩≡

```
#include "../ulixlib.h"
int main () {
  char password[32];  printf ("Enter root password: ");
  ureadline ((char*)&password, sizeof(password)-1, false);  // no echo
  printf ("\n");
  if (login (0, password) == -1) { printf ("Login failed\n"); exit (1); }

  // exec shell
  char *args[] = { "/bin/sh", 0 };
  execv (args[0], args);
}
```

15.1.4 The getuid() and getgid() Functions

Processes sometimes need to query the user and group IDs with which they are running. For this purpose they can use the following functions:

[586c] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁584e 591a▷

```
word getuid  ();
word geteuid ();
word getgid  ();
word getegid ();
```

Since such functions are never needed in the kernel (where functions can simply look at the thread control block), we provide a simplified mechanism for quick access to these IDs:

⟨*public constants* 46a⟩+≡ (44a 48a) ◁584a [587a]
```
#define QUERY_UID  0
#define QUERY_EUID 1
#define QUERY_GID  2
#define QUERY_EGID 3
#define __NR_query_ids 524
```

⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁585a 591b▷ [587b]
```
word getuid  () { return syscall2 (__NR_query_ids, QUERY_UID);  }
word geteuid () { return syscall2 (__NR_query_ids, QUERY_EUID); }
word getgid  () { return syscall2 (__NR_query_ids, QUERY_GID);  }
word getegid () { return syscall2 (__NR_query_ids, QUERY_EGID); }
```
Uses `__NR_query_ids`, `QUERY_EGID`, `QUERY_EUID`, `QUERY_GID`, `QUERY_UID`, and `syscall2` 203c.

The system call handler in the kernel directly returns the queried value:

⟨*syscall prototypes* 173b⟩+≡ (202a) ◁582b 590a▷ [587c]
```
void syscall_query_ids (context_t *r);
```
Uses `context_t` 142a and `syscall_query_ids` 587d.

⟨*syscall functions* 174b⟩+≡ (202b) ◁583a 590b▷ [587d]
```
void syscall_query_ids (context_t *r) {
  // ebx: type of ID
  switch (r->ebx) {
    case QUERY_UID:  eax_return (thread_table[current_task].uid);
    case QUERY_EUID: eax_return (thread_table[current_task].euid);
    case QUERY_GID:  eax_return (thread_table[current_task].gid);
    case QUERY_EGID: eax_return (thread_table[current_task].egid);
    default:         eax_return (-1);
  }
}
```
Defines:
`syscall_query_ids`, used in chunk 587.
Uses `context_t` 142a, `current_task` 192c, `eax_return` 174a, `egid` 573a, `euid` 573a, `gid` 573a, `QUERY_EGID`, `QUERY_EUID`, `QUERY_GID`, `QUERY_UID`, `thread_table` 176b, and `uid` 573a.

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁583b 590c▷ [587e]
```
install_syscall_handler (__NR_query_ids, syscall_query_ids);
```
Uses `__NR_query_ids`, `install_syscall_handler` 201b, and `syscall_query_ids` 587d.

15.1.5 Changing Owner, Group and Permissions

All Unix systems provide system calls and library functions which allow users to change the owner, group and access permissions of any file or directory for which they have write access; Ulix is no different. We start with the kernel-internal functions:

[588a] ⟨*function prototypes* 45a⟩+≡ (44a) ◁580b 589b▷
```
int u_chown (const char *path, short owner, short group);
int u_chmod (const char *path, word mode);
```

Note how `u_chown`$_{588b}$ accepts negative arguments for the owner and group: if one or both of them are –1, they are ignored. Thus,

- `u_chown(file, 1000, 0)` will set `file`'s UID to 1000 and the GID to 0,
- `u_chown(file, 500, -1)` will only set the UID to 500,
- `u_chown(file, -1, 0)` ignores the UID argument and sets the GID to 0 and
- `u_chown(file, -1, -1)` does nothing at all.

The implementation looks similar to other functions which take a path name as an argument; we start with converting the path into an absolute path and checking which device with which filesystem the file resides on. Then we call filesystem-specific functions. As ULIX only supports the Minix filesystem for disk drives and we do not want `/dev` entries to change, the only option is to call `mx_chown`$_{589d}$:

[588b] ⟨*function implementations* 100b⟩+≡ (44a) ◁582a 589a▷
```
int u_chown (const char *path, short owner, short group) {
  char localpath[256], abspath[256];
  short device, fs;

  // only root may change file ownership / group
  if (scheduler_is_active && thread_table[current_task].euid != 0)  return -1;

  // check relative/absolute path
  if (*path != '/')  relpath_to_abspath (path, abspath);
  else               strncpy (abspath, path, 256);
  get_dev_and_path (abspath, &device, &fs, (char*)&localpath);
  switch (fs) {
    case FS_MINIX: return mx_chown (device, localpath, owner, group);
    case FS_FAT:   return -1;  // not possible (and FAT is not implemented)
    case FS_DEV:   return -1;  // not allowed
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}
```
Defines:
`u_chown`, used in chunks 588a and 590b.
Uses `current_task` 192c, `euid` 573a, `FS_DEV` 410a, `FS_ERROR` 410a, `FS_FAT` 410a, `FS_MINIX` 410a, `get_dev_and_path` 408c, `mx_chown` 589d, `relpath_to_abspath` 412b, `scheduler_is_active` 276e, `strncpy` 594b, and `thread_table` 176b.

The implementation of `u_chmod`$_{589a}$ looks almost identical to `u_chown`$_{588b}$, except that the number of arguments taken and passed to `mx_chown`$_{589d}$ or `mx_chmod`$_{589d}$ is different and regular users are allowed to change the access permissions (but not the ownership or owner group), so we don't need the check for the `root` user:

⟨*function implementations* 100b⟩+≡ (44a) ◁588b 589d▷ [589a]
```
int u_chmod (const char *path, word mode) {
  char localpath[256], abspath[256];
  short device, fs;

  // check relative/absolute path
  if (*path != '/')  relpath_to_abspath (path, abspath);
  else               strncpy (abspath, path, 256);

  if (scheduler_is_active) {
    ⟨u_chmod: check permissions 591c⟩   // see user/group chapter
  }

  get_dev_and_path (abspath, &device, &fs, (char*)&localpath);
  switch (fs) {
    case FS_MINIX: return mx_chmod (device, localpath, mode);
    case FS_FAT:   return -1;  // not possible, no FAT implementation
    case FS_DEV:   return -1;  // not allowed
    case FS_ERROR: return -1;  // error
    default:       return -1;
  }
}
```
Defines:
  u_chmod, used in chunk 590b.
Uses FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, get_dev_and_path 408c, mx_chmod 589d, relpath_to_abspath 412b, scheduler_is_active 276e, and strncpy 594b.

We only implement chown, chgrp and chmod for the Minix filesystem. Here are the corresponding mx_∗ functions

⟨*function prototypes* 45a⟩+≡ (44a) ◁588a 589c▷ [589b]
```
int mx_chown (int device, const char *path, short owner, short group);
int mx_chmod (int device, const char *path, word mode);
```

which use a more general function

⟨*function prototypes* 45a⟩+≡ (44a) ◁589b 601b▷ [589c]
```
int mx_chinode (int device, const char *path, short owner,
                short group, short mode);
```

that is able to change user ID, group ID and permissions in one step. It is called by both mx_chown$_{589d}$ and mx_chmod$_{589d}$ and will only modify the fields for which the provided values are $\neq -1$:

⟨*function implementations* 100b⟩+≡ (44a) ◁589a 598a▷ [589d]
```
int mx_chown (int device, const char *path, short owner, short group) {
  return mx_chinode (device, path, owner, group, -1);  // change UID or GID, not mode
}

int mx_chmod (int device, const char *path, word mode) {
  return mx_chinode (device, path, -1, -1, mode);      // change mode, not UID or GID
}
```

```
int mx_chinode (int device, const char *path, short owner,
                short group, short mode) {
  int ext_ino = mx_pathname_to_ino (device, path);
  if (ext_ino == -1) {
    printf ("file not found: %s\n", path);
    return -1;   // file not found
  }

  struct minix2_inode inode;
  mx_read_inode (device, ext_ino, &inode);
  if (owner != -1) inode.i_uid  = owner;  // change owner (if != -1)
  if (group != -1) inode.i_gid  = group;  // change group (if != -1)
  if (mode  != -1)
    // change mode  (if != -1)
    inode.i_mode = (inode.i_mode & ~07777) | (mode & 07777);
  mx_write_inode (device, ext_ino, &inode);
  return 0;
}
```

Defines:
mx_chinode, used in chunk 589c.
mx_chmod, used in chunk 589a.
mx_chown, used in chunks 588b and 589b.
Uses minix2_inode 442a, mx_pathname_to_ino 461d, mx_read_inode 451b, mx_write_inode 452a, and printf 601a.

We provide two system call handlers:

[590a] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁587c 610c▷

```
void syscall_chown (context_t *r);
void syscall_chmod (context_t *r);
```

[590b] ⟨*syscall functions* 174b⟩+≡ (202b) ◁587d 610d▷

```
void syscall_chown (context_t *r) {
  // ebx: path, ecx: owner, edx: group
  eax_return ( u_chown ((char *)r->ebx, r->ecx, r->edx) );
}

void syscall_chmod (context_t *r) {
  // ebx: path, ecx: new mode
  eax_return ( u_chmod ((char *)r->ebx, r->ecx) );
}
```

Defines:
syscall_chmod, used in chunk 590c.
syscall_chown, used in chunk 590.
Uses context_t 142a, eax_return 174a, u_chmod 589a, and u_chown 588b.

(which we need to initialize)

[590c] ⟨*initialize syscalls* 173d⟩+≡ (44b) ◁587e 611a▷

```
install_syscall_handler (__NR_chown, syscall_chown);
install_syscall_handler (__NR_chmod, syscall_chmod);
```

Uses __NR_chmod 204c, __NR_chown 204c, install_syscall_handler 201b, syscall_chmod 590b, and syscall_chown 590b.

and also two user mode library functions $\texttt{chown}_{591b}$ and $\texttt{chmod}_{591b}$ (the `chgrp` program will use the $\texttt{chown}_{591b}$ function):

⟨ *ulixlib function prototypes* 174c⟩+≡ (48a) ◁586c 598b▷ [591a]

```
int chown (const char *path, short owner, short group);
int chmod (const char *path, short mode);
```

and

⟨ *ulixlib function implementations* 174d⟩+≡ (48b) ◁587b 598c▷ [591b]

```
int chown (const char *path, short owner, short group) {
  return syscall4 (__NR_chown, (unsigned int)path, owner, group);
}

int chmod (const char *path, short mode) {
  return syscall3 (__NR_chmod, (unsigned int)path, mode);
}
```

Defines:
`chown`, used in chunk 591a.
Uses `__NR_chmod` 204c, `__NR_chown` 204c, `syscall3` 203c, and `syscall4` 203b.

15.2 Exercises

38. The code chunk ⟨`u_execv`: *check permissions* 580a⟩ is empty: When executing a program, this version of the Ulix kernel does not check whether the user is allowed to run the program. In general, users can run programs if they can read them and also have an execute …

    Implement the empty code chunk and test your checks against some binaries which have or do not have appropriate access permissions.

39. The $\texttt{u\_chmod}_{589a}$ function also needs to check whether it may change the access permissions. Fill the following code chunk:

    ⟨`u_chmod`: *check permissions* 591c⟩≡ (589a) [591c]

    ```
    // TO DO
    ```

    and test that you can only change permissions of files for which you have write access.

16
Small Standard Library

Some standard functions which are normally included with an operating system must be provided by us; here's a list of functions that are often used. Some of these functions will be part of both the kernel and the user mode library since features like formatting and printing a string are needed in both environments.

Looking at the implementations of these functions will add nothing new to your understanding of operating system concepts—that is why they have been moved to this late chapter. Following the literate programming concept that the document is the program, we decided to include them here even though they could have been moved to a separate code file. We will only provide few comments on these functions.

Some of these functions have not been implemented by us but were copied from online resources. In those cases, the first line after the function name lists the source.

16.1 Strings

Let's start with some string functions which compare, copy and convert strings to numbers:

⟨*public function prototypes* 454b⟩+≡ (44a 48a) ◁454b 596b▷ [593]

```
size_t strlen (const char *str);
int    strcmp (const char *str1, const char *str2);
int    strncmp (const char *str1, const char *str2, uint n);
char   *strncpy (char *dest, const char *src, size_t count);
char   *strcpy  (char *dest, const char *src);
int    atoi (char *s);
int    atoi8 (char *s);
```

The function `strlen`$_{594a}$ returns the length of a null-terminated string, `strcmp`$_{594a}$ and `strncmp`$_{594a}$ compare two strings and return 0 if they are equal and -1 or 1 if the first string is lexicographically smaller than the second one or vice versa. The `strncmp`$_{594a}$ variant stops comparing after n characters have been seen.

[594a] ⟨*public function implementations* 455a⟩+≡ (44a 48b) ◁455b 594b▷

```
size_t strlen (const char *str) {
    size_t retval;
    for (retval = 0; *str != '\0'; str++) retval++;
    return retval;
}

int strcmp (const char *s1, const char *s2) {
  // source: http://en.wikibooks.org/wiki/C_Programming/Strings
  while (*s1 != '\0' && *s1 == *s2) {
    s1++;  s2++;
  }
  byte b1 = (*(byte *) s1);
  byte b2 = (*(byte *) s2);
  return ((b1 < b2) ? -1 : (b1 > b2));
}

int strncmp (const char *s1, const char *s2, uint n) {
  // source: http://en.wikibooks.org/wiki/C_Programming/Strings
  if (n == 0) { return 0; }                    // nothing to compare? return 0
  while (n-- > 0 && *s1 == *s2) {
    if (n == 0 || *s1 == '\0') { return 0; }  // equality
    s1++;  s2++;
  }
  byte b1 = (*(byte *) s1);
  byte b2 = (*(byte *) s2);
  return ((b1 < b2) ? -1 : (b1 > b2));
}
```

Defines:
`strcmp`, used in chunk 596a.
`strlen`, used in chunks 232a, 234b, 408, 409, 412b, 419a, 455a, 484b, 577c, 641d, and 642a.
`strncmp`, used in chunks 229a, 562b, and 641e.
Uses `size_t` 46b.

The `strcmp`$_{594a}$ and `strncmp`$_{594a}$ functions copy a null-terminated string. While the first of the two will potentially go on copying forever if the source string is not terminated, the second function stops copying after `count` bytes. Note that if `strncpy`$_{594b}$ fills the whole target string (buffer), that string will not be null-terminated which can cause problems when correct termination is not checked and otherwise enforced.

[594b] ⟨*public function implementations* 455a⟩+≡ (44a 48b) ◁594a 595▷

```
char *strcpy (char *dest, const char *src) {
  char *ret = dest;
  while ((*dest++ = *src++) != '\0') ;
  return ret;
```

```
}

char *strncpy (char *dest, const char *src, size_t count) {
  // like memcpy (see next section), but copies only until first \0 character
  const char *sp = (const char *)src;
  char *dp = (char *)dest;
  for (; count != 0; count--) {
    *dp = *sp;
    if (*dp == 0) break;
    dp++; sp++;
  }
  return dest;
}
```

Defines:
strcpy, used in chunks 640 and 642b.
strncpy, used in chunks 224c, 234b, 367b, 409c, 411e, 412b, 419, 431, 432e, 455a, 461d, 488a, 490d, 492, 500, 577c, 586a, 588b, 589a, 593, and 641.
Uses size_t 46b.

`atoi`$_{595}$ converts a string into an integer value. It goes on reading until the first non-digit occurs, so it can also be used to convert strings like `"123 KByte"` to 123. It does not recognize negative values; for example, trying to convert the string `"-1234"` will lead to a result of 0 as the first character is found to be a non-digit.

⟨*public function implementations* 455a⟩+≡ (44a 48b) ◁594b 596c▷ [595]

```
int atoi (char *s) {
  int res = 0;
  while ( ('0' ≤ *s) && (*s ≤ '9') ) {
    res = res*10 + (*s-'0');
    s++;
  }
  return res;
};

int atoi8 (char *s) {   // same as atoi, but with octal numbers
  int res = 0;
  while ( ('0' ≤ *s) && (*s ≤ '7') ) {
    res = res*8 + (*s-'0');
    s++;
  }
  return res;
};
```

Defines:
atoi, used in chunks 582a, 585a, and 586a.

`atoi8`$_{595}$ is not a standard function. It works like `atoi`$_{595}$ but expects the string to contain an octal number instead of a decimal number.

We define two macros `strequal`$_{596a}$ (strings are equal) and `strdiff`$_{596a}$ (strings are different) which use `strcmp`$_{594a}$:

[596a] ⟨*public macro definitions* 596a⟩≡ (44a 48a)
```
#define strequal(s1,s2) (!strcmp((s1),(s2)))
#define strdiff(s1,s2)  (strcmp((s1),(s2)))
```
Defines:
strequal, used in chunks 432e, 462a, 480c, 495c, 499d, 582a, 585a, 608b, 610a, and 631.
Uses strcmp 594a.

They are more intuitive to use than the standard (in-)equality comparisons via $\texttt{strcmp}_{594a}$. Wherever we need to compare strings in this book, we only use our $\texttt{strequal}_{596a}$ and $\texttt{strdiff}_{596a}$ functions. However, if you want to integrate other code in ULIX or one of the user mode programs, the default $\texttt{strcmp}_{594a}$ function is available.

16.2 Memory

The standard functions $\texttt{memcpy}_{596c}$, $\texttt{memset}_{596c}$ and $\texttt{memsetw}_{596c}$ compare two chunks of memory and fill a memory area with a byte or word constant:

[596b] ⟨*public function prototypes* 454b⟩+≡ (44a 48a) ◁593 597b▷
```
void *memcpy  (void *dest, const void *src, size_t count);
void *memset  (void *dest, char val, size_t count);
word *memsetw (word *dest, word val, size_t count);
```

[596c] ⟨*public function implementations* 455a⟩+≡ (44a 48b) ◁595
```
void *memcpy (void *dest, const void *src, size_t count) {
  const char *sp = (const char *)src;
  char *dp = (char *)dest;
  for (; count != 0; count--)
    *dp++ = *sp++;
  return dest;
}

void *memset (void *dest, char val, size_t count) {
  char *temp = (char *)dest;
  for ( ; count != 0; count--) *temp++ = val;
  return dest;
}

word *memsetw (word *dest, word val, size_t count) {
  word *temp = (word *)dest;
  for ( ; count != 0; count--) *temp++ = val;
  return dest;
}
```
Defines:
memcpy, used in chunks 190a, 209b, 223e, 232a, 327a, 332b, 334a, 449, 451a, 453b, 455a, 456, 468b, 471c, 475c, 487a, 496d, 497, 509d, 510b, 518b, 519d, 521a, 530, 549c, 550b, and 597a.
memset, used in chunks 100c, 103b, 112, 121c, 122a, 164, 166a, 197a, 211a, 232c, 255c, 257c, 480c, 487a, and 509b.
memsetw, used in chunks 326c, 329b, 333e, 334a, and 609.
Uses size_t 46b.

Only in the kernel we provide the

⟨ *macro definitions* 35a⟩+≡ (44a) ◁471d 601c▷ [597a]
```
#define memcpy_debug(dest, src, count) \
  debug_printf ("DEBUG: memcpy() called in line %d\n", __LINE__); \
  memcpy (dest, src, count);
```
Uses debug_printf 601d and memcpy 596c.

macro which creates a debug message and calls memcpy$_{596c}$.

16.3 Formatted Output

We use a small implementation of the standard functions printf$_{601a}$ and sprintf$_{601a}$ that is dual-licensed unter the LGPL and the BSD license and available from http://www.menie.org/georges/embedded/ [Men02]—there is no need to reinvent the wheel. We modified the code so that printf$_{601a}$ can also handle the 'o' format character for octal numbers and we changed the code indentation to match the style of the other code in this book. There were also some minor modifications that enable the functions to use the Ulix output functions. (Thankfully the function already knew how to print numbers to any base; we just copied the code for 'x' and changed the base 16 to 8.)

The code was also changed a bit to make it shorter, and we have turned the leading comments into normal text to make them better readable.

/* Copyright 2001, 2002 Georges Menie (http://www.menie.org)

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA */

/* putchar$_{598a}$ is the only external dependency for this file, if you have a working putchar$_{598a}$, just remove the following define. If the function should be called something else, replace outbyte(c) by your own function call. */

16.3.1 printf in the Kernel

The kernel uses the functions

⟨ *public function prototypes* 454b⟩+≡ (44a 48a) ◁596b [597b]
```
int printf  (const char *format, ...);
int sprintf (char *out, const char *format, ...);
```

and we have to define `putchar`$_{598a}$ as `kputch`$_{335b}$ so that `printf`$_{601a}$ will call the right character output function.

[598a] ⟨*function implementations* 100b⟩+≡ (44a) ◁589d 601d▷

```
#define putchar(c) kputch (c)

static void printchar (char **str, int c) {
  if ((int)str == -1) {
    // debug output: goes to qemu serial console via uartputc
    if (c == 0x100) {  //  backspace
      uartputc ('\b'); uartputc (' '); uartputc ('\b');
    } else {
      uartputc (c);
    }
  } else if (str) {
    **str = c; ++(*str);
  } else {
    (void)putchar (c);
  }
}
```

⟨*public* `printf` *implementation* 599a⟩

Defines:
`printchar`, used in chunks 599 and 600.
`putchar`, used in chunk 556.
Uses `kputch` 335b and `uartputc` 336b.

The `printchar`$_{598a}$ function uses `uartputc`$_{336b}$ to write output (only) to the serial port for debugging purposes if `str` is -1. `kputch`$_{335b}$ always writes to the serial port, too.

16.3.2 `printf` in the User Mode Library

For the library, we use the same `printf`$_{601a}$ code and only need to modify the implementation of `putchar`$_{598a}$ and `printchar`$_{598a}$. That is why we can provide most of the code via the ⟨*public* `printf` *implementation* 599a⟩ chunk. Here we also define the `ulixlib_printchar`$_{598c}$ function for printing single characters that writes to the standard output descriptor (1).

[598b] ⟨*ulixlib function prototypes* 174c⟩+≡ (48a) ◁591a

```
int ulixlib_printchar (byte c);
```

[598c] ⟨*ulixlib function implementations* 174d⟩+≡ (48b) ◁591b

```
int ulixlib_printchar (byte c) { write (STDOUT_FILENO, &c, 1); }

#define putchar(c) ulixlib_printchar(c)

static void printchar (char **str, int c) {
  if (str) { **str = c; ++(*str); }
  else     (void)putchar(c);
}
```

⟨*public* `printf` *implementation* 599a⟩

Defines:
printchar, used in chunks 599 and 600.
putchar, used in chunk 556.
ulixlib_printchar, used in chunk 598b.
Uses STDOUT_FILENO 415b and write 429b.

16.3.3 The Generic Implementation

⟨*public* printf *implementation* 599a⟩≡ (598) 599b▷ [599a]

```
#define PAD_RIGHT 1
#define PAD_ZERO 2

static int prints (char **out, const char *string, int width, int pad) {
  register int pc = 0, padchar = ' ';
  if (width > 0) {
    register int len = 0; register const char *ptr;
    for (ptr = string; *ptr; ++ptr) ++len;
    if (len ≥ width) width = 0;
    else width -= len;
    if (pad & PAD_ZERO) padchar = '0';
  }
  if (!(pad & PAD_RIGHT)) {
    for ( ; width > 0; --width) { printchar (out, padchar); ++pc; }
  }
  for ( ; *string ; ++string) { printchar (out, *string); ++pc; }
  for ( ; width > 0; --width) { printchar (out, padchar); ++pc; }
  return pc;
}
```

Defines:
PAD_RIGHT, used in chunk 600.
PAD_ZERO, used in chunks 599b and 600.
prints, used in chunks 599b and 600.
Uses printchar 598a 598c.

The $\texttt{printi}_{599b}$ functions deals with 32-bit integers whose textual representation fits inside a 34-byte buffer:

⟨*public* printf *implementation* 599a⟩+≡ (598) ◁599a 600▷ [599b]

```
#define PRINT_BUF_LEN 34

static int printi (char **out, int i, int b, int sg, int width, int pad, int letbase) {
  char print_buf[PRINT_BUF_LEN];
  register char *s; register int t, neg = 0, pc = 0; register unsigned int u = i;
  if (i == 0) {
    print_buf[0] = '0';  print_buf[1] = '\0';
    return prints (out, print_buf, width, pad);
  }
  if (sg && b == 10 && i < 0) { neg = 1; u = -i; }
  s = print_buf + PRINT_BUF_LEN-1;  *s = '\0';
  while (u) {
    t = u % b;        if ( t ≥ 10 )  t += letbase - '0' - 10;
```

```
    *--s = t + '0';   u /= b;
  }
  if (neg) {
    if ( width && (pad & PAD_ZERO) ) { printchar (out, '-'); ++pc; --width; }
    else                             { *--s = '-'; }
  }
  return pc + prints (out, s, width, pad);
}
```

Defines:
printi, used in chunk 600.
Uses PAD_ZERO 599a, printchar 598a 598c, and prints 599a.

We have modified the $\texttt{print}_{600}$ function so that it recognizes the `%o` (octal) and `%b` (binary) format words and prints numbers with base 8 or base 2, respectively.

[600] ⟨*public* printf *implementation* 599a⟩+≡ (598) ◁599b 601a▷

```
static int print (char **out, int *varg) {
  register int width, pad; register int pc = 0;
  register char *format = (char *)(*varg++); register char *s; char scr[2];
  for (; *format != 0; ++format) {
    if (*format == '%') {
      ++format; width = pad = 0;
      if (*format == '\0') break;
      if (*format == '%')  goto outlabel;
      if (*format == '-')  { ++format; pad = PAD_RIGHT; }
      while (*format == '0') { ++format; pad |= PAD_ZERO; }
      for ( ; *format ≥ '0' && *format ≤ '9'; ++format) {
        width *= 10;  width += *format - '0';
      }
      switch (*format) {
        case 's': s = *((char **)varg++);
                  pc += prints (out, s?s:"(null)", width, pad);          continue;
        case 'd': pc += printi (out, *varg++, 10, 1, width, pad, 'a');  continue;
        case 'o': pc += printi (out, *varg++,  8, 0, width, pad, 'a');  continue;
        case 'b': pc += printi (out, *varg++,  2, 0, width, pad, 'a');  continue;
        case 'x': pc += printi (out, *varg++, 16, 0, width, pad, 'a');  continue;
        case 'X': pc += printi (out, *varg++, 16, 0, width, pad, 'A');  continue;
        case 'u': pc += printi (out, *varg++, 10, 0, width, pad, 'a');  continue;
        case 'c': // char are converted to int then pushed on the stack
                  scr[0] = *varg++;  scr[1] = '\0';
                  pc += prints (out, scr, width, pad);                  continue;
      }
    } else {
      outlabel:
      printchar (out, *format);  ++pc;
    }
  }
  if ( (int)out != -1 && out )  **out = '\0';
  return pc;
}
```

Defines:
print, used in chunks 431, 601, and 626b.
Uses PAD_RIGHT 599a, PAD_ZERO 599a, printchar 598a 598c, printi 599b, and prints 599a.

⟨*public* printf *implementation* 599a⟩+≡ (598) ◁600 [601a]
```
int printf (const char *format, ...) {
  register int *varg = (int *)(&format); return print (0, varg);
}

int sprintf (char *out, const char *format, ...) {
  register int *varg = (int *)(&format); return print (&out, varg);
}
```
Defines:
printf, used in chunks 45d, 151c, 152a, 164, 165a, 168d, 170e, 191a, 201d, 214, 290, 291, 293b, 297, 299e, 308c, 311b, 321a, 324a, 326c, 337c, 340b, 349, 406, 416d, 417, 431, 450a, 456, 471c, 476b, 480, 488a, 513e, 532d, 534b, 536b, 537a, 539c, 547d, 551, 552c, 562b, 564, 585b, 589d, 603–8, 610–14, 626a, and 639c.
sprintf, used in chunks 280a, 342b, 343b, 369c, 597b, and 608b.
Uses print 600.

The debug_printf$_{601d}$ function

⟨*function prototypes* 45a⟩+≡ (44a) ◁589c 610b▷ [601b]
```
int debug_printf (const char *format, ...);
```

exists only in the kernel, it is similar to printf$_{601a}$ if the DEBUG macro is set but passes the target argument −1 instead of 0 so that output will go *only* to the serial port (and not to the Ulix screen). If DEBUG is not set, it does nothing.

Since writing many messages makes the system a bit slower, debug output is disabled by default:

⟨*macro definitions* 35a⟩+≡ (44a) ◁597a [601c]
```
// #define DEBUG
```

⟨*function implementations* 100b⟩+≡ (44a) ◁598a 603▷ [601d]
```
#ifdef DEBUG
  int debug_printf (const char *format, ...) {
    register int *varg = (int *)(&format); return print ((char**)-1, varg);
  }
#else
  inline int debug_printf (const char *format, ...) { return 0; }  // do nothing
#endif
```
Defines:
debug_printf, used in chunks 277b, 366c, 597a, and 601b.
Uses print 600.

17 Debugging Help

17.1 The Kernel Mode Shell

In cases when user mode does not work or when we need to look at kernel structures that are not accessible from user mode, we can launch a simple kernel shell `kernel_shell`$_{610a}$ that provides a few commands which are helpful for debugging.

This code is even less interesting than the one in the previous chapter unless you want to modify Ulix and run into problems that require some debugging.

For most internal commands `CMD` of the kernel shell we provide corresponding functions named `ksh_command_CMD` which will be executed.

You can enter the kernel shell by pressing [Shift-Escape], and you can leave it with `exit` which brings you back to user mode. When the system detects a fault from which it cannot recover it will also drop you in the kernel mode shell, but in that case returning to user mode is not possible.

The `test` command displays addresses of the current page directory and page table and a hexdump of the frame table.

⟨ *function implementations* 100b ⟩+≡ (44a) ◁601d 604b▷ [603]

```
void ksh_command_test () {
  kputs ("current_pd as INT:              ");
    printbitsandhex (*(uint*)(current_pd)); kputs ("\n");
  kputs ("current_pd->ptds[0].frame_addr.:");
    printbitsandhex (current_pd->ptds[0].frame_addr<<12); kputs ("\n");
  kputs ("current_pt as INT:              ");
    printbitsandhex (*(uint*)(current_pt)); kputs ("\n");
  kputs ("address of current_pd:          "); printf ("%08x\n",(uint)current_pd);
  kputs ("address of current_pt:          "); printf ("%08x\n",(uint)current_pt);
```

```
    kputs ("size of current_pd:             ");
      printf ("%08x\n", sizeof (*current_pd));
    kputs ("size of current_pt:             ");
      printf ("%08x\n", sizeof (*current_pt));
    kputs ("address of frame table:         "); printf ("%08x\n", (uint)ftable);
    kputs ("hexdump ftable\n");
    hexdump ((uint)&place_for_ftable, ((uint)&place_for_ftable) + 1);
  };
```

Defines:
ksh_command_test, used in chunk 608b.
Uses current_pd 105a, current_pt 105a, ftable 112c, hexdump 612c, kputs 335b, place_for_ftable 112c, printbitsandhex 612a, and printf 601a.

mem displays similar data, but for specific page table entries.

[604a] ⟨*global variables* 92b⟩+≡ (44a) ◁547b

```
  extern memaddress stack_first_address, stack_last_address;
```

Uses memaddress 46c, stack_first_address 95a, and stack_last_address 95a.

[604b] ⟨*function implementations* 100b⟩+≡ (44a) ◁603 605a▷

```
  void ksh_command_mem () {
    kputs ("kernel_pd as INT:               ");
      printbitsandhex (*(int*)(&kernel_pd)); kputs ("\n");
    kputs ("kernel_pd.ptds[0].frame_addr:   ");
      printbitsandhex (kernel_pd.ptds[0].frame_addr<<12); kputs ("\n");
    kputs ("kernel_pd.ptds[768].frame_addr: ");
      printbitsandhex (kernel_pd.ptds[768].frame_addr<<12); kputs ("\n");
    kputs ("kernel_pd.ptds[831].frame_addr: ");
      printbitsandhex (kernel_pd.ptds[831].frame_addr<<12); kputs ("\n");
    kputs ("kernel_pd.ptds[832].frame_addr: ");
      printbitsandhex (kernel_pd.ptds[832].frame_addr<<12); kputs ("\n");
    kputs ("kernel_pd.ptds[833].frame_addr: ");
      printbitsandhex (kernel_pd.ptds[833].frame_addr<<12); kputs ("\n");
    kputs ("kernel_pt as INT:               ");
      printbitsandhex (*(int*)(&kernel_pt)); kputs ("\n");
    kputs ("address of kernel_pd:           ");
      printf ("%08x\n", (uint)&kernel_pd);
    kputs ("address of kernel_pt:           ");
      printf ("%08x\n", (uint)&kernel_pt);
    kputs ("stack_first_address:            ");
      printf ("%08x\n", (uint)&stack_first_address);
    kputs ("stack_last_address:             ");
      printf ("%08x\n", (uint)&stack_last_address);
    kputs ("free_frames:                    "); printf ("%d\n", free_frames);
    uint esp; asm volatile ("mov %%esp, %0": "=r"(esp));
    kputs ("ESP:                            "); printf ("%08x\n", esp);
  };
```

Defines:
ksh_command_mem, used in chunk 608b.
Uses free_frames 112b, kernel_pd 105a, kernel_pt 105a, kputs 335b, printbitsandhex 612a, printf 601a, stack_first_address 95a, and stack_last_address 95a.

`time` and `uname` show the current time and the ULIX version string.

⟨*function implementations* 100b⟩+≡ (44a) ◁604b 605b▷ [605a]
```
void ksh_command_time () {
  short int hour, min, sec;
    hour = (system_time/60/60)%24; min = (system_time/60)%60; sec = system_time%60;
    printf ("The time is %02d:%02d:%02d.\n", hour, min, sec);
};

void ksh_command_uname () { printf ("%s; Build: %s \n", UNAME, BUILDDATE); };
```
Defines:
`ksh_command_time`, used in chunk 608b.
`ksh_command_uname`, used in chunk 608b.
Uses `BUILDDATE` 35a, `hour`, `min`, `printf` 601a, `sec`, `system_time` 338a, and `UNAME` 35a.

`div0` causes a division by zero fault.

⟨*function implementations* 100b⟩+≡ (44a) ◁605a 605c▷ [605b]
```
void ksh_command_div0 () {
  int zero = 0; int i = 10 / zero; kputch (i);  // Test for exception
}
```
Defines:
`ksh_command_div0`, used in chunk 608b.
Uses `kputch` 335b.

`hexdump` prints a hex dump of the specified address range. Since kernel shell commands do not take parameters, the addresses have to be changed in the source code in order to show a different range.

⟨*function implementations* 100b⟩+≡ (44a) ◁605b 605d▷ [605c]
```
void ksh_command_hexdump () {
  int as = current_as;
  activate_address_space (10);
  hexdump (0xaffffdf8, 0xaffffdf8 + 128);   // modify this to show other regions
  activate_address_space (as);
};
```
Defines:
`ksh_command_hexdump`, used in chunk 608b.
Uses `activate_address_space` 170c, `current_as` 170b, and `hexdump` 612c.

`ps` shows the process list.

⟨*function implementations* 100b⟩+≡ (44a) ◁605c 606▷ [605d]
```
void ksh_command_ps () {
  int i;
  printf (" TID  PID PPID ESP      EIP      EBP      ESP0     AS  State "
          "Exi Cmdline\n");
  for (i=0;i<MAX_THREADS; i++) {
    if (thread_table[i].used) {
      printf ("%4d %4d %4d %08x %08x %08x %08x %2d  %-5s %3d %s\n",
        thread_table[i].tid,                thread_table[i].pid,
        thread_table[i].ppid,               thread_table[i].regs.esp,
        thread_table[i].regs.eip,           thread_table[i].regs.ebp,
```

```
            thread_table[i].esp0,               thread_table[i].addr_space,
            state_names[thread_table[i].state], thread_table[i].exitcode,
            thread_table[i].cmdline);
        }
      }
    }
```

Defines:
ksh_command_ps, used in chunk 608b.
Uses MAX_THREADS 176a, printf 601a, state_names 180b, and thread_table 176b.

queues shows all blocked queues and the processes or threads on those queues (as well as the ready queue). The ksh_command_queues$_{606}$ function uses the ksh_print_queue$_{606}$ helper function which displays a single queue. Similarly, locks shows all processes or threads waiting for a lock.

[606] ⟨*function implementations* 100b⟩+≡ (44a) ◁605d 607a▷

```
void ksh_print_queue (char *name, blocked_queue *bq) {
  printf ("%s: ", name);
  int pid = bq->next;
  while (pid != 0) {
    printf ("%d, ", pid);
    pid = thread_table[pid].next;
  }
  printf ("\n");
}

void ksh_command_queues () {
  printf ("Queues: \n");
  printf ("ready: ");
  int pid = 0;
  while ((pid = thread_table[pid].next) != 0)  printf ("%d, ", pid);
  printf ("\n");
  ksh_print_queue ("keyboard", &keyboard_queue);
  ksh_print_queue ("harddisk", &harddisk_queue);
  ksh_print_queue ("floppy",   &floppy_queue);
  ksh_print_queue ("waitpid",  &waitpid_queue);
  ksh_print_queue ("buffer",   &(buffer_lock->bq));
}

void ksh_command_locks () {
  for (int i = 1;  i < MAX_LOCKS;  i++) {
    if (kernel_locks[i].used) {
      ksh_print_queue (kernel_locks[i].lockname, &kernel_locks[i].bq);
    }
  }
}
```

Defines:
ksh_command_locks, used in chunk 608b.
ksh_command_queues, used in chunk 608b.
Uses blocked_queue 183a, buffer_lock 509a, floppy_queue 544d, harddisk_queue 529a, kernel_locks 365c, keyboard_queue 323d, MAX_LOCKS 365b, printf 601a, thread_table 176b, waitpid 220d, and waitpid_queue 218b.

`inode` displays the hex dump of an inode. The device and number must be set directly in the source code.

⟨*function implementations* 100b⟩+≡ (44a) ◁606 607b▷ [607a]
```
void ksh_command_inode () {
  struct minix2_inode in;
  int dev = DEV_HDA;
  int ino = 79;
  int res = mx_read_inode (dev, ino, &in);
  printf ("mx_read_inode(%d, %d) returns %d\n", dev, ino, res);
  if (res != 0) {
    hexdump ((uint)&in, (uint)&in+sizeof(struct minix2_inode));
    printf ("size: %d, blocks: ", in.i_size);
    for (int i = 0; i < 10; i++) printf ("%d, ", in.i_zone[i]); printf ("\n");
  }
}
```
Defines:
`ksh_command_inode`, used in chunk 608b.
Uses `DEV_HDA` 508a, `hexdump` 612c, `minix2_inode` 442a, `mx_read_inode` 451b, and `printf` 601a.

`lsof` displayes the list of open Minix files.

⟨*function implementations* 100b⟩+≡ (44a) ◁607a 607c▷ [607b]
```
void ksh_command_lsof () {
  for (int i = 0;  i < MX_MAX_FILES;  i++) {
    struct int_minix2_inode *inode = mx_status[i].int_inode;
    if (inode != NULL) {
      printf ("mfd=%d inode-addr=%08x size=%d\n",
              i, (unsigned int)inode, inode->i_size);
    }
  }
}
```
Defines:
`ksh_command_lsof`, used in chunk 608b.
Uses `int_minix2_inode` 459a, `MX_MAX_FILES` 461a, `mx_status` 461b, `NULL` 46a, and `printf` 601a.

`longhelp` explains (some of) the commands in the kernel shell.

⟨*function implementations* 100b⟩+≡ (44a) ◁607b 608b▷ [607c]
```
void ksh_command_longhelp () {
  printf ("ex" "it         return to user mode\n"
          "te" "st\n"
          "pfault, div0 test faults\n"
          "mem          show memory (frames, pages) info\n"
          "st" "at\n"
          "uname        show Ulix version\n"
          "hex" "dump      show hex" "dump of some memory area\n"
          "clear        clear the screen\n"
          "gf, g" "p, gp1k get a frame, a page, 1000 pages\n"
          "rp           release page\n"
          "bdump\n"
          "malloc       test kernel malloc\n"
```

```
            "time         show time\n"
            "cloneas <n>  clone address space (argument: size)\n"
            "listas       show address spaces\n"
            "ps           process list\n"
            "disable      disable sche" "duler\n"
            "enable       (re-)enable sche" "duler\n"
    );
  }
```

Defines:
ksh_command_longhelp, used in chunk 608b.
Uses faults 148a, g, and printf 601a.

The ksh_run_command$_{608b}$ function tests whether the command that was entered is known. If so, it executes one of the ksh_command_∗ functions (or directly performs some action).

[608a] ⟨*constants* 112a⟩+≡ (44a) ◁579b

```
#define SHELL_COMMANDS "help, ex" "it, test, div0, mem, stat, uname, "\
  "hexdump, clear, gf, gp, rp, gp1k, bdump, malloc, time, listas, "\
  "init, exec, testdisk, enable, longhelp, ps, queues, lsof"
```

Defines:
SHELL_COMMANDS, used in chunks 608b and 610a.
Uses gp 92b, hexdump 612c, and stat 429b 489b.

[608b] ⟨*function implementations* 100b⟩+≡ (44a) ◁607c 609▷

```
void ksh_run_command (char *s) {
  if      ( strequal (s, "help") )     { printf ("Commands: %s \n",
                                                 SHELL_COMMANDS); }
  else if ( strequal (s, "uname") )    { ksh_command_uname (); }
  else if ( strequal (s, "test") )     { ksh_command_test (); }
  else if ( strequal (s, "div0") )     { ksh_command_div0 (); }
  else if ( strequal (s, "hexdump") )  { ksh_command_hexdump (); }
  else if ( strequal (s, "clear") )    { vt_clrscr (); }
  else if ( strequal (s, "mem") )      { ksh_print_page_table (); }
  else if ( strequal (s, "mem2") )     { ksh_command_mem (); }
  else if ( strequal (s, "ps") )       { ksh_command_ps (); }
  else if ( strequal (s, "queues") )   { ksh_command_queues (); }
  else if ( strequal (s, "locks") )    { ksh_command_locks (); }
  else if ( strequal (s, "longhelp") ) { ksh_command_longhelp (); }
  else if ( strequal (s, "enable") )   { ⟨enable scheduler 276a⟩ }
  else if ( strequal (s, "disable") )  { ⟨disable scheduler 276b⟩ }
  else if ( strequal (s, "listas") )   { list_address_spaces (); }
  else if ( strequal (s, "time") )     { ksh_command_time (); }
  else if ( strequal (s, "lsof") )     { ksh_command_lsof (); }
  else if ( strequal (s, "inode") )    { ksh_command_inode ();
  } else if ( strequal (s, "gf") ) {
    uint newframeid = request_new_frame ();
    printf ("New frame ID: %d\n", newframeid);
  } else if ( strequal (s, "gp") ) {
    /* uint* page = */ request_new_page ();
    // kputs (", Page @ "); printf ("%08x\n", (uint)page);
  } else if ( strequal (s, "rp") ) {
```

```
    printf ("releasing page range 0xc03fe..0xc07e6 \n");
    release_page_range (0xc03fe,0xc07e6);
  } else if ( strequal (s, "gp1k") ) {
    char buf[20]; uint *page;
    for (int i = 0;  i < 1024;  i++) {
      sprintf ((char*)&buf, "Create: %d   ", i); set_statusline ((char*)&buf);
      page = request_new_page ();
    }
  } else if ( strequal (s, "gp10k") ) {
    char buf[20]; uint *page;
    for (int i = 0;  i < 10;  i++) {
      sprintf ((char*)&buf, "Create: %d   ", i); set_statusline ((char*)&buf);
      page = request_new_pages (1024);
    }
  }
  else if ( strequal (s, "") ) { return; } // no command
  else { printf ("Error: >%s< - no such command\n", s); }
}
```

Defines:
ksh_run_command, used in chunk 610a.
Uses gp 92b, hexdump 612c, kputs 335b, ksh_command_div0 605b, ksh_command_hexdump 605c, ksh_command_inode 607a, ksh_command_locks 606, ksh_command_longhelp 607c, ksh_command_lsof 607b, ksh_command_mem 604b, ksh_command_ps 605d, ksh_command_queues 606, ksh_command_test 603, ksh_command_time 605a, ksh_command_uname 605a, ksh_print_page_table 613b, list_address_spaces 171a, printf 601a, release_page_range 123d, request_new_frame 118b, request_new_page 120a, request_new_pages 120b, set_statusline 337b, SHELL_COMMANDS 608a, sprintf 601a, strequal 596a, and vt_clrscr 329b.

The two statusline_* functions change the color of the status line so that it is always obvious whether you are using a regular shell (blue blackground) or the kernel shell (red).

⟨*function implementations* 100b⟩+≡ (44a) ◁608b 610a▷ [609]

```
void statusline_red () {
  // make status line red
  memsetw (textmemptr + 24 * 80, 0x20 | VT_RED_BACKGROUND, 80);
}

void statusline_blue () {
  // make status line blue
  memsetw (textmemptr + 24 * 80, 0x20 | VT_BLUE_BACKGROUND, 80);
  set_statusline (UNAME);
}
```

Defines:
statusline_blue, used in chunk 610a.
statusline_red, used in chunk 610a.
Uses memsetw 596c, set_statusline 337b, textmemptr 116c, UNAME 35a, VT_BLUE_BACKGROUND 326b, and VT_RED_BACKGROUND 326b.

Finally, this is the kernel shell. It reads in a command and calls ksh_run_command$_{608b}$.

[610a] ⟨*function implementations* 100b⟩+≡ (44a) ◁609 611b▷

```
void kernel_shell () {
  ⟨enable interrupts 47b⟩
  statusline_red ();
  char s[101];

  system_kbd_pos = 0;
  system_kbd_lastread = -1;
  system_kbd_count = 0;

  printf ("\nUlix Kernel Shell. Commands: %s\n", SHELL_COMMANDS);
  printf ("Type 'ex" "it' to enter user mode.\n");
  for (;;) {
    set_statusline (UNAME);
    kputs ("kernel@ulix# ");
    kreadline ((char*)&s,sizeof (s)-1);
    if ( strequal ((char*)&s, "ex" "it") ) {
      statusline_blue ();  // restore normal color
      ⟨enable scheduler 276a⟩
      return;
    }
    ksh_run_command ((char*)&s);
  };
};
```

Defines:
`kernel_shell`, used in chunks 151c, 290b, 321a, and 610b.
Uses `kputs` 335b, `kreadline` 324b, `ksh_run_command` 608b, `printf` 601a, `set_statusline` 337b, `SHELL_COMMANDS` 608a, `statusline_blue` 609, `statusline_red` 609, `strequal` 596a, `system_kbd_count` 318d, `system_kbd_lastread` 318d, `system_kbd_pos` 318d, and `UNAME` 35a.

[610b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁601b 611c▷

```
void kernel_shell ();
```

17.2 A System Call That Displays an Inode

For testing the Minix filesystem implementation we provide a system call that reads a Minix inode from the disk and displays it. It also shows the first seven entries of the `i_zone[]` array (which contain the direct block numbers).

[610c] ⟨*syscall prototypes* 173b⟩+≡ (202a) ◁590a

```
void syscall_print_inode (context_t *r);
```

Uses `context_t` 142a and `syscall_print_inode`.

[610d] ⟨*syscall functions* 174b⟩+≡ (202b) ◁590b

```
void syscall_print_inode (context_t *r) {
  int ino = r->ebx;   // requested inode
  printf ("syscall; ino = %d\n", ino);
```

```
    struct minix2_inode in;
    mx_read_inode (DEV_HDA, ino, &in);
    printf ("i_mode:   0%o\n", in.i_mode);
    printf ("i_nlinks: %d\n", in.i_nlinks);
    printf ("i_size:   %d\n", in.i_size);
    printf ("i_zone:   [");
    for (int i = 0; i < 7; i++) printf ("%d, ", in.i_zone[i]);  printf ("]\n");
  }
```

Uses context_t 142a, DEV_HDA 508a, minix2_inode 442a, mx_read_inode 451b, printf 601a, and syscall_print_inode.

⟨*initialize syscalls* 173d⟩+≡ (44b) ◁590c [611a]

```
  install_syscall_handler (777, syscall_print_inode);
```

Uses install_syscall_handler 201b and syscall_print_inode.

The system call should not be used for regular programs, instead the stat$_{429b}$ function is intended to return information about a file.

17.3 Printing the Page Directory

The following function prints parts of the page directory.

⟨*function implementations* 100b⟩+≡ (44a) ◁610a 612a▷ [611b]

```
  void print_page_directory () {
    int i;
    kputs ("The Page Directory:\n");
    for ( i = 700 ; i<800 ; i++ ) {
      if ( current_pd->ptds[i].present ) {
        printf ("%04d ", i);
        printf ("%08x\n", current_pd->ptds[i].frame_addr);
      };
    };

    unsigned int z=(unsigned int)current_pd;
    printf ("hexdump for %08x\n", z);
    hexdump (z,z+128);
    kputch ('\n');
  };
```

Uses current_pd 105a, hexdump 612c, kputch 335b, kputs 335b, and printf 601a.

17.4 Helper Functions for Printing

Some functions that belong to the kernel mode shell use the following helper functions to print number in binary and hexadecimal format and to print a hex dump.

⟨*function prototypes* 45a⟩+≡ (44a) ◁610b 612b▷ [611c]

```
  void printbitsandhex (uint i);
```

[612a] ⟨*function implementations* 100b⟩+≡ (44a) ◁611b 612c▷
```
void printbitsandhex (uint i) { printf ("%032b %08X", i, i); };
```
Defines:
printbitsandhex, used in chunks 603, 604b, and 611c.
Uses printf 601a.

[612b] ⟨*function prototypes* 45a⟩+≡ (44a) ◁611c 613a▷
```
void hexdump (uint startval, uint endval);
```

[612c] ⟨*function implementations* 100b⟩+≡ (44a) ◁612a 612d▷
```
void hexdump (uint startval, uint endval) {
  for (uint i=startval; i < endval; i+=16) {
    printf ("%08x  ", i);                    // address
    for (int j = i;  j < i+16;  j++) {       // hex values
      printf ("%02x ", (byte)PEEK(j));
      if (j==i+7) printf (" ");
    };
    printf (" ");
    for (int j = i;  j < i+16;  j++) {       // characters
      char z = PEEK(j);
      if ((z≥32) && (z<127)) {
        printf ("%c", PEEK(j));
      } else {
        printf (".");
      };
    };
    printf ("\n");
  };
};
```
Defines:
hexdump, used in chunks 290a, 436c, 437, 603, 605c, 607, 608, 611b, and 612b.
Uses PEEK 117 and printf 601a.

17.5 Printing the Frame Table and Page Table

Here's a function for displaying the current page tables.

Since we want to output information from the free frame list (we will use test_frame$_{114a}$ to check frame states), we write a simple function that can print status information for a memory region (going from start$_{94}$ to end):

[612d] ⟨*function implementations* 100b⟩+≡ (44a) ◁612c 613b▷
```
void ksh_print_page_table_helper (unsigned sta, unsigned end, unsigned used) {
  if (used) { kputs ("Used: "); }
  else      { kputs ("Free: "); }
  printf ("%05x-%05x    %5d-%5d   (%5d frames)\n",
          sta, end, sta, end, end-sta+1);
};
```
Defines:
ksh_print_page_table_helper, used in chunk 613c.
Uses kputs 335b and printf 601a.

The following function prints the frame and page tables:

⟨*function prototypes* 45a⟩+≡ (44a) ◁612b [613a]
```
void ksh_print_page_table ();
```

⟨*function implementations* 100b⟩+≡ (44a) ◁612d [613b]
```
void ksh_print_page_table () {
  unsigned int cr3;
  ⟨print frame table 613c⟩
  kputch ('\n');
  ⟨print page table 614a⟩
  __asm__ __volatile__("mov %%cr3, %0": "=r"(cr3));
  printf ("cr3: %08x\n", cr3);
}
```
Defines:
ksh_print_page_table, used in chunks 608b and 613a.
Uses kputch 335b and printf 601a.

⟨*print frame table* 613c⟩≡ (613b) [613c]
```
kputs ("Current Frame Info:\n");
unsigned int frameno = 0;
unsigned int totalfree = NUMBER_OF_FRAMES;  // total number of free frames
unsigned int test = test_frame (frameno);   // check first frame

for (unsigned int i = 1;  i < NUMBER_OF_FRAMES;  i++) {
  if (test_frame (i) != test) {
    ksh_print_page_table_helper (frameno, i-1, test);
    if (test) totalfree -= (i-frameno);
    test = 1-test;
    frameno = i;
  };
};
ksh_print_page_table_helper (frameno, NUMBER_OF_FRAMES-1, test);
if (test) totalfree -= (NUMBER_OF_FRAMES-frameno);
printf ("Total free frames:    %6d\n", totalfree);
printf ("Value of free_frames: %6d\n", free_frames);
```
Uses free_frames 112b, kputs 335b, ksh_print_page_table_helper 612d, NUMBER_OF_FRAMES 112a, printf 601a, test_frame 114a, and totalfree.

The output of ⟨*print frame table* 613c⟩ looks like this:

```
Current Frame Info:
Used: 0x00000000-0x000003FF    0000000-0001023   (0001024 frames)
Free: 0x00000400-0x000007FE    0001024-0002046   (0001023 frames)
Used: 0x000007FF-0x000007FF    0002047-0002047   (0000001 frames)
Free: 0x00000800-0x00004000    0002048-0016384   (0014337 frames)
Total free frames:   0015359
```

The following code for the page table seems overly complicated because we want to print mappings of ranges and not each single mapping of a page to a frame in order to

save space in the output (and keep it readable). We use a variable `started` to memorize whether we're right now in a mapped region while skipping through the page tables.

[614a] ⟨*print page table* 614a⟩≡ (613b)

```
printf ("Current Paging Info: Address Space #%d\n", current_as);

boolean started=false;
int save_i=0; int save_f=0;
unsigned int start_i=0; unsigned int start_f=0;
for (unsigned int i = 0;  i < 1024*1024;  i++) {
  frameno = mmu_p (current_as, i);  // get frameno with respect to current AS
  if (frameno == -1) {
    if (started) {                              // frame NOT found
      ⟨print pages to frames block 614b⟩
      started = false;
    }
    continue;  // dont act on non-mapped pages
  } else {                                      // frame found
    if (!started) {
      start_i = i; start_f = frameno;
      save_i  = i; save_f  = frameno;
      started = true;
    } else {
      if (i-start_i != frameno-start_f) {
        // pages continue, but frames are elsewhere
        ⟨print pages to frames block 614b⟩
        start_i=i; start_f=frameno;
      };

      save_i = i; save_f = frameno;
    };
  };
};
if (started) {  ⟨print pages to frames block 614b⟩ }
```

Uses `current_as` 170b, `mmu_p` 171c, and `printf` 601a.

This is just the code for formatting the output:

[614b] ⟨*print pages to frames block* 614b⟩≡ (614a)

```
printf ("PTEs 0x%05x..0x%05x -> frames 0x%05x..0x%05x  (%5d pages)\n",
  start_i, save_i, start_f, save_f, save_i-start_i+1);
```

Uses `printf` 601a.

The output of ⟨*print page table* 614a⟩ will look like this:

```
Current Paging Info:
PTEs 0x00000000..0x000003FF -> frames 0x00000000..0x000003FF   (0001024 pages)
PTEs 0x000C0000..0x000C03FF -> frames 0x00000000..0x000003FF   (0001024 pages)
PTEs 0x000D0000..0x000D3FFF -> frames 0x00000000..0x00003FFF   (0016384 pages)
```

18 The ULIX Build Process

You have almost reached the end of the book—now we describe the whole process that is needed in order to turn a literate program (the `ulix-book.nw` file) into a booting operating system disk image and some other files needed for execution of the system.

18.1 Required Software

If you want to build Ulix yourself, you need several tools which might not be installed on your machine. Check that the following requirements are fulfilled:

- **Linux operating system**

  Any 32-bit version of Linux will work, provided that you can install the correct version of the C compiler (see next point). There was also one positive report of a developer using FreeBSD, and Ulix can also be compiled on Mac OS X, but that requires some more work (see Section 18.1.1). In principle a 64-bit Linux system should work as well, but that would require some extra work because in default 64-bit installations the compiler cannot create 32-bit binaries.

- **GNU C compiler, version 4.4**

  The Ulix sources can be compiled with older or newer versions of the GNU C compiler `gcc` (`https://gcc.gnu.org/`), but when we experimentally picked a different version than 4.4, the resulting kernel did not work. This is likely caused by different code optimization. We successfully used GCC 4.4.5 on a 32-bit version of Debian Linux 6.0.1 (Squeeze, `https://www.debian.org/releases/squeeze/`). If it turns out that your compiler version cannot compile Ulix and you do not have the option to install GCC 4.4, then you will need to download the development environment, see Section 18.2.

- **NASM assembler**

  You need the `nasm` assembler (`http://nasm.us/`). On two development machines (Debian Linux 6.0.1 and OS X 10.6.8) `nasm -v` displayed the following version strings:

  ```
  Debian: NASM version 2.08.01 compiled on Jun  2 2010
  OS X:   NASM version 0.98.40 (Apple Computer, Inc. build 11) compiled on May 18
          2009
  ```

  Both versions worked well, but others should, too, because the assembler must always produce the same object files from the code: Assembler code will not be optimized.

- **NoWEB**

  You have to install the `noweb` package (`http://www.cs.tufts.edu/~nr/noweb/`) which can extract the C and assembler source code files and the makefiles from the literate program `ulix-book.nw`. On a Debian Linux machine you can type `apt-get install noweb`.

- **LaTeX, the XeLaTeX variant**

  In order to reproduce a PDF version of this book, you will need the XeLaTeX variant of the LaTeX document preparation system (`http://www.xelatex.org/`, `http://www.latex-project.org/`). Depending on your Linux distribution, installing LaTeX might not lead to a full installation (that contains XeLaTeX). On Debian Linux `apt-get install texlive-xetex` should fetch and install the required packages. You will also need a `noweb` package for XeLaTeX that is available from the Ulix project website via

  ```
  wget http://ulixos.org/files/0.12/noweb.sty
  ```

  You can instead use the default `noweb.sty` file that might be installed on your machine, but then the layout will look a bit different.

- **mtools**

  Install the *mtools* package (`http://www.gnu.org/software/mtools/`) if it is not present yet. (You can check by typing `mtools` in the shell; if you get a "Command not found" error, you need to install it.) On Debian systems `apt-get install mtools` finds the right package.

- **qemu**

  You will also need the `qemu` PC emulator (`http://www.qemu.org/`). While Ulix might run on other Intel-x86-based hardware, we only tested it in the `qemu` and Bochs PC emulators, and of those two only `qemu` was able to boot and run it. We also successfully used the Q program on OS X (`http://www.kju-app.org/`) which is a GUI for `qemu`; the package contains a `qemu` version, so installing Q is enough for running Ulix.

18.1.1 Toolchain on Mac OS X

It is possible to compile Ulix on an Apple Mac, but the information in this section will not be fully applicable if you use a newer version of OS X. However, it might still be helpful for finding the right files for your setup. We used Mac OS X 10.6.8 and started with installing a GCC Cross Compiler as documented in `http://www.fanofblitzbasic.de/prettyos/PrettyOSMacOSX.pdf`). We downloaded the file `http://www.fanofblitzbasic.de/prettyos/`

i586-elf-binutils-gcc-macos.zip and unpacked it. (Note: When we attempted to re-download the file during the final preparation stage of this book, the website was offline. We could not find that PDF file or the cross compiler archive elsewhere, but on http://wiki.osdev.org/Talk:GCC_Cross-Compiler#On_Mac_OS_X_Lion the creation of a cross-compiler is discussed, so that site might help you. In the end you will need a gcc version called i586-elf-gcc that creates ELF-i386 binaries.)

We also had to install GMP and MPFR which could be automated using the port tool (https://www.macports.org/). (On newer OS X versions port is replaced by brew; http://brew.sh/.)

```
port install gmp
port install mpfr
```

Then we set some links:

```
ln -s /opt/local/var/macports/software/mpfr/3.0.0-p8_0/opt/local/lib/libmpfr
    .4.dylib /usr/local/lib/libmpfr.1.dylib
ln -s /opt/local/var/macports/software/gmp/5.0.1_0/opt/local/lib/libgmp.3.
    dylib /usr/local/lib/libgmp.3.dylib
```

The nasm assembler was installed by default; it is also available via the MacPorts package collection.

18.1.2 Other Useful Tools

You might find the following tools helpful though we have not used all of them for the development of Ulix.

- **All in one boot disk**, http://rescup.winbuilder.net/bootdisk/

  This is a FAT-formatted GRUB boot disk (with other tools on there, e. g., a free DOS clone and tools). It is useful because you can use the mtools utilities to copy a new Ulix kernel to the disk by typing

  ```
  mcopy -i bootdisk.img kernel.img ::kernel.img
  ```

  (the first : in :: is a "drive letter" used for talking to the disk image referenced by -i).

  We modified the boot disk so that it has only one menu entry to boot /ulix.bin, and we removed the contents of the TOOLS directory that provided DOS tools such as fdisk.exe or ntfsdos.exe.

- **mfstool**, http://mfstool.sourceforge.net/

  The mfstool can access Minix filesystem images. It works on Linux, Mac OS and other Unix versions. For example,

  ```
  mfstool dir minix1.img
  ```

  displays the contents of the root directory in the Minix filesystem image minix1.img. However, the version we tested had problems with writing files to an image. It was good for reading files or listing directories, though.

- **minixfs (MacFUSE)**, `http://osxbook.com/software/unixfs/`

MacFUSE, UnixFS

`minixfs` is a driver that is part of the MacFUSE-based UnixFS package and lets OS X users mount Minix-formatted volumes—but only in read-only mode. Accessing a mounted volume is simpler than using `mfstool`.

18.2 Downloading the Development Environment

VirtualBox

If you run into problems with your installed version of the development tools, you can either attempt to fix them or simply download a virtual machine image for the VirtualBox virtualization program (`https://www.virtualbox.org/`) that contains a Debian Linux 6.0.1 installation and the ULIX sources. It is distributed as an `ova` appliance file (Open Virtualization Format) that you can import in VirtualBox using the *File / Import Appliance* menu entry. Visit the `http://ulixos.org/files/0.12/ova/` directory and read the instructions in `readme.txt` which contain updated information about the installation process.

18.3 Bootstrapping: How to Start

Assuming that you have a development environment with all the needed tools installed and the ULIX noweb source code file `ulix-book.nw` in your home directory, you can start by extracting the needed files from the noweb source file.

Create a directory `ulix` somewhere in your home directory and change into it with `cd`. The directory must be empty. Move the literate program `ulix-book.nw` that you can download with

```
wget http://ulixos.org/files/0.12/ulix-book.nw
```

into that folder and execute the command

```
notangle ulix-book.nw | sh
```

That command will extract the following root chunk ⟨* 618⟩ of the document which contains a simple shell script that in turn creates some directories and makefiles. You will also need to download and decompress the disk images and some additional files that help with the PDF file generation. If you don't press [Ctrl-C], the script will do that for you automatically.

The Makefiles are intended to work on a Debian Linux 6.0.1 system that has all the required tools installed. If they do not work, you might want to inspect the `Makefile` files in `bin-build/`, `lib-build/` and `tex-build/` which are extracted from the ⟨*bin-build/Makefile* 620b⟩, ⟨*lib-build/Makefile* 622⟩ and ⟨*tex-build/Makefile* 623b⟩ code chunks (see below).

[618] ⟨* 618⟩≡

```
#!/bin/bash
echo This is the ULIX source code extractor
litprog=ulix-book.nw
files=$( ls -1 | wc -l )
```

```
if [ $files != 1 ]; then
  echo "~/ulix directory is not empty, it must contain only ulix-book.nw. Aborting."
  exit
fi

for dir in bin-build lib-build/tools lib-build/diskfiles/bin mountpoint tex-build
do
  mkdir -p ${dir}
done
for file in Makefile bin-build/Makefile lib-build/Makefile lib-build/process.ld \
lib-build/tools/Makefile lib-build/tools/process.ld tex-build/Makefile \
bin-build/assembler-parser.py tex-build/filter-uses.py module.nw lib-build/init.c
do
  notangle -R${file} -t8 ${litprog} > ${file}
done
chmod a+x bin-build/assembler-parser.py tex-build/filter-uses.py

webroot="http://ulixos.org/files/0.12"
echo "You can download the disk images if you don't have them yet:"
echo "cd to bin-build/ and type:"
echo "  wget ${webroot}/ulix-fd0.img.gz"
echo "  wget ${webroot}/ulix-fd1.img.gz"
echo "  wget ${webroot}/ulix-hda.img.gz"
echo "  wget ${webroot}/ulix-hdb.img.gz"
echo "Then uncompress them with"
echo "  gunzip *.gz"
echo "Similarly, change to tex-build/ and type:"
echo "  wget ${webroot}/noweb.sty.gz"
echo "  wget ${webroot}/grep-patterns.gz"
echo "and uncompress them as well."
echo
echo "This script will download all files for you if you don't press Ctrl-C"
echo -n "in the next eight seconds... "
for (( i=1; i<9; i++ )); do echo -n ${i}...; sleep 1; done
echo ""
echo "Downloading files"
cd bin-build
for image in fd0 fd1 hda hdb; do
  echo wget ${webroot}/ulix-${image}.img.gz
  wget ${webroot}/ulix-${image}.img.gz
  gunzip ulix-${image}.img.gz
done
cd ../tex-build
for file in noweb.sty grep-patterns; do
  echo wget ${webroot}/${file}.gz
  wget ${webroot}/${file}.gz
  gunzip ${file}.gz
done
echo 'Done. Type "make" to build the kernel, type "make run" to run it in qemu.'
```

18.3.1 Makefiles

We start with the makefiles which control the build processes for the kernel, the user mode library, the applications and the PDF document (this book).

The development root folder contains the following makefile that will make kernel, library and tools (when you execute `make`) and start ULIX in the `qemu` emulator when you type `make run`. With `make pdf` you can create the PDF file (if XeLaTeX is installed).

[620a] ⟨*Makefile* 620a⟩≡

```
all: tools bin

pdf: ulix-book.nw
        make -C tex-build

bin: ulix-book.nw
        make -C bin-build

run:
        make -C bin-build run

runs:
        make -C bin-build runs

tools: ulix-book.nw
        make -C lib-build

clean:
        make -C lib-build clean
        make -C bin-build clean
        make -C tex-build clean
```

The `bin-build/` directory is used for compiling the kernel source file `ulix.c`, assembling the Assembler source file `start.asm` and linking the generated object files to create the kernel binary `ulix.bin`.

[620b] ⟨*bin-build/Makefile* 620b⟩≡

```
OS=Linux
LD=ld
CC=/usr/bin/gcc-4.4
OBJDUMP=objdump
CFLAGS=-O0 -m32 -mstackrealign

HDA_IMG=ulix-hda.img
HDB_IMG=ulix-hdb.img
FD0_IMG=ulix-fd0.img
FD1_IMG=ulix-fd1.img

ASM=nasm
ASMFLAGS=-f elf
TEXSRC_FILE=../ulix-book.nw
TEXSRC_MODULE_FILE=../module.nw
EXTRACT_FILES=ulix.c start.asm ulix.ld

all: build

build:  extract parse asm compile linking objdump mtools

extract:
        notangle -L -Rulix.c $(TEXSRC_FILE) > ulix.c; true
        notangle -Rstart.asm $(TEXSRC_FILE) > start.asm
```

```
        notangle -Rulix.ld $(TEXSRC_FILE) > ulix.ld
        notangle -L -Rmodule.c $(TEXSRC_MODULE_FILE) > module.c
        notangle -L -Rmodule.h $(TEXSRC_MODULE_FILE) > module.h

parse:
        mv ulix.c ulix.c.pre
        ./assembler-parser.py ulix.c.pre ulix.c
        sed -ie "s/Mon Nov  2 17:33:51 CET 2015/`date`/" ulix.c

asm:
        mv module.c module.c.pre
        ./assembler-parser.py module.c.pre module.c
        $(ASM) $(ASMFLAGS) -o start.o start.asm

compile:
        $(CC) $(CFLAGS) -fno-stack-protector -std=c99 -g -nostdlib -nostdinc \
        -fno-builtin -I./include -c -o module.o module.c
        $(CC) $(CFLAGS) -fno-stack-protector -std=c99 -g -nostdlib -nostdinc \
        -fno-builtin -I./include -c -o ulix.o -aux-info ulix.aux ulix.c

linking:
        $(LD) $(LDFLAGS) -T ulix.ld -o ulix.bin *.o

mtools:
        mcopy -o -i $(FD0_IMG) ulix.bin ::

objdump:
        $(OBJDUMP) -M intel -D ulix.bin > ulix.dump
        cat ulix.dump | grep -e '^[^ ]* <' | sed -e 's/<//' -e 's/>://' > ulix.sym

clean:
        rm -f ./*.o ./*.c ./*.h ./*.pre ./ulix.bin ./ulix.aux ./ulix.ce
        rm -f ./ulix.dump* ./*asm ./*.objdump ./*sym

run:
        qemu -m 64 -rtc base=localtime -boot a -fda $(FD0_IMG) -fdb $(FD1_IMG) \
        -hda $(HDA_IMG) -hdb $(HDB_IMG) -d cpu_reset -s -serial mon:stdio | \
        tee ulix.output

nolog:
        qemu -m 64 -rtc base=localtime -boot a -fda $(FD0_IMG) -fdb $(FD1_IMG) \
        -hda $(HDA_IMG) -hdb $(HDB_IMG) -d cpu_reset
```

The `lib-build/` directory is used for compiling the user mode library (from its source files `ulixlib.c` and `ulixlib.h`) and the user mode applications in `lib-build/tools/`. The generated Ulix binaries will be placed in `lib-build/diskfiles/` and then copied to the hard disk image file `bin-build/ulix-hda.img`.

You might want to set up your Linux system so that you can run `sudo` without entering a password, otherwise making the files in this directory will ask for your password.

[622] ⟨*lib-build/Makefile* 622⟩≡

```
OS=Linux
LD=ld
CC=/usr/bin/gcc-4.4
OBJDUMP=objdump

NOWEBFILE=../ulix-book.nw
ROOTDISK=../bin-build/ulix-hda.img

CCOPTIONS=-nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer \
-fno-function-cse -nostartfiles -mtune=i386 -momit-leaf-frame-pointer -O0
CCASMOPTIONS=-fverbose-asm -masm=intel
LDOPTIONS=-static -s

all: build

build: extract compile image

extract:
        notangle -L -Rulixlib.c < $(NOWEBFILE) > ulixlib.c  ; true
        notangle -L -Rulixlib.h < $(NOWEBFILE) > ulixlib.h  ; true

compile:
        $(CC) $(CCOPTIONS) -g $(CCTESTOPTIONS) -c ulixlib.c
        $(CC) $(CCOPTIONS) $(CCTESTOPTIONS) -c init.c
        # link it with linker script "process.ld"
        $(LD) $(LDOPTIONS) -T process.ld -o init init.o ulixlib.o
        touch tools/*.c
        make -C tools

image:
        sudo mount -o loop $(ROOTDISK) ../mountpoint
        cp init ../mountpoint/
        sudo umount ../mountpoint

clean:
        rm -f ./*.o
```

We've already shown the ⟨*lib-build/tools/Makefile* 236⟩ code chunk when we introduced ELF binaries.

18.3.2 Linker Configuration Files

The two code chunks ⟨*lib-build/process.ld* 191b⟩ and ⟨*lib-build/tools/process.ld* 623a⟩ contain `process.ld` files which are used to configure the behavior of the GNU linker `ld`. One of those files belongs in the `lib-build/` folder and is only used for creating the flat binary file `init`, the other one belongs in `lib-build/tools/` and is used for linking all the regular programs which are ELF binaries. (The code chunk ⟨*lib-build/process.ld* 191b⟩ was already

shown in Chapter 6.3.)

⟨ *lib-build/tools/process.ld* 623a ⟩≡ [623a]

```
OUTPUT_FORMAT("elf32-i386")
ENTRY(main)
virt = 0x00000000;
SECTIONS {
  . = virt;

  .setup : AT(virt) {
    *(.setup)
  }

  .text : AT(code) {
    code = .;
    *(.text)
    *(.rodata*)
    . = ALIGN(4096);
  }

  .data : AT(data) {
    data = .;
    *(.data)
    . = ALIGN(4096);
  }

  .bss : AT(bss) {
    bss = .;
    *(COMMON*)
    *(.bss*)
    . = ALIGN(4096);
  }
  end = .;
}
```

`tex-build/` is the folder in which you can recreate the PDF file of this book.

⟨ *tex-build/Makefile* 623b ⟩≡ [623b]

```
TEX=xelatex -8bit -shell-escape

all:
        nodefs -auto cee tmp.nw | sort -u > noweb.defs
        grep -v -f grep-patterns noweb.defs > noweb.filtered.defs
        noweave -indexfrom noweb.filtered.defs -delay tmp.nw > tmp.tex.in
        ./filter-uses.py < tmp.tex.in > tmp.tex
        sed -ie "s/Mon Nov  2 17:33:51 CET 2015/`LANG=C date`/" tmp.tex
        sed -ie 's/≤/≤/g' tmp.tex
        sed -ie 's/≥/≥/g' tmp.tex
        $(TEX) tmp
        bibtex tmp
        makeindex tmp.idx
        noindex tmp
        $(TEX) tmp
        $(TEX) tmp
        mv tmp.pdf ../ulix-book.pdf

clean:
        rm -f ./tmp.*
```

(Note that the three `sed` commands are shown wrong in this code chunk, the first one replaces `SCRIPTBUILD` with today's date, the second and third ones replace `<=` and `>=` with ≤ and ≥ which you cannot see here because the transformation was also applied to those lines. Extracting the Makefile gives you a correct file.)

18.3.3 The Assembler Pre-Parser

The following Python program performs transformations of a simplified inline assembler syntax to the regular syntax (as expected by the GNU C compiler).

[624] ⟨*bin-build/assembler-parser.py* 624⟩≡

```
#!/usr/bin/python

"""
This Parser replaces code of the following form:
    asm {
      starta: mov eax, 0x1001   // comment
      mov ebx, 'A'              // more comment
      int 0x80
    }
with code that looks like this:
    asm ("\
      .intel_syntax noprefix; \
      starta: mov eax, 0x1001; \
      mov ebx, 'A'; \
      int 0x80; \
      .att_syntax; \
    ");
It also understands asm volatile. What it cannot cope with is variable / register
usage. Note that it does not change the number or position of code lines.
"""

from sys import argv, exit
if len(argv)<3:
  print ("Error: give input and output filenames")
  exit (1)
infilename = argv[1]
outfilename = argv[2]

global ReplaceMode
ReplaceMode = False

def count_leading_blanks (line):
  counter = 0
  while line and (line[0] == " "):
    counter+=1
    line = line[1:]
  return counter

def remove_trailing_blanks (line):
  if (line == ""): return line
  while (line != "") and (line[-1] == " "):
    line = line[:-1]
  return line
```

```
def transform (line):
  global ReplaceMode
  if ReplaceMode:
    if "}" in line:
      # reached the end; skip this line
      blanks = count_leading_blanks (line)
      line = (blanks+2) * " " + '.att_syntax; ");'
      ReplaceMode = False
      return line
    else:
      # do something to the line
      if '//' in line:
        # remove comment
        pos = line.find ("//")
        line = line[:pos]
        line = remove_trailing_blanks (line)
      line = line + "; \\"
      return line

def process (line):
  global ReplaceMode
  line = line[:-1]
  if ReplaceMode:
    # we're already in ReplaceMode, working on assembler
    line = transform (line)
  else:
    # we're in normal C mode, check for asm {
    if ("asm volatile{" in line) or ("asm volatile {" in line):
      blanks = count_leading_blanks (line)
      line = blanks * " " + 'asm volatile (".intel_syntax noprefix; \\'
      ReplaceMode = True
    elif ("asm{" in line) or ("asm {" in line):
      blanks = count_leading_blanks (line)
      line = blanks * " " + 'asm (".intel_syntax noprefix; \\'
      ReplaceMode = True
  return line

infile  = file (infilename,  "r")
outfile = file (outfilename, "w");

EndOfLoop = False

for line in infile:
  line = process (line)
  outfile.write (line+"\n")

infile.close()
outfile.close()
```

18.3.4 Creating Modules with `module.nw`

The file `module.nw` is intended for students who want to work on a ULIX-related project but create their own literate program document. From the file two code chunks are extracted, resulting in `bin-build/module.c` and `bin-build/module.h`. The C file will also be compiled, and the resulting object file `module.o` is linked with the other kernel object files.

[626a] ⟨*module.nw* 626a⟩≡

```
<<module.c>>=
#include "module.h"
void initialize_module () {}
@
<<module.h>>=
void initialize_module ();
extern int printf(const char *format, ...);
@
```

The `module.c` function must provide an `initialize_module`$_{45a}$ function which will be called during kernel initialization.

18.3.5 Pretty Printing for the Book

The `filter-uses.py` script enables the limited pretty-printing that we have used in this book. It replaces brackets (`()[]{}`), exclamation marks, `#include` and `#define` statements and C and Assembler comments with highlighted versions.

[626b] ⟨*tex-build/filter-uses.py* 626b⟩≡

```
#!/usr/bin/env python

import fileinput
from re import sub

deletemode = False
codemode = False
asmmode = False
breakmode = False
nosyntaxmode = False

for line in fileinput.input():
  line = line[:-1]   # remove \n
  if line == "%nouse":
    deletemode = True
    # print "% DELETE MODE ON"

  if "%BEGIN ASM CHUNK" in line:
    asmmode = True

  if "%END ASM CHUNK" in line:
    asmmode = False
```

```
    if "%BEGIN NOSYNTAX" in line:
      nosyntaxmode = True

    if "%END NOSYNTAX" in line:
      nosyntaxmode = False

    if "%BREAK BEFORE DEFINES" in line:
      breakmode = True

    if "nwendcode" in line:
      codemode = False

    if breakmode and "nwindexdefn" in line:
      line = sub (r"\\nwindexdefn", r"\\pagebreak\\nwindexdefn", line)
      breakmode = False

    if not nosyntaxmode:

      if codemode == True:
        if asmmode == False:
          # highlight C comments
          line = sub (r"(//.*)$", r"{\\green\\emph{\1}}", line)
          line = sub (r"/\*", r"{\\green\\emph{/*", line)
          line = sub (r"\*/", r"*/}}", line)
        else:
          # highlight ASM comments
          line = sub (r"(;.*)$", r"{\\green\\emph{\1}}", line)
        line = sub (r"\(", r"{\\lightblue{}(}", line)
        line = sub (r"\)", r"{\\lightblue{})}", line)
        line = sub (r"\[", r"{\\red{}[}", line)
        line = sub (r"\]", r"{\\red{}]}", line)
        line = sub (r"!", r"{\\red{}!}", line)
        line = sub (r"\\\{", r"{\\orange{}\\{}", line)
        line = sub (r"\\\}", r"{\\orange{}\\}}", line)
        for keyword in (r"#define", r"#include"):
          line = sub (keyword, r"\\emph{"+keyword+r"}", line)

    if "nwenddeflinemarkup" in line:
      codemode = True

    if deletemode and "nwidentuses" in line:
      line = sub (r"nwidentuses.*nwindexuse", r"nwindexuse", line)
      deletemode = False

    print line
```

Uses print 600.

For those readers who might want to modify the build process, we give some more details in the following sections.

18.4 Directory Hierarchy and Makefiles

In the `ulix/` directory you find several files and directories after the initial build process:

- `ulix.pdf` is a PDF version of this book. It will only be generated if you have XeLaTeX installed (which is not a requirement for simply testing Ulix).
- `bin-build/` contains the kernel source files `ulix.c` and `start.asm` which are compiled and linked into the kernel binary with `gcc`, `nasm` and `ld`.
- `lib-build/` holds the user mode library source files `ulixlib.c` and `ulixlib.h` as well as a sub-directory for the user mode applications.
- `lib-build/tools/` is the place where the application source files reside. All of those are C programs.
- `lib-build/tools/diskfiles/` collects files which shall be placed in the Ulix root disk image file (that is located in `bin-build/minixdata.img`).

  The image `bin-build/ulix-fd0.img` only contains the boot loader GRUB and the Ulix kernel (and no other data).
- `tex-build/` is used for generating the PDF documentation. It also contains some extra LaTeX files which are not part of a standard LaTeX distribution, such as the noweb package (`noweb.sty`).
- The Ulix source root directory (where you placed `ulix-book.nw`) and all `*-build` directories contain makefiles which can be executed by simply changing to the directory and calling `make`. They do also provide some options for partial builds or cleanup operations (see next section).

18.5 Making and Booting

In this section we provide some further details about the build process and the ways to execute Ulix.

18.5.1 User Mode Applications

In order to compile a user mode program, place its source code file in the `lib-build/tools` folder. We assume that the source file is called `myprogram.c`. If you simply want to check whether it compiles, type `make myprogram` (while in the `lib-build/tools` directory), that will generate a `myprogram` binary. For installing it in the disk image, change the directory to `lib-build` and type `make`. You can then test the program by entering the Ulix development root directory and typing `make run`.

Note that each user mode program must start with the line

```
#include "../ulixlib.h"
```

which includes the ULIX library headers.

If you also made changes to the library (by modifying the `ulix-book.nw` file) you need to call `make` twice in `lib-build`.

18.5.2 The Disk Images

ULIX uses four disk image files:

- `ulix-fd0.img`: This file is FAT-formatted and contains the boot loader GRUB and the ULIX kernel file `ulix.bin`. It is updated whenever you rebuild the kernel.
- `ulix-fd1.img`: This is a Minix-formatted floppy image that is mounted on the `/mnt` directory when ULIX boots. It does not contain any relevant files, so you can reformat it with `mkfs.minix -2 ulix-fd1.img`.
- `ulix-hda.img`: This disk image is used as the first hard disk, even though it has the layout of a 1.44 MB floppy disk. It is the root disk, i. e., it is mounted to `/`. You can reformat it, but then you need to reinstall the `init` program and the other applications (via `make` in the `lib-build` directory).
- `ulix-hdb.img`: The 100 MByte hard disk image is mounted on the `/tmp` directory and holds the 64 MByte swap file `/tmp/swap` that ULIX uses for paging out page frames. It is required, but you can also add other files to it.

18.5.3 Alternative Boot Options

If you look at the `Makefile` in `bin-build`, you will notice that there is another make target besides `run` which also starts the PC emulator: By typing `make nolog` you can start `qemu` without the option that gathers the serial line output, displays it in the terminal and also writes it to the log file `ulix.output`.

For experiments, you can add further make targets which use modified options.

18.5.4 Informative Files

When you compile the kernel, the files `ulix.sym` and `ulix.dump` are created. The first one contains a listing of symbols with their addresses, and the second one contains the generated assembler code, also with addresses. When you modify ULIX and the system hangs because of invalid memory access or some other fault, the fault handler will display the faulting address. You can then use these files to check where the error occurred.

18.5.5 Manually Inspecting the C Files

If you want to have a look at the C files which are extracted from `ulix-book.nw` because you prefer to see functions in a complete version (instead of the chunk-based presentation in this book), you will notice that the files are garbled with hundreds of source line modifiers of the form

```
#line 19651 "../ulix-book.nw"
```

They allow the C compiler to show the line number in `ulix-book.nw` (instead of the line number in the current C file) when printing error messages. In order to get rid, `untangle` the C file without the `-L` option, i. e., run

```
notangle -Rulix.c ../ulix-book.nw > ulix.c
```

instead of

```
notangle -L -Rulix.c ../ulix-book.nw > ulix.c
```

18.5.6 Other Emulators

Early versions of ULIX were also compatible with the Bochs PC emulator which has a comfortable graphical debugger (if you install the right version of Bochs). However, the current version does not boot on the Bochs machine.

You could also try to use ULIX with virtualization software (such as VirtualBox or VMware Workstation), but we have not tested that.

18.6 Online Resources: the `ulixos.org` Website

We already mentioned the website as the download resource for all the files we have discussed above. You may find updated information in a `readme.txt` file in the `http://ulixos.org/files/0.12/` directory. Also check the start page, `http://ulixos.org/`, for information about new ULIX releases.

18.7 Tools

The last section is not strictly related to the build process. Here we merely present the `bindump` tool that was mentioned in the Minix implementation chapter (Chapter 12.5).

18.7.1 bindump

Similar to `hexdump`, here's an implementation of `bindump`. The tool has an option `-r` which reverses the output order of 8-bit-strings (bytes; e. g. `10100000` instead of `00000101`). `bindump` accepts no filename, you must use it as a filter (e. g. `bindump -r < image.img`).

The tool was helpful during the early implementation phase of the Minix filesystem since it allowed to print the inode and zone bitmaps in a readable form. It is not automatically extracted from the book sources, but you can copy and paste its code from `ulix-book.nw` if you want to use it, too.

⟨ *bindump source code* 631 ⟩≡ [631]

```
// bindump.c

// use as filter:
//   bindump < image.img      (for regular output, lower bits on the right)
//   bindump -r < image.img   (for reversed output, lower bits on the left)
//   cat image.img | bindump

#include <stdio.h>

int rev;  // reverse output?

void binwrite (byte c) {
  unsigned int v = (unsigned int)c;
  int i;
  for (i = 7;  i > -1;  i--) {
    if (rev == 0)  printf ("%d", (v>>i)%2);       // regular output
    else           printf ("%d", (v>>(7-i))%2);   // reversed output
  };
  printf (" ");
}

void bindump (byte *bytes, int offset, int num) {
  int i; byte c;
  printf ("%08x  ", offset);
  for (i = 0;  i < num;  i++)  binwrite (bytes[i]);
  printf (" ");
  for (i = 0;  i < num;  i++) {
    c = bytes[i];
    if ((c > 31) && (c < 128))  printf ("%c",c);
    else                        printf (".");
  };
  printf ("\n");
};

int main (int argc, char *argv[]) {
  byte buf[8]; int count; int pos = 0;
  rev = 0;                                              // reverse?
  // Test if option -r is set:
  if ((argc > 1) && (strequal (argv[1], "-r")))  rev = 1;  // reverse!
  do {
    count = read (0, &buf, 8);
    if (count > 0)  bindump ((byte*)&buf, pos, count);
    pos += 8;
  } while (count > 0);
  return 0;
}
```

19
Where to Go Now?

You've done it: you finished the book (unless you skipped to this chapter early), and that means you've seen the whole source code of the Ulix operating system. Now you know how a Unix-like system works internally, and that tells you a lot about how most other systems function. Of course, Ulix differs a lot from Linux or Windows, but many of the differences are about hardware support (systems intended for practical purposes need lots of drivers for all sorts of devices), performance, stability, failure handling and of course the list of provided features.

However, there is one important topic that you have not seen in this book at all: Operating systems for multi-core (or multi-processor) architectures are more complex since they have to handle a lot more parallelism; after all, on such systems several cores or CPUs execute instructions at the same time, and it may happen that two or more processes simultaneously make a system call or run a faulting instruction. Similarly the scheduler may be required to pick a new process on several cores at the same time. This has many consequences for the operating system code which needs to be protected better against the typical problems that parallelism causes.

So if you want to understand why Linux or Windows is able to use your quad-core machine so efficiently, you need to go on reading.

Herlihy and Shavit's book "The Art of Multiprocessor Programming" [HS12] discusses synchronization problems on multiprocessor platforms in detail, and Schimmel's "UNIX Systems for Modern Architectures" [Sch94] also deals with this topic, but focuses on Unix.

Looking at the Linux sources (or those of one of the free BSD versions) could be a next step, though that would require lots of time. If you're more interested in Windows, Microsoft used to provide a stripped-down but very well-documented version of the Windows 2003 Server kernel, called the "Windows Research Kernel" (WRK) which was available to instructors via Microsoft's Academic Alliance website and was later moved

to `http://www.microsoft.com/resources/sharedsource/Licensing/researchkernel.mspx`, however that section of the website has been moved once more and we are currently unaware of any WRK download resources—perhaps someone in your faculty still has a copy of it.

If you want to test your understanding of the ULIX code (and have already worked on the exercises) you might want to continue with a bigger project. Here are some suggestions for improvements of the ULIX kernel:

- Enable partition support: Currently ULIX treats hard disks like floppies, i. e., unpartitioned. Understanding either the classical MBR or the new GUID partition tables (GPT) and adding code to ULIX so that partitions can be accessed via `/dev/hda1`, `/dev/hda2` etc. is not too complicated but will still require some time to get it right.
- Add network support: ULIX would get closer to being a proper Unix system if it could access the network. The task would be twofold: a) Write a hardware driver for a standard network adapter (e. g. the one that is provided as a virtual network card by `qemu`), and b) Write or port a TCP/IP stack to ULIX.
- Port some interesting user mode applications to ULIX. For example, there is a simple implementation of a `vi` clone which can do very limited editing of text files that are no longer than 23 lines (because it does not support scrolling). You could take this code and build it into a proper editor.

If you're able to read the German language, you can also have a look at the publication list on the ULIX website: There are links to several Bachelor's theses which describe the implementations of various ULIX components.

As a closing remark, we'd like to rephrase what we wrote in the foreword: We hope that you've found this book interesting and helpful for gaining some understanding of operating system concepts. We believe that our approach of presenting the whole source code of a simplified Unix system in the literate programming style is unique and worthwhile. If you agree (or disagree), then please drop us a note and tell us how the ULIX book worked for you.

A Introduction to C

In this chapter we give you a very short introduction to programming with C—and we expect that you have some previous knowledge of one of the object-oriented successors of C, such as C++ or C#.

A.1 No Classes, no Objects

The most important difference between C and the other languages is that C is no object-oriented language. It knows neither classes nor objects. This means that you have to change your way of conceptually thinking of code: Where you have been used to define a *class* and implement *methods* that can manipulate objects of that class, this is not possible with C. Instead you need to write *functions*, and these functions are independent of specific "data objects". If you want to store data, you declare *variables*, and you must provide a function with that variable (while calling it).

class, method
function

Imagine a string class that has a reverse function. If you have an object `s`, you might put the statement `s.reverse();` in your code and expect that this changes the order of the characters in the string `s`. In C, you could implement a function `reverse()` which has the following prototype:

⟨*example function* 635⟩≡ [635]
```
void reverse (char *arg);
```

You would then call the function by using the statement `reverse (s);`. (We will explain why the argument is written `char *arg` in the next section.)

C is a *typed language*, and it does not allow you to *overload* its functions. So, by continuing the above example, if you also had a list class in an object-oriented language, you might find that that list class also provides a `reverse()` method, using the same name.

overloading

Then, if you had a string `s` and a list `l`, you could use the syntactically identical method invocations `s.reverse();` and `l.reverse();` to have both objects reversed—even though the technical details of the methods' implementations might differ a lot.

It is not possible to have two C functions of the same name, so in this situation the best solution would be to write two functions with appropriate names, such as `reverse_string()` and `reverse_list()`.

A.2 Data Types, Arrays and Pointers

structure

Since classes are not available, C needs to provide an alternative for declaring user-defined complex data types (which have several simpler elements, similar to member fields of an object). The C keyword `struct` is used for defining a *structure*. For example, the following definition declares a complex number that consists of two real numbers:

[636a] ⟨*example structure 1* 636a⟩≡
```
struct complex {
  float re;
  float im;
};
```

After you have defined this structure, you can declare variables of that new type, for example by writing `struct complex c;`. The `struct` keyword is required, though it is possible to get rid of it: instead of the above code, you can also write

[636b] ⟨*example structure 2* 636b⟩≡
```
typedef struct {
  float re;
  float im;
} complex;
```

typedef

This creates the same kind of structure, but via the `typedef` keyword you assign the name `complex` to that structure. Then, you can declare variables by simply stating `complex c;`.

dot notation

In both cases, you can access the fields of the variable `c` with a syntax that is similar to the one that C++ and Java use for member access: the *dot notation*. Typing `c.re` will give you the real component of the number, and `c.im` is the imaginary component.

array

Often several instances of a variable are needed, and for this purpose C provides *arrays*. If you want `cnumbers[]` to be an array that can hold 20 complex numbers, you would write

[636c] ⟨*example array 1* 636c⟩≡
```
struct complex cnumbers[20];
```

or

[636d] ⟨*example array 2* 636d⟩≡
```
complex cnumbers[20];
```

(depending on whether you chose the first or second method of defining the new type). You can then access the 20 individual entries of the array by putting the index in square brackets. Note that C starts counting at 0, thus valid index numbers for the example array range from 0 to 19: `cnumbers[0]` is the first number, and `cnumbers[19]` is the last. Getting

the real and imaginary parts is done via the dot notation again, so `cnumbers[0].re` is the real part of the first complex number.

If you want to add all the complex numbers in the `cnumbers[]` array and store the sum in the `sum` variable, you could write the following loop:

⟨*summing up the complex numbers* 637a⟩≡ [637a]
```
complex sum = { 0, 0 };        // set sum.re = sum.im = 0
int i;
for (i = 0;  i < 20;  i++) {
  sum.re += cnumbers[i].re;
  sum.im += cnumbers[i].im;
}
```

(`x += y;` is a short form for `x = x + y;`, and the first line shows how you can initialize a structure without using the field names.)

It is impossible to directly add two complex numbers, because you cannot create your own version of the + or += operator—the following loop cannot be expressed in C:

⟨*impossible way to build the sum* 637b⟩≡ [637b]
```
complex sum = { 0, 0 };        // set sum.re = sum.im = 0
int i;
for (i = 0;  i < 20;  i++) {
  sum += cnumbers[i];          // !  cannot do that
}
```

You could, however, write a function `addto()` that takes two complex numbers and adds the second one to the first one:

⟨*function for adding* 637c⟩≡ [637c]
```
void addto (complex *c1, complex *c2) {
  *c1.re += *c2.re;
  *c1.im += *c2.im;
}
```

and then rewrite the add loop as

⟨*summing up the complex numbers with* `addto` 637d⟩≡ [637d]
```
complex sum = { 0, 0 };        // set sum.re = sum.im = 0
int i;
for (i = 0;  i < 20;  i++) {
  addto (&sum, &cnumbers[i]);
}
```

What's happening here or, more specifically, what are the * and & operators doing?

address-of operator

Let's start with the & operator which is called the *address-of operator*: It "gets" the memory address of the variable, i. e., a numerical value that says where in memory the variable is stored. In the above loop this happens with both `sum` and `cnumbers[i]`. The two addresses are then provided to the `addto` function.

When you look at the functions's prototype, you see that `addto()` does not expect two complex *values* but something else, namely, two addresses of complex variables. These are just 32-bit integers (if you're working on a 32-bit machine), but the function knows to

expect complex numbers (and not something else) at those addresses. Inside the function you cannot directly access the numbers by writing `c1` or `c2`, because those two arguments are not complex numbers, but addresses. In order to access the contents, you have to *dereference* the address, and that is what the `*` operator is for. It is called the *dereference operator*.

dereference operator

`c1` and `c2` are what C calls *pointers*: internally they store the addresses of two complex variables, and practically that turns them into pointers to those variables. Prefixing them with the `*` operator turns them into the (wanted) complex variables. Thus, `*c1` and `*c2` are of type `complex`. You already know how to access the real and imaginary parts of a complex variable, so it should be obvious why `*c1.re` and `*c2.re` deliver the intended values.

pointer

Why can we use `addto()` to actually change the value of the first operand? That's because the memory addresses of the real variables are provided. Using pointers as function arguments is a form of *call-by-reference* (as opposed to *call-by-value* where a function gets to work with a copy of the values).

call-by-reference

Since expressions of the form `*var.element` are often needed, there is a different way to write them which is better readable and also makes it clearer that we deal with a pointer: that alternative is `var->element`. The `->` combination looks like an arrow (it points). The prettier way to write the `addto` function is this:

-> notation

[638a] ⟨*function for adding, with pointer syntax* 638a⟩≡

```
void addto (complex *c1, complex *c2) {
  c1->re += c2->re;
  c1->im += c2->im;
}
```

A.3 Strings? There is no String

One of the properties of C is that there is no built-in *string type*. But obviously, strings are much needed. What C does have, is an elementary *character type* (`char`) that simply is a signed byte, storing values between -128 and 127; the ASCII table holds only 128 values, so the positive numbers of this range are sufficient to store any ASCII character. You can also use the `unsigned char` type which ranges from 0 to 255.

no string type

char

unsigned char

The simplest way to introduce a string in your program is using a *character array*:

[638b] ⟨*string definition as* `char` *array* 638b⟩≡

```
char my_string[128];
```

(or with `unsigned char` instead of `char`).

This defines a character array of length 128. All functions in the standard C libraries use a *null character* (ASCII value 0) to mark the end of such a string, so the above definition actually provides a string that can hold no more than 127 characters: The last position cannot be used other than to store a 0 value (to indicate that this string consists of 127 characters). This is a popular source of programming errors, where a developer thinks "my password string can consist of 16 characters" and then proceeds to write `char password[16]` which lets him store only 15 characters.

\0

An alternative way to think of a string is as just a consecutive chunk of memory (where a character is stored at each of its addresses). Then it is enough to simply store the starting address of the string. A pointer can do just that, and in case of strings, a pointer to `char` makes the most sense. So typing

⟨*string definition as* `char*` *pointer* 639a⟩≡ [639a]
```
char *my_string;
```

does also declare a string. However, there's an important difference: The first example (with the array) reserves a defined amount of memory where the string can be stored—the new version does not do so. It simply says: `my_string` points to a chunk of memory which is interpreted as a string. That address may or may not be initialized to 0. In any case, just declaring the string does nothing that helps us store a string.

How can we use such a pointer? First of all, if you already have a string (say, one declared as an array), you can assign its address to the pointer. Consider the following code:

⟨*char pointer assignment* 639b⟩≡ 639c▷ [639b]
```
char array_string[12] = "Hello World";   // declares and assigns
char *pointer_string;
pointer_string = array_string;
```

The first line shows how strings can be filled with content at the time of declaration. The *string literal* `"Hello World"` is just a shorthand for `{ 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\0' }` which is the standard way to initialize arrays during declaration. When you type a string in `"..."` signs, the string-terminating `'\0'` character is automatically added. The second line declares a pointer to `char`. The interesting instruction is in the last line: it assigns `array_string` to `pointer_string`. When you use just the name of an array, that is automatically interpreted as the (starting) address of the array. Thus, `array_string` is a pointer to `char`, and this address is assigned to `pointer_string` by the instruction. Afterwards, both `pointer_string` and `array_string` reference *the same* memory address. You can check that by changing a single character in one string and then printing the other one:

string literal

⟨*char pointer assignment* 639b⟩+≡ ◁639b [639c]
```
array_string[5] = '_';                          // replace blank with '_'
printf ("array_string:   %s\n", array_string);  // print the first string
printf ("pointer_string: %s\n", pointer_string); // print the second string
```

The output will be identical (`Hello_World` with an underscore). The assignment does not work vice versa, i. e., you cannot have an instruction `array_string = pointer_string` in the same situation: the compiler knows that the `array_string` variable points to a fixed memory address, whereas `pointer_string` is truly a variable. What is changeable in an array variable, are the elements, not the position and size of the array.

If you have no array-type strings, then how to get a free memory location? You can use the `malloc()` function (memory allocation) to request space. If you want to store a string with 100 characters (plus the terminating zero), you could type

⟨*char pointer memory allocation* 639d⟩≡ [639d]
```
pointer_string = (char*) malloc (101);
```

The `malloc(101);` call will reserve memory and return its start address which is then stored in the variable. The new memory may or may not be initialized. The extra `(char*)` part before `malloc` performs a type conversion: the return value of `malloc` is `(void*)`, a pointer to data of unknown type. Placing the desired type (`char*`) in round brackets in front of it changes the type to `char*`. This kind of conversion is called a *type cast* or simply a *cast*. You can always convert pointers to something into pointers to something else, though some conversions make no sense. A conversion in the C program does not translate into the execution of any code (in the generated assembler language): a pointer is an address, and all addresses have the same type when looked at at the machine level; on a 32-bit machine every address is a 32-bit integer. But the C compiler keeps track of how you define your pointers and issues warnings when you assign a pointer to a pointer of a different type without adding the explicit cast.

type cast

A.4 String Operations

Once you have the memory that is necessary to store a string (either via a direct array declaration of via `malloc`), you want to work with the string. As C has no string type and no methods, you cannot write `s1 = s1+s2;` or `s1.append(s2);` in order to append a string `s2` to another string `s1`. Instead you need to use a function that does just that. The same holds for copying: You cannot assign a string to another one (and expect that to result in a copy), so even `s1 = s2;` is illegal for character arrays. It is legal when `s1` is a pointer, but does not duplicate the string: It copies the address. The standard functions for copying a string and appending a string (to another one) are

[640a] ⟨*standard string functions* 640a⟩≡ 641a ▷

```
char *strcpy (char *dest, char *src);
char *strcat (char *dest, char *src);
```

Instead of `s1 = s2;` you would write `strcpy (s1, s2);` and instead of `s1 = s1 + s2;` you need `strcat (s1, s2);`. However, these functions may not always have the intended effect, and that is the source of many security holes in applications.

You can find our implementation of `strcpy`$_{594b}$ in the book: ULIX provides its own version since it cannot use the functions which are available on the development (Linux) system.

The functions `strcpy` and `strcat` should be used very carefully because they can write beyond the end of the memory area that was reserved for the string. Typing

[640b] ⟨*string buffer overflow* 640b⟩≡

```
char s1[10];
char s2[25] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
strcpy (s1, s2);   // copy long string s2 into s1
```

is perfectly legal C code, but `strcpy()` will not stop when it reaches the end of `s1`. Instead it will just fill the following bytes as well, and that may result in a number of things, for example `s2` being destroyed (if it is located directly behind `s1`) or an application crashing if the addresses behind `s1` are not available.

That is why it is best to replace all uses of `strcpy` and `strcat` with their safe variants which are called `strncpy` and `strncat`: they have a third argument via which you can limit the number of bytes that are actually written:

⟨*standard string functions* 640a⟩+≡ ◁640a 641d▷ [641a]
```
char *strncpy (char *dest, char *src, size_t n);
char *strncat (char *dest, char *src, size_t n);
```

Using them (and using the right value for `n`) the following code causes no problems (though it still cannot achieve what can't be done, i. e., copying a large string into a small one:

⟨*no string buffer overflow* 641b⟩≡ [641b]
```
char s1[10];
char s2[25] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
strncpy (s1, s2, 10);   // copy up to 10 characters of s2 into s1
```

This code will not write beyond the end of `s1`, but it will also leave `s1` in a bad state: the string will not be null-terminated, but instead contain the first ten characters of `s1` without termination. The only way to avoid this (if the size limit cannot be helped) is this:

⟨*truly creating a string* 641c⟩≡ [641c]
```
char s1[10];
char s2[25] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
strncpy (s1, s2, 9);   // copy 9 characters...
s1[9] = '\0';          // and manually terminate the string
```

Of course there are other ways that let you avoid such situations. For example, when you want to copy a string you can first use the function

⟨*standard string functions* 640a⟩+≡ ◁641a 641e▷ [641d]
```
size_t strlen (const char *s);
```

to discover the length of a string, then reserve the appropriate amount of memory for a copy (using `malloc()`) and then create the copy.

For comparing two strings you need yet another function since a simple comparison like `(s1==s2)` will only compare the strings' starting addresses.

⟨*standard string functions* 640a⟩+≡ ◁641d [641e]
```
int strcmp  (const char *s1, const char *s2);
int strncmp (const char *s1, const char *s2, size_t n);
```

compare `s1` and `s2` character-by-character and return 0 if the strings are equal. If `s1` $<$ `s2` (lexicographically), they return a negative number, and if `s1` $>$ `s2`, they return a positive number. So test for `(strcmp(s1,s2)==0)` to check whether two strings are identical. For both functions the comparison ends when the null-termination occurs in one of the strings, the safe "n" version also stops after `n` characters have been read.

A.5 Pointer Arithmetic

Consider the following, naive implementation of a `strcpy()` function:

[642a] ⟨*simple* `strcpy` *implementation* 642a⟩≡

```
char *strcpy (char *dest, char *src) {
  int i;
  for (i = 0; i < strlen(src); i++) {
    dest[i] = src[i];     // copy i'th character
  dest[i] = '\0';         // terminate dest
  return dest;
}
```

It works because `dest` and `src` can be treated like arrays (though the parameters are declared as pointers). However it is neither necessary to use a counter variable nor the `strlen()` function. Instead the following version is preferred in the realm of C programmers:

[642b] ⟨*typical* `strcpy` *implementation* 642b⟩≡

```
char *strcpy (char *dest, char *src) {
  char *tmp = dest;
  while ( (*dest++ = *src++) != '\0' )
    ;
  return tmp;
}
```

For most people who are new to C, this looks like garbage though it's perfectly correct C (and does what you want). Let's explain the code in detail.

- First of all `*dest++ = *src++` is C shorthand for the three commands `*dest = *src; dest++; src++;`, and the value of the whole expression is the value of `*dest` or `*src` *before* the two increment commands. (`x++` is another shorthand, meaning `x = x+1`.)
- Adding 1 to a `char` pointer increases the memory address by 1 (as would be expected). At the beginning `dest` points to the first character of the destination string, `dest[0]`. After the increment it points to what was previously `dest[1]`, the second character.
- The `while` loop continues the byte-wise copying until a null character is found (and copied). The destination string is now complete (null-terminated), but `dest` no longer points to the beginning of the target string since mutiple `++` operations have increased its value. That is why we kept the initial value in `tmp`.
- That saved value is then returned. (The `strcpy()` function must always return a pointer to the new string.)

Now why is this section titled *Pointer Arithmetic*? If we were to work with integer arrays (instead of character arrays), we might also be interested in a copy function, let's call it `intarrcpy()` (integer array copy). Its implementation looks almost identical to the second `strcpy()` version, with all occurrences of "`char`" replaced with "`int`":

⟨*copying a 0-terminated integer array* 643a⟩≡ [643a]
```
int *intarrcpy (int *dest, int *src) {
  int *tmp = dest;
  while ( (*dest++ = *src++) != 0 )
    ;
  return tmp;
}
```

But let's look at the ++ operator again: If it also increased the values of `dest` and `src` by 1, the function would have to fail since a 32-bit integer needs four bytes for storage, not just one. So what we want is to add 4 to the addresses in every step of the loop. The surprising news is: This is exactly what ++ does, and that is why it is called pointer arithmetic. The ++ operator (as well as -- and the regular addition and subtraction) consider the size of the base type that the pointer points to. For an `int` pointer `ptr`, the `ptr++` commands actually increases `ptr`'s address by `sizeof(int)` (which is 4).

Pointer arithmetic is not restricted to base types. If you define a `struct something` structure that contains a lot of data, totaling in 2604 bytes of data for each such variable, and declare a pointer of that type (via `struct something *ptr;`), then each `ptr++;` command will modify the address by adding 2604. This makes it easy to walk through array-like structures even when they were never declared as arrays.

There is one problem with pointer arithmetic that sometimes leads to wrong code. We already mentioned that you can cast pointer types to different pointer types. Look at the following example to see what can go wrong:

⟨*cast and pointer arithmetic gone bad* 643b⟩≡ [643b]
```
char s[10] = "ABCDEFGHI";
char *charptr;
int  *intptr;

charptr = s;              // points to the 'A' in s
intptr = (int*) charptr;  // same address
intptr++;                 // pointer arithmetic!
charptr = (char*) intptr; // cast/copy it back
```

If you expect `charptr` to point to the `'B'` in `s`, then you've made the mistake that we want to explain. It actually points to the `'E'` because the pointer arithmetic was performed on an `int` pointer, so addresses are always modified in multiples of 4.

A.6 C Pre-Processor

The C compiler runs a pre-processor before actually compiling the code. That pre-processor looks for commands that begin with # and acts on them. These can be used for including other source files, for conditional compilation and for macro definitions. Since we use some of these features, we give a short explanation of each of these three possibilities but only discuss the details which are relevant for reading and/or modifying the Ulix code.

- Including files: Using the command

  ```
  #include "path/to/file.h"
  ```

  you can include other files in the source file. Typically those are header files (ending in `.h`), but you can include any file you want. If the path name starts with a slash, it is treated as an absolute path, otherwise as a relative one. So you can include the file `xyz.h` from the upper directory by writing

  ```
  #include "../xyz.h"
  ```

  —regardless of where the files are placed absolutely.

- Macro definition: The `#define` command declares a macro. In its simplest version that leads to a simple search-and-replace. For example,

  ```
  #define BLOCK_SIZE 1024
  ```

  lets the compiler search the source file for the string `BLOCK_SIZE` and replace every occurrence with `1024`.

macros with parameters

A more advanced version of macros uses parameters, so macros provide an alternative method to writing (simple) functions. A typical example is finding the smaller of two values:

```
#define MIN(x,y) ((x<y) ? x : y)
```

The expression `(x<y) ? x : y` evaluates to `x` if `x<y` is true and to `y` otherwise. Using `MIN(x,y)` in the code makes it more readable. However, this must be used with care, as the following example shows which attempts to increase two variables while picking their minimum: `MIN(v1++,v2++)` does not do what is expected because the macro is expanded to `((v1++ < v2++) ? v1++ : v2++)`. Here, `v1` and `v2` are compared. Let's assume that `v1` is the smaller one. Then both variables are incremented (as expected). However, in the next step `v1` is increased again: The condition is true, so `v1++` is evaluated. The total result is that the value of `MIN(v1++,v2++)` is the old value of `v1` $+1$, and `v1` gets incremented twice (while `v2` is incremented only once).

side effect

When you work with macros, use them only with constant arguments or arguments which have no *side effects*. (In the example, `MIN(f(x1),f(x2))` with some function `f()` would also call `f(x1)` twice, not once, if `(f(x1)<f(x2))` evaluates as true.)

- Conditional Compilation:You can create simple if-then-else constructions which can remove parts of the code before compilation. This is often used for inserting debug code during the development which is then removed for the final release of the software. As an example consider the following code block:

  ```
  #define DEBUG
  int somefunc (int x) {
    int res;
    #ifdef DEBUG
      printf ("DEBUG: somefunc() called with argument %d\n", x);
      res = x / 3;
      printf ("DEBUG: somefunc() going to return %d\n", res);
  ```

```
  #else
    res = x / 3;
  #endif
  return res;
}
```

When you compile this code it will contain the debug output because the `DEBUG` macro is defined. (Note that it has no specific value, it is just defined, so `#ifdef` will evaluate it as true.) Simply remove that `#define DEBUG` line and recompile to get the version which contains only the "else" case: the lines between `#else` and `#endif`.

Of course, the same effect could be achieved without a macro (by using a `DEBUG` variable and a regular `if` expression), but then all of the code would be compiled. With the macro, the pre-processor removes the unwanted lines from the source code before the compiler runs.

The pre-processor does not touch occurrences of a macro name inside a string literal: the arguments of `printf()` in the example contain `DEBUG` in the string argument, and they remain intact.

You can check the effect of pre-processor commands by calling the `gcc` compiler with the `-E` option: `gcc -E file.c -o file.i` creates a new file `file.i` where all pre-processor commands have been executed. That file is now free of pre-processor commands. An interesting alternative to the `-E` option is the `-save-temps` option which performs all compilation steps, but keeps the intermediate files which are normally deleted. When you run the command

```
gcc -save-temps testprog.c -o testprog
```

you can find four new files: `testprog.i` is the pre-processor-modified version of the source file, `testprog.s` is the compiled assembler version (with readable assembler source code), `testprog.o` is the assembled object file, and finally `testprog` is the binary executable file which contains library code or links to dynamically loaded libraries. Figure A.1 shows an example of the created files.

A.7 Further Reading

Since pointers seem to be a crucial topic for most students who are new to C programming, we suggest reading a whole book about pointers: "Pointers on C" [Ree98] by Kenneth Reek is an excellent read and comes with many exercises, both practical and theoretical. The author uses diagrams to show what points where and what content is stored in which memory locations.

For those who truly want to delve into the language, the description of the C compiler in David Hanson and Christopher Fraser's LCC book [HF95] offers deep insight. If you understand the workings of a C compiler, you've also mastered the language. Plus: the book is another literate programming example which in itself makes it worth reading—provided that you like this style.

```
$ cat testprog.c
#define TEST 1024

int main () {
  printf ("%d",
          TEST);
}
```

```
$ cat testprog.i
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "test.c"


int main () {
  printf ("%d",
          1024);
}
```

```
$ cat testprog.s
        .cstring
LC0:
        .ascii "%d\0"
        .text
.globl _main
_main:
LFB2:
        pushq   %rbp
LCFI0:
        movq    %rsp, %rbp
LCFI1:
        movl    $1024, %esi
        leaq    LC0(%rip), %rdi
        movl    $0, %eax
        call    _printf
        leave
        ret
...
```

Figure A.1: When called with the `-save-temps` option, `gcc` keeps the intermediate files.

Another interesting book about C is Peter van der Linden's "Expert C Programming: Deep C Secrets" [vdL94]. The author looks at some obscure details and explains unexpected phenomena with direct quotations from the ANSI C standard definition [Ame89]. There are many comparisons of language features in C and C++ which are especially helpful if you're used to writing C++ code. The publisher's website has a 60-page sample.

Many good C books are rather old as the language was created in the eighties. Some of them are still available in print. If you want to have some fun with C programs, look at the International Obfuscated C Code Contest website, `http://www.ioccc.org/`.

B

Introduction to Intel x86 Assembler

Most of the ULIX code is written in C, but some small parts had to be done in Assembler since C cannot directly access the CPU's registers or execute specific CPU instructions such as the ones that enable or disable interrupts. In many cases it is sufficient to use `gcc`'s inline assembler feature that lets you drop a few lines of assembler in the middle of a C function (see Section B.4), but we also needed a separate assembler source file for the early steps in the system initialization. In this chapter we give a very short introduction to some of the features available on Intel i386 and higher CPUs and the syntax of the commands.

When you use assembly language, you are somewhat limited in the way you can structure the code. For example, where C has several types of loops (`for`, `while`, `do`) and nestable if-then-else expressions, assembler does not. Instead, you can make comparisons and jump elsewhere in the code (depending on the result of that comparison). That is closer to early Basic dialects where branching worked via "IF *condition* GOTO *line number*" statements. However, there's no need to learn the assembler ways of expressing `for` and `while` loops, because we don't want to write *all* our code in assembler.

B.1 CPU Registers

Just like a C program that accesses a variable (which is stored somewhere in RAM), assembler code often works with memory, too. For example, the CPU provides instructions that inspect the contents of two memory cells, add or substract them and store the result in the first of the two cells. That is basically what the C compiler creates when it compiles

a C command like `var1 += var2;` or `var1 -= var2;`. But memory access is expensive: it takes some time to translate a virtual address into a physical address and fetch the memory contents via the memory bus. It is too slow to perform all operations that way. Every CPU has a set of faster memory cells: the CPU-internal *registers*. They are even faster than the first level cache which is embedded in the same chip as the CPU's core: they provide instant access.

registers

Intel's CPUs (like many other processors) have a set of *general purpose registers* which can be used to hold arbitrary data and perform calculations on them, and then there's also a set of *special purpose registers* which is what we're really interested in, because we need to read or write some of those registers to influence and control paging, interrupt handling and other critical tasks.

general registers

There are eight *general registers* [Int86, p. 29]: *EAX*, *EBX*, *ECX*, *EDX*, *EBP*, *ESP*, *ESI*, and *EDI*. You can use them to hold values and perform calculations and comparisons. Each of these registers is 32 bits wide, and you can also access the lower 16 bits of them by using a different name (*AX*, *BX*, *CX*, *DX*, *BP*, *SP*, *SI*, and *DI*; all without the leading "E"). The first four registers can be separated even further into higher and lower halves which are only eight bits wide (and hold a byte): *AH*, *BH*, *CH* and *DH* are the higher halves, *AL*, *BL*, *CL* and *DL* are the lower halves (see Figure B.1) which are sometimes needed for I/O when data is read from or written to a *port* that grants the CPU access to the internal 8-bit register of some chip, e. g. a disk controller.

I/O port

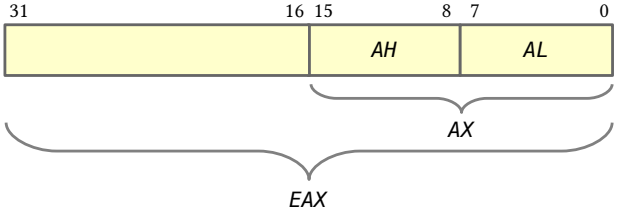


Figure B.1: The lower half of *EAX* is *AX* which in turn is split into *AH* (high) and *AL* (low).

For example, in order to add the contents of *EBX* to *EAX* you could use the assembler instruction `add eax, ebx`. The `mov` instruction copies (not: moves) a value from one register to another, so `mov eax, ebx` performs an `eax := ebx` action. You can also access memory locations with these commands: `mov eax, [ebx]` will load a 32-bit value from the memory addresses pointed to by *EBX* and copy it to *EAX*—the C equivalent would be `eax := *ebx` (with `ebx` interpreted as an `int*` pointer).

The registers *ESP* (extended stack pointer) and *EBP* (extended base pointer) are used for working with stacks. *ESP* points to the top of the stack and is changed whenever a subroutine is called or it exits and when a value is pushed onto or popped from the stack. The base pointer helps with staying oriented on the stack: While a subroutine executes, it may often modify the stack (thus changing *ESP*). But the stack also holds the arguments which were provided to the subroutine as well as its local variables, and by letting *EBP* point to the address between arguments and local variables, it is possible to access them

stack

without a need to consider changes to *ESP*. For example, *EBP*+8, *EBP*+12, *EBP*+16 store the first, second and third argument, and *EBP*−4, *EBP*−8 and *EBP*−12 store the first, second and third local variable (assuming that all those values are 32-bit integers). Between those two areas there is still some room: address *EBP*+4 holds the return address, and at the address that *EBP* points directly to you find the "old value" of *EBP* that was used for the previous subroutine call (in a setting where subroutines call other subroutines).

B.2 A Few Standard Commands

We will not give a detailed introduction to the instructions that the Intel x86 CPU provides, but here is a short overview of some of the most important ones. For the Intel processor platform, two "dialects" exist, the Intel and the AT&T one. The GNU C compiler supports both but defaults to the AT&T variant. We have decided to use the Intel syntax, because it is closer to C's syntax: For example, you can load the *EAX* register with the value 0 via the command `mov eax, 0` (in Intel syntax). So the target of the `mov` command comes first which resembles the C command `eax = 0`. In AT&T syntax, the operands are reversed, with the target coming last and extra syntactical elements being needed (`mov $0, %eax`). In the following examples we will show the Intel syntax on the left hand side, and the AT&T syntax on the right.

B.2.1 Moving Data Around

The simplest way of filling a register is loading an *immediate* value. The `mov` instruction does that. In Intel syntax, the register name comes first, followed by the number expressed as in the following examples. In the AT&T syntax the order of arguments is reversed, and immediate values are prefixed with a dollar sign, whereas register names have a percent sign as prefix. Also, AT&T syntax appends a size identifier to the `mov` instruction, so instead of `mov` it is called `movb` (byte, 8 bits), `movw` (word, 16 bits) or `movl` (long, 32 bits). immediate

If you want to copy the contents of a memory location, you can load the address in one of the registers and tell `mov` to look at the memory cell(s) that it points to. The last two examples in the following table show how this is done, once without an offset and once with an offset.

Intel Syntax	AT&T Syntax	Description
`mov eax, 0xABCD`	`movl $0xABCD, %eax`	direct load, hexadecimal
`mov ebx, 1`	`movl $1, %ebx`	direct load, decimal
`mov eax, [ebx]`	`movl (%ebx), %eax`	copy memory at *EBX* to *EAX*
`mov eax, [ebx+0xF0]`	`movl 0xF0(%ebx), %eax`	with offset 0xF0

In those last two lines, if *EBX* holds the value `0xABCD0000`, then the first line will read the 32-bit integer stored at address `0xABCD0000`, wheras the second one reads address `0xABCD00F0`. In both cases the found value is written to the *EAX* register.

The original Intel syntax for hexadecimal numbers is `0ABCDh` with a `h` suffix (and a required `0` prefix if the number starts with a letter digit), but `nasm` supports both variants in

Intel mode, so we have chosen to use the `0xABCD` notation that is also C's way of expressing hexadecimal numbers.

B.2.2 Different Integer Sizes

In the register overview you have already seen that some registers can be accessed in ways which only use the lowest eight or 16 bits. For using these smaller versions, use the alternative names (e. g., *ax* for the 16-bit version and *al* for the 8-bit version). In the AT&T version you need to use a suffix again for expressing that you want to move a byte, word or long.

Intel Syntax	AT&T Syntax	Description
`mov al, bl`	`movb %bl, %al`	move a byte
`mov ax, bx`	`movw %bx, %ax`	move a word
`mov eax, ebx`	`movl %ebx, %eax`	move a long (32 bits)
`mov al, byte ptr [ebx]`	`movb (%ebx), %al`	move byte from memory
`mov ax, word ptr [ebx]`	`movw (%ebx), %ax`	move word from memory
`mov eax, dword ptr [ebx]`	`movl (%ebx), %eax`	move long from memory

Making the value size explicit makes even more sense when you access memory: Copying a byte is not the same as copying a long integer. In the Intel syntax explicit `byte ptr`, `word ptr` and `long ptr` keywords are used to state that a byte, word or long integer shall be read from memory, as shown in the last three lines. The equivalent AT&T commands don't need this since the `mov` suffix already makes the length explicit.

B.2.3 Arithmetic Operations

When talking about arithmetic operations, we only consider integer operations. The Intel CPUs also provides floating point operations, but we do not need them for Ulix.

For adding and subtracting you can use the `add` and `sub` instructions which take two arguments and add the source to the target; multiplication and integer division are handled by `mul` and `div`, but they don't take two arguments but only one and use the *EAX* register as *accumulator* (i. e., *EAX* is implicitly both one of the source operands and the target):

accumulator

Intel Syntax	AT&T Syntax	Description
`add eax, ebx`	`addl %ebx, eax`	add *EBX* to *EAX*, `eax += ebx`
`add eax, [ebx]`	`addl (%ebx), eax`	add (long) memory contents at *EBX* to *EAX*
`sub eax, ebx`	`subl %ebx, eax`	subtract *EBX* from *EAX*, `eax -= ebx`
`sub eax, [ebx]`	`subl (%ebx), eax`	subtract (long) memory contents at *EBX* from *EAX*
`mul ebx`	`mull %ebx`	multiply *EAX* with *EBX*, result in *EAX*, `eax *= ebx`
`div ebx`	`divl %ebx`	divide *EAX* by *EBX*, result in *EAX*, `eax /= ebx`

B.2.4 Jumps, Calls, Comparisons and Conditional Jumps

Sometimes you want to jump to a specific program address in order to continue execution elsewhere. For those cases the `jmp` instruction can be used. When creating an assembler source file (for use with `nasm`) you will normally assign a label to an instruction that you want to jump to and then use it in the `jmp` instruction, like this:

```
infinite_loop: mov  al,  byte ptr [eax]
               call printchar
               add  eax, 1
               jmp  infinite_loop
```

This example shows a further instruction for jumping to a new address: `call` also jumps to the supplied address, but before that it pushes the address of the following instruction (in the example: of `add eax, 1`) onto the stack. The assembler code at `printchar` can then execute the `ret` instruction which will pop that address from the stack and continue execution inside the above loop. To summarize the difference: When you `jmp`, there's no easy way to get back; when you `call`, you can return with `ret`.

Simply jumping (or calling) unconditionally does not allow for any case distinctions: Code that only uses `jmp` and `call` will always execute the same commands in the same sequence. For a distinction of cases we need comparison operations and based on the result we want to decide whether we jump elsewhere or not.

The `cmp` instruction takes two arguments and compares them. They are either identical or one value is bigger than the other. The `j*` instructions in the following table jump if a certain condition (such as: first value is smaller than the second one) is met.

Intel Syntax	**AT&T Syntax**	**Description**
`cmp eax, ebx`	`cmpl %ebx, eax`	compare *EAX* and *EBX*, then:
`jl label`	`jl label`	jump to `label` if *EAX* $<$ *EBX*
`jle label`	`jle label`	jump to `label` if *EAX* $\leq$ *EBX*
`je label`	`je label`	jump to `label` if *EAX* $=$ *EBX*
`jge label`	`jge label`	jump to `label` if *EAX* $\geq$ *EBX*
`jg label`	`jg label`	jump to `label` if *EAX* $>$ *EBX*
`jne label`	`jne label`	jump to `label` if *EAX* $\neq$ *EBX*
`sub eax, ebx`	`subl %ebx, eax`	subtract *EBX* from *EAX*, then:
`jz label`	`jz label`	jump to `label` if result of last arith. operation $= 0$
`jnz label`	`jnz label`	jump to `label` if result of last arith. operation $\neq 0$

The last three commands show that there are also conditional jumps that depend on the last arithmetic operation (instead of the last comparision). `jz` jumps if the *zero flag* is set which is the case if the last arithmetic operation resulted in a zero value. Similarly you can jump if the *overflow flag* is set which happens when the last operation caused an overflow, e. g., after adding 100 to `0xFFFFFFF0`.

zero flag

overflow flag

B.2.5 Pushing and Popping With the Stack

We already mentioned the stack which is used by the `call` instruction for storing the return address. Using `push` and `pop` you can also push data on the stack and pop it back. If you want to call an assembler function with arguments, first `push` the values and then `call` the function. The function can then either `pop` all values from the stack into registers (though it has to remember the return address and restore the stack later) or it can access the stack contents directly via `[ebp+8]`, `[ebp+12]` etc. if it saves the original *ESP* in the base pointer *EBP*. `push` and `pop` also accept memory locations, so you can push the value stored at the memory address that *EAX* points to by executing `push [EAX]`.

In AT&T syntax, `push` and `pop` need a suffix to indicate how large the data are which must be pushed or popped, so you get `pushw`, `pushl`, `popw` and `popl`. (You cannot push or pop a single byte, see [Hyd10, p. 137].) Some assemblers support a `pushb` or `push byte` instruction, but that will actually turn a byte into a word (by filling it with zero bits) and then push that.

B.3 Special Commands

There are three kinds of special commands that we'll introduce in this section: You can define constants (which are similar to C-`defined` macro constants), you can use macros (similar to C's macros that look like functions), and you can store data bytes for creating data structures.

All of these require using the `nasm` assembler. If you want to work with another assembler, it is likely that the same features are available, but they might use a different syntax.

B.3.1 Data Storage (`db`, `dw`, `dd`)

Assembler code contains instructions and data. Since you cannot define data types (like in C), your only option is to *know* what a data structure should look like and then directly encode data in the binary (and later refer to the address where the data are located).

There are three commands which can store data:

- `db` is used to store individual bytes (or sequences of bytes). For example, `dd 0x32, 0x38, 0x3b` will store the three bytes `0x32`, `0x38` and `0x3b` (in this order). If you place a label before that command you can later reference the address where those three bytes can be found. Instead of hexadecimal (or regular) numbers, you can only provide a single character or a string: `db 'xyz'` will store the three bytes whose character representations are x, y and z. Note that such a string will not be null-terminated.
- `dw` does the same as `db`, but stores a double byte (a word), and
- `dd` stores a double word (or quad byte, consisting of four bytes). You need to use `dd` to store 32-bit addresses.

`dw` and `dd` get the order of the bytes right so that they are stored in memory according to the *endianness* of your platform, for example, `dd 0x12345678` will write bytes `0x78`, `0x56`, `0x34` and `0x12` because Intel x86 machines are of the *little-endian* type where the smaller bit blocks come first.

endianness
little-endian

There are also commands for storing floating-point numbers (`dq`, `dt`), but we do not need them for Ulix which uses only integers.

B.3.2 Constants (`equ`)

With `equ` statements you can define identifiers which you can use in later code lines instead of the constants or expressions that you provided in the identifier definition. The syntax is always of the form

```
IDENTIFIER  equ  EXPRESSION
```

For example, the Ulix assembler file `start.asm` contains the following lines:

```
MB_HEADER_MAGIC   equ 0x1BADB002
MB_HEADER_FLAGS   equ 11b
MB_CHECKSUM       equ - (MB_HEADER_MAGIC + MB_HEADER_FLAGS)
```

They define three local identifiers `MB_HEADER_MAGIC`, `MB_HEADER_FLAGS` and `MB_CHECKSUM` that are used to create the multiboot header:

```
; GRUB Multiboot header, boot signature
dd MB_HEADER_MAGIC   ; 00..03: magic string
dd MB_HEADER_FLAGS   ; 04..07: flags
dd MB_CHECKSUM       ; 08..11: checksum
```

B.3.3 Macros (`%macro`)

If your code contains repetitive sequences that you would turn into a function (when programming in C), you can use a *macro* to keep the level of repetitions down. We show you the macro that we used in `start.asm` to write the assembler functions $\texttt{irq0}_{144}$, $\texttt{irq1}_{144}$, ...

The macro definition looked like this:

```
%macro irq_macro 1
       push byte 0          ; error code (none)
       push byte %1         ; interrupt number
       jmp irq_common_stub  ; rest is identical for all handlers
%endmacro
```

and we called the macro this way:

```
...
irq12: irq_macro 44
irq13: irq_macro 45
irq14: irq_macro 46
...
```

which the assembler expands to the following lines:

```
  irq12: push byte 0          ; error code (none)
         push byte 44         ; interrupt number
         jmp irq_common_stub  ; rest is identical for all handlers

  irq13: push byte 0          ; error code (none)
         push byte 45         ; interrupt number
         jmp irq_common_stub  ; rest is identical for all handlers

  irq14: push byte 0          ; error code (none)
         push byte 46         ; interrupt number
         jmp irq_common_stub  ; rest is identical for all handlers
%endmacro
```

In the macro's definition, the parameter `1` in the first line states that the macro can be used with one argument. That argument can be referred to via `%1`. So, while expanding the macro, every occurrence of `%1` is replaced with the argument. In the above examples we called the macro with arguments `44`, `45` and `46`.

If you need more than one argument (say: three), you set a different number, e. g. with `%macro name 3`. Then you access those arguments via `%1`, `%2` and `%3`.

B.4 `gcc` Inline Assembler

The GNU C compiler `gcc` lets developers write inline assembler statements in the middle of C code. That is very helpful if only a few lines of assembler code are required—storing them in a separate assembler source file and making sure that both the C and the assembler code can see the variables which are needed in both places is laborious, and it also costs (a little) CPU time because an external assembler routine must be called via the function call mechanism whereas inline assembler code is just inserted between the translations of the preceding and successive C code.

You can find the best example for a quick line of assembler code in the book: Whenever we need to disable or enable the interrupts, we do this via the ⟨ *disable interrupts* 47a⟩ and ⟨ *ensable interrupts* ⟩ code chunks which use the inline assembler instructions

```
asm ("cli");    // disable interrupts (clear interrupt flag)
```

or

```
asm ("sti");    // enable interrupts (set interrupt flag)
```

to execute the `cli` and `sti` instructions.

However, this is the simple case. It is more typical that values (which are stored in C variables) have to be used as a parameter of the assembler instruction. In a pure assembler file you can write

```
mov eax, some_label
```

to load the address of `some_label` in the *EAX* register, but you cannot similarly write an inline assembler statement like

```
asm ("mov eax, &some_C_variable")
```

to do the same for the address of `some_C_variable`. Instead, values or addresses must be passed by preparing registers before the assembler instructions are executed (and fetching possible result values via registers as well). `gcc` defines a special syntax to provide *input operands* and *output operands*; the general instruction format is this:

input/output operands

```
asm ("assembler instructions",
     :    // optional output operands
     :    // optional input  operands
     :    // optional "clobber" list
    )
```

Operands are always written in the `"reg" (variable)` form; if there is more than one register that we want to use, we use several such expressions and put commas between them. The following registers can be used:

Register	Reading	Writing
EAX	`"a"`	`"=a"`
EBX	`"b"`	`"=b"`
ECX	`"c"`	`"=c"`
EDX	`"d"`	`"=d"`
ESI	`"S"`	`"=S"`
EDI	`"D"`	`"=D"`
any	`"r"`	`"=r"`

We use that syntax in the `syscall*` functions of the user mode library: as an example, here is the code for $\texttt{syscall3}_{\texttt{203c}}$ which takes three parameters, stores them in *EAX*, *EBX* and *ECX*, raises the interrupt `0x80` and returns a result from *EAX*:

```
inline int syscall3 (int eax, int ebx, int ecx) {
  int result;
  asm ( "int $0x80"       : "=a" (result)     : "a" (eax), "b" (ebx), "c" (ecx) );
  //    |- instruction -|   |- output regs -|   |- input registers           -|
  return result;
}
```

The instruction explicitly copies the variables `eax`, `ebx` and `ecx` into the corresponding registers and then, after `int 0x80`, explicitly copies the contents of *EAX* into the `result` variable.

`"r"` (or `"=r"` for output) can be used for an arbitrary register, i. e., the compiler will choose a register as it sees fit. But then we don't know which register will hold the value. We can reference the variables by observing the order in which they appear in the output and input operand lists, and then address them as `%0`, `%1`, ... in the assembler code.

For example, in order to add two variables `var1` and `var2` and store the sum in `sum`, we could write

```
asm ("movl %1, %%eax \n"
     "addl %2, %%eax \n"
     "movl %%eax, %0"
     : "=r" (sum)               // output
     : "r" (var1), "r" (var2)   // input
     : "%eax"                   // "clobber" list
    );
```

We don't know what registers will hold the values of `var1` and `var2`, but we simply move or add them to *EAX* and finally write the sum (which is in *EAX* after the `addl` instruction) back to register 0. That is then copied to `sum`.

clobber list

We do not want the compiler to use *EAX* for one of the variable values, and that it where the *clobber list* comes in: With it we can tell the compiler which registers it must not use, so in the case of the above addition we put *EAX* on that list. Note that the syntax for the register name is different in the instructions (`%%eax`) and in the clobber list (`%eax`)! In the clobber list you could also write `eax` instead of `%eax`, but not `%%eax`.

B.5 Further Reading

A freely downloadable introductory text is Paul Carter's "PC Assembly Language Book" [Car06]. It is available in English and a few other languages via the author's website, `http://www.drpaulcarter.com/pcasm/`.

If you're more interested in concepts of assembler programming than in the actual syntax, "The Art of Assembly Language" [Hyd10] is a good alternative introduction: The author, Randall Hyde, has created his own dialect of Intel 32-bit assembler called HLA (High Level Assembler) which looks more natural since it has commands for conditional loops and standard input/output. HLA programs have more similarity with C programs than normal assembler code, but writing programs in HLA still teaches all the basic principles of assembler.

"Linux Assembly Language Programming" [Nev00] by Bob Neveln focuses on Linux; for example there is a description of the ELF binary format used by Linux (and Ulix).

For `gcc` inline assembler the "GCC Inline Assembly How-to" [S03] is available online at `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

C

Other Educational Operating Systems

As mentioned in the introduction chapter, we have assembled a list of other educational operating systems. We already discussed Minix and Xinu (see page 38) and suggested having a look at these two systems. But there are many more:

- One of the oldest instructional texts in this area is the Lions' commentary [Lio96], originally written by John Lions in 1976 but only published 20 years later. Its roughly 100 pages of code documentation are intended to be read side-by-side with a specially enumerated printout of the source code of Unix Version 6. Each section of the commentary explains a specific code section that is identified by its first line number.
- Xv6 is a simple Unix-like teaching OS [CKM12] originally developed in the summer of 2006 for MIT's OS course, and its documentation mimics the Lions' commentary in that the source code is available in a document with full line numbering (throughout all source files) and a descriptive document refers to those line numbers.

  Our Ulix implementation has borrowed some code from Xv6, for example for dealing with the hard disk controller and with serial ports.
- Topsy (Teachable Operating System) was developed at ETH Zurich [FCZP95], the original version runs on the MIPS architecture. Later it was ported to Intel i386 [Ruf98] and to the Pentium 4 [Ryf07] by students of the same university.
- Thix is a Unix-like operating system developed by Tudor Hulubei and documented in a technical report [Hul95]. While the author's goal was not to create an educational resource but "to learn about operating systems design and architecture, kernel algorithms, resource allocation, process scheduling, memory management policies, etc.",

the resulting code and report make an interesting read.

The Ulix implementation uses parts of the floppy driver code from Thix.

- Nachos (Not another completely heuristic operating system) [CPA93, And30] was originally developed in Berkeley and is currently supported at University of Washington. It was written in C++ and the sources were well-commented, though classically (i. e., directly in the source files). The system has to be run on a specific MIPS hardware emulator (also provided by the authors). Development of the original code was stopped, but there is a Java version [HC30] now used in Berkeley. The concept of the authors is to provide a system with only the most basic properties which is then extended by students during a course. Another successor to Nachos is Pintos.
- The goal of the Pintos [Pfa10, Pin09] developers was to replace Nachos. It differs from it in several aspects: it runs on 32-bit Intel x86 hardware (the authors suggest to execute it in the Bochs [LDA+14] or `qemu` [B+14] PC emulators), and it uses the C programming language instead of C++. Its feature set is similar to the one of Ulix. Pintos is also similar to Unix systems, but does not provide a `fork` function. It is currently used for teaching at Stanford and other universities.
- OS/161 is a kernel that was written in C for classes at Harvard University. It runs on an emulator for a machine called System/161 and is available for download at `http://www.eecs.harvard.edu/~syrah/os161/`. The authors have described their experiences with using OS/161 in class in a short conference paper [HLS02].
- iPoƨix (spelled iPosix) is a small Unix-like kernel for 32-bit Intel machines that was written in C++ by two students of Oldenburg University [MT09]. Later a practical course with exercises based on iPoƨix was designed by another student [Phl10].
- L4 is a family of micro-kernels that is sometimes used for educational purposes, for example at Technical University of Dresden (in the "Building Microkernel-Based Operating Systems" course). Reference manuals for the x86 [Lie96] and MIPS architectures [EHL97] describe the system.
- FreeDOS is a clone of Microsoft's MS-DOS, and its author has documented the development in a book [Vil96]. When he created FreeDOS, sources of MS-DOS were not available, so at that time studying FreeDOS was an alternative to reverse engineering the MS-DOS binaries. (In 2014 Microsoft published the source code of MS-DOS [Lev14, Shu14].)
- MicroC-OS II by Jean J. Labrosse is a real-time kernel that is described in a freely available book [Lab02]. The author states that the intended audience of the book includes "students interested in real-time operating systems". There is also a newer version (MicroC-OS III) that runs on several platforms.
- OOStuBS and MPStuBS are object-oriented operating systems for the Intel x86 platform which are used for classes at Friedrich Alexander University Erlangen-Nuremberg [Loh13]. MPStuBS is a version of OOStuBS that supports multiprocessor systems. Students taking the Operating Systems course can decide which system they want to work with throughout the semester.

The following entries do not refer to educational operating systems, but to books which describe "real world" systems. However, they do it so thoroughly that these texts can also be used as learning materials.

- Marshall Kirk McKusick and George V. Neville-Neil give a very detailed description of the FreeBSD kernel in their book "The Design and Implementation of the FreeBSD Operating System" [MNN05]. There is a lot of code mixed with explanations and figures, but the code is only pseudo code that leaves out the details.
- The original Unix system is described in Bach's book "The Design of the Unix Operating System" [Bac86]. It is based on Unix System V, Release 2 (from 1984) and describes the internal data structures and algorithms in a similar way as the FreeBSD book does: Real code is replaced with more accessible descriptions of what needs to be done.
- Lixiang et al. have written "The Art of Linux Kernel Design" [LWD+14]. Even though it was published in 2014, it is based on Linux version 0.11 which was released in December 1991. The old code is not as complex as that of current versions, and the authors use it for in-depth descriptions of the internal data structures and algorithms.

Chunk Index

Identifier Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Bibliography

[AC00] Werner Almesberger and John R. Coffman. LILO – Generic boot loader for Linux Version 21.5 – Technical overview, 2000. http://lilo.alioth.debian.org/olddoc/pdf/tech_21-5.pdf.

[ADAD14] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces.* Arpaci-Dusseau Books, 0.80 edition, May 2014.

[Ame89] American National Standards Institute. *Programming language, C: ANSI X3.159-1989.* Number 160 in FIPS Publication. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1989.

[And30] Tom Anderson. Nachos Code and Documentation. http://www.cs.washington.edu/homes/tom/nachos/, accessed 2014/10/30.

[B+14] Fabrice Bellard et al. QEMU: machine emulator. http://www.qemu.org/, 2014.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986. ISBN: 0-13-201799-7.

[Ber13] Liviu Beraru. Implementation eines Dateisystems und einer RAM-Disk für das Betriebssystem ULIX. Bachelor's thesis, Georg-Simon-Ohm-Hochschule Nürnberg, 2013. ISBN: 3-639-62622-2. http://ulixos.org/publications/ulix-beraru-fs-ramdisk.pdf.

[Bro09] Andries Brouwer. Keyboard scancodes, 2009. http://www.win.tue.nl/~aeb/linux/kbd/scancodes.html, accessed 2014/07/17.

[But97] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley Professional, 1997. ISBN: 0201633922.

[Car06] Paul A. Carter. *PC Assembly Language.* July 26, 2006. http://www.drpaulcarter.com/pcasm/.

[CF88] Douglas E. Comer and Timothy Fossum. *Operating System Design: The XINU Approach, Volume 1, PC Edition (Operating System Design Vol. 1).* Prentice Hall, 1988. ISBN: 0136381804.

[CKM12] Russ Cox, Frans Kaashoek, and Robert Morris. xv6 – a simple, Unix-like teaching operating system, 2012. Rev. 7, http://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf, accessed 2014/10/28.

[CM89] Douglas Comer and Steven Munson. *Operating System Design: The Xinu Approach (Macintosh Edition).* Prentice Hall, 1989. ISBN: 013638529X.

[Com84] Douglas Comer. *Operating System Design: The XINU Approach.* Prentice Hall, 1984.

[Com11] Douglas Comer. *Operating System Design: The Xinu Approach, Linksys Version.* CRC Press, 2011. ISBN: 143988109X.

[CPA93] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The Nachos Instructional Operating System. In *USENIX Winter*, pages 481–488, 1993.

[CT04] Steve Chamberlain and Ian Lance Taylor. Using ld. The GNU linker, 2004. http://wwwcdf.pd.infn.it/localdoc/ld.pdf, accessed 2011/08/07.

[Cus94] Helen Custer. *Inside the Windows NT File System.* Microsoft Press, 1994. ISBN: 155615660X.

[Dij] Edsger W. Dijkstra. Over seinpalen. EWD-74. E. W. Dijkstra Archive. Center for American History, University of Texas at Austin. Circulated privately, date unknown, http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF, retrieved 2014/07/07.

[Dij68] Edsger W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, 11(5):341–346, 1968. CODEN CACMA2. ISSN: 0001-0782.

[Dow08] Allen B. Downey. *The Little Book of Semaphores.* 2008. http://greenteapress.com/semaphores/.

[ECM87] ECMA. *ECMA-119: Volume and File Structure of CDROM for Information Interchange.* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 1987. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-119.pdf.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. L4 Reference Manual, MIPS R4x00. Version 1.0. Technical report, IBM T. J. Watson Research Center, New York, December 1997. UNSW-CSE-TR-9709, ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9709.pdf.

[FCZP95] George Fankhauser, Christian Conrad, Eckart Zitzler, and Bernhard Plattner. Topsy—A Teachable Operating System. Technical report, Computer Engineering and Networks Laboratory, ETH Zürich, 1995. http://www.tik.ee.ethz.ch/~topsy/Book/Topsy_1.1.pdf, accessed 2012/11/09.

[Fel13] Markus Felsner. Implementation eines Schedulers für das Betriebssystem ULIX. Bachelor's thesis, Hochschule für Oekonomie und Management (FOM), 2013. http://ulixos.org/publications/ulix-felsner-scheduler.pdf.

[Fre05] Free Software Foundation. GNU GRUB Manual 0.97, 2005. http://www.gnu.org/software/grub/manual/legacy/grub.html.

[Fri05] Brandon Friesen. Bran's Kernel Development. A tutorial on writing kernels. Version 1.0, 2005. http://www.osdever.net/bkerndev/Docs/title.htm; accessed 2013/10/19.

[HC30] Dan Hettena and Rick Cox. A Guide to Nachos 5.0j. Internet: http://inst.eecs.berkeley.edu/~cs162/fa07/Nachos/walk/walk.html, accessed 2014/10/30.

[HF95] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley Professional, 1995. ISBN: 0805316701.

[HLS02] David A. Holland, Ada T. Lim, and Margo I. Seltzer. A New Instructional Operating System. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 111–115. ACM, New York, NY, USA, 2002. ISBN: 1-58113-473-8. http://doi.acm.org/10.1145/563340.563383.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, Amsterdam, 2012. ISBN: 978-0-12-397337-5.

[Hul95] Tudor Hulubei. The Thix Operating System Kernel – Design and Architecture. Technical report, Department of Computer Science and Engineering, Politehnica University of Bucharest, Romania, 1995. http://hulubei.net/tudor/thix/download/doc/thix.ps, accessed 2014/10/30.

[Hyd10] Randall Hyde. *The Art of Assembly Language.* No Starch Press, San Francisco, 2010. ISBN: 1593272073.

[IBM83] IBM. *The IBM Personal Computer XT Technical Reference Manual.* IBM, 1983. http://www.retroarchive.org/dos/docs/ibm5160techref.pdf.

[IEE95] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language).* IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. http://www.ansi.org/.

[Int86] Intel. Intel 80386 Programmer's Reference Manual, 1986. http://microsym.com/editor/assets/386intel.pdf, accessed 2013/11/17.

[Int88] Intel. Intel 8259A Programmable Interrupt Controller (8259A/8259A-2), December 1988. Datasheet for PC interrupt controller chip, http://i30www.ira.uka.de/~sdi/doc/8259a.pdf.

[Int00] International Electrotechnical Commission (IEC). IEC 60027-2, Second edition, Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics, November 2000.

[Int01] Intel. Intel (R) Itanium (TM) Processor-specific Application Binary Interface (ABI), May 2001. http://download.intel.com/design/itanium/downloads/245370.pdf.

[Int08] Intel. TLBs, Paging-Structure Caches, and Their Invalidation. Application Note, Rev. 3, December 2008. Also incorporated in [Int11].

[Int11] Intel. Intel (R) 64 and IA-32 Architectures Software Developer's Manual – Volume 3, System Programming Guide, May 2011.

[IT13] IEEE and The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition (also named: Single UNIX Specification, Version 4, 2013 Edition), 2013. http://pubs.opengroup.org/onlinepubs/9699919799/.

[JL83] Andy Johnson-Laird. *The Programmer's CP/M Handbook.* Osborne/McGraw-Hill, Berkeley, California, USA, 1983. ISBN: 0-88134-119-3.

[JM98] Bruce Jacob and Trevor Mudge. Virtual Memory: Issues of Implementation. *Computer*, 31(6): 33–43, June 1998. ISSN: 0018-9162. http://dx.doi.org/10.1109/2.683005. http://www.ece.cmu.edu/~ece548/localcpy/r6033.pdf.

[KL01] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation.* Addison-Wesley, third edition, 2001. Contains details on Version 3.6 of CWEB.

[KL03] Donald E. Knuth and Silvio Levy. The CWEB System of Structured Documentation. http://www-cs-faculty.stanford.edu/~knuth/cweb.html, 2003. Accessed 2014/10/30.

[Kno65] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10):623–624, October 1965. ISSN: 0001-0782. http://doi.acm.org/10.1145/365628.365655.

[Knu86] Donald E. Knuth. *Computers & Typesetting, Volume B: TEX: The Program.* Addison Wesley Publishing Company, 1986.

[Knu93] Donald E. Knuth. Mini-indexes for literate programs. Internet: http://www.literateprogramming.com/milp.pdf, December 1993.

[Koh13] Frank Kohlmann. Implementierung eines ELF-Programm-Loaders für das ULIX-Betriebssystem. Bachelor's thesis, Georg-Simon-Ohm-Hochschule Nürnberg, 2013. http://ulixos.org/publications/ulix-kohlmann-elfloader.pdf.

[Lab02] Jean J. Labrosse. *MicroC/OS-II. The Real-Time Kernel.* CMP Books, 2nd edition, 2002. ISBN: 1-57820-103-9. https://doc.micrium.com/download/attachments/10753158/100-uC-OS-II-002.pdf, accessed 2014/10/15.

[Lam94] Lawrence J. Lamers, editor. *Information technology – AT Attachment Interface for Disk Drives (ATA 1).* Computer and Business Equipment Manufacturers Association, 1994. Working draft; withdrawn 1999; http://www.t13.org/documents/UploadedDocuments/project/d0791r4c-ATA-1.pdf.

[LDA+14] Kevin Lawton, Bryce Denney, Greg Alexander, Todd Fries, Donald Becker, and Tim Butler. BOCHS: x86 PC emulator. http://bochs.sourceforge.net/, 2014.

[Lev14] Roy Levin. Microsoft makes source code for MS-DOS and Word for Windows available to public, 2014. http://blogs.technet.com/b/microsoft_blog/archive/2014/03/25/microsoft-makes-source-code-for-ms-dos-and-word-for-windows-available-to-public.aspx, accessed 2014/03/29.

[Lie96] Jochen Liedtke. L4 Reference Manual, 486, Pentium, Pentium Pro. Version 2. Technical report, IBM T. J. Watson Research Center, New York, September 1996. Research Report RC 20549, http://os.inf.tu-dresden.de/L4/l4refx86.ps.gz.

[Lin12a] Linux man-pages project. fork(2) man-page, release 3.44 of the Linux man-pages, October 2012. https://www.kernel.org/pub/linux/docs/man-pages/Archive/man-pages-3.44.tar.gz, accessed 2014/08/07.

[Lin12b] Linux man-pages project. pthread_exit(3) man-page, release 3.44 of the Linux man-pages, October 2012. https://www.kernel.org/pub/linux/docs/man-pages/Archive/man-pages-3.44.tar.gz, accessed 2014/08/07.

[Lio96] John Lions. *Lions' Commentary on UNIX 6th Edition, with Source Code.* Peer-to-Peer Communications, 1996. ISBN: 1-57398-013-7. http://www.lemis.com/grog/Documentation/Lions/index.html. Actually written in 1976, but published later.

[Loh13] Daniel Lohmann. Betriebssysteme (Operating Systems), 2013. http://www4.informatik.uni-erlangen.de/Lehre/WS13/V_BS.

[Lov03] Robert Love. *Linux Kernel Development.* Sams Publishing, Indianapolis, Ind., 2003. ISBN: 0672325128.

[LWD+14] Yang Lixiang, Liang Wenfeng, Chen Dazhao, Liu Tianhou, Wu Ruobing, Song Qi, and Feng Ke. *The Art of Linux Kernel Design: Illustrating the Operating System Design Principle and Implementation.* CRC Press, Taylor & Francis Group, 2014. ISBN: 1466518030.

[Mau08] Wolfgang Mauerer. *Professional Linux Kernel Architecture.* Wiley Publishing, Inc., Indianapolis, Indiana, 2008. ISBN: 978-0-470-34343-2.

[McL02] Peter T. McLean, editor. *Information Technology – AT Attachment with Packet Interface – 6 (ATA/ATAPI-6).* T13 Technical Committee, 2002. Working draft, version 3b, http://www.t13.org/documents/UploadedDocuments/project/d1410r3b-ATA-ATAPI-6.pdf.

[Men02] Georges Menie. Small printf source code, 2002. http://www.menie.org/georges/embedded/.

[Mic00a] Microsoft Corporation. Keyboard Scan Code Specification (Windows Platform Design Notes), Rev. 1.3a. Specification, March 2000. http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/scancode.doc, accessed 2014/07/17.

[Mic00b] Microsoft Corporation. Microsoft EFI FAT32 File System Specification. Hardware white paper, December 2000. http://msdn.microsoft.com/en-us/gg463080.aspx, accessed 2014/06/25.

[MNN05] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley, 2005. ISBN: 0-201-70245-2. xxviii + 683 pp. LCCN QA76.76.O63 M398745 2005. http://www.mckusick.com/FreeBSDbook.html.

[Mol08] James Molloy. Roll your own toy UNIX-clone OS – JamesM's kernel development tutorials, 2008. http://www.jamesmolloy.co.uk/tutorial_html/, accessed 2014/07/27.

[MT09] Eike Möhlmann and Janko Timmermann. iPosix. Ein objektorientierter Kernel in C++. Project Documentation, University of Oldenburg, September 2009. http://www.svs.informatik.uni-oldenburg.de/download/thesis/IP-20090914-moehlmann_timmermann-iPosix.pdf, accessed 2014/10/30.

[Nev00] Bob Neveln. *Linux Assembly Language Programming.* Prentice Hall, 2000. ISBN: 0130879401.

[NS01] Jürgen Nehmer and Peter Sturm. *Systemsoftware: Grundlagen moderner Betriebssysteme.* dpunkt, 2nd edition, 2001. ISBN: 3-89864-115-5.

[Nyb11] Jens Nyberg. cpu_usermode function. Forum post on osdev.org, July 2011. http://f.osdev.org/viewtopic.php?t=23890&p=194213; accessed 2014/08/07.

[OFBI09] Yoshinori K. Okuji, Bryan Ford, Erich Stefan Boleyn, and Kunihiro Ishiguro. The Multiboot Specification version 0.6.96, 2009. http://www.gnu.org/software/grub/manual/multiboot/multiboot.pdf.

[OSD13] OSDev.org. Interrupt Descriptor Table, July 2013. http://wiki.osdev.org/Interrupt_Descriptor_Table, last accessed 2013/11/16.

[Pat03] Steve D. Pate. *UNIX Filesystems: Evolution, Design, and Implementation.* Veritas series. Wiley, 2003. ISBN: 9780471164838. LCCN 2003266051. http://books.google.de/books?id=zj9E4_6zsHgC. ISBN: 9780471164838.

[Pfa10] Ben Pfaff. Pintos. http://www.stanford.edu/class/cs140/projects/pintos/pintos.pdf, accessed 2014/10/29, 2010.

[Phl10] Mark Phlippen. Konzeption eines Betriebssystem-Programmierpraktikums auf Grundlage von iPosix. Project Documentation, University of Oldenburg, August 2010. http://www.svs.informatik.uni-oldenburg.de/download/thesis/IP-20100831-phlippen-BSP_iPosix.pdf, accessed 2014/10/30.

[Pin09] Pintos Project: Table of Contents. http://www.stanford.edu/class/cs140/projects/pintos/pintos.html, accessed 2014/10/29, 2009.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[Ree98] Kenneth Reek. *Pointers on C.* Addison-Wesley Longman, Reading, Mass, 1998. ISBN: 0673999866.

[Rit93] Dennis M. Ritchie. The development of the C language. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 201–208. ACM, New York, NY, USA, 1993. ISBN: 0-89791-570-4. http://doi.acm.org/10.1145/154766.155580.

[Rob01] Tim Robinson. Memory Management 1, 2001. http://www.osdever.net/tutorials/view/memory-management-1, accessed 2014/10/30.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX Time-sharing System. *Commun. ACM*, 17(7): 365–375, July 1974. ISSN: 0001-0782. http://doi.acm.org/10.1145/361011.361061.

[Ruf98] Lukas Ruf. Topsy i386—A Teachable Operating System. The Port to the ia32 Architecture. Semesterarbeit, Computer Engineering and Networks Laboratory, ETH Zurich, 1998. http://www.tik.ee.ethz.ch/~topsy/theses/Topsyi386.pdf, accessed 2014/10/30.

[Ryf07] Sebastian Ryffel. Portierung von Topsy v3 auf den Intel Pentium 4. Semesterarbeit, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, 2007. http://www.sryffel.com/files/ETH/Topsy-v3_on_P4.pdf, accessed 2014/10/30.

[S03] Sandeep S. GCC-Inline-Assembly-HOWTO, 2003. http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html, accessed 2014/08/18.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* Addison-Wesley Professional, 1994. ISBN: 0201633388.

[Sch13] Joachim Schrod. cweb-latex — A LaTeX Version of CWEB. CTAN Comprehensive TEX Archive Network, http://www.ctan.org/pkg/cweb-latex, accessed 2014/10/01, 2013.

[Sea93] ATA Interface Reference Manual. Reference, Seagate Technology, ftp://ftp.seagate.com/acrobat/reference/111-1c.pdf, May 1993.

[Sey13] Ian Seyler. BareMetal website, 2013. http://www.returninfinity.com/baremetal.html, accessed 2014/06/08.

[Shu14] Len Shustek. Microsoft MS-DOS early source code, 2014. http://www.computerhistory.org/_static/atchm/microsoft-ms-dos-early-source-code/, accessed 2014/03/29.

[Son07] Marcel Sondaar. Higher Half with GDT, 2007. http://wiki.osdev.org/Higher_Half_With_GDT, accessed 2011/08/06.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1987. ISBN: 0-13-637331-3.

[TIS95] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[TW06] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: design and implementation.* Pearson Prentice Hall, third edition, 2006. xvii + 1054 pp.

[vdL94] Peter van der Linden. *Expert C Programming: Deep C Secrets.* SunSoft Press, Englewood Cliffs, NJ, 1994. ISBN: 0131774298.

[vG94] Frank van Gilluwe. *The Undocumented PC (The Andrew Schulman Programming Series).* Addison-Wesley Publishing Company, 1994. ISBN: 0201622777.

[vH02] William von Hagen. *Linux Filesystems.* Number 1 in Kaleidoscope Series. Sams, 2002. ISBN: 9780672322723. LCCN 2001093563. http://books.google.de/books?id=Lm5GAAAAYAAJ. ISBN: 9780672322723.

[Vil96] Pat Villani. *FreeDOS Kernel: An MS-DOS Emulator for Platform Independence & Embedded System Development.* Master OS development. Taylor & Francis, 1996. ISBN: 9780879304362. http://books.google.de/books?id=1yozoAjCy7sC.

[Wik] Wikipedia. Semaphore (programming). http://en.wikipedia.org/wiki/Semaphore_(programming), accessed 2014/07/07.

[XeL14] XeLaTeX Documentation Initiative. XeLaTeX Wiki, 2014. http://wiki.xelatex.org/doku.php, accessed 2014/06/08.

Image Credits

"UNIX-Licence-Plate" (title page of book resp. part 2 of the thesis) by KHanger - Own work. Licensed under Creative Commons Attribution 3.0 via Wikimedia Commons,
http://commons.wikimedia.org/wiki/File:UNIX-Licence-Plate.JPG#mediaviewer/File:UNIX-Licence-Plate.JPG

Hard disk icon (Figure 3.14): Oxygen Team, LGPL-licensed,
http://www.iconarchive.com/show/oxygen-icons-by-oxygen-icons.org/Devices-drive-harddisk-icon.html

Punch card image (Figure 12.1): Wikimedia Commons,
http://commons.wikimedia.org/wiki/File:Blue-punch-card-front-horiz.png

DVD logo (Figures 12.5, 12.6):
Wikipedia, http://de.wikipedia.org/wiki/Datei:DVD-ROM-Logo.svg

Other hard disk icon (Figures 12.5, 12.6):
Wikimedia Commons, http://commons.wikimedia.org/wiki/File:Harddisk.svg

The font used for Figure 10.1 and Figure 10.2 is "Aa Qwertz-Tasten" and was designed by Anke Arnold, http://www.anke-art.de/.

.

GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

   The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b) The work must carry prominent notices stating that it is

released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the

entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims

owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.