# Entwicklung eines Overlay-Dateisystems für das Lehrbetriebssystem Ulix mit Literate Programming

## Bachelorarbeit
## Medieninformatik

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM


| | |
|---|---|
| vorgelegt von: | Andreas Kurz |
| Matrikelnummer: | 2335486 |
| Erstgutachter: | Prof. Dr. Michael Zapf |
| Zweitgutachter: | Prof. Dr. Hans-Georg Eßer |


23. September 2016

# Zusammenfassung

In dieser Bachelorarbeit wird ein Overlay-Dateisystem als Erweiterung für Ulix entwickelt. Dieses Dateisystem wird als Copy-On-Read implementiert. Bei der Entwicklung kommt außerdem Literate Programming zum Einsatz, welches es dem Leser ermöglicht, die einzelnen Schritte detailliert zu verfolgen und zu verstehen. Da Ulix selbst ein unixartiges Betriebssystem ist, kann mit diesem Ansatz für ein Overlay-Dateisystem die Integration in andere unixartige Betriebssysteme verstanden werden.

## Prüfungsrechtliche Erklärung

Ich, Andreas Kurz, Matrikel-Nr. 2335486, versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____

Andreas Kurz

# Inhaltsverzeichnis

# 1. Einleitung

Betriebssysteme verwandeln etwas Hässliches in etwas Schönes. So schreiben es Andreas S. Tanenbaum und Herbert Bos in ihrem Buch „Moderne Betriebssysteme“. Was sie damit meinen ist, dass es zu einer der Hauptaufgaben des Betriebssystems gehört, dem Software-Entwickler eine schöne Schnittstelle zur Hardware zu bieten. Die Hardware sollte versteckt und durch eine hübsche, saubere, elegante und konsistente Abstraktion bereitgestellt werden. Täten Betriebssysteme das nicht, so müsste ein Entwickler zur Ansteuerung einer SATA-Festplatte, im Jahre 2007, 450 Seiten lesen. Benutzerprogramme arbeiten allerdings mit Dateien. Für diese Abstraktionsschicht zur Hardware, ist ebenfalls das Betriebssystem zuständig. [1, S. 31–32]

Auch in dieser Arbeit geht es um die Abstraktion von Hardware und den Umgang mit Dateien. Genauer gesagt wird mit Hilfe der Methodik des Literate Programmings erklärt, wie der Arbeitsspeicher eines Rechners auch als RAM-Disk genutzt werden kann. Ferner wird gezeigt, wie sich dieser Treiber dafür benutzen lässt ein Overlay-Dateisystem zu realisieren. Damit dies an einem konkreten Beispiel durchgeführt werden kann, wird das Lehrbetriebssystem Ulix verwendet. „Richtige“ Betriebssysteme, wie Linux, sind viel zu komplex, als dass sie einen einfachen Einstieg für die Implementierung eines solchen Treibers bieten können. So hat der Linux Kernel, in Version 3.14.5 für x86, bereits 318881 Zeilen an Quellcode [2, S. 18].

## 1.1. Zielsetzung

Im Rahmen dieser Arbeit soll ein Overlay-Dateisystem für Ulix entstehen. Diese Implementierung wird mit der Methode „Literate Programming“ durchgeführt. Das entwickelte Overlay-Dateisystem soll den Charakter eines Moduls besitzen und somit wenig Änderungen am Ulix -Kernelcode erfordern. Weiterhin soll der Speicher effizient genutzt werden, was eine dynamische Speicherverwaltung erfordert. Hierfür werden zuerst die nötigen Datenstrukturen geschaffen und anschließend Funktionen implementiert, um auf dieser Basis zu arbeiten. In einem letzten Schritt wird das entwickelte Dateisystem in Ulix integriert, so dass es ein Überlagern eines vorhandenen Dateisystems ermöglicht. Ferner soll dieses Overlay-Dateisystem unsichtbar für den Benutzer arbeiten, was Änderungen an der Benutzerbibliothek ausschließt.

## 1.2. Ulix-Lehrbetriebssystem

Das Betriebssystem Ulix wird von Hans-Georg Eßer und Felix Freiling entwickelt und hat sich zum Ziel gesetzt, interessierten Lesern einen Einstieg in die Betriebssystementwicklung zu bieten. Hierbei wurde sich auch darauf konzentriert, nicht alle möglichen Lösungen für ein Problem zu zeigen, sondern nur die, die in Ulix Anwendung findet.[2, S. 5] Der Name Ulix deutet schon die Verwandtschaft zu UNIX an und manifestiert sich noch mehr in der Implementierung, sodass die im Buch beschriebenen Lösungsansätze nicht nur theoretischer Natur sind, sondern auch zum Verständnis von UNIX bzw. Linux beitragen.

Eine Besonderheit dieses Betriebssystem ist die Art der Programmierung, was Ulix auch für den Einsatz in der Lehre interessant macht. Die Programmiersprache selbst ist C, aber die Art und Weise, wie Dokumentation und Quellcode zusammenspielen ist hier anders. Hierfür wurde die Methode des Literate Programmings gewählt, was eine Kombination aus Quellcode und Dokumentation darstellt. Daraus resultiert, dass der Quellcode nicht in einer anderen Datei liegt als die Beschreibung – das Buch *ist* der Quellcode. So können Studierende, oder andere Interessenten, allein durch Lesen des Buches die komplette Implementierung anschaulich und gut erläutert erfahren.

## 1.3. Verwandte Arbeiten

Diese Arbeit stellt eine Erweiterung für Ulix dar. Da dieses aktiv in der Lehre eingesetzt wird, sind bereits zahlreiche Erweiterungen im Rahmen von Abschlussarbeiten entstanden [3]. Themenverwandt zu dieser Arbeit sind die Bachelorarbeiten „Implementation eines FAT-Treibers für das Lehrbetriebssystem ULIX“ von Simon Brugger und „Implementation eines Dateisystems und einer RAM-Disk für das Betriebssystem ULIX“ von Liviu Beraru. Eine Kombination der beiden Arbeiten würde schon fast für ein Overlay-Dateisystem genügen. Warum dem nicht so ist, wird im Folgenden dargelegt. Ebenfalls wird kurz erläutert, weshalb „Unionfs“ auch nicht für die Verwendung in Ulix geeignet ist.

**FAT-Treiber von Simon Brugger** Das Overlay-Dateisystem aus dieser Arbeit soll das bestehende Minix-Dateisystem von Ulix überlagern können. Hierfür ist es unter anderem notwendig, dass das Konzept von Zugriffsrechten aufrechterhalten bleibt. Allerdings unterstützt das FAT-Dateisystem nur einfache Dateimerkmale, aber keine Zugriffsrechte, wie sie für die Überlagerung des Minix-Dateisystems erforderlich wären [4, S. 39]. Weiterhin arbeitet diese Implementierung, bzw. das FAT-Dateisystem selbst, mit einem vorher in der Größe festgelegten Speichermedium [4, S. 47], was im Rahmen dieser Arbeit nicht als effiziente Speichernutzung des RAM verstanden wird. Aus diesen Gründen scheidet die Nutzung der Implementierung dieser Erweiterung für diese Arbeit aus.

**RAM-Disk von Liviu Beraru** Das in der Arbeit von Beraru implementierte Dateisystem „UlixFS“ enthält alle notwendigen Eigenschaften, um es für die Überlagerung des Minix-Dateisystem in Ulix zu verwenden [5, S. 25]. Allerdings basiert die Nutzung der RAM-Disk wieder auf einem Speicherbereich mit fester Größe [5, S. 48]. Ein weiteres Problem bei der Nutzung dieser Implementierung, wird ebenfalls bei der Initialisierung ersichtlich [5, S. 48]. Die Funktion `kmalloc()` ist in der Ulix-Version 0.13 nicht bzw. nicht mehr verfügbar. Da für den Test in der Arbeit von Beraru noch Ulix 0.06 zum Einsatz kam [5, S. 123] wurden keine weiteren Nachforschungen unternommen, den Sachverhalt um `kamlloc()` aufzudecken, da die Implementierung schon wegen der festen Speichergröße ausschied.

**Unionfs** ist eine Implementierung eines Overlay-Dateisystems, allerdings für den Linux. Dieses ist quasi über UNIX mit Ulix verwandt [1, S. 43] [2, S. 21], doch ist es für die Entwickler von Unionfs schon schwierig verschiedene Linux-Versionen zu unterstützen [6, S. 2–3]. Aus diesem Grund wurde eine mögliche Portierung von Unionfs für Ulix nicht weiter in Betracht gezogen.

# 2. Grundlagen

In diesem Abschnitt soll die Funktionsweise der beiden Kernaspekte Literate Programming und Overlay-Dateisysteme erläutert werden. Ferner wird auf das Virtuelle-Dateisystem in Ulix eingegangen, da dies den Grundstein für die Integration in das Betriebssystem legt.

## 2.1. Literate Programming

Dies ist eine Art der Software-Entwicklung, die von Donald Knuth 1984 in einem Paper des „THE COMPUTER JOURNAL“ vorgestellt wurde. Er war so angetan von dieser neuen Methode, dass er sich selbst als Fanatiker, der glaubt erleuchtet worden zu sein, bezeichnet. Der Fokus bei der Software-Entwicklung sollte seiner Meinung nach nicht mehr darin bestehen einem Computer zu erklären, was er zu tun hat, sondern einem Menschen zu erklären, was der Computer tun soll. Er hat beim Einsatz dieser Methodik festgestellt, dass er besser beschriebene Programme entwickelt und ist davon überzeugt, dass auch die Programme selbst besser sind, da er sich mehr Mühe gibt. Diese Überzeugung rührt daher, dass sich der Entwickler nicht mehr als reiner Programmierer sieht, der sein Programm dokumentiert, sondern als Author, der sein Programm in einer für Menschen möglichst bestens zu verstehenden Weise niederschreibt. Daraus resultiert nicht unbedingt eine Reihenfolge, wie sie der Computer verlangen würde. [7, S. 1]

Die technischen Voraussetzungen hierfür hat Knuth im Rahmen seiner Forschungsarbeit an der Stanford University ebenfalls selbst geschaffen. Diese Sprache und die Sammlung an nötigen Programmen bezeichnet er als „WEB“. Die Sprache selbst wiederum besteht aus der Dokumentenbeschreibungssprache TEX und der Programmiersprache Pascal. Um nun Programmdatei und Dokumentation aus der gemeinsamen WEB-Datei zu extrahieren, existieren hierfür die Programme „TANGLE“ und „WEAVE“, deren Ergebnisse wiederum ganz normal vom Pascal-Compiler in ein Programm übersetzt und vom TEX-Interpreter weiterverarbeitet werden. [7, S. 1–2] Eine genauere Darstellung dieser Vorgehensweise wird in Abschnitt 2.1.1 erläutert.

Obwohl die Veröffentlichung von Knuths Paper doch einige Zeit zurückliegt, ist die Forschung im Bereich des Literate Programming nicht zum Stillstand gekommen. So zeigt ein Paper von Meik Teßmer der Universität Bielefeld ausführlich auf, wie Literate Programming selbst zur Verwaltung und Dokumentation in der Systemadministration eingesetzt werden kann. Ferner wird im Rahmen dieser Forschungsarbeit auch das Problem angegangen, dass WEB stets davon ausgeht, dass es nur eine einzige Datei gibt, in die das Dokument geschrieben wird. Da die Systemadministration an der Universität aber von mehreren Personen durchgeführt wird und somit auch dokumentiert werden muss, löst er dieses Problem mit einem Programm, dass in seinem Paper als „includer“ betitelt wird. Dieses Projekt ist seit 2008 im produktiven Einsatz und zeigt damit auch die Möglichkeiten von Literate Programming abseits von reinen Programmiersprachen. [8]

### 2.1.1. Funktionsweise

Sowohl in der Arbeit von Teßmer als auch in Ulix kommt nicht das von Knuth entwickelte WEB zum Einsatz, sondern eine, in der Funktionsweise ähnliche, Implementierung „noweb“. Das „no“ in diesem Namen ist hierbei ein Verweis auf den Entwickler Norman Ramsey. Hauptmerkmal dieser Implementierung ist die Unabhängigkeit der genutzten Programmiersprache. Wie oben bereits beschrieben, ist WEB für Pascal entwickelt worden und eine Änderung der Sprache würde eine Änderung an der Implementierung von WEB und erneutes Kompilieren bedeuten, was bei noweb nicht erforderlich ist. [9]

Die noweb-Datei enthält nun neben der Dokumentation, in der Auszeichnungssprache LaTeX, auch den Programmcode, organisiert in sogenannten Code-Chunks. Diese dienen nun dazu, das Programm in gut verständliche Einheiten zu zerlegen, die dann beschrieben werden können. Ein Chunk ist technisch gesehen nie abgeschlossen, das heißt konkret, er kann immer „nach unten“ erweitert werden - egal wo im Dokument diese Erweiterung stattfindet. Allerdings findet die Zusammensetzung des Chunks, beim Extrahieren des Programmcodes aus der noweb-Datei, stets von „oben nach unten“ statt. So gesehen ist zwar die Erweiterbarkeit von Chunks unabhängig von der Position im Dokument, allerdings ist bei der Reihenfolge trotzdem Vorsicht geboten.

Eine weitere, wichtige Eigenschaft von Chunks ist, dass diese ineinander geschachtelt werden können, wobei der Name hierbei als Platzhalter dient. Es ist ebenfalls möglich, einen Platzhalter zu verwenden, auch wenn dieser noch nicht im Dokument definiert wurde - anders als in der Programmiersprache C, wo ein Platzhalter, eine Variable, immer erst definiert werden muss, bevor sie verwendet werden kann. Dies wird am Code-Beispiel 2.1 aufgezeigt, was die Syntax in der noweb-Datei veranschaulicht.

Zuerst wird eine Variable benötigt, um eine Zahl zu speichern, was in der Definition von „first chunk“ geschieht. Anschließend fällt dem Entwickler ein, dass noch eine weitere Variable zu speichern ist. Nun muss aber keine spezielle Syntax beachtet werden, um „first chunk“ zu erweitern, sondern es kann wie bei der ersten Definition verfahren werden. Ebenfalls fügt er noch einen Chunk namens „initilize variables“ ein, in dem er später im Dokument die Variablen initialisiert. Wieviel Text nun konkret zwischen den einzelnen Chunks steht spielt keine Rolle. Auch können weitere Chunks dazwischen definiert werden. Das Beispiel ist daher auf ein Minimum reduziert, um die Verwendung von Chunks anschaulich zu halten.

Listing 2.1: Beispiel für die Nutzung von Chunks

```
We need a variable to store a number.
<<first chunk>>=
    int first_variable;
@

Now we need a second variable to store another number and later we
initialize them with specific values.
<<first chunk>>=
    int second_variable;
    <<initialize variaibles>>
@

Our variables are initialized as follows.
```

```
<<initilize variables>>=
    first_variable = 100;
    second_variable = 200;
@
```

Wie aus der Syntax im Listing 2.1 ersichtlich ist, kann diese nicht ohne weiteres dem C-Compiler übergeben werden. Für die Extraktion des eigentlichen Codes aus der noweb-Datei ist das Programm „notangle“ verantwortlich. Es setzt nun die Code-Chunks, ausgehend von einem Anfangs-Chunk, der per Argument übergeben wird, zusammen, so dass eine korrekte C-Datei entsteht. Die Ausgabe von notangle mit dem Anfangs-Chunk „first chunk“ würde wie in Listing 2.2 aussehen.

Listing 2.2: Ausgabe von notangle

```
int first_variable;
int second_variable;
first_variable = 100;
second_variable = 200;
```

Ebenfalls erlaubt und möglich ist der mehrfache „Aufruf“ von Chunks. So könnten im obigen Beispiel später im Code durch ein erneutes Einfügen von „«initialize variables»“ wieder auf die dort definierten Werte zurückgesetzt werden. Damit wird die Möglichkeit geschaffen, auch kleinere Routinen in Chunks statt in Funktionen der Programmiersprache zu kapseln und somit Funktionsaufrufe zu sparen. Genau wie bei Funktionen muss hier ebenfalls nur an einer Stelle im Dokument bei einer Änderung der Code angepasst werden. Einzige Einschränkung ist hierbei, dass die Variablen, die innerhalb des Chunks zum Einsatz kommen, entweder in diesem definiert werden oder bereits vorher schon definiert worden sind. In Ulix wird diese Methodik beispielsweise verwendet, um Pfadangaben bei Funktionsaufrufen im virtuellen Dateisystem zu korrigieren [2, 411e S.411].

Die Ausgabe von „noweave“ zur Erzeugung der tex-Datei ist in diesem Fall weniger interessant, da für eine korrekte LaTeX-Datei noch einiges an Syntax fehlt, was das Beispiel in Listing 2.1 nur unnötig aufblähen würde. Um nun ein Verständnis dafür zu erhalten, wie von einer noweb-Datei die letztendliche Programmdatei und das PDF mit der Dokumentation entstehen, soll Abbildung 2.1 Aufschluss geben. Aus der gemeinsamen Datei „MyProg.nw“ werden mit den Programmen noweave und notangle die Dateien „MyProg.tex“ und „MyProg.c“ erzeugt. Erstere wird anschließend durch das Programm „pdflatex“ in ein PDF überführt und Letztere mit Hilfe des Compilers, hier „gcc“, zu einem ausführbaren Programm übersetzt.

### 2.1.2. Notation von Code-Chunks

Wie im vorherigen Abschnitt 2.1.1 aufgezeigt wurde, kann die freie Definition von Chunks und deren Einbindung in Andere schnell unübersichtlich werden. Damit diese Verknüpfungen für den Leser ersichtlich bleiben, werden von noweave umfangreiche Referenzierungen durchgeführt. Der Chunk in Abbildung 2.2 befindet sich auf S. 122 des Dokuments, was der aktuellen Chunk-Nummer zu entnehmen ist. Das kleine „c“ am Ende dieser Nummer bedeutet, dass es der dritte Chunk auf dieser Seite ist, da diese in alphabetischer Reihenfolge mit Kleinbuchstaben gekennzeichnet werden, um Verwechslungen auszuschließen.

Diese Nummer befindet sich selbst im Randbereich, symbolisiert durch die gestrichelte grüne Linie der jeweiligen Seite, was bei zweiseitigem Druck links oder rechts sein kann - abhängig

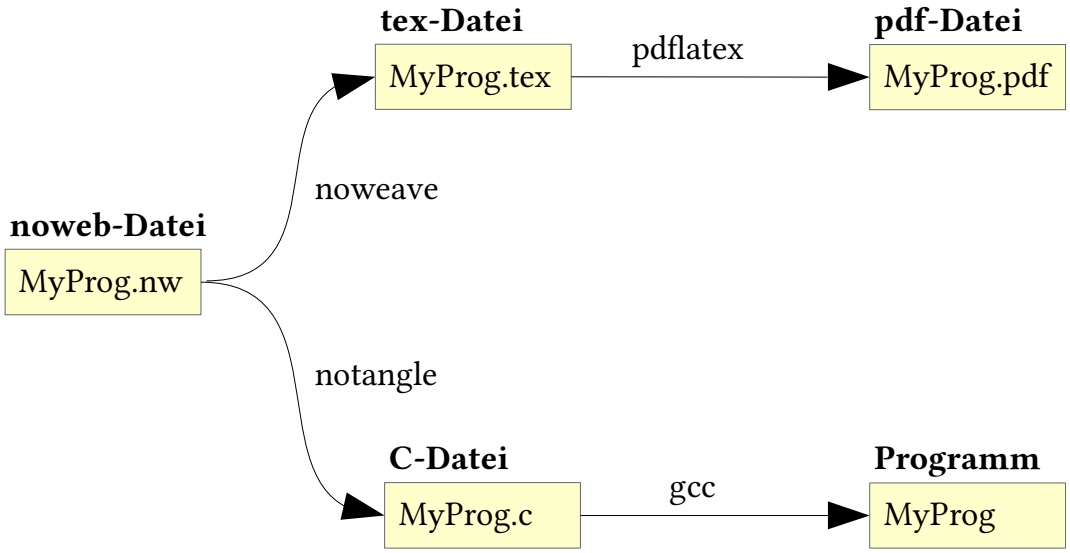


Abbildung 2.1.: Beispiel der noweb-Programmkette zur Erzeugung von PDF und Programmdatei

davon, ob es sich um eine linke oder rechte Seite handelt. Nach dem Namen des Chunks folgt ein Verweis auf die Stelle, an der dieser Chunk das erste Mal definiert wurde. In gleicher Höhe auf der rechten Seite befinden sich nacheinander die Chunk-Nummern für die erste Referenzierung, eine Weiterführung des Chunks an einer vorherigen Stelle sowie für eine etwaige Weiterführung an einer nachfolgenden Stelle. Die Chunk-Nummer auf die erste Referenzierung ist in dieser Abbildung deshalb niedriger als die erste Definition, weil der Chunk vorher schon einmal „aufgerufen“ wurde, aber noch nicht definiert war.

Am Ende eines Chunks befinden sich noch Namen und Referenzen von Identifiern, die in diesem Chunk zum Einsatz kamen und entweder dort definiert wurden, und eventuell an anderer Stelle verwendet werden, oder an anderer Stelle definiert wurden und in diesem Chunk verwendet werden.

## 2.2. Overlay-Dateisysteme

Dieser Dateisystemtyp schafft die Möglichkeit, verschiedene Dateisysteme für den Anwender als ein einziges, zusammenhängendes System zur Verfügung zu stellen. Als Synonym wird deshalb auch der Begriff Union-Dateisystem verwendet, um die Funktion der Vereinigung zu verdeutlichen. Die Funktionsweise solcher Overlay-Dateisysteme lässt sich einfach mit einem Schichtenmodell visualieren, wobei die Schichten von oben nach unten priorisiert sind.

Das heißt im Anwendungsfall, dass beim Öffnen einer Datei zuerst in der obersten Schicht nachgesehen wird, ob diese dort vorhanden ist. Falls nicht, werden so lange die Schichten nach unten durchwandert, bis die Datei gefunden wird und somit geöffnet werden kann. Ist die Datei nicht vorhanden und der Modus beim Öffnen auf „Erstellen“ gesetzt, so wird wiederum in der obersten Schicht die Datei angelegt und dem Anwender zur Verfügung gestellt.

Ein Durchlaufen dieser Schichten von oben nach unten kann ebenfalls wie ein Stapel betrachtet werden. Grundeigenschaft eines Stapels in der Informatik ist, dass jeweils nur das

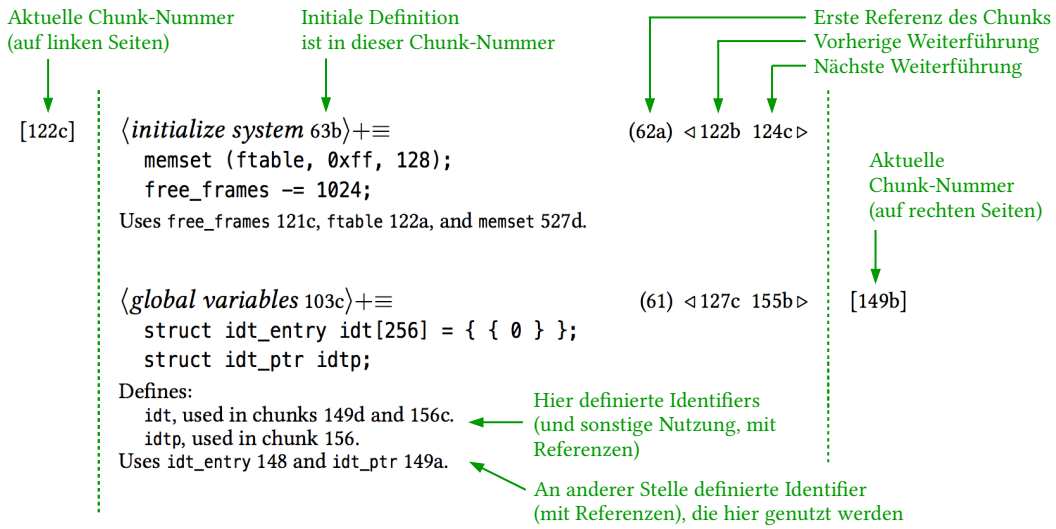


Abbildung 2.2.: Beispiel von Referenzierungen in noweb. Das original Bild wurde dahingehend modifiziert, dass die Beschreibungen in grün in deutsche Sprache übersetzt wurden [2, S. 27]

oberste Element direkt angesprochen werden kann [10, S. 67–68]. Im oben aufgezeigten Beispiel wäre dies das Overlay-Dateisystem. Daher wird bei der Implementierung von „Unionfs“ ein solches auch als „stackable file system“ bezeichnet [6].

Ein Anwendungsfall hierfür wäre ein Netzlaufwerk, das nur über eine langsame Internetverbindung erreichbar ist. Dies soll die unterste Schicht sein und eine Schicht darüber ein lokales Dateisystem. Nun können die Dateien geöffnet und bearbeitet werden. Beim Speichern werden diese aber zunächst nur in das lokale Dateisystem geschrieben, um Bandbreite zu sparen. Auch neue Dateien werden, wie bereits beschrieben, nur lokal angelegt. Für den Anwender ist das Overlay- Dateisystem unsichtbar, denn er soll von diesem möglichst nichts mitbekommen. So kann während der Arbeit mit den lokalen Kopien der Dateien gearbeitet und erst zum Feierabend eine Rücksynchronisation mit dem Netzlaufwerk gestartet werden.

Beim Öffnen der Dateien müssen diese natürlich auch erst vom Netzlaufwerk geladen werden, falls diese noch nicht lokal zur Verfügung stehen. Wird allerdings die Tatsache hinzugezogen, dass die meisten Internetanschlüsse in Haushalten oder auch kleineren Unternehmen über eine asymmetrische Verbindung verfügen, so stellt sich klar das Speichern als Flaschenhals heraus. Ein Anschluss über das Koaxkabel stellt zum Beispiel 200MBit/s zum Herunter-, aber nur 12MBit/s zum Hochladen von Daten zur Verfügung [11, S. 1].

### 2.2.1. Kopierverfahren

Wird obiges Beispiel weiter verfolgt, so muss unterschieden werden zwischen dem Laden der Daten vom Netzlaufwerk und dem tatsächlichen Kopieren bzw. Speichern der Daten in das Overlay-Filesystem. Ersteres ist immer erforderlich, falls die Datei noch nicht im lokalen Dateisystem existiert. Um zu steuern, wann die Daten dann in das Overlay-Dateisystem abgelegt werden, gibt es zwei Verfahren, die spiegelbildlich zueinander arbeiten.

#### Copy-On-Write

Das erste Verfahren wäre Copy-On-Write. Bezeichnend hierfür ist, dass das Kopieren von Daten als ein „teuerer Vorgang“ angesehen wird und deshalb so lange damit gewartet wird, bis die Daten verändert wurden und zwangsläufig geschrieben werden muss [1, S. 888]. Diese Variante ist dann im Vorteil, wenn mehr lesende als schreibende Zugriffe erfolgen und möglichst wenig Speicherplatz belegt werden soll, da so die Zeit für das Kopieren der Dateien minimal gehalten wird.

#### Copy-On-Read

Das Copy-On-Read-Verfahren ist das genaue Gegenteil zum vorher erwähnten Copy-On-Write. Hier werden die Daten bereits beim Lesen kopiert. Wird ebenfalls der beschriebene Anwendungsfall herangezogen, hat dies zur Folge, dass jeder lesende Zugriff auf eine Datei auf dem Netzlaufwerk eine lokale Kopie erzeugt. Zur Vermeidung von Konflikten ist auch immer die ganze Datei zu übertragen und nicht nur die tatsächlich gelesenen Bytes.

Auf den ersten Blick scheint dieses Verfahren wesentlich verschwenderischer mit Ressourcen umzugehen, als das vorher aufgezeigte Copy-On-Write. Das ist aber nur dann richtig, wenn von einem einzigen, lesenden Zugriff auf eine spezifische Datei ausgegangen wird. Bei einem zweiten Lesezugriff, auf genau die selbe Datei, müssen bei Copy-On-Write wieder die selben Daten übertragen werden. Copy-On-Read hätte hier bereits beim ersten Lesezugriff die Datei komplett übertragen und würde beim Zweiten den Vorteil der erhöhten Zugriffsgeschwindigkeit bieten, da nicht noch einmal auf die langsamere Internetverbindung gewartet werden muss.

### 2.2.2. Vergleich der Verfahren

Das eben aufgezeigte Rechenbeispiel funktioniert natürlich nur dann, wenn davon ausgegangen wird, dass beim ersten und zweiten Lesezugriff die selben Teile der, oder die komplette, Datei gelesen werden. Bei partiellen Lesezugriffen an unterschiedlichen Stellen in der Datei ist Copy-On-Write erst einmal im Vorteil.

Das Optimum aus Geschwindigkeit und Bandbreitennutzung wäre natürlich eine Kombination von Copy-On-Read und Copy-On-Write. Hier würden nur die tatsächlich gelesenen Daten in eine lokale Kopie geschrieben werden, so dass bei einem erneuten Zugriff von der Geschwindigkeit dieses Speichers profitiert werden kann. Problematisch wird es erst bei Schreibzugriffen, da hier die Position der Teildaten in der lokalen Kopie nicht mehr mit denen in der kompletten Datei auf dem Netzlaufwerk übereinstimmen.

Dieses Problem könnte wieder mit Verfahren aus Versionsverwaltungen angegangen werden. Allerdings könnten hier verschiedene Versionen einer Datei leicht miteinander in Konflikt geraten, was der Geschwindigkeits- und Speichernutzungsvorteil nicht aufwiegen kann. Diese Vermutung zu untersuchen wäre aber zu komplex, als dass sie in dieser Arbeit überprüft werden könnte. Ein weiterer Punkt, der hierbei nicht außer Acht gelassen werden darf, ist die Tatsache, dass beim Lesen von Daten nun immer die lokalen Teildaten mit der Gesamtdatei abgeglichen werden müssen. Dies benötigt mehr Rechenzeit in der CPU als ein einfacher Lesezugriff, da immer unterschieden werden muss, wann vom Netzlaufwerk und wann von der lokalen Kopie gelesen werden soll.

### 2.2.3. Verwendung in Ulix

Die Nutzung eines Overlay-Dateisystems in Ulix ist leider stark limitiert. Der bereits beschriebene Anwendungsfall mit dem Netzlaufwerk kann in Ulix nicht integriert werden, da hierfür die Netzwerkfunktionalität noch zu implementieren ist. Zwar ist bereits eine Bachelorarbeit zur „Implementation eines SLIP Moduls“ [12] vorhanden, allerdings fehlen hier noch sämtliche Schichten, Schnittstellen und Dienstprogramme, die ein Einbinden eines Netzlaufwerks ermöglichen. Desweiteren fehlt ebenfalls in Ulix selbst die Funktion zur Laufzeit Dateisysteme einzubinden [2, S. 405].

Die Vereinigung von zwei Minix-Dateisystemen würde nur zu Demonstrationszwecken entwickelt werden. Deshalb wurde sich dazu entschieden, das Overlay-Dateisystem mit dem RAM zu verwenden. Dies hat den Vorteil, dass Ulix auch von einem schreibgeschützten Medium starten und verwendet werden könnte. Dieses Verhalten ist auch bei so genannten Live-Systemen von Linux Distributionen zu beobachten, die dazu dienen das System bereits vorab von einer CD bzw. DVD gestartet zu testen [6, S.1]. Eine weitere Anwendung von Systemen, die von einem schreibgeschützten Medium gestartet werden, ist für sicherheitsrelevante Systeme denkbar. Ist dieses kompromittiert, so kann mit einem simplen Neustart das System auf den Ursprungszustand zurückgesetzt werden und die Arbeit wieder aufgenommen werden.

Nun ist noch zu entscheiden, welche Art von Overlay-Dateisystem in Ulix zum Einsatz kommt. Wie bereits beschrieben bietet Copy-On-Write einige Vorteile was die Nutzung von Ressourcen angeht. Dies ist in Ulix aber nicht zwingend erforderlich, da es sich um ein Lehrbetriebssystem handelt, welches nicht für den produktiven Einsatz gedacht ist [2, S. 21]. Deshalb fällt die Entscheidung hier auf die Variante mit Copy-On-Read, da diese bei der Verwendung als Überlagerung des Startmediums bzw. Wurzeldateisystems noch einen Vorteil bietet. Systemdateien und -programme werden in der Regel häufig auch von Benutzerprogrammen genutzt. Sind diese also schon vom ersten Zugriff an im RAM gespeichert, so erfolgen weitere Zugriffe schneller, da nicht mehr der Umweg über eine langsame Festplatte gegangen werden muss. Auch für die Implementierung ist die Verwendung von Copy-On-Read von Vorteil, da hier beim ersten Zugriff die entsprechende Datei erst in den RAM kopiert und dann von dort aus gelesen werden kann. Eine Entscheidung, ob von der Festplatte oder dem RAM bei einem Zugriff zu lesen ist, muss nicht getroffen werden. Diese Einfachheit folgt ebenfalls dem Designprinzip von Ulix [2, S. 21], da so der Quellcode übersichtlicher gehalten werden kann und leichter zu verstehen ist.

## 2.3. Virtuelles-Dateisystem in Ulix

Das Virtuelle-Dateisystem in Ulix schafft die Abstraktion vom konkreten Dateisystemtreiber hin zu generischen Funktionen wie `open()`, zum Öffnen von Dateien, bei der ein Benutzer nur den Dateipfad, aber nicht das Dateisystem angeben muss [2, S. 407]. Für den Anwender schafft dies den Vorteil, dass Programme nicht für ein bestimmtes Dateisystem geschrieben werden müssen, sondern sich immer auf die generischen Funktionen verlassen können. Das Virtuelle- Dateisystem entscheidet dann anhand des Einhängepunktes in selbiges, an welchen konkreten Dateisystemtreiber der Funktionsaufruf weitergereicht werden muss [2, S. 403–404]. Dies ist möglich, da die Dateisysteme in einem Baum zusammenhängen, der von unten nach oben durchwandert werden kann, bis an einem Knoten ein Gerät gefunden wird, auf dem die Datei dann gespeichert ist [2, S. 401–403].

Soll dem Benutzer mehr als ein Dateisystem angeboten werden, so entsteht zwangsläufig

das Problem, dass das Betriebssystem entscheiden muss, welcher Dateisystemtreiber zur Anwendung kommt. Betriebssysteme, die ihre Geräte und deren Dateisysteme nicht in einem einzigen Baum organisieren, gehen hier einen anderen Lösungsansatz. Windows-Betriebssysteme zum Beispiel weisen jedem Dateisystem einen Laufwerksbuchstaben zu. Bei einem Zugriff auf eine Datei, wird nun immer der Pfad inklusive Laufwerksbuchstabe angegeben. So kann das Betriebssystem anhand des Laufwerksbuchstaben entscheiden, welcher Dateisystemtreiber angesprochen werden muss. [1, S. 367]

Abgesehen davon, dass der Windows-Kern nicht Open-Source-Software ist, lässt sich aus obigem Verhalten ableiten, dass eine Integration eines Overlay-Dateisystems, die unabhängig vom Laufwerksbuchstaben sein soll, schwierig ist, da das Betriebssystem schon anhand dieses Buchstabens die Entscheidung getroffen hat, welcher Treiber anzusprechen ist. Das Virtuelle-Dateisystem von Ulix bietet hier jedoch sehr gute Möglichkeiten, das Betriebssystem um weitere Dateisystemtreiber zu erweitern. Da zu jeder generischen Funktion, wie `open()`, auch eine Kernel-Funktion, `u_open()`, existiert, muss nur in dieser an der richtigen Stelle die Integration vorgenommen werden und das Virtuelle-Dateisystem erledigt den Rest. Dies wird im Abschnitt 4 auf S. 54 noch genauer erläutert.

# 3. Implementation

Auf Basis der in Kapitel 2 beschriebenen Grundlagen, soll nun schrittweise ein Overlay-Dateisystem für Ulix entstehen. Dabei wird zuerst der Aufbau der Quellcode-Dateien beschrieben bevor die Datentypen, nötige Hilfsfunktionen und Funktionen der Schnittstelle zum Virtuellen-Dateisystem folgen. Das Overlay-Dateisystem selbst wird als flüchtiger Speicher implementiert, sodass nach einem Neustart des Systems alle Änderungen verschwunden sind. Da diese Arbeit eine Erweiterung für Ulix darstellt, wird sie ebenfalls mit Literate Programming entwickelt.

## 3.1. Aufbau der Code-Dateien

Der Quellcode für das Modul des Overlay-Dateisystems wird nicht direkt in die Ulix -Datei geschrieben, da es so nicht möglich wäre ein eigenes Dokument zur Beschreibung zu erstellen. Daher wird eine eigene Datei angelegt, aus der dann die nötigen Komponenten extrahiert werden. Das Modul besteht aus einer Datei `module.c`, die die Implementation der Funktionen enthält, und aus einer Header-Datei `module.h`, die die Datentyp-Definitionen und die Funktionsprototypen bereitstellt. Der Aufbau dieser beiden Dateien ist den nächsten beiden Code-Chunks zu entnehmen.

In der Header-Datei werden erst die Typdefinitionen aus Ulix geschrieben, bevor die eigenen Konstanten, Datentypen und Funktionsprototypen folgen.

⟨*module.h* 16a⟩≡ [16a]

```
/* Header-Datei */

#ifndef __MODULE_H
#define __MODULE_H

⟨public elementary type definitions 73⟩

/* Diese Funktion wird aus der Initialisierung des
   Betriebssystems heraus aufgerufen. */
void initialize_module ();

⟨ofs constants 19b⟩
⟨ofs data types 18a⟩
⟨ofs prototypes 23a⟩
#endif
```

Defines:
`__MODULE_H`, never used.
Uses `initialize_module` 16b.

In der C-Datei für den Quellcode wird anschließend die Header-Datei eingebunden, Datentypen initialisiert und die Funktionsimplementationen geschrieben. Zur Ausgabe der ersten 20 Inodes im Overlay-Dateisystem, wird zusätzlich ein Systemcall implementiert und installiert, der über ein Usermode-Programm aufgerufen werden kann.

⟨*module.c* 16b⟩≡ [16b]

```
/* Code */
#include "module.h"
⟨ofs data initialize 19a⟩
⟨ofs functions 23b⟩

void ofs_test_syscall(context_t *r) {
    int i = 0;
    for(; i < 20; i++) {
        printf("%s\n", ofs_index[i].path);
        struct ofs_file *file = ofs_index[i].data;
        if(file != 0) {
            printf("mode: %x size: %d\n", file->i_mode, file->i_size);
            if(i < 0) {
                char *ptr_c = file->data[0];
                for(int j = 0; j < 200; j++) {
                    printf("%x", *(ptr_c + j));
                }
                printf("\n");
            }
        }
    }
}

void initialize_module () {
    install_syscall_handler(780, ofs_test_syscall);
    printf ("Initializing student's module. \n");
    ofs_init_complete = 1;
    return;
}
```

Defines:
`initialize_module`, used in chunk 16a.
`ofs_test_syscall`, never used.
Uses `context_t` 73, `file` 32b 33c 38b 44c 47d 50a 53a, `ofs_index` 19a, and `ofs_init_complete` 54c.

## 3.2. Datentypen

Das Overlay-Dateisystem soll für ULIX die Speicherung der Dateien im RAM übernehmen. Hierfür ist es notwendig eigene Datenstrukturen zu definieren. Die Datenhaltung geschieht in einem zweistufigen Verzeichnis. Die erste Stufe stellt den Inode-Index mit Pfadangaben und die Zweite die eigentlichen Daten zur Verfügung. In erster Überlegung war die Speicherung in einer einfachen Tabelle angedacht, die sich aber bei genauerer Betrachtung als nachteilig erwies.

### 3.2.1. Einfache Tabelle

Bei dieser Art der Speicherung wird beim Öffnen einer Datei nur überprüft, ob in `ofs_files` ein `path` Eintrag mit dem entsprechenden Dateinamen vorhanden ist. Wird dieser gefunden, so wird der Inhalt aus der Speicherstelle `data` zurückgeliefert.

⟨ofs unused 17⟩≡ [17]

```
struct ofs_table {
    char path[255];
    char* data;
}

struct ofs_table ofs_files[255];
```

Defines:
`ofs_files`, never used.

In diesem Beispiel wäre eine einfache Verwendung schon möglich, allerdings mit folgenden Einschränkungen: Verknüpfungen würden zu Duplikaten im RAM führen und eigentlich nicht mehr ihren Zweck erfüllen, da sie so unterschiedliche Dateien erzeugten und nicht auf eine andere verwiesen. Die Dateigröße stellt hierbei ein weiteres Problem dar. Im Kernelmodus ist kein `malloc()` und `realloc()` implementiert, so dass ein nachträgliches Vergrößern bzw. Verkleinern der Datei nur über eine zwischenzeitliche Kopie möglich ist. Ebenfalls kann der Speicher für die Datei nur seitenweise allokiert und beim Anfordern von mehr als einer Speicherseite, besteht die Einschränkung, dass auch ausreichend zusammenhängender Speicher vorhanden ist [2, S. 119].

Ein weiterer Nachteil sind die Dateizugriffsrechte und andere Metainformationen wie Zeitstempel der letzten Modifkation. Diese könnten zwar einfach in obigem `ofs_table` ergänzt werden, hätten aber wieder den Nachteil, dass sie nicht für Verknüpfungen geeignet sind. Damit diese Nachteile möglichst ausgeschlossen werden, kommt dieser einfache Ansatz, eine einfache Tabelle, nicht für das Overlay-Dateisystem zum Einsatz. Der Lösungsweg hierfür wird im folgenden Abschnitt 3.2.2 aufgezeigt.

### 3.2.2. Zweistufiges Verzeichnis

Um die vorhin genannten Nachteile zu umgehen soll das Overlay-Dateisystem mit einem zweistufigen Verzeichnis realisiert werden. Der Zugriff erfolgt jetzt nicht nur ausschließlich über die absolute Pfadangabe `path` sondern zusätzlich noch über die Inode-Nummer `inode` der Datei im Minix-Dateisystem. So wird vermieden, dass Verknüpfungen zu unverknüpften Duplikaten im RAM führen. Dieser Zusammenhang wird später in Unterabschnitt 4.2.2 genauer erläutert.

Die maximale Dateilänge wird in `path` auf 248 Bytes festgelegt. Dies hat zur Folge, dass ein `ofs_inode` exakt 256 Bytes groß ist und somit 16 dieser Strukturen in eine Speicherseite passen. Als „Inode“ wird hier ein Eintrag im darüberliegenden Index bezeichnet, der nur die oben aufgeführten Parameter enthält. Was in Ulix als Inode bezeichnet wird, wird im Rahmen dieser Arbeit mit `ofs_file`, also einer Datei, bezeichnet. Sollte für Ulix einmal `malloc()` im Kernel zur Verfügung stehen, wäre es hier natürlich eine schönere Lösung exakt so viel Speicher wie nötig für den Dateinamen zu allokieren. Für die jetzige Implementation ist aber der aufgeführte Lösungsansatz in Ordnung, da Ulix nicht als Produktivsystem gedacht ist.

⟨*ofs data types* 18a⟩≡ (16a) 18b▷ [18a]
```
struct ofs_inode {
    char path[248];
    int inode;
    struct ofs_file* data;
};
```

Für den eigentlichen Index `ofs_index` ist es nun sinnvoll ein vielfaches von 16 von `ofs_inode` anzulegen. Um den Code variabel zu gestalten, wird dies mit Hilfe einer Konstanten `OFS_MAX_FILES` realisiert.

⟨*ofs data types* 18a⟩+≡ (16a) ◁18a 19d▷ [18b]
```
extern struct ofs_inode ofs_index[OFS_MAX_FILES];
```
Uses `ofs_index` 19a and `OFS_MAX_FILES` 19b.

⟨*ofs data initialize* 19a⟩≡ (16b) 21d▷ [19a]
```
struct ofs_inode ofs_index[OFS_MAX_FILES] = {{0}};
```
Defines:
`ofs_index`, used in chunks 16b, 18b, 24–30, 32–34, 38b, 46–51, and 53.
Uses `OFS_MAX_FILES` 19b.

Da das Wurzeldateisystem über ein 1,44 MB Floppy-Image eingebunden wird, ist es zunächst ausreichend `OFS_MAX_FILES` auf 1024 festzulegen, da nicht davon ausgegangen wird, auf diesem mehr als 1024 Dateien unterzubringen.

⟨*ofs constants* 19b⟩≡ (16a) 19c▷ [19b]
```
#define OFS_MAX_FILES 1024
```
Defines:
`OFS_MAX_FILES`, used in chunks 18b, 19a, 24a, 26b, 27a, and 34b.

Damit wäre die erste Stufe des Datenspeichers fertiggestellt. Die zweite Stufe stellt nun die eigentlichen Informationen zur Datei und den Dateiinhalt selbst zur Verfügung. Das Lesen und Schreiben im Minix-Dateisystem läuft zwar blockorientiert, jedoch ist es beim Arbeiten im RAM einfacher diese Zugriffe auf Byte-Ebene durchzuführen. Da im Kernelmodus Speicher nur Seitenweise allokiert werden kann, wäre auch eine Operation auf Speicherseitenbasis vorstellbar, allerdings ist diese Einheit etwas zu groß und unhandlich. Zwar ist es möglich mit `request_new_pages()` einen ganzen Speicherbereich anzufordern, was aber wegen der genannten Nachteile in Unterabschnitt 3.2.1 nicht zum Einsatz kommt.

Die größe einer Speicherseite beträgt 4 KiB und könnte somit 4 Minix-Blöcke speichern [2, S. 440]. An dieser Stelle wird deutlich, dass das Overlay-Dateisystem eher ungeeignet für viele kleine Dateien ist, da diese immer mindestens 4 KiB an Speicher belegen. In der ersten Speicherseite für eine Datei müssen allerdings noch die Metainformationen untergebracht werden, welche die ersten 512 Bytes dieser Seite belegen. Zur besseren Anpassung ist es auch an dieser Stelle angebracht diesen Offset in einer Konstanten `OFS_DATA_OFFSET` zu speichern. Um später Berechnungen zu erleichtern, wird auch die resultierende Größe der ersten Speicherseite in einer Konstante `OFS_FIRST_PAGE_SIZE` festgehalten.

⟨*ofs constants* 19b⟩+≡ (16a) ◁19b 20a▷ [19c]
```
#define OFS_DATA_OFFSET 512
#define OFS_FIRST_PAGE_SIZE 3584
```
Defines:
`OFS_DATA_OFFSET`, used in chunks 28b and 34d.
`OFS_FIRST_PAGE_SIZE`, used in chunks 35 and 36b.

Für die zweite Stufe wird ebenfalls eine neue Datenstruktur benötigt. Diese Struktur `ofs_file` kann zum großen Teil aus der Minix-Implementation von ULIX übernommen werden [2, S. 442]. Der einzige Unterschied ist hierbei, dass die Zeiger in `data` immer auf den Anfang einer Speicherseite zeigen.

Da der erste Block der Datei nicht mehr für Daten selbst verwendet werden kann, wird das `data`-Array so gewählt, dass die 512 Bytes des Blocks möglichst ausgefüllt sind. In diesem Fall ergibt sich folgende Rechnung: 512 Bytes abzüglich der 24 Bytes, die für die Metainformationen genutzt werden, dividiert durch 4 Bytes, für den Zeiger auf die Datenbereiche, gleich 122 Einträge. $(512 - 24)/4 = 122$. Daraus resultiert eine maximale Dateigröße von $(3 \times 512) + 121 \times 4096 = 497152$ Bytes. Für die Speicherung von größeren Dateien ist `OFS_DATA_OFFSET` anzupassen, da somit mehr Platz für `data` geschaffen wird.

⟨*ofs data types* 18a⟩+≡ (16a) ◁18b 20b▷ [19d]
```
struct ofs_file {
    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;
    char* data[OFS_MAX_FILE_PAGES];
};
```

Uses OFS_MAX_FILE_PAGES 20a, uint16_t 73, and uint32_t 73.

Da ofs_file im ersten Teil einer Datei im Overlay-Dateisystem stehen soll, ist auch ersichtlich, wofür OFS_DATA_OFFSET eingesetzt wird. Die Funktionen zum Lesen und Schreiben erwarten, dass in einem ofs_file.data immer eine Speicherseite zu verarbeiten ist. Die Ausnahme bildet die erste Speicherseite, die ofs_file selbst enthält.

Die oben berechnete, maximale Dateigröße im Overlay-Dateisystem, und die maximale Anzahl im data-Array werden ebenfalls noch in Konstanten gespeichert, um spätere Überprüfungen zur Fehlerbehandlung zu ermöglichen.

⟨*ofs constants* 19b⟩+≡ (16a) ◁19c 20c▷ [20a]

```
#define OFS_MAX_FILE_SIZE 479152
#define OFS_MAX_FILE_PAGES 122
```

Defines:
OFS_MAX_FILE_PAGES, used in chunks 19d, 40d, and 46b.
OFS_MAX_FILE_SIZE, used in chunk 44c.

Was nun noch fehlt sind Verzeichnisse innerhalb des Overlay-Dateisystems. Da diese aber nur Dateien mit einem besonderen Inhalt, nämlich dir_entry-Strukturen darstellen, müssen nur diese zusätzlich definiert werden. Um kompatibel mit Ulix zu bleiben, wird die Definition dieser Struktur einfach übernommen.

⟨*ofs data types* 18a⟩+≡ (16a) ◁19d 21a▷ [20b]

```
struct dir_entry {
    word inode;
    byte filename[64];
};
```

Uses byte 73 and word 73.

Um beim Lesen und Schreiben dieser Struktur in Dateien besser agieren zu können, wird die Größe dieser Struktur ebenfalls in einer Konstanten OFS_DIR_ENTRY_SIZE hinterlegt.

⟨*ofs constants* 19b⟩+≡ (16a) ◁20a 21b▷ [20c]

```
#define OFS_DIR_ENTRY_SIZE 66
```

Defines:
OFS_DIR_ENTRY_SIZE, used in chunks 24, 31–33, 51, and 52.

### 3.2.3. Geöffnete Dateien

Mit der bisherigen Datenstruktur ist es bereits möglich Dateien im Overlay-Dateisystem abzulegen. Für die Zugriffe auf diese Daten ist allerdings noch eine weitere Datenstruktur erforderlich, die weitere Informationen über den aktuellen Zugriff auf eine Datei speichert. Diese sind der Modus in dem die Datei geöffnet wurde und die aktuelle Lese- und Schreibposition des Dateizeigers.

Würde sich dazu entschieden werden eine Datei nur ein einziges Mal zu öffnen, so könnten diese Informationen auch direkt in den Datenstrukturen des zweistufigen Verzeichnisses untergebracht werden. Da dies allerdings eine Einschränkung wäre, die eine vernünftige Nutzung eines Dateisystems nicht zu lässt, sind noch weitere Datenstrukturen notwendig.

Zunächst wird die Struktur ofs_open_file angelegt, die die eigentlichen Informationen zu einer geöffneten Datei bereit hält. pos ist die aktuelle Position des Dateizeigers für Lese- und Schreibzugriffe. mode definiert den Zugriffsmodus auf die Datei und werden später noch

erläutert. `file` zeigt auf eine `ofs_file` Struktur, um die eigentlichen Daten der Datei erreichen zu können.

⟨*ofs data types* 18a⟩+≡ (16a) ◁20b 21c▷ [21a]
```
struct ofs_open_file {
    int pos;
    short mode;
    struct ofs_file *file;
};
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a.

Nun wird noch ein Verzeichnis benötigt, in dem alle geöffneten Dateien gespeichert werden. Ähnlich wie bei der maximalen Anzahl an möglichen Dateien im Overlay-Dateisystem legt auch hier eine Kontante `OFS_MAX_OPEN_FILES` die maximale Anzahl an geöffneten Dateien fest. Diese wird zuerst definiert und anschließend das Verzeichnis `ofs_open_files`.

⟨*ofs constants* 19b⟩+≡ (16a) ◁20c [21b]
```
#define OFS_MAX_OPEN_FILES 256
```
Defines:
`OFS_MAX_OPEN_FILES`, used in chunks 21, 23b, 27c, and 39a.

⟨*ofs data types* 18a⟩+≡ (16a) ◁21a [21c]
```
struct ofs_open_file ofs_open_files[OFS_MAX_OPEN_FILES];
```
Defines:
`ofs_open_files`, used in chunks 23c, 27c, 38, and 40–45.
Uses `OFS_MAX_OPEN_FILES` 21b.

⟨*ofs data initialize* 19a⟩+≡ (16b) ◁19a [21d]
```
struct ofs_open_file ofs_open_files[OFS_MAX_OPEN_FILES] = {{0}};
```
Defines:
`ofs_open_files`, used in chunks 23c, 27c, 38, and 40–45.
Uses `OFS_MAX_OPEN_FILES` 21b.

Damit besteht nun die Möglichkeit den Zustand von geöffneten Dateien festzuhalten. Die allgemeinen Datentypen für das Overlay-Dateisystem sind damit vollständig.

### 3.2.4. Grafische Übersicht

Da der Zusammenhang der einzelnen Datenstrukturen aus dem obigen Fließtext nicht so einfach ersichtlich ist, folgt nun eine grafische Darstellung. Die Abbildung 3.1 auf Seite 22 zeigt im oberen und unteren Bereich die beiden Verzeichnisse `ofs_index` und `ofs_open_files`. Beide haben eine Referenz auf die in der Mitte abgebildete Datei. Dort ist zu erkennen, wie die `ofs_file`-Struktur in die erste Speicherseite eingebettet ist, die dann noch 3584 Bytes zur Speicherung von Nutzdaten übrig hat. Die Referenzierung dieses Datenspeichers ist in `ofs_files.data[0]` abgelegt. Ebenfalls wird aufgezeigt, wie ein Verweis auf eine weitere Speicherseite aussieht, die nun 4096 Bytes für Nutzdaten zur Verfügung stellt. Die linke, obere Ecke der Speicherseiten soll jeweils den Anfang, also die erste Speicheradresse, symbolisieren.

## 3.3. Hilfsfunktionen

Die hier beschriebenen und implementierten Funktionen stellen selbst keine direkte Schnittstelle zwischen Betriebssystem und Overlay-Dateisystem dar, sind aber notwendig um Berechnung und wiederkehrende Routinen zu kapseln. Zwar werden einzelne Funktionen auch vom

Kernel aus aufgerufen, aber an dieser Stelle ist die Bezeichnung „Schnittstelle“ so zu verstehen, dass es keine korrespondierende `u_*()`-Funktion im Kernel gibt. Für `u_open()` wäre dies beispielsweise `ofs_open()`.

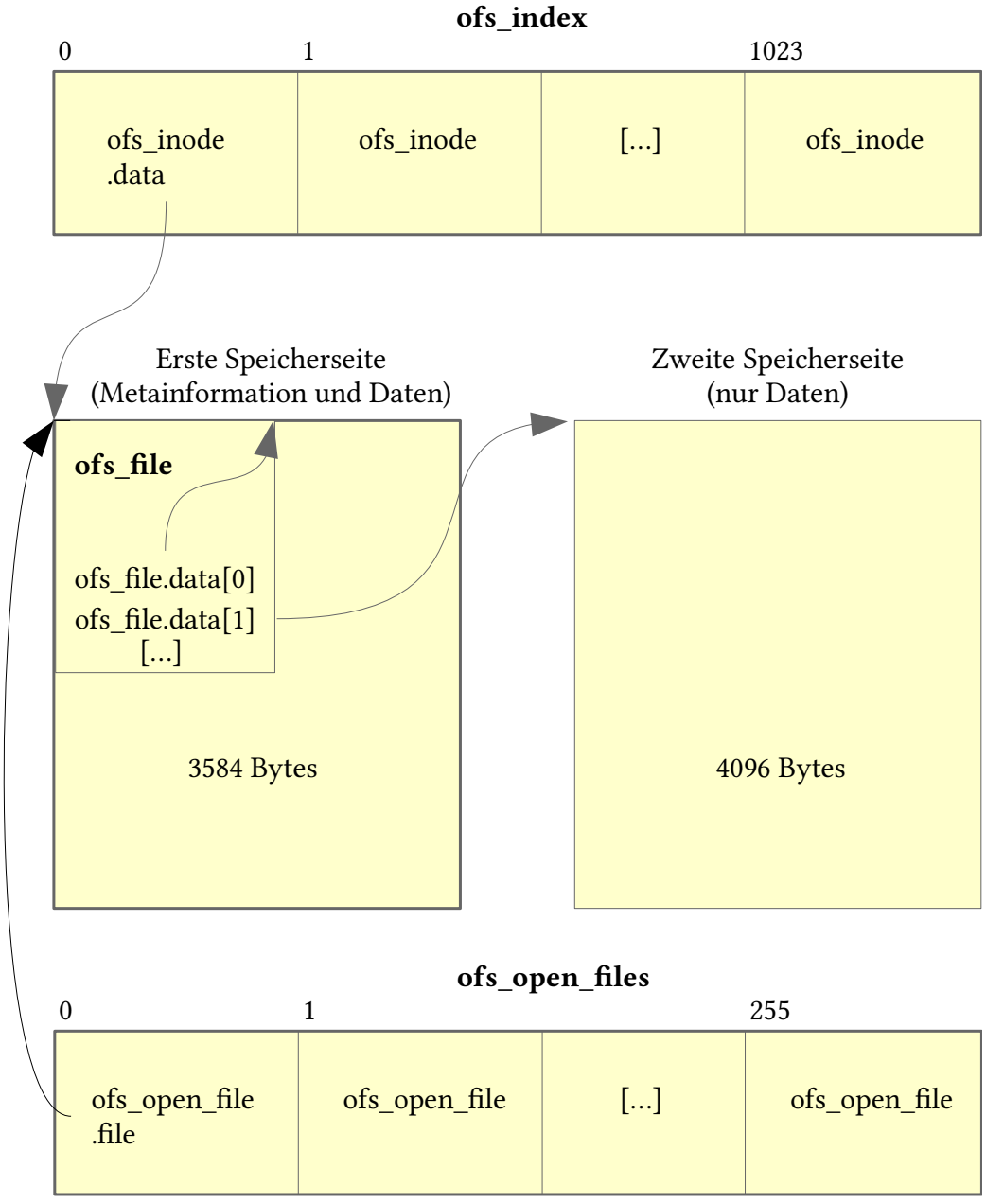


Abbildung 3.1.: Übersicht des Zusammenhangs der Datenstrukturen im Overlay-Dateisystem

### 3.3.1. Abfragen von Eigenschaften

Funktionen aus diesem Abschnitt dienen dazu, zuverlässig bestimmte Eigenschaften im Overlay-Dateisystem abzufragen. Es wird davon ausgegangen, dass mit den Ergebnissen dieser Funktionen weitergearbeitet werden kann und Fehler eine tatsächliche Ausnahme darstellen.

#### ofs_get_free_fd

Um eine Datei öffnen zu können wird ein freier Eintrag in `ofs_open_files` benötigt. Die Funktion `ofs_get_free_fd()` durchsucht dieses Verzeichnis und liefert den Index eines freien Eintrags zurück. Ein freier Eintrag ist durch einen Zeiger auf `0` in `ofs_open_file.file` zu erkennen.

Zunächst wird der Prototyp der Funktion angelegt. Die Funktion benötigt keine Argumente und hat als Rückgabewert `int`.

⟨*ofs prototypes* 23a⟩≡ (16a) 23d▷ [23a]
```
int ofs_get_free_fd();
```
Uses `ofs_get_free_fd` 23b.

Die Funktion selbst besteht nur aus einer Schleife, die die Einträge in `ofs_open_files` von `0` bis `OFS_MAX_OPEN_FILES` durchläuft. Wird in der Schleife kein freier Eintrag gefunden, so wird `-1` zurückgegeben um dies zu signalisieren.

⟨*ofs functions* 23b⟩≡ (16b) 24a▷ [23b]
```
int ofs_get_free_fd() {
    int i = 0;
    for(i = 0; i < OFS_MAX_OPEN_FILES; i++) {
        ⟨ofs_get_free_fd loop 23c⟩
    }
    return -1;
}
```
Defines:
`ofs_get_free_fd`, used in chunks 23a, 37b, and 72.
Uses `OFS_MAX_OPEN_FILES` 21b.

In der Schleife wird nun überprüft, ob das `file`-Attribut eines Eintrags auf `0` zeigt. Ist dies der Fall, so ist ein freier Eintrag gefunden und der Index, repräsentiert durch `i`, wird zurückgegeben.

⟨*ofs_get_free_fd loop* 23c⟩≡ (23b) [23c]
```
if(ofs_open_files[i].file == 0) { return i; }
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a and `ofs_open_files` 21c 21d.

#### ofs_get_free_inode

Diese Funktion ist dafür zuständig in `ofs_index` einen freien Eintrag zu suchen und im Erfolgsfall dessen Index zurückzugeben. Ein freier Eintrag ist an einer `0` in `ofs_inode.data` zu erkennen.

Die Funktion selbst benötigt keinen Parameter und hat als Rückgabetyp `int`. Der Prototyp muss also folgendermaßen angelegt werden:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁23a 24b▷ [23d]
```
int ofs_get_free_inode();
```
Uses `ofs_get_free_inode` 24a.

In der Funktion existiert nur eine Schleife, die alle `ofs_inode` Einträge in `ofs_index` durchläuft und deren `data`-Attribut überprüft. Ist ein Eintrag mit einer `0` in `data` gefunden, so wird der Index in `i` zurückgeliefert. Wird kein freier Eintrag gefunden ist `-1` der Rückgabewert.

⟨*ofs functions* 23b⟩+≡ (16b) ◁23b 24c▷ [24a]

```
int ofs_get_free_inode() {
    int i = 0;
    for(i = 0; i < OFS_MAX_FILES; i++) {
        if(ofs_index[i].data == 0) { return i; }
    }
    return -1;
}
```

Defines:
`ofs_get_free_inode`, used in chunks 23d, 28d, 46d, and 72.
Uses `ofs_index` 19a and `OFS_MAX_FILES` 19b.

## ofs_get_free_dir_entry

Das Schreiben von `dir_entry`-Strukturen soll den Speicher möglichst effizient nutzen. Daher ist es erforderlich, nach eventuellen Lücken in der `data`-Sektion eines `ofs_file`, welches als Verzeichnis genutzt wird, zu suchen. Als Parameter erhält die Funktion den Pfadnamen `path` des Verzeichnisses. Zurückgegeben wird die Index-Nummer eines freien Eintrags.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁23d 25a▷ [24b]

```
int ofs_get_free_dir_entry(const char *path);
```

Uses `ofs_get_free_dir_entry` 24c.

Die Funktion öffnet nun das Verzeichnis und liest in einer Schleife die enthaltenen `dir_entry`-Strukturen. Diese werden in `buf` zwischengespeichert. Die Anzahl an gelesenen Bytes wird in `read_bytes` gespeichert. Da ein Verzeichnis immer 2 Einträge, „.“ und „..“, enthält, kann mit der Suche bei `i = 2` begonnen werden. Hierfür muss zum einen `i` auf `2` und zum anderen der Dateizeiger mit `ofs_lseek()` entsprechend verschoben werden.

⟨*ofs functions* 23b⟩+≡ (16b) ◁24a 25b▷ [24c]

```
int ofs_get_free_dir_entry(const char *path) {
    int fd = ofs_open(path, O_RDONLY);
    int i = 2; int read_bytes = 0;
    char buf[OFS_DIR_ENTRY_SIZE] = {0};
    ofs_lseek(fd, i * OFS_DIR_ENTRY_SIZE, SEEK_SET);
    ⟨ofs_get_free_dir_entry search loop 24d⟩
    ofs_close(fd);
    return i;
}
```

Defines:
`ofs_get_free_dir_entry`, used in chunks 24b and 31e.
Uses `O_RDONLY` 73, `ofs_close` 38d, `OFS_DIR_ENTRY_SIZE` 20c, `ofs_lseek` 44b, `ofs_open` 37b, and `SEEK_SET` 73.

Die Suchschleife selbst gestaltet sich relativ einfach. In der Abbruchbedingung wird ein `dir_entry` gelesen und `read_bytes` geprüft. Sollte `read_bytes` gleich `0` sein, so ist ein freier Eintrag am Ende von `data` gefunden worden. In der Schleife selbst wird nun der Inhalt eines `dir_entry` geprüft. Ist sowohl `dir_entry.inode` als auch `dir_entry.filename` gleich `0`, so ist der Eintrag als leer zu betrachten und die Suchschleife zu beenden.

⟨*ofs_get_free_dir_entry search loop* 24d⟩≡ (24c) [24d]

```
for(; (read_bytes = ofs_read(fd, buf, OFS_DIR_ENTRY_SIZE)) > 0; i++) {
    struct dir_entry *entry = (struct dir_entry *) buf;
    if(entry->inode == 0 && entry->filename[0] == '\0') break;
}
```

Uses `entry` 47a, `OFS_DIR_ENTRY_SIZE` 20c, and `ofs_read` 42b.

### ofs_get_dir_and_filename

Bei der Arbeit mit Verzeichnissen ist es oftmals von Nöten aus einer absoluten Pfadangabe den Verzeichnis- und Dateinamen separat zu erhalten. Dazu werden der Funktion `ofs_get_dir_and_filename()` drei Paramter übergeben. Der erste Parameter `path` ist der absolute Dateifpad. Als nächstes folgen der Verzeichnisname `dirname` und der Dateiname `filename`. Auf Fehlerüberprüfung und RÜckgabe wird an dieser Stelle verzichtet. Der resultierende Prototyp sieht nun folgendermaßen aus:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁24b 25c▷ [25a]
```
void ofs_get_dir_and_filename(const char *path, char *dirname, char *filename);
```
Uses `dirname` 37d 47a 49a and `ofs_get_dir_and_filename` 25b.

Der Ablauf der Funktion selbst ist relativ einfach gehalten. Zuerst wird eine Kopie von `path` in `dirname` angelegt und der Zeiger `ptr_path` auf das Ende dieser Kopie gesetzt. Anschließend wird `dirname` nach dem nächsten `'/'` durchsucht – da ein absoluter Pfad immer mit einem `'/'` beginnt, wird automatisch ein Unterlaufen des Puffers verhindert. Ist der `'/'` gefunden, wird dieser einfach mit `'\0'` ersetzt, sodass nun in `dirname` schon der 0-terminierte, richtige Name enthalten ist. Zum Schluss wird noch der Dateiname, beginnend ab `ptr_path`, in `filename` kopiert und die Funktion hat ihre Arbeit erledigt.

⟨*ofs functions* 23b⟩+≡ (16b) ◁24c 25d▷ [25b]
```
void ofs_get_dir_and_filename(const char *path, char *dirname, char *filename) {
    char *ptr_path;
    strncpy(dirname, path, 247);
    ptr_path = &dirname[247];
    while(*ptr_path != '/') ptr_path-;
    *ptr_path = '\0'; ptr_path++;
    strncpy(filename, ptr_path, 64);
}
```
Defines:
`ofs_get_dir_and_filename`, used in chunks 25a, 34b, 37d, 47a, and 49a.
Uses `dirname` 37d 47a 49a.

### ofs_get_path_from_index

Diese Funktion soll anhand eines übergebenen Index `index` den entsprechenden Dateinamen `path` aus der `ofs_inode`-Struktur in den Puffer `buf` schreiben. Da bei diesem simplen Lesevorgang nicht mit einem Fehler gerechnet wird, ist der Rückgabetyp `void`.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁25a 26a▷ [25c]
```
void ofs_get_path_from_index(int index, char *buf);
```
Uses `index` 47d and `ofs_get_path_from_index` 25d.

Die Funktion führt nur ein einfaches `strncpy()` der Zeichenketten von `ofs_index[index].path` nach `buf` durch.

⟨*ofs functions* 23b⟩+≡ (16b) ◁25b 26b▷ [25d]
```
void ofs_get_path_from_index(int index, char *buf) {
    strncpy(buf, ofs_index[index].path, 248);
}
```
Defines:
`ofs_get_path_from_index`, used in chunks 25c, 59a, and 72.
Uses `index` 47d and `ofs_index` 19a.

### 3.3.2. Suchen im Overlay-Dateisystem

Im Gegensatz zu den Funktionen im vorherigen Abschnitt, wird bei den Such-Funktionen nicht davon ausgegangen, dass auch tatsächlich ein Resultat geliefert wird mit dem unmittelbar weitergearbeitet werden kann. Bei `ofs_find_pathname` kann beispielsweise nicht damit gerechnet werden, dass mit dem zurückgelieferten Index ein erfolgreicher Zugriff auf das Dateisystem gelingen kann.

#### ofs_find_pathname

Ziel dieser Funktion ist es, den absoluten Dateinamen `path` in `ofs_index` zu suchen und im Erfolgsfall dessen Index zurückzugeben.

Hierfür is folgender Prototyp anzulegen:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁25c 26d▷ [26a]
```
int ofs_find_pathname(const char* path);
```
Uses `ofs_find_pathname` 26b.

Die Implementierung der Funktion gestaltet sich derart, dass in einer Schleife alle `ofs_inode` Einträge in `ofs_index` mit dem Übergebenen Pfad verglichen werden. Im Erfolgsfall wird mit `i` der Index zurückgegeben und im Fehlerfall, Eintrag nicht gefunden, `-1`.

⟨*ofs functions* 23b⟩+≡ (16b) ◁25d 27a▷ [26b]
```
int ofs_find_pathname(const char* path) {
    int i = 0;
    for(i = 0; i < OFS_MAX_FILES; i++) {
        ⟨ofs_find_pathname loop 26c⟩
    }
    return -1;
}
```
Defines:
`ofs_find_pathname`, used in chunks 26a, 32, 37, 46–51, 53, 56, 57b, 72, and 77–83.
Uses `OFS_MAX_FILES` 19b.

In der Schleife selbst findet nun der Vergleich der beiden Zeichenketten `path` und `ofs_inode.path` statt.

⟨*ofs_find_pathname loop* 26c⟩≡ (26b) [26c]
```
if(strcmp(path, ofs_index[i].path) == 0) { return i; }
```
Uses `ofs_index` 19a.

#### ofs_find_inode

Diese Funktion soll, ähnlich wie `ofs_find_pathname`, `ofs_index` durchlaufen und überprüfen, ob bereits die Daten eines Minix-Inodes gespeichert wurden und falls ja, dessen Index in `ofs_index` zurückliefern.

Als Parameter hat diese Funktionen die Minix-Inode-Nummer `mx_inode` und als Rückgabewert den Index oder `-1` falls die Inode-Nummer nicht existiert.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁26a 27b▷ [26d]
```
int ofs_find_inode(int mx_inode);
```
Uses `ofs_find_inode` 27a.

Die Implementierung ist analog zu `ofs_find_pathname` nur dass nicht die Pfadangabe sondern die Inode-Nummer verglichen werden.

⟨*ofs functions* 23b⟩+≡ (16b) ◁26b 27c▷ [27a]

```
int ofs_find_inode(int mx_inode) {
    int i = 0;
    for(i = 0; i < OFS_MAX_FILES; i++) {
        if(ofs_index[i].inode == mx_inode) { return i; }
    }
    return -1;
}
```

Defines:
`ofs_find_inode`, used in chunks 26d, 57a, and 72.
Uses `ofs_index` 19a and `OFS_MAX_FILES` 19b.

### ofs_find_open_file

Um später überprüfen zu können, ob eine Datei schon geöffnet ist, soll `ofs_find_open_file()` implementiert werden. Als Parameter wird ein Zeiger `file` auf eine `ofs_file`-Struktur erwartet. Der Rückgabewert ist `-1` oder der Index, für den Fall, dass die Datei bereits geöffnet ist.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁26d 28a▷ [27b]

```
int ofs_find_open_file(struct ofs_file *file);
```

Uses `file` 32b 33c 38b 44c 47d 50a 53a and `ofs_find_open_file` 27c.

Wie bei den anderen `ofs_find_*`-Funktionen ist die Implementierung wieder eine Schleife, die alle Einträge durchläuft und dabei einen Vergleich mit `file` vornimmt. Diesmal wird allerdings nicht `ofs_index` sondern `ofs_open_files` durchlaufen.

⟨*ofs functions* 23b⟩+≡ (16b) ◁27a 28b▷ [27c]

```
int ofs_find_open_file(struct ofs_file *file) {
    int i = 0;
    for(i = 0; i < OFS_MAX_OPEN_FILES; i++) {
        if(ofs_open_files[i].file == file) { return i; }
    }
return -1;
}
```

Defines:
`ofs_find_open_file`, used in chunks 27b and 48b.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `OFS_MAX_OPEN_FILES` 21b, and `ofs_open_files` 21c 21d.

## 3.3.3. Anlegen von Dateien

Funktionen in diesem Abschnitt erledigen sowohl Speicherallokierung für neue Dateien, als auch das Eintragen von Informationen in Inodes im `ofs_index`.

### ofs_new_inode_file

Damit tatsächlich Daten im Overlay-Dateisystem gespeichert werden können, muss die Möglichkeit bestehen, zu den bisher statisch angelegten Datentypen auch Speicher im RAM dynamisch zu allokieren. Diese Funktion weist einem `ofs_inode.data`- Eintrag im `ofs_index` eine neue `ofs_file` zu.

Als Parameter erwartet `ofs_new_inode_file` den Index in `ofs_index`. Im Fehlerfall, dass keine neue Speicherseite mehr verfügbar ist, wird `-1` zurückgegeben, ansonsten `0`. Der Prototyp gestaltet sich also folgendermaßen:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁27b 28c▷ [28a]
```
int ofs_new_inode_file(int index);
```
Uses `index` 47d and `ofs_new_inode_file` 28b.

Die Funktion muss mehrere Schritte abarbeiten. Zu erst wird versucht eine neue Speicherseite anzufordern und deren Adresse in `new_page` zu speichern. Anschließend wird eine neue `ofs_file`-Struktur erzeugt und deren `ofs_file.data[0]`-Eintrag auf `new_page` + `OFS_DATA_OFFSET` gesetzt. Danach wird `temp_file` an den Anfang der neuen Speicherseite `new_page` kopiert. Abschließend wird noch `ofs_index[index].data` auf `new_page` gesetzt, um die Verbindung zum Verzeichnis herzustellen.

⟨*ofs functions* 23b⟩+≡ (16b) ◁27c 28d▷ [28b]
```
int ofs_new_inode_file(int index) {
    void *new_page = request_new_page();
    if(new_page == 0) { return -1; }

    struct ofs_file temp_file = {0};
    temp_file.data[0] = (char *) new_page + OFS_DATA_OFFSET;

    memcpy(new_page, (void *) &temp_file, OFS_DATA_OFFSET);

    ofs_index[index].data = (struct ofs_file *) new_page;

return 0;
}
```
Defines:
`ofs_new_inode_file`, used in chunks 28 and 72.
Uses `index` 47d, `memcpy` 41a 43b, `OFS_DATA_OFFSET` 19c, and `ofs_index` 19a.

### ofs_create_empty_file

Mit `ofs_new_inode_file` besteht schon die Möglichkeit einen neuen Datenbreich in einem `ofs_inode` einzutragen. `ofs_create_empty_file` führt nun die nötigen Operationen aus, damit auch im `ofs_index` die richtigen Informationen zu einer neuen Datei stehen – etwa die Zeit der Erstellung etc.

Benötigt wird dafür zum einen ein Dateiname `path` und der Zugriffsmodus `mode` als Parameter für die Funktion. Rückgabetyp soll der Index im `ofs_index` sein, um zum Beispiel mit der eben erstellen Datei gleich weiter arbeiten zu können. Der Prototyp hat also folgende Gestalt:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁28a 29c▷ [28c]
```
int ofs_create_empty_file(const char *path, int mode);
```
Uses `ofs_create_empty_file` 28d.

Die Funktion selbst muss nun erst einmal einen freien `ofs_inode`-Eintrag suchen und dort mittels `ofs_create_inode_file` eine neue Datenstruktur anlegen. Wird kein freier Eintrag gefunden, wird `-1` zurückgegeben - in jedem anderen Fehlerfall ebenso.

⟨*ofs functions* 23b⟩+≡ (16b) ◁28b 30a▷ [28d]
```
int ofs_create_empty_file(const char *path, int mode) {
    int free_idx = ofs_get_free_inode();
    if(free_idx < 0) { return -1; }

    int ret_new_inode_file = ofs_new_inode_file(free_idx);
```

```
    if(ret_new_inode_file < 0) { return -1; }
    ⟨ofs_create_empty_file set path 29a⟩
    ⟨ofs_create_empty_file set meta data 29b⟩

    return free_idx;
}
```

Defines:
`ofs_create_empty_file`, used in chunks 28c, 37c, and 72.
Uses `ofs_get_free_inode` 24a and `ofs_new_inode_file` 28b.

Nun ist der Eintrag bereit dafür mit den nötigen Informationen gefüllt zu werden. Als erstes wird der Dateiname aus `path` übernommen.

⟨*ofs_create_empty_file set path* 29a⟩≡ (28d) [29a]
```
strcpy(ofs_index[free_idx].path, path);
```
Uses `ofs_index` 19a.

Anschließend werden noch die Attribute in `ofs_inode.data` gesetzt.

⟨*ofs_create_empty_file set meta data* 29b⟩≡ (28d) [29b]
```
struct ofs_file *new_file = ofs_index[free_idx].data;
int current_systime = system_time;
new_file->i_mode = S_IFREG | mode;
new_file->i_nlinks = 1;
new_file->i_uid = thread_table[current_task].uid;
new_file->i_gid = thread_table[current_task].gid;
new_file->i_size = 0;
new_file->i_atime = current_systime;
new_file->i_ctime = current_systime;
new_file->i_mtime = current_systime;
```
Defines:
`current_systime`, never used.
`new_file`, never used.
Uses `ofs_index` 19a and `S_IFREG` 73.

Da sich die Metainformationen nicht im `ofs_inode`-Eintrag sondern eine Ebene darunter befinden, wird `new_file->i_nlinks` standardmäßig auf `1` gesetzt, da es immer mindestens einen Verweis auf diesen `ofs_file`-Eintrag gibt. Ist `i_nlinks` gleich `0` so kann der Speicher wieder freigegeben werden, da keine weiteren Verweise mehr existieren.

### 3.3.4. Bearbeiten von Dateien

Mit „Bearbeiten“ ist an dieser Stelle nicht das Lesen und Schreiben von Nutzdaten von und in Dateien gemeint, sondern die tatsächliche Manipulation der Metainformationen oder des Speicherbereichs der Datei.

#### ofs_write_stat

Da zu der Grundfunktionalität des Overlay-Dateisystems das Kopieren von Dateien des darunterliegenden Dateisystems gehört, müssen auch deren Metainformationen mit kopiert werden und nicht nur der eigentliche Dateinhalt.

`ofs\_write\_stat` erhält als Paramter den Index `idx` in `ofs_index` und einen Zeiger `buf` auf einen Puffer, der die Informationen bereit hält. Als Rückgabetyp dient `void`, da nicht von fehleranfälligen Operationen ausgegangen wird.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁28c 30b▷ [29c]
```
void ofs_write_stat(int idx, struct stat *buf);
```
Uses `idx` 53a and `ofs_write_stat` 30a.

Die Funktion an sich kopiert nur die Informationen aus dem Puffer `buf` in die entsprechenden Felder in `ofs_inode` und `ofs_file`.

⟨*ofs functions* 23b⟩+≡ (16b) ◁28d 30c▷ [30a]

```
void ofs_write_stat(int idx, struct stat *buf) {
    struct ofs_file *file = ofs_index[idx].data;
    ofs_index[idx].inode = buf->st_ino;
    file->i_mode = buf->st_mode;    file->i_size = buf->st_size;
    file->i_uid = buf->st_uid;      file->i_gid = buf->st_gid;
    file->i_atime = buf->st_atime;  file->i_ctime = buf->st_ctime;
    file->i_mtime = buf->st_mtime;
}
```

Defines:
`ofs_write_stat`, used in chunks 29c, 57b, 59a, 72, and 83b.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `idx` 53a, `ofs_index` 19a, `st_atime` 73, `st_ctime` 73, `st_gid` 73, `st_ino` 73, `st_mode` 73, `st_mtime` 73, `st_size` 73, and `st_uid` 73.

### ofs_fill_gaps

Durch Verschiebung des Dateizeigers, z.B. mit `ofs_lseek()`, kann es passieren, dass der Datenbereich in einem `ofs_file` nicht durchgängig belegt ist. Um diesen eventuellen Missstand zu korrigieren, füllt `ofs_fill_gaps()` die fehlenden Datenbereiche mit leeren Speicherseiten auf.

Als Parameter benötigt die Funktion einen Zeiger auf eine `ofs_file`-Struktur sowie den Index `page_index` der letzten Seite, auf die geschrieben wurde. Für den Fehlerfall, dass keine neue Seite angefordert werden kann, wird `-1` zurückgegeben.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁29c 31d▷ [30b]

```
int ofs_fill_gaps(struct ofs_file *file, int page_index);
```

Uses `file` 32b 33c 38b 44c 47d 50a 53a and `ofs_fill_gaps` 30c.

Die Funktion läuft nun von `page_index` rückwärts durch den `data`-Bereich von `file` und prüft deren Speicheradressen. Sollte eine `0` gefunden werden, wird versucht mit `request_new_page()` eine neue Speicherseite dort zu hinterlegen. Schlägt dies fehl, wird mit `-1` abgebrochen.

⟨*ofs functions* 23b⟩+≡ (16b) ◁30a 31e▷ [30c]

```
int ofs_fill_gaps(struct ofs_file *file, int page_index) {
    while(page_index > 0) {
        if(file->data[page_index] == 0) {
           file->data[page_index] = request_new_page();
            if(file->data[page_index] == 0) {
                return -1;
            }
        }
        page_index-;
    }
    return 0;
}
```

Defines:
`ofs_fill_gaps`, used in chunks 30b, 41c, and 45c.
Uses `file` 32b 33c 38b 44c 47d 50a 53a.

### Zeitstempel setzen

Beim Anlegen von Dateien und bei `ofs_write_stat` sind bereits drei Zeitstempel in Erscheinung getreten, die später auch modifiziert werden. Für eine bessere Lesbarkeit des Codes

werden diese Modifikationen hier in Code-Chunks hinterlegt. Voraussetzung ist jedoch, dass es einen Zeiger `file` auf eine `ofs_file`-Struktur gibt.

⟨*update atime* 31a⟩≡ (42b) [31a]
```
file->i_atime = system_time;
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a.

⟨*update mtime* 31b⟩≡ (39c 45b) [31b]
```
file->i_mtime = system_time;
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a.

⟨*update ctime* 31c⟩≡ (46–48 50c 51b) [31c]
```
file->i_ctime = system_time;
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a.

### 3.3.5. Bearbeiten von Verzeichnissen

Beim Verwalten von Verzeichnissen fallen immer wieder Operationen an, die an dieser Stelle in eigene Funktionen gekapselt wurden, um sie bei Bedarf auch außerhalb der regulären Anwendungsbereiche, z.B. `ofs_mkdir()`, nutzen zu können.

#### ofs_write_dir_entry

Zum Schreiben von `dir_entry`-Strukturen, soll die Funktion `ofs_write_dir_entry()` dienen. Sie erwartet als Parameter das Verzeichnis als Pfadangabe `path` in welches die, als Parameter `entry` übergebene, `dir_entry`-Struktur geschrieben wird. Der Rückgabewert ist im Erfolgsfall `0` und im Fehlerfall `-1`.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁30b 32c▷ [31d]
```
int ofs_write_dir_entry(const char *path, struct dir_entry *entry);
```
Uses `entry` 47a and `ofs_write_dir_entry` 31e.

Das Schreiben selbst erfolgt nun nach folgendem Schema: das Verzeichnis wird geöffnet, auf ein eventuell zu schreibendes Duplikat überprüft, nach einem freien Platz gesucht und dessen Index in `idx` gespeichert. Anschließend wird der Dateizeiger von `fd` auf die Position für `idx` gesetzt und `entry` geschrieben. Danach wird `fd` wieder geschlossen und `bytes_written` geprüft. Ist `bytes_written` größer als `-1`, so war der Vorgang erfolgreich.

⟨*ofs functions* 23b⟩+≡ (16b) ◁30c 32d▷ [31e]
```
int ofs_write_dir_entry(const char *path, struct dir_entry *entry) {
    int idx = 0; int fd = 0; int bytes_written = 0;
    fd = ofs_open(path, O_RDWR);
    if(fd < 0) return -1;
    ⟨ofs_write_dir_entry check duplicate 32a⟩
    idx = ofs_get_free_dir_entry(path);
    ofs_lseek(fd, idx*OFS_DIR_ENTRY_SIZE, SEEK_SET);
    bytes_written = ofs_write(fd, (void *) entry, OFS_DIR_ENTRY_SIZE);
    ofs_close(fd);
    if(bytes_written < 0) return -1;
    ⟨ofs_write_dir_entry update link count 32b⟩
    return 0;
}
```
Defines:
`ofs_write_dir_entry`, used in chunks 31d, 34b, 37e, 47a, 72, and 83b.
Uses `entry` 47a, `idx` 53a, `O_RDWR` 73, `ofs_close` 38d, `OFS_DIR_ENTRY_SIZE` 20c, `ofs_get_free_dir_entry` 24c, `ofs_lseek` 44b, `ofs_open` 37b, `ofs_write` 39c, and `SEEK_SET` 73.

Bevor das Verzeichnis einen neuen Eintrag erhält wird überprüft, ob es dieser nicht bereits existiert. Dazu wird das Verzeichnis durchwandert und die Dateinamen darin mit dem zu Schreibenden verglichen. Ist eine Übereinstimmung gefunden, kann das Verzeichnis geschlossen und 0 zurückgegeben werden.

⟨*ofs_write_dir_entry check duplicate* 32a⟩≡ (31e) [32a]
```
int bytes_read = 0;
struct dir_entry dentbuf = {0};
while((bytes_read = ofs_read(fd, (void *) &dentbuf, OFS_DIR_ENTRY_SIZE)) > 0) {
    if(strcmp(dentbuf.filename, entry->filename) == 0) {
        ofs_close(fd);
        return 0;
    }
    idx++;
    ofs_lseek(fd, idx*OFS_DIR_ENTRY_SIZE, SEEK_SET);
}
```
Defines:
`bytes_read`, used in chunks 42 and 43c.
`dentbuf`, used in chunks 51d and 52a.
Uses `entry` 47a, `idx` 53a, `ofs_close` 38d, `OFS_DIR_ENTRY_SIZE` 20c, `ofs_lseek` 44b, `ofs_read` 42b, and `SEEK_SET` 73.

Was nach erfolgreichem Schreiben noch erledigt werden muss, ist das Erhöhen der Link-Anzahl des Verzeichnisses. Dazu wird einfach direkt die `ofs_file`-Struktur manipuliert. Dieses Vorgehen ist vorallem dafür gedacht, dass bei einem späteren Löschen des Verzeichnis überprüft werden kann, ob dieses leer ist oder nicht.

⟨*ofs_write_dir_entry update link count* 32b⟩≡ (31e) [32b]
```
idx = ofs_find_pathname(path);
struct ofs_file *file = ofs_index[idx].data;
file->i_nlinks++;
```
Defines:
`file`, used in chunks 16b, 21a, 23c, 27, 30, 31, 38d, 40–43, 45, 46, 48–51, 53c, 73, and 78b.
Uses `idx` 53a, `ofs_find_pathname` 26b, and `ofs_index` 19a.

## ofs_remove_from_dir

Bisher ist es nur möglich, neue Einträge in ein Verzeichnis zu schreiben, aber beim Löschen von Dateien ist es auch nötig, diese Einträge wieder zu entfernen. Die Funktion soll nun so implementiert werden, dass sie einen Verzeichnisnamen `dirname` und Dateinamen `filename` übergeben bekommt und diesen löscht, falls überhaupt ein Verzeichnis und dieser Dateiname existieren. Beim Funktionsaufruf soll also keine Überprüfung auf einen möglichen Fehler stattfinden müssen. Daraus ergibt sich nachfolgender Prototyp:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁31d 34a▷ [32c]
```
void ofs_remove_from_dir(const char *dirname, const char *filename);
```
Uses `dirname` 37d 47a 49a and `ofs_remove_from_dir` 32d.

Beim Aufruf der Funktion, wird nun als erstes geprüft, ob überhaupt ein Verzeichnis existiert - falls nicht, ist auch nichts weiter zu tun. Wird ein Verzeichnis gefunden, so wird dieses geöffnet und durchsucht. Da der exakte Index zum Löschen eines Eintrags benötigt wird, kann hierfür nicht `ofs_getdent` genutzt werden, da dort etwaige Lücken, die durch das Löschen entstehen, übersprungen werden. Zum Suchen wird nun einfach das Verzeichnis geöffnet und der erste Verzeichniseintrag gelesen.

⟨*ofs functions* 23b⟩+≡ (16b) ◁31e 34b▷ [32d]
```
void ofs_remove_from_dir(const char *dirname, const char *filename) {
    int dir_exists = ofs_find_pathname(dirname);
```

```
    if(dir_exists > -1) {
        int fd = 0; int read_bytes = 0; int index = 0;
        struct dir_entry entry = {0};
        fd = ofs_open(dirname, O_RDWR);
        if(fd < 0) return;
        ofs_lseek(fd, index*OFS_DIR_ENTRY_SIZE, SEEK_SET);
        read_bytes = ofs_read(fd, (void *) &entry, OFS_DIR_ENTRY_SIZE);
        ⟨ofs_remove_from_dir look for filename 33a⟩
        ⟨ofs_remove_from_dir delete filename 33b⟩
        ⟨ofs_remove_from_dir update link count 33c⟩
        ofs_close(fd);
    }
}
```

Defines:
ofs_remove_from_dir, used in chunks 32c, 49a, and 53c.
Uses dir_exists 37e, dirname 37d 47a 49a, entry 47a, index 47d, O_RDWR 73, ofs_close 38d, OFS_DIR_ENTRY_SIZE 20c, ofs_find_pathname 26b, ofs_lseek 44b, ofs_open 37b, ofs_read 42b, and SEEK_SET 73.

Jetzt wird in einer Schleife geprüft, ob Bytes gelesen wurden und ob `filename` eine Übereinstimmung erzielt.

⟨*ofs_remove_from_dir look for filename* 33a⟩≡ (32d) [33a]

```
for(index = 1; read_bytes > 0 &&
                    strcmp(entry.filename, filename) != 0; index++) {
    ofs_lseek(fd, index*OFS_DIR_ENTRY_SIZE, SEEK_SET);
    read_bytes = ofs_read(fd, (void *) &entry, OFS_DIR_ENTRY_SIZE);
}
```

Uses entry 47a, index 47d, OFS_DIR_ENTRY_SIZE 20c, ofs_lseek 44b, ofs_read 42b, and SEEK_SET 73.

Da die Schleife auch beendet werden kann, wenn das Ende des Verzeichnisses erreicht wurde, kann der Verzeichniseintrag nur dann gelöscht werden, wenn `read_bytes` größer `0` ist. Zum Löschen selbst wird einfach eine `OFS_DIR_ENTRY_SIZE` lange Folge von `'\0'` an die richtige Position in der Datei geschrieben.

⟨*ofs_remove_from_dir delete filename* 33b⟩≡ (32d) [33b]

```
if(read_bytes > 0) {
    index-;
    char null_string[OFS_DIR_ENTRY_SIZE] = { 0 };
    ofs_lseek(fd, index*OFS_DIR_ENTRY_SIZE, SEEK_SET);
    ofs_write(fd, (void *) null_string, OFS_DIR_ENTRY_SIZE);
}
```

Uses index 47d, OFS_DIR_ENTRY_SIZE 20c, ofs_lseek 44b, ofs_write 39c, and SEEK_SET 73.

Wie schon bei `ofs_write_dir_entry()` muss auch beim Entfernen die Link-Anzahl auf das Verzeichnis entsprechend angepasst werden.

⟨*ofs_remove_from_dir update link count* 33c⟩≡ (32d) [33c]

```
struct ofs_file *file = ofs_index[dir_exists].data;
file->i_nlinks-;
```

Defines:
file, used in chunks 16b, 21a, 23c, 27, 30, 31, 38d, 40–43, 45, 46, 48–51, 53c, 73, and 78b.
Uses dir_exists 37e and ofs_index 19a.

### ofs_append_files_to_dir

Im Overlay-Dateisystem können Dateien angelegt werden, auch wenn das Verzeichnis noch nicht in diesem existiert. Um später diese Dateien auch im Verzeichnis auflisten zu können,

ist es erforderlich diese fehlenden Einträge im Verzeichnis zu ergänzen. Einziger Paramter soll der Dateiname `path` des Verzeichnisses sein. Als Rückgabetyp dient `void`.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁32c 34c▷ [34a]
```
void ofs_append_files_to_dir(const char *path);
```
Uses ofs_append_files_to_dir 34b.

Die Funktion durchläuft nun `ofs_index` und vergleicht mit `ofs_get_dir_and_filename()`, ob der übergebene Wert in `path` mit `dirname` übereinstimmt. Ist dies der Fall, so wird ein neuer `dir_entry` in das Verzeichnis geschrieben.

⟨*ofs functions* 23b⟩+≡ (16b) ◁32d 34d▷ [34b]
```
void ofs_append_files_to_dir(const char *path) {
    for(int i = 0; i < OFS_MAX_FILES; i++) {
        char dirname[248] = {0}; char filename[248] = {0};
        if(*ofs_index[i].path != '\0') {
            ofs_get_dir_and_filename(ofs_index[i].path, dirname, filename);
            if(strcmp(path, dirname) == 0) {
                struct dir_entry entry = {0};
                entry.inode = 1;
                strncpy(entry.filename, filename, 64);
                ofs_write_dir_entry(path, &entry);
            }
        }
    }
}
```
Defines:
ofs_append_files_to_dir, used in chunks 34a, 72, and 83b.
Uses dirname 37d 47a 49a, entry 47a, ofs_get_dir_and_filename 25b, ofs_index 19a, OFS_MAX_FILES 19b, and ofs_write_dir_entry 31e.

### 3.3.6. Berechnungen für den Datenbereich

Da das hier implementierte Overlay-Dateisystem den Speicher für die Nutzdaten selbst verwaltet und nicht in einem Stück Speicher vom Betriebssystem anfordert, werden die hier beschriebenen Funktionen benötigt, um beispielsweise zu einer Position eines Dateizeigers die richtige Speicherseite zu finden.

#### ofs_position_to_data_index

Für das Overlay-Dateisystem ist es wichtig zu berechenen, auf welcher Speicherseite sich der aktuelle Dateizeiger befindet. Da die erste Seite zusätzlich mit der `ofs_file`-Struktur belegt ist, muss dieser Offset mit einkalkuliert werden.

Der Rückgabewert der Funktion soll der Index für ein `ofs_file.data` sein – also ein Integer. Als Parameter erwartet die Funtkion die Position `position` des Dateizeigers.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁34a 35a▷ [34c]
```
int ofs_position_to_data_index(int position);
```
Uses ofs_position_to_data_index 34d.

Würde die erste Speicherseite auch komplett für Daten zur Verfügung stehen, wäre die Berechnung des Index eine einfache Ganzzahldivision durch die Speicherseitengröße `PAGE_SIZE`. Da dem aber nicht der Fall ist, muss der Offset, den `ofs_file` einnimmt, vor der Division noch addiert werden.

⟨*ofs functions* 23b⟩+≡ (16b) ◁34b 35b▷ [34d]
```
int ofs_position_to_data_index(int position) {
    return (position+OFS_DATA_OFFSET)/PAGE_SIZE;
```

```
}
```

Defines:
ofs_position_to_data_index, used in chunks 34c, 40c, 42e, 45b, and 72.
Uses OFS_DATA_OFFSET 19c and PAGE_SIZE 73.

### ofs_relative_data_position

Diese Funktion dient der Umrechnung der absoluten Position des Dateizeigers zur Position des Dateizeigers innerhalb einer Speicherseite in `ofs_file.data`. Da die erste Seite einen Sonderfall darstellt, muss neben der Position `position` auch der Index der Seite `page_index` mit übergeben werden. Die Funktion könnte diesen zwar auch durch `ofs_position_to_data_index` berechnen lassen, allerdings ist davon ausgzugehen, dass `ofs_relative_data_position` nur aus Funktionen heraus aufgerufen wird, die vorher schon Kenntnis über den aktuellen Index haben und so ein Funktionsaufruf eingespart werden kann.

Als Rückgabewert hat die Funktion die relative Position des Dateizeigers innerhalb einer Speicherseite.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁34c 35c▷ [35a]
```
int ofs_relative_data_position(int page_index, int position);
```
Uses ofs_relative_data_position 35b.

Die Berechnung ist eigentlich nur eine Division mit Rest durch die Speicherseitengröße `PAGE_SIZE`. Allerdings muss der Offset der ersten Seite wieder mit berücksichtigt werden. Für den Fall, dass sich der Dateizeiger innerhalb der ersten Seite befindet, ist die relative Position gleich der absoluten. Für alle anderen Seiten muss erst die Verschiebung durch die erste Seite abgezogen werden bevor durch eine Modulo-Division mit `PAGE_SIZE` die Position ermittelt werden kann.

⟨*ofs functions* 23b⟩+≡ (16b) ◁34d 35d▷ [35b]
```
int ofs_relative_data_position(int page_index, int position) {
    if(page_index == 0) {
        return position;
    }
    return (position - OFS_FIRST_PAGE_SIZE)%PAGE_SIZE;
}
```
Defines:
ofs_relative_data_position, used in chunks 35a, 36c, 41a, 43b, 45e, and 72.
Uses OFS_FIRST_PAGE_SIZE 19c and PAGE_SIZE 73.

### ofs_absolute_data_position

Die Umrechnung von der relativen zur absoluten Position des Dateizeigers innerhalb einer Datei erfolgt in umgekehrter Weise zu `ofs_relative_data_position`. Der Prototyp ist daher analog aufgebaut:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁35a 36a▷ [35c]
```
int ofs_absolute_data_position(int page_index, int position);
```
Uses ofs_absolute_data_position 35d.

Auch die Implementierung ist sehr ähnlich, nur die Rechenoperationen sind umgekehrt. Einziger Unterschied ist, dass `page_index` um `1` vermindert werden muss, da sonst vom Ende der Seite aus gezählt werden würde.

⟨*ofs functions* 23b⟩+≡ (16b) ◁35b 36b▷ [35d]
```
int ofs_absolute_data_position(int page_index, int position) {
    if(page_index == 0) {
        return position;
```

```
    }
    page_index-;
    return (position + OFS_FIRST_PAGE_SIZE + page_index * PAGE_SIZE);
}
```

Defines:
ofs_absolute_data_position, used in chunks 35c and 72.
Uses OFS_FIRST_PAGE_SIZE 19c and PAGE_SIZE 73.

### ofs_bytes_in_page

Da beim späteren Lesen und Schreiben von Dateien nicht immer davon ausgegangen werden kann, dass sich der Dateizeiger am Anfang einer Speicherseite befindet, soll diese Funktion in Abhängigkeit der Position `position` des Zeigers, die noch verfügbaren Bytes in der Speicherseite zurückliefern. Da die Funktion aus den selben Funktionen wie `ofs_relative_data_position` heraus aufgerufen werden wird, wird zusätlich noch der Index `page_index` mit übergeben.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁35c 37a▷ [36a]

```
int ofs_bytes_in_page(int page_index, int position);
```

Uses ofs_bytes_in_page 36b.

Für die erste Speicherseite ist wieder ein Sonderfall zu treffen, dessen Berechnung aber sehr simpel ist. Befindet sich der Zeiger auf besagter Seite, so ist die Anzahl an verfügbaren Bytes einfach die Differenz aus `OFS_FIRST_PAGE_SIZE` und der aktuellen Position `position`.

⟨*ofs functions* 23b⟩+≡ (16b) ◁35d 37b▷ [36b]

```
int ofs_bytes_in_page(int page_index, int position) {
    if(page_index == 0) {
        return OFS_FIRST_PAGE_SIZE - position;
    }
    ⟨ofs_ bytes_ in_ page calculate for other pages 36c⟩
}
```

Defines:
ofs_bytes_in_page, used in chunks 36a, 40c, 42e, 45e, and 72.
Uses OFS_FIRST_PAGE_SIZE 19c.

Für alle anderen Speicherseiten ist die Berechnung ähnlich. Allerdings wird hier die Differenz aus relativer Speicherseitenposition und `PAGE_SIZE` gebildet. Zur Bestimmung der relativen Position dient die Funktion `ofs_relative_data_position`.

⟨*ofs_ bytes_ in_ page calculate for other pages* 36c⟩≡ (36b) [36c]

```
return PAGE_SIZE-ofs_relative_data_position(page_index, position);
```

Uses ofs_relative_data_position 35b and PAGE_SIZE 73.

## 3.4. Funktionen

Die in diesem Abschnitt implementierten Funktionen stellen die Hauptschnittstelle zwischen Betriebssystem und Overlay-Dateisystem zur Verfügung. Dies bedeutet konkret, dass es zu jeder `ofs_*()`-Funktion auch eine `u_*()`-Funktion im Kernel gibt, die beispielsweise durch einen Systemcall aufgerufen werden kann.

### 3.4.1. Öffnen und Schließen von Dateien

In diesem Abschnitt werden nun die beiden Funktionen `ofs_open()` und `ofs_close()` implementiert, mit denen es später möglich ist, aus den jeweiligen Systemcall-Funtionen Dateien im Overlay-Dateisystem zu öffnen und auch wieder zu schließen.

### ofs_open

Diese Funktion dient dazu eine Datei im Overlay-Dateisystem zu öffnen und gibt im Erfolgsfall einen Filedeskriptor zurück, der für die weiteren Standardfunktionen verwendet werden kann.

Als Parameter benötigt die Funktion den Dateinamen `path` und den Zugriffsmodus `mode`. Der Prototyp für diese Funktion sieht nun folgendermaßen aus:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁36a 38c▷ [37a]
```
int ofs_open(const char *path, int mode);
```
Uses `ofs_open` 37b.

Die Implementation der Funktion wird nun schrittweise durchgeführt. Zu Anfang stehen Überprüfungen an, ob die Datei bereits existiert und ob ein freier Filedeskriptor vorhanden ist. Existiert die Datei nicht und steht ein Filedeskriptor zur Verfügung, so wird versucht eine neue Datei anzulegen – falls `mode` entsprechend gesetzt wurde. Abschließend wird der `ofs_open_file`-Eintrag mit den nötigen Informationen gefüllt und der Filedeskriptor zurückgegeben.

⟨*ofs functions* 23b⟩+≡ (16b) ◁36b 38d▷ [37b]
```
int ofs_open(const char *path, int mode) {
    int fd = 0;
    int idx_ofs_index = ofs_find_pathname(path);
    fd = ofs_get_free_fd();
    ⟨ofs_ open create file if O_ CREAT 37c⟩
    ⟨ofs_ open check idx_ ofs_ index 38a⟩
    ⟨ofs_ open set open file information 38b⟩
    return fd;
}
```
Defines:
`ofs_open`, used in chunks 24c, 31e, 32d, 37a, 47c, 51d, 52c, 54d, 56, 57b, and 72.
Uses `ofs_find_pathname` 26b and `ofs_get_free_fd` 23b.

Falls die Datei nicht existiert, wird nun noch `mode` auf die `O_CREAT`-Option geprüft und ggf. die Datei neu angelegt.

⟨*ofs_ open create file if O_ CREAT* 37c⟩≡ (37b) [37c]
```
if(idx_ofs_index < 0 && (mode & O_CREAT) > 0) {
    idx_ofs_index = ofs_create_empty_file(path, S_IRUSR | S_IWUSR);
    ⟨ofs_ open insert file into dir_ entry 37d⟩
}
```
Uses `O_CREAT` 73, `ofs_create_empty_file` 28d, `S_IRUSR` 73, and `S_IWUSR` 73.

Nun wird noch der Verzeichniseintrag für die eben erstellte Datei angelegt. Dazu werden zunächst zwei Puffer-Variablen `dirname` und `filename` angelegt und mittels `ofs_get_dir_and_filename()` befüllt.

⟨*ofs_ open insert file into dir_ entry* 37d⟩≡ (37c) 37e▷ [37d]
```
char dirname[248] = {0}; char filename[64] = {0};
ofs_get_dir_and_filename(path, dirname, filename);
```
Defines:
`dirname`, used in chunks 25, 32, 34b, 37e, and 81b.
Uses `ofs_get_dir_and_filename` 25b.

Eigentlich könnte nun einfach der Verzeichniseintrag für `filename` angelegt werden, allerdings kann es vorkommen, dass das Verzeichnis noch nicht im Overlay-Dateisystem existiert. Daher wird vor dem Anlegen geprüft, ob das Verzeichnis schon vorhanden ist und nur im Erfolgsfall der Verzeichniseintrag geschrieben.

⟨*ofs_ open insert file into dir_ entry* 37d⟩+≡ (37c) ◁37d [37e]
```
int dir_exists = ofs_find_pathname(dirname);
if(dir_exists > -1) {
```

```
    struct dir_entry entry = {0};
    entry.inode = 1;
    strncpy(entry.filename, filename, 64);
    ofs_write_dir_entry(dirname, &entry);
}
```

Defines:
dir_exists, used in chunks 32d, 33c, and 83b.
Uses dirname 37d 47a 49a, entry 47a, ofs_find_pathname 26b, and ofs_write_dir_entry 31e.

Damit wäre das Erstellen einer Datei abgeschlossen und es kann mit dem normalen Vorgang fortgefahren werden. Nun wird noch einmal idx_ofs_index auf Gültigkeit überprüft. Ist dort immer noch der Wert -1 hinterlegt, war es weder möglich eine bereits existierende Datei zu öffnen noch eine neue Datei anzulegen. In diesem Fall kann die Datei nicht erfolgreich geöffnet werden und es wird -1 zurückgegeben.

⟨*ofs_ open check idx_ ofs_ index* 38a⟩≡ (37b) [38a]
```
if(idx_ofs_index < 0) { return -1; }
```

Wurde bisher noch nicht aus der Funktion gesprungen, so enthält idx_ofs_index einen gültigen Index auf einen ofs_inode-Eintrag und es kann der entsprechende Eintrag in ofs_-open_files befüllt werden.

⟨*ofs_ open set open file information* 38b⟩≡ (37b) [38b]
```
struct ofs_file *file = ofs_index[idx_ofs_index].data;
ofs_open_files[fd].file = file;
ofs_open_files[fd].mode = (short) mode;
if((mode & O_APPEND) == O_APPEND) {
    ofs_open_files[fd].pos = file->i_size + 1;
} else {
    ofs_open_files[fd].pos = 0;
}

return fd;
```

Defines:
file, used in chunks 16b, 21a, 23c, 27, 30, 31, 38d, 40–43, 45, 46, 48–51, 53c, 73, and 78b.
Uses O_APPEND 73, ofs_index 19a, and ofs_open_files 21c 21d.

## ofs_close

Das Schließen einer Datei gestaltet sich wesentlich einfacher als das Öffnen. Da die Operationen auch alle direkt im RAM ablaufen, muss sich beim Schließen nicht darum gekümmert werden, eventuelle Zwischenspeicher wieder mit der Festplatte zu synchronisieren, damit keine Daten verloren gehen.

Die Funktion erhält als einzigen Parameter den Filedeskriptor und hat als Rückgabewert 0 um Unix-Konform zu bleiben.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁37a 39b▷ [38c]
```
int ofs_close(int fd);
```

Uses ofs_close 38d.

Beim Schließen muss nur das file-Attribut des Eintrags auf 0 gesetzt werden, da dies das Kriterium für einen freien Eintrag ist. Da die anderen Werte durch ofs_open immer überschrieben werden, benötigen diese keine weitere Aufmerksamkeit.

⟨*ofs functions* 23b⟩+≡ (16b) ◁37b 39c▷ [38d]
```
int ofs_close(int fd) {
    ⟨ofs check fd range 39a⟩
    ofs_open_files[fd].file = 0;
    return 0;
```

```
}
```
Defines:
`ofs_close`, used in chunks 24c, 31, 32, 38c, 47c, 51d, 52c, 55, and 72.
Uses `file` 32b 33c 38b 44c 47d 50a 53a and `ofs_open_files` 21c 21d.

Der einzige Fehler, der hier auftreten könnte, wäre, dass `file` nicht beschreibbar ist. Allerdings hätte dies wohl als Ursache irgendeinen Hardware-Defekt, sodass davon auszugehen ist, dass eine weitere Behandlung dieses Fehlers in weiteren Programmteilen nicht mehr möglich ist. Somit kann immer `0` zurückgegeben werden.

Ein Ausnahmefall, der behandelt werden muss, ist, wenn `fd` außerhalb des gültigen Bereiches liegt. In diesem Fall wird sofort mit `-1` als Rückgabewert abgebrochen. Da diese Operation in mehreren Funktionen benötigt wird, wird diese in einen eigenen Code-Chunk gekapselt.

⟨*ofs check fd range* 39a⟩≡ (38d 39c 42b 44b 45b) [39a]
```
if(fd < 0 || fd > OFS_MAX_OPEN_FILES) { printf("wrong fd\n"); return -1; }
```
Uses `OFS_MAX_OPEN_FILES` 21b.

## 3.4.2. Schreiben und Lesen von Dateien

Neben dem Öffnen und Schließen von Dateien ist es auch erforderlich, mit diesen Dateien zu arbeiten. Deshalb werden nun Funktionen zum Schreiben `ofs_write()` und zum Lesen `ofs_read()` implementiert.

### ofs_write

Das eigentliche Schreiben von Daten in eine Datei im Overlay-Dateisystem wird durch die Funktion `ofs_write` erledigt. Sie erhält als Parameter einen Filedeskriptor `fd`, einen Puffer `buf` aus dem gelesen wird und die Anzahl an Bytes `count`, die geschrieben werden sollen. Zurück liefert die Funktion die Anzahl an geschriebenen Bytes.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁38c 42a▷ [39b]
```
int ofs_write(int fd, const void *buf, int count);
```
Uses `ofs_write` 39c.

Die Implementation der Funktion ist ein wenig umfangreicher. Im Groben sind die Schritte zum Schreiben das Überprüfen des Filedeskriptors und des Zugriffsmodus, das Definieren einiger Variablen zum Zwischenspeichern von Informationen, das eigentliche Schreiben der Daten in einer Schleife und zum Schluss das Zurückgeben der Anzahl an geschriebenen Bytes. Bevor dies geschieht, wird allerdings noch der Datenbereich auf eventuelle Lücken überprüft und die neue Dateigröße berechnet.

⟨*ofs functions* 23b⟩+≡ (16b) ◁38d 42b▷ [39c]
```
int ofs_write(int fd, const void *buf, int count) {
    ⟨ofs check fd range 39a⟩
    ⟨ofs_ write check access mode 40a⟩
    ⟨ofs_ write define buffer variables 40b⟩
    while(bytes_to_write > 0) {
        ⟨ofs_ write loop 40c⟩
    }
    ⟨ofs_ write finish steps 41c⟩
    ⟨update mtime 31b⟩
    return bytes_written;
}
```
Defines:
`ofs_write`, used in chunks 31e, 33b, 39b, 47c, 52c, and 72.
Uses `bytes_to_write` 40b.

Das Überprüfen des Filedeskriptors auf den gültigen Bereich, wurde bereits in einem anderen Chunk erledigt. Nun wird noch anhand Zugriffsmodus überprüft, ob die Datei tatsächlich zum Schreiben geöffnet wurde.

⟨*ofs_ write check access mode* 40a⟩≡ (39c) [40a]
```
if((ofs_open_files[fd].mode & O_WRONLY) != O_WRONLY &&
   (ofs_open_files[fd].mode & O_RDWR) != O_RDWR) {
    return -1;
}
```
Uses O_RDWR 73, O_WRONLY 73, and ofs_open_files 21c 21d.

Als nächstes werden ein paar Variablen definiert, damit das abarbeiten der Schleife übersichtlich gehalten werden kann. `bytes_to_write` enthält die Anzahl an noch zu schreibenden Bytes und `bytes_written` die bereits geschriebenen. In `bytes_in_page` wird gespeichert, wieviel Platz noch in der aktuellen Speicherseite `page_index` vorhanden ist. `ptr_buf` ist ein Zeiger auf den Puffer zum Lesen und `file` auf die `ofs_file`-Struktur, die zum aktuellen Filedeskriptor `fd` gehört. `current_page` ist ein Zeiger auf die aktuelle Speicherseite zum späteren Schreiben der Daten.

⟨*ofs_ write define buffer variables* 40b⟩≡ (39c) [40b]
```
int bytes_to_write = count;  int bytes_written = 0;
int bytes_in_page = 0;       int page_index = 0;
char *ptr_buf = (char* )buf; struct ofs_file *file = ofs_open_files[fd].file;
char *current_page = 0;
```
Defines:
bytes_in_page, used in chunks 40–43.
bytes_to_write, used in chunks 39–41.
current_page, used in chunks 40e, 41a, and 43.
ptr_buf, used in chunks 41–43.
Uses file 32b 33c 38b 44c 47d 50a 53a and ofs_open_files 21c 21d.

In der Schleife wird nun zuerst `page_index` und `bytes_in_page` mit den richtigen Werten befüllt. Anschließend wird überprüft, ob `page_index` im gültigen Bereich liegt und ob überhaupt noch genug Bytes zum Schreiben vorhanden wären, um `bytes_in_page` komplett zu nutzen. Falls nicht wird `bytes_in_page` entsprechend angepasst.

⟨*ofs_ write loop* 40c⟩≡ (39c) 40e▷ [40c]
```
page_index = ofs_position_to_data_index(ofs_open_files[fd].pos);
⟨ofs check page_ index range 40d⟩
bytes_in_page = ofs_bytes_in_page(page_index, ofs_open_files[fd].pos);
if(bytes_in_page > bytes_to_write) {
    bytes_in_page = bytes_to_write;
}
```
Uses bytes_in_page 40b 42d, bytes_to_write 40b, ofs_bytes_in_page 36b, ofs_open_files 21c 21d, and ofs_position_to_data_index 34d.

Die Überprüfung von `page_index` wird in einen separaten Chunk ausgelagert, damit sie in andere Funktionen leicht integriert werden kann.

⟨*ofs check page_ index range* 40d⟩≡ (40c 42e) [40d]
```
if(page_index < 0 || page_index >= OFS_MAX_FILE_PAGES) { return -1; }
```
Uses OFS_MAX_FILE_PAGES 20a.

Anschließend wird `current_page` auf die richtigen Daten gesetzt. Für den Fall, dass noch keine Speicherseite hinterlegt wurde, wird eine neue angefordert, damit dann an `current_page` geschrieben werden kann.

⟨*ofs_ write loop* 40c⟩+≡ (39c) ◁40c 41a▷ [40e]
```
current_page = file->data[page_index];
if(current_page == 0) {
    file->data[page_index] = (char *) request_new_page();
```

```
    current_page = file->data[page_index];
    if(current_page == 0) {
        return -1;
    }
}
```

Uses `current_page` 40b 42d and `file` 32b 33c 38b 44c 47d 50a 53a.

Danach kann das eigentliche Schreiben der Daten erfolgen, was durch `strncpy` erledigt wird. Vorher muss allerdings noch die relative Position des Dateizeigers in der Speicherseite bestimmt werden. Diese wird in `rel_page_pos` zwischengespeichert.

⟨*ofs_ write loop* 40c⟩+≡ (39c) ◁40e 41b▷ [41a]

```
int rel_page_pos = ofs_relative_data_position(page_index, ofs_open_files[fd].pos);
memcpy((void *) (current_page + rel_page_pos),
       (void *) ptr_buf, (size_t) bytes_in_page);
```

Defines:
`memcpy`, used in chunks 28b, 52a, and 73.
`rel_page_pos`, never used.
Uses `bytes_in_page` 40b 42d, `current_page` 40b 42d, `ofs_open_files` 21c 21d, `ofs_relative_data_position` 35b, `ptr_buf` 40b 42d, and `size_t` 73.

Am Ende der schleife werden noch alle Zähler und Zeiger auf die neuen Positionen verschoben.

⟨*ofs_ write loop* 40c⟩+≡ (39c) ◁41a [41b]

```
bytes_to_write          -= bytes_in_page;
bytes_written           += bytes_in_page;
ptr_buf                 += bytes_in_page;
ofs_open_files[fd].pos += bytes_in_page;
```

Uses `bytes_in_page` 40b 42d, `bytes_to_write` 40b, `ofs_open_files` 21c 21d, and `ptr_buf` 40b 42d.

Abschließend ist es wichtig, dass der Datenbereich für spätere Operationen keine Lücken enthält. Diese können beispielsweise entstehen, wenn man den Dateizeiger mit `ofs_lseek()` verschiebt. Eventuelle Lücken schließt die Funktion `ofs_fill_gaps()`. Außerdem wird die richtige Größe der Datei angepasst, falls sich der Dateizeiger außerhalb von `file->i_size` befindet.

⟨*ofs_ write finish steps* 41c⟩≡ (39c) [41c]

```
int ret_fill_gaps = ofs_fill_gaps(file, page_index);
if(ret_fill_gaps == -1) { printf("fill fail\n");return -1; }
if(ofs_open_files[fd].pos > file->i_size) {
    file->i_size = (size_t) ofs_open_files[fd].pos;
}
```

Defines:
`ret_fill_gaps`, never used.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `ofs_fill_gaps` 30c, `ofs_open_files` 21c 21d, and `size_t` 73.

### ofs_read

Das Lesen aus dem Overlay-Dateisystem verläuft ähnlich dem Schreiben in selbiges. Da die Logik zum Durchwandern der `ofs_file`-Struktur schon ausführlich in `ofs_write()` beschrieben wurde, erfolgt die Ausführung hier weniger detailliert. Theoretisch könnten auch ein paar Chunks aus `ofs_write()` übernommen werden, da aber die Variablennamen sprechend und passend zum Vorgang sein sollen, sind hierfür doch eigene Chunks notwendig.

Der Prototyp für `ofs_read()` ist analog dem zu `ofs_write()` nur wird aus `buf` nicht gelesen sondern geschrieben. Aus diesem Grund fehlt hier das Schlüsselwort `const` im Vergleich zu

`ofs_write()`.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁39b 44a▷ [42a]
```
int ofs_read(int fd, void *buf, int count);
```
Uses `ofs_read` 42b.

Die Funktion besteht im Wesentlichen wieder aus einer Schleife, die die Speicherseiten in der `ofs_file`-Struktur durchwandert und die Daten nach und nach in `buf` schreibt. Im Überblick sieht die Funktion folgendermaßen aus:

⟨*ofs functions* 23b⟩+≡ (16b) ◁39c 44b▷ [42b]
```
int ofs_read(int fd, void *buf, int count) {
    ⟨ofs check fd range 39a⟩
    ⟨ofs_ read check access mode 42c⟩
    ⟨ofs_ read define buffer variables 42d⟩
    while(bytes_to_read > 0 &&
        ofs_open_files[fd].pos < file->i_size ) {
        ⟨ofs_ read loop 42e⟩
    }
    ⟨update atime 31a⟩
    return bytes_read;
}
```
Defines:
`ofs_read`, used in chunks 24d, 32, 33a, 42a, 51d, 52a, and 72.
Uses `bytes_read` 32a, `bytes_to_read` 42d, `file` 32b 33c 38b 44c 47d 50a 53a, and `ofs_open_files` 21c 21d.

Die Abbruchbedingung in der Schleife enthält zudem eine Überprüfung, ob sich der Dateizeiger zum Lesen hinter dem Dateiende befindet. Diese Situation entsteht beispielsweise durch Setzen des Dateizeigers mit `lseek()`. In diesem Fall können keine Bytes gelesen werden und es wird einfach `bytes_read` zurückgegeben, was mit `0` initialisiert wurde.

Der Zugriffsmodus wird hier auf `O_RDONLY` und `O_RDWR` überprüft.

⟨*ofs_ read check access mode* 42c⟩≡ (42b) [42c]
```
if((ofs_open_files[fd].mode & O_RDONLY) != O_RDONLY &&
   (ofs_open_files[fd].mode & O_RDWR) != O_RDWR) {
    return -1;
}
```
Uses `O_RDONLY` 73, `O_RDWR` 73, and `ofs_open_files` 21c 21d.

Zum Durchlaufen der Schleife werden vorher wieder ähnliche Variablen wie in `ofs_write()` benötigt - lediglich ein paar Bezeichnungen sind anders.

⟨*ofs_ read define buffer variables* 42d⟩≡ (42b) [42d]
```
int bytes_to_read = count;  int bytes_read = 0;
int bytes_in_page = 0;      int page_index = 0;
char *ptr_buf = buf;        struct ofs_file *file = ofs_open_files[fd].file;
char *current_page = 0;
```
Defines:
`bytes_in_page`, used in chunks 40–43.
`bytes_to_read`, used in chunks 42 and 43c.
`current_page`, used in chunks 40e, 41a, and 43.
`ptr_buf`, used in chunks 41–43.
Uses `bytes_read` 32a, `file` 32b 33c 38b 44c 47d 50a 53a, and `ofs_open_files` 21c 21d.

Anschließend werden die Variablen für den aktuellen Schleifendurchlauf entsprechend gefüllt und `page_index` auf den Gültigkeitsbereich hin geprüft. Die Berechnung von `size_difference` ist deshalb wichtig, damit beim Lesen mit einer festen Blockgröße nicht über das Ende der Datei hinaus gelesen werden kann.

⟨*ofs_ read loop* 42e⟩≡ (42b) 43a▷ [42e]
```
page_index = ofs_position_to_data_index(ofs_open_files[fd].pos);
⟨ofs check page_ index range 40d⟩
bytes_in_page = ofs_bytes_in_page(page_index, ofs_open_files[fd].pos);
```

```
if(bytes_in_page > bytes_to_read) {
    bytes_in_page = bytes_to_read;
}
int size_difference = file->i_size - ofs_open_files[fd].pos;
if(size_difference <= 0) { *ptr_buf = '\0'; return 0; }
else if (bytes_in_page > size_difference) {
    bytes_in_page = size_difference;
}
```

Defines:
`size_difference`, never used.
Uses `bytes_in_page` 40b 42d, `bytes_to_read` 42d, `file` 32b 33c 38b 44c 47d 50a 53a, `ofs_bytes_in_page` 36b, `ofs_open_files` 21c 21d, `ofs_position_to_data_index` 34d, and `ptr_buf` 40b 42d.

Danach wird `current_page` auf die zu lesende Seite gesetzt und auf eine korrekte Speicheradresse geprüft. Sollte `current_page` auf eine nicht vorhande Seite zeigen, so wurde das Ende der Datei überschritten und ein weiteres Lesen ist nicht mehr möglich. Für diesen Fall wird die Schleife abgebrochen.

⟨*ofs_ read loop* 42e⟩+≡ (42b) ◁42e 43b▷ [43a]

```
current_page = file->data[page_index];
if(current_page == 0) {
    break;
}
```

Uses `current_page` 40b 42d and `file` 32b 33c 38b 44c 47d 50a 53a.

Jetzt geht es für den Normalfall, dass `current_page` Daten enthält, weiter. Erst wird wieder die relative Position des Dateizeigers innerhalb der Speicherseite bestimmt, damit an der richtigen Stelle gelesen wird und anschließend werden die Daten über den Zeiger `ptr_buf` in `buf` geschrieben.

⟨*ofs_ read loop* 42e⟩+≡ (42b) ◁43a 43c▷ [43b]

```
int rel_page_pos = ofs_relative_data_position(page_index, ofs_open_files[fd].pos);
memcpy((void *) ptr_buf,
       (void *) (current_page + rel_page_pos), (size_t) bytes_in_page);
```

Defines:
`memcpy`, used in chunks 28b, 52a, and 73.
`rel_page_pos`, never used.
Uses `bytes_in_page` 40b 42d, `current_page` 40b 42d, `ofs_open_files` 21c 21d, `ofs_relative_data_position` 35b, `ptr_buf` 40b 42d, and `size_t` 73.

Das Anpassen der Schleifenvariablen erfolgt wieder analog zu `ofs_write()`.

⟨*ofs_ read loop* 42e⟩+≡ (42b) ◁43b [43c]

```
bytes_to_read          -= bytes_in_page;
bytes_read             += bytes_in_page;
ptr_buf                += bytes_in_page;
ofs_open_files[fd].pos += bytes_in_page;
```

Uses `bytes_in_page` 40b 42d, `bytes_read` 32a, `bytes_to_read` 42d, `ofs_open_files` 21c 21d, and `ptr_buf` 40b 42d.

Da beim Lesen nichts an der Datei verändert wird, sind auch keine weiteren Schritte mehr erforderlich, so dass nur noch die Anzahl an gelesenen Bytes zurückgegeben wird und die Funktion damit beendet ist.

### 3.4.3. Modifizieren von Dateizeiger und -größe

Eine weitere wichtige Funktion eines Dateisystems ist es, dem Anwender die Möglichkeit zu bieten, den Dateizeiger innerhalb einer geöffneten Datei frei bewegen zu können, was mit `ofs_lseek()` ermöglicht wird. Desweiteren wird in diesem Abschnitt auch beschrieben, wie mit `ofs_ftruncate()` die Dateigröße einer Datei verändert werden kann.

### ofs_lseek

Damit die Position des Dateizeigers manuell angepasst werden kann, wird die Funktion `ofs_lseek` implementiert. Diese soll konform zu den bisherigen `*_lseek`-Funktionen sein und erwartet als Parameter einen Filedeskriptor `fd`, einen Versatz, um den der Zeiger verschoben werden soll, `offset` und eine Richtung `whence`, in die verschoben werden soll. Zurückgeliefert wird die aktuelle Position, gemessen vom Anfang der Datei, oder `-1` im Fehlerfall.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁42a 45a▷ [44a]
```
int ofs_lseek(int fd, int offset, int whence);
```
Uses `ofs_lseek` 44b.

Zunächst gilt es den Fildeskriptor `fd` auf Gültigkeit zu überprüfen.

⟨*ofs functions* 23b⟩+≡ (16b) ◁42b 45b▷ [44b]
```
int ofs_lseek(int fd, int offset, int whence) {
    int new_pos = 0;
    ⟨ofs check fd range 39a⟩
    ⟨ofs_ lseek calc new offset 44c⟩
    return new_pos;
}
```
Defines:
`ofs_lseek`, used in chunks 24c, 31–33, 44a, 51d, 52a, and 72.

Anschließend wird die neue Position `new_pos` des Dateizeigers in Abhänigkeit von `whence` berechnet. Hierbei ist zu beachten, dass die neue Position nicht kleiner als `0` und nicht größer als `OFS_MAX_FILE_SIZE` sein darf. Abschließend wird noch `ofs_open_files[fd].pos` auf die neue Position gesetzt.

⟨*ofs_ lseek calc new offset* 44c⟩≡ (44b) [44c]
```
struct ofs_file *file = ofs_open_files[fd].file;
int old_pos = ofs_open_files[fd].pos;
switch(whence) {
    case SEEK_SET:
        new_pos = offset;
    break;
    case SEEK_CUR:
        new_pos = old_pos + offset;
    break;
    case SEEK_END:
        new_pos = file->i_size + offset;
    break;
}
if(new_pos < 0 || new_pos >= OFS_MAX_FILE_SIZE) { return -1; }
ofs_open_files[fd].pos = new_pos;
```
Defines:
`file`, used in chunks 16b, 21a, 23c, 27, 30, 31, 38d, 40–43, 45, 46, 48–51, 53c, 73, and 78b.
`old_pos`, never used.
Uses `OFS_MAX_FILE_SIZE` 20a, `ofs_open_files` 21c 21d, `SEEK_CUR` 73, `SEEK_END` 73, and `SEEK_SET` 73.

### ofs_ftruncate

Zur Veränderung der Größe einer Datei soll die Funktion `ofs_ftruncate` implementiert werden. Das Verhalten dieser Funktion soll sich wie folgt darstellen: Ist die neue Dateigröße kleiner als die Bisherige, so sind alle Daten ab dem letzten Byte der neuen Dateigröße verloren. Für den Fall, dass die Datei größer sein soll als bisher, werden die neuen Daten einfach mit `'\0'` vorbelegt.

Die Funktion gibt im Fehlerfall `-1` und im Erfolgsfall `0` zurück. Als Parameter wird ein gültiger Filedeskriptor `fd` und eine neue Größe `length` der Datei erwartet.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁44a 46c▷ [45a]
```
int ofs_ftruncate(int fd, int length);
```
Uses `ofs_ftruncate` 45b.

Das Gerüst der Funktion sieht wie folgt aus: zunächst wird `fd` geprüft, anschließend werden die oben beschriebenen Fälle unterschieden und entsprechend gehandelt. Zum Abschluss wird noch die Dateigröße auf den neuen Wert gesetzt.

⟨*ofs functions* 23b⟩+≡ (16b) ◁44b 46d▷ [45b]
```
int ofs_ftruncate(int fd, int length) {
    ⟨ofs check fd range 39a⟩
    struct ofs_file *file = ofs_open_files[fd].file;
    int page_index = ofs_position_to_data_index(length);
    if(length > file->i_size) {
        ⟨ofs_ftruncate extend 45c⟩
    } else if (length < file->i_size) {
        ⟨ofs_ftruncate shrink 45e⟩
    }
    file->i_size = length;
    ⟨update mtime 31b⟩
    return 0;
}
```
Defines:
`ofs_ftruncate`, used in chunks 45a and 72.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `ofs_open_files` 21c 21d, and `ofs_position_to_data_index` 34d.

Das Vergrößern der Datei gestaltet sich einfach, da hierfür nur die Lücken in der Datei vom letzten Byte der neuen Größe `length` mit `'\0'` aufgefüllt werden müssen, was die Funktion `ofs_fill_gaps()` erledigt.

⟨*ofs_ftruncate extend* 45c⟩≡ (45b) 45d▷ [45c]
```
int ret_fill = ofs_fill_gaps(file, page_index);
```
Defines:
`ret_fill`, used in chunk 45d.
Uses `file` 32b 33c 38b 44c 47d 50a 53a and `ofs_fill_gaps` 30c.

Beim Auffüllen der Datei könnte es passieren, dass nicht mehr genug freie Speicherseiten vorhanden sind, weshalb `ret_fill` noch geprüft werden muss.

⟨*ofs_ftruncate extend* 45c⟩+≡ (45b) ◁45c [45d]
```
if(ret_fill < 0) { return -1; }
```
Uses `ret_fill` 45c.

Das Verkleinern gestaltet sich ein wenig aufwändiger, da hier die vorhandenen Daten mit `'\0'` innerhalb einer Speicherseite überschrieben werden müssen, falls `length` nicht genau auf dem Ende einer Speicherseite liegt. Dies geschieht durch `ofs_bytes_in_page()` und einem `memset()`. Bei der Berechnung von `bytes_to_fill` und `rel_position` ist `length` um `1` zu erhöhen, da erst nach dem letzten Byte aufgefüllt werden soll.

⟨*ofs_ftruncate shrink* 45e⟩≡ (45b) 46a▷ [45e]
```
int bytes_to_fill = ofs_bytes_in_page(page_index, length + 1);
int rel_position = ofs_relative_data_position(page_index, length +1);
memset((void *) file->data[page_index] + rel_position, '\0', bytes_to_fill);
```
Defines:
`bytes_to_fill`, never used.
`rel_position`, never used.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `ofs_bytes_in_page` 36b, and `ofs_relative_data_position` 35b.

Anschließend werden noch alle nicht mehr benötigten Speicherseiten wieder freigegeben.

⟨*ofs_ftruncate shrink* 45e⟩+≡ (45b) ◁45e [46a]
```
page_index++;
⟨ofs release page_index to OFS-MAX-FILE-PAGES 46b⟩
```

Da hier nur innerhalb bereits benutzten Speichers gearbeitet wird, ist keine weitere Überprüfung auf Fehler notwendig. Für die Verwendung in anderen Funktionen, wird die Methodik zum Freigeben der Seiten ab einem bestimmten Index `page_index` in folgendem Chunk implementiert.

⟨*ofs release page_index to OFS-MAX-FILE-PAGES* 46b⟩≡ (46a 49b) [46b]
```
while(page_index < OFS_MAX_FILE_PAGES && file->data[page_index] != 0) {
    release_page((unsigned int) file->data[page_index] / PAGE_SIZE);
    page_index++;
}
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `OFS_MAX_FILE_PAGES` 20a, and `PAGE_SIZE` 73.

Das Pendant zu `ofs_ftruncate` ohne Filedeskriptor, `ofs_truncate`, wurde an dieser Stelle nicht etwa vergessen, sondern wird bereits durch den Aufruf im Virtuellen-Dateisystem von Ulix auf `ofs_ftruncate` umgeleitet [2, S. 419–420].

### 3.4.4. Verknüpfen und Löschen

Zur Vermeidung von unnötigen Duplikaten, soll mit `ofs_link()` und `ofs_symlink()` die Erstellung von Verknüpfungen implementiert werden. Um diese wieder zu lösen und eventuell nicht mehr benötigten Speicher freizugeben, wird weiter die Funktion `ofs_unlink()` zur Verfügung gestellt.

#### ofs_link

Für die Erstellung von Hardlinks soll die Funktion `ofs_link()` implementiert werden. Diese nimmt als Parameter den alten Dateinamen `path_old` sowie einen neuen Dateinamen `path_new` entgegen und gibt im Erfolgsfall `0` und im Fehlerfall `-1` zurück.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁45a 47b▷ [46c]
```
int ofs_link(const char *path_old, const char *path_new);
```
Uses `ofs_link` 46d and `path_old` 59a.

Ein Hardlink ist nun nichts weiter als ein neuer Eintrag in `ofs_index`, dessen `data`-Attribut auf die selbe `ofs_file`-Struktur zeigt wie der Eintrag von `path_old`. Am Anfang wird hierfür überprüft, ob sich `path_old` im Overlay-Dateisystem befindet. Ist dies der Fall, so wird ein neuer Eintrag in `ofs_index` angelegt, dessen `data`-Attribut gesetzt und die Link-Anzahl von `file` erhöht.

⟨*ofs functions* 23b⟩+≡ (16b) ◁45b 47c▷ [46d]
```
int ofs_link(const char *path_old, const char *path_new) {
    int idx_old = ofs_find_pathname(path_old);
    if(idx_old < 0) return -1;
    struct ofs_file *file = ofs_index[idx_old].data;
    int idx_new = ofs_get_free_inode();
    struct ofs_inode *inode = &ofs_index[idx_new];
    inode->data = ofs_index[idx_old].data;
    strncpy(inode->path, path_new, 248);
    file->i_nlinks++;
    ⟨link/symlink write dir entry 47a⟩
    ⟨update ctime 31c⟩
    return 0;
```

```
}
```

Defines:
ofs_link, used in chunks 46c, 59a, 72, and 81a.
Uses file 32b 33c 38b 44c 47d 50a 53a, ofs_find_pathname 26b, ofs_get_free_inode 24a, ofs_index 19a, and path_old 59a.

Abschließend muss der neue Dateiname in das entsprechende Verzeichnis geschrieben werden. Da dies auch für `ofs_symlink()` gebraucht wird, ist es sinnvoll dieses Vorgehen in einen eigenen Chunk auszulagern.

⟨*link/symlink write dir entry* 47a⟩≡ (46d 47c) [47a]
```
char dirname[248] = {0};
struct dir_entry entry = {0};
entry.inode = 1;
ofs_get_dir_and_filename(path_new, dirname, entry.filename);
ofs_write_dir_entry(dirname, &entry);
```

Defines:
dirname, used in chunks 25, 32, 34b, 37e, and 81b.
entry, used in chunks 24d, 31–34, 37e, 52c, and 72.
Uses ofs_get_dir_and_filename 25b and ofs_write_dir_entry 31e.

## ofs_symlink

Da Hardlinks nur innerhalb des selben Dateisystems funktionieren, braucht es für Verknüpfungen über Dateigrenzen hinweg eine andere Art: die symbolische Verknüpfung, auch Symlink genannt. Dies ist nichts anderes, als eine Datei, deren Inhalt eine absolute Pfadangabe zum Ziel ist. Allerdings muss der Modus der Datei von `S_IFREG` auf `S_IFLNK` geändert werden. In einem Symlink darf auch ein ungültiger Pfad enthalten sein, damit entfällt auch die Überprüfung des Link-Ziels. Der Prototyp gleicht dem von `ofs_link()`.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁46c 48a▷ [47b]
```
int ofs_symlink(const char *path_old, const char *path_new);
```

Uses ofs_symlink 47c and path_old 59a.

Wie bereits geschildert, muss nun einfach nur `path_old` in die spezielle, zu erstellende Datei in `path_new` geschrieben werden. Dies ist mit den vorhandenen `ofs_open()` etc. Funktionen schnell erledigt.

⟨*ofs functions* 23b⟩+≡ (16b) ◁46d 48b▷ [47c]
```
int ofs_symlink(const char *path_old, const char *path_new) {
    int fd = ofs_open(path_new, O_WRONLY | O_CREAT);
    if(fd < 0) return -1;
    ofs_write(fd, path_old, strlen(path_old));
    ofs_close(fd);
    ⟨ofs_symlink set S_IFLNK 47d⟩
    ⟨link/symlink write dir entry 47a⟩
    ⟨update ctime 31c⟩
    return 0;
}
```

Defines:
ofs_symlink, used in chunks 47b, 72, and 82a.
Uses O_CREAT 73, O_WRONLY 73, ofs_close 38d, ofs_open 37b, ofs_write 39c, and path_old 59a.

Um nun den Symlink von einer normalen Datei unterscheiden zu können, muss noch das `i_mode`-Attribut der eben erstellten Datei angepasst werden.

⟨*ofs_symlink set S_IFLNK* 47d⟩≡ (47c) [47d]
```
int index = ofs_find_pathname(path_new);
struct ofs_file *file = ofs_index[index].data;
file->i_mode = S_IFLNK | 0777;
```

Defines:
  `file`, used in chunks 16b, 21a, 23c, 27, 30, 31, 38d, 40–43, 45, 46, 48–51, 53c, 73, and 78b.
  `index`, used in chunks 25, 28, 32, 33, 48–53, 72, 83b, and 84.
Uses `ofs_find_pathname` 26b, `ofs_index` 19a, and `S_IFLNK` 73.

## ofs_unlink

Mit `ofs_unlink()` soll die Möglichkeit geschaffen werden, erstellte Dateien auch wieder zu löschen. Dies beinhaltet zum einen das Entfernen aus `ofs_index` als auch das tatsächliche Freigeben des Speichers, falls dies der letzte Verweis auf die `ofs_file`-Struktur war und die Datei derzeit nicht geöffnet ist.

Als Parameter wird ein Dateiname `path` übergeben. Der Rückgabewert ist im Erfolgsfall `0` und im Fehlerfall `-1`.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁47b 49c▷ [48a]
```
int ofs_unlink(const char *path);
```
Uses `ofs_unlink` 48b.

Am Anfang der Funktion wird der Index in `ofs_index` von `path` ermittelt. Anschließend wird `in_use` zwischengespeichert, ob die Datei `file` gerade in Gebrauch ist. Danach wird anhand von `in_use` und `file->i_nlinks` entschieden wie weiter vorzugehen ist.

⟨*ofs functions* 23b⟩+≡ (16b) ◁47c 49d▷ [48b]
```
int ofs_unlink(const char *path) {
    int index = ofs_find_pathname(path);
    if(index < 0) { return -1; }
    struct ofs_file *file = ofs_index[index].data;
    int in_use = ofs_find_open_file(file);
    ⟨ofs_ unlink check cases 48c⟩
    return 0;
}
```
Defines:
  `ofs_unlink`, used in chunks 48a, 53c, 72, and 82c.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `index` 47d, `ofs_find_open_file` 27c, `ofs_find_pathname` 26b, and `ofs_index` 19a.

Nun werden folgende Fälle unterschieden: Ist die Datei gerade geöffnet und `path` der letzte Verweis auf `file`, wird mit `-1` abgebrochen.

⟨*ofs_ unlink check cases* 48c⟩≡ (48b) 48d▷ [48c]
```
if(in_use == 1 && file->i_nlinks == 1) { return -1; }
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a.

Für alle anderen Fälle ist `in_use` nicht weiter relevant, da Zugriffe über einen offenen Filedeskriptor nicht mehr ins Leere laufen. Ist `file->i_nlinks` größer als `1`, so muss dieser Wert nur um `1` vermindert und `ofs_index[index]` freigeben werden, um die gewünschte Operation abzuschließen.

⟨*ofs_ unlink check cases* 48c⟩+≡ (48b) ◁48c 49b▷ [48d]
```
else if(file->i_nlinks > 1) {
    file->i_nlinks-;
    struct ofs_inode *inode = &ofs_index[index];
    ⟨ofs clear inode 49a⟩
    ⟨update ctime 31c⟩
}
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `index` 47d, and `ofs_index` 19a.

Das Bereinigen eines Eintrags in `ofs_index` wird in folgendem Code-Chunk erledigt, da dieses auch für den letzten Fall in `ofs_unlink()` benötigt wird. Da beim Löschen eines Inodes auch der zugehörige Dateipfad obsolet wird, muss dieser aus einem eventuell existierenden Verzeichnis gelöscht werden. Hierzu werden Verzeichnisname `dirname` und Dateiname `filename` aus `inode->path` mit `ofs_get_dir_and_filename()` ermittelt.

⟨*ofs clear inode* 49a⟩≡ (48d 49b) [49a]
```
char dirname[248] = {0}; char filename[64] = {0};
ofs_get_dir_and_filename(inode->path, dirname, filename);
ofs_remove_from_dir(dirname, filename);
inode->inode = 0;
inode->data = 0;
memset(inode->path, '\0', 248);
```
Defines:
`dirname`, used in chunks 25, 32, 34b, 37e, and 81b.
Uses `ofs_get_dir_and_filename` 25b and `ofs_remove_from_dir` 32d.

Für den letzten Fall ist die Ausgangssituation so, dass sowohl der Eintrag in `ofs_index` bereinigt werden muss, als auch der Speicher, den `file` und die zugehörigen Daten belegen, freigegeben werden kann.

⟨*ofs_ unlink check cases* 48c⟩+≡ (48b) ◁48d [49b]
```
else {
    struct ofs_inode *inode = &ofs_index[index];
    int page_index = 1;
    ⟨ofs release page_ index to OFS-MAX-FILE-PAGES 46b⟩
    release_page((unsigned int) file / PAGE_SIZE);
    ⟨ofs clear inode 49a⟩
}
```
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `index` 47d, `ofs_index` 19a, and `PAGE_SIZE` 73.

### 3.4.5. Anzeigen und Bearbeiten von Metainformationen

Eine Datei besteht aus mehr als nur einem Dateinamen und dem eigentlichen Inhalt. Zum Abrufen der zusätzlich gespeicherten Metainformationen wird `ofs_stat()` implementiert. Da das Overlay-Dateisystem sowohl Zugriffsmodi als auch die Zugehörigkeit einer Datei zu einem Benutzer und einer Gruppe unterstützt, werden zur Bearbeitung dieser Paramter die Funktionen `ofs_chown()` und `ofs_chmod()` bereitgestellt.

#### ofs_stat

Zur Anzeige von Metainformationen von einer Datei `path` soll diese Funktion dienen. Sie liest die relevanten Daten aus der `ofs_file`-Struktur und befüllt damit die `stat`-Struktur, welche über `buf` erreicht werden kann. Der Rückgabewert ist `-1` im Fehlerfall und `0` im Erfolgsfall. Daraus ergibt sich folgender Prototyp:

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁48a 50b▷ [49c]
```
int ofs_stat(const char *path, struct stat *buf);
```
Uses `ofs_stat` 49d.

Die Implementierung der Funktion selbst gestaltet sich wieder etwas einfacher, als es bei `ofs\_write()` und `ofs_read()` der Fall war. Zunächst wird die Datei `path` im Overlay-Dateisystem gesucht. Ist diese nicht vorhanden, kann sofort mit `-1` abgebrochen werden.

⟨*ofs functions* 23b⟩+≡ (16b) ◁48b 50c▷ [49d]
```
int ofs_stat(const char *path, struct stat *buf) {
    int index = ofs_find_pathname(path);
```

```
    if(index < 0) { return -1; }
    ⟨ofs_ stat fill buf 50a⟩
    return 0;
}
```

Defines:
ofs_stat, used in chunks 49c, 59a, 72, and 79a.
Uses index 47d and ofs_find_pathname 26b.

Wurde die Datei gefunden, kann damit fortgefahren werden buf mit den nötigen Informationen zu füllen.

⟨ofs_ stat fill buf 50a⟩≡ (49d) [50a]

```
struct ofs_file *file = ofs_index[index].data;
buf->st_mode  = file->i_mode;  buf->st_nlink = file->i_nlinks;
buf->st_uid   = file->i_uid;   buf->st_gid   = file->i_gid;
buf->st_size  = file->i_size;  buf->st_atime = file->i_atime;
buf->st_ctime = file->i_ctime; buf->st_mtime = file->i_mtime;
```

Defines:
file, used in chunks 16b, 21a, 23c, 27, 30, 31, 38d, 40–43, 45, 46, 48–51, 53c, 73, and 78b.
Uses index 47d, ofs_index 19a, st_atime 73, st_ctime 73, st_gid 73, st_mode 73, st_mtime 73, st_nlink 73, st_size 73, and st_uid 73.

## ofs_chown

Dateien werden immer unter dem aktuellen Benutzer mit der aktuellen Gruppe angelegt. Datim dies nicht fest ins Dateisystem geschrieben bleibt, wird mit ofs_chown() die Möglichkeit geschaffen, sowohl den Benutzer owner als auch die Gruppe group zu verändern. Hierzu muss der Funktion, neben eben genannten Parametern, noch der Dateiname path übergeben werden. Bei Erfolg wird 0 und bei einem Fehler -1 zurückgegeben.

⟨ofs prototypes 23a⟩+≡ (16a) ◁49c 51a▷ [50b]

```
int ofs_chown(const char *path, short owner, short group);
```

Uses ofs_chown 50c.

Die Funktion selbst ändert nun einfach die i_uid bzw. i_gid der ofs_file-Struktur, falls diese nicht mit -1 übergeben wurden.

⟨ofs functions 23b⟩+≡ (16b) ◁49d 51b▷ [50c]

```
int ofs_chown(const char *path, short owner, short group) {
    int index = ofs_find_pathname(path);
    if(index < 0) return -1;
    struct ofs_file *file = ofs_index[index].data;
    if(owner > -1) file->i_uid = owner;
    if(group > -1) file->i_gid = group;
    ⟨update ctime 31c⟩
    return 0;
}
```

Defines:
ofs_chown, used in chunks 50b, 72, and 78a.
Uses file 32b 33c 38b 44c 47d 50a 53a, index 47d, ofs_find_pathname 26b, and ofs_index 19a.

## ofs_chmod

Mit ofs_chown() ist es möglich Benutzer und Gruppe zu ändern. Durch ofs_chmod() soll auch der Zugriffsmodus selbst geändert werden können. Neben dem Modus mode muss der

Funktion hierfür auch der Dateiname `path` übergeben werden. Ist die Änderung erfolgreich, wird `0` und im Fehlerfall `-1` zurückgegeben.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁50b 51c▷ [51a]
```
int ofs_chmod(const char *path, short mode);
```
Uses `ofs_chmod` 51b.

Die Methodik zum Ändern ist analog zu `ofs\_chown()`, nur dass `file->i_mode` über eine Bitmanipulation auf den gewünschten Wert gebracht wird.

⟨*ofs functions* 23b⟩+≡ (16b) ◁50c 51d▷ [51b]
```
int ofs_chmod(const char *path, short mode) {
    int index = ofs_find_pathname(path);
    if(index < 0) return -1;
    struct ofs_file *file = ofs_index[index].data;
    if(mode > -1) file->i_mode = (file->i_mode & ~07777) | (mode & 07777);
    ⟨update ctime 31c⟩
    return 0;
}
```
Defines:
`ofs_chmod`, used in chunks 51a, 72, and 77a.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `index` 47d, `ofs_find_pathname` 26b, and `ofs_index` 19a.

### 3.4.6. Arbeiten mit Verzeichnissen

Zur besseren Strukturierung von Dateien werden diese üblicherweise in Verzeichnissen abgelegt. Dies dient einerseits der Übersichtlichkeit, andererseits aber dem Umstand, dass so auch Dateien mit dem selben Namen in unterschiedlichen Verzeichnissen existieren können, was sonst nicht möglich wäre. Um Verzeichnisse zu unterstützen, werden die Funktionen `ofs_getdent()`, `ofs_mkdir()` und `ofs_rmdir()` implementiert.

#### ofs_getdent

Um auf Verzeichnisse im Overlay-Dateisystem zugreifen zu können, ist es wichtig deren Einträge zu lesen. Hierfür wird `ofs_getdent()` implementiert. Der Rückgabewert ist `-1` bei Fehlern, sonst `0`. Die Paramter sind der Verzeichnisname `path`, der zu lesende Index `index` und der Puffer `buf` in den geschrieben wird.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁51a 52b▷ [51c]
```
int ofs_getdent(const char *path, int index, struct dir_entry *buf);
```
Uses `index` 47d and `ofs_getdent` 51d.

Im Wesentlichen ist die Funktionslogik analog zu `ofs_write_dir_entry()`. Unterschiede bestehen darin, dass der zu lesende Index übergeben und gelesen statt geschrieben wird. Was das Lesen allerdings komplizierter macht, sind entstandene Lücken durch Löschen von Dateien.

⟨*ofs functions* 23b⟩+≡ (16b) ◁51b 52c▷ [51d]
```
int ofs_getdent(const char *path, int index, struct dir_entry *buf) {
    int fd = 0; int read_bytes = 0;
    struct dir_entry dentbuf = {0};
    fd = ofs_open(path, O_RDONLY);
    if(fd < 0) return -1;
    ofs_lseek(fd, index*OFS_DIR_ENTRY_SIZE, SEEK_SET);
    read_bytes = ofs_read(fd, (void *) &dentbuf, OFS_DIR_ENTRY_SIZE);
    if(strcmp(dentbuf.filename, buf->filename) == 0) {
        dentbuf.inode = 0;
    }
    ⟨ofs_ getdent check for gaps 52a⟩
    ofs_close(fd);
```

```
    return 0;
}
```

Defines:
ofs_getdent, used in chunks 51c, 72, and 83b.
Uses dentbuf 32a, index 47d, O_RDONLY 73, ofs_close 38d, OFS_DIR_ENTRY_SIZE 20c, ofs_lseek 44b, ofs_open 37b, ofs_read 42b, and SEEK_SET 73.

Bis hierher wäre ein vorhandener Eintrag bereits erfolgreich in `buf` gelesen worden. Nun folgt die Überprüfung, ob nach dem letzten Eintrag gelesen wurde oder ob es sich um eine Lücke handelt und weiter gesucht werden kann, bis ein gütliger Eintrag gefunden wird. Ist `read_bytes` gleich `0`, so ist das Ende erreicht und der Inhalt von `dentbufbuf` muss auf `0` gesetzt werden. Wurden Bytes gelesen, aber `dentbuf->inode` ist gleich `0`, so ist eine Lücke vorhanden und ein gültiger Eintrag wird gesucht.

⟨*ofs_ getdent check for gaps* 52a⟩≡ (51d) [52a]
```
if (read_bytes > 0 && dentbuf.inode == 0) {
    while(read_bytes > 0) {
        ofs_lseek(fd, (++index)*OFS_DIR_ENTRY_SIZE, SEEK_SET);
        read_bytes = ofs_read(fd, (void *) &dentbuf, OFS_DIR_ENTRY_SIZE);
        if(dentbuf.inode > 0) break;
    }
}
if(read_bytes == 0 || dentbuf.inode == 0) {
    buf->inode = 0; buf->filename[0] = '\0';
    return -1;
}
memcpy((void *) buf, (void *) &dentbuf, OFS_DIR_ENTRY_SIZE);
```

Uses dentbuf 32a, index 47d, memcpy 41a 43b, OFS_DIR_ENTRY_SIZE 20c, ofs_lseek 44b, ofs_read 42b, and SEEK_SET 73.

## ofs_mkdir

Damit Verzeichnisse ausgelesen werden können, muss es Verzeichnisse geben. Zum Anlegen dieser wird die Funktion `ofs_mkdir()` implementiert. Da Verzeichnisse nur spezielle Dateien sind, können diese mit den Standardfunktionen `ofs_open()` angelegt und nachträglich modifiziert werden. Der Prototyp ähnelt daher `ofs_open()`.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁51c 53b▷ [52b]
```
int ofs_mkdir(const char *path, int mode);
```

Uses ofs_mkdir 52c.

Zunächst wird also mit `ofs_open()` und der `O_CREAT`-Option eine Datei erzeugt. Anschließend werden zwei `dir_entry`-Strukturen in die Datei geschrieben. `entry.inode` wird im Overlay-Dateisystem nicht genutzt, zeigt aber für die Standardfunktionen im Kernel an, ob der Eintrag belegt ist oder nicht und wird deshalb immer auf `1` gesetzt.

⟨*ofs functions* 23b⟩+≡ (16b) ◁51d 53c▷ [52c]
```
int ofs_mkdir(const char *path, int mode) {
    int fd = ofs_open(path, O_CREAT | O_RDWR);
    if(fd < 0) return -1;
    struct dir_entry entry = { 0 };
    entry.filename[0] = '.';
    entry.inode = 1;
    ofs_write(fd, (void *) &entry, OFS_DIR_ENTRY_SIZE);
    entry.filename[1] = '.';
    ofs_write(fd, (void *) &entry, OFS_DIR_ENTRY_SIZE);
    ofs_close(fd);
    ⟨ofs_ mkdir set S_ IFDIR 53a⟩
    return 0;
```

```
}
```

Defines:
`ofs_mkdir`, used in chunks 52b, 72, 80a, and 83b.
Uses `entry` 47a, `O_CREAT` 73, `O_RDWR` 73, `ofs_close` 38d, `OFS_DIR_ENTRY_SIZE` 20c, `ofs_open` 37b, and `ofs_write` 39c.

Anschließend muss noch die Bitmaske in `ofs_file.i_mode` modifiziert und `ofs_file.i_nlinks` auf 2 gesetzt werden.

⟨*ofs_ mkdir set S_ IFDIR* 53a⟩≡ (52c) [53a]
```
int idx = ofs_find_pathname(path);
struct ofs_file *file = ofs_index[idx].data;
file->i_nlinks = 2;
file->i_mode = S_IFDIR | mode;
```

Defines:
`file`, used in chunks 16b, 21a, 23c, 27, 30, 31, 38d, 40–43, 45, 46, 48–51, 53c, 73, and 78b.
`idx`, used in chunks 29–32 and 72.
Uses `ofs_find_pathname` 26b, `ofs_index` 19a, and `S_IFDIR` 73.

## ofs_rmdir

Das Löschen eines Verzeichnisses gestaltet sich ein wenig aufwändiger als das Löschen einer Datei. Dabei muss beachtet werden, dass nur leere Verzeichnisse gelöscht werden können. Als Parameter nimmt diese Funktion die absolute Pfadangabe `path` entgegen und gibt im Erfolgsfall `0` und im Fehlerfall `-1` zurück.

⟨*ofs prototypes* 23a⟩+≡ (16a) ◁52b [53b]
```
int ofs_rmdir(const char *path);
```

Uses `ofs_rmdir` 53c.

Der Ablauf ist nun folgender: Die Funktion prüft am Anfang, ob `i_nlinks` des Verzeichnisses kleiner oder gleich `2` ist, denn dies signalisiert, dass das Verzeichnis leer ist. Ist dies der Fall, so wird der Eintrag `..` aus dem Verzeichnis entfernt. Anschließend kann das Verzeichnis mit `ofs_unlink()` gelöscht werden.

⟨*ofs functions* 23b⟩+≡ (16b) ◁52c [53c]
```
int ofs_rmdir(const char *path) {
    printf("%s \n", path);
    int index = ofs_find_pathname(path);
    if(index < 0) return -1;
    struct ofs_file *file = ofs_index[index].data;
    if(file->i_nlinks <= 2) {
        ofs_remove_from_dir(path, "..");
        ofs_unlink(path);
        return 0;
    }
    return -1;
}
```

Defines:
`ofs_rmdir`, used in chunks 53b, 72, and 80c.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `index` 47d, `ofs_find_pathname` 26b, `ofs_index` 19a, `ofs_remove_from_dir` 32d, and `ofs_unlink` 48b.

# 4. Integration in Ulix und Funktionstest

Die bisherige Implementation beinhaltet bei genauerer Betrachtung nur eine RAM-Disk. Ein Overlay-Dateisystem wird erst bei der Integration in den Ulix -Kernel daraus. Wie dies von statten geht, wird in diesem Abschnitt dargelegt. Da der Programmcode in dieser Arbeit und der Ulix -Kernel in separaten Dateien liegen, müssen die in diesem Abschnitt gezeigten Code-Chunks in die Datei des Ulix -Kernels eingefügt werden.

## 4.1. Konstanten und Funktionsprototypen

Für die Definition von neuen Konstanten und Funktionsprototypen muss erst das Layout des Ulix -Kernel betrachtet werden, damit in die korrekten Code-Chunks die jeweiligen Eintragungen vorgenommen werden können. Diese sind `<constants>` und `<function prototypes>` [2, S. 44].

Zur Nutzung des Overlay-Dateisystems werden in `<constants>` zwei neue Konstanten angefügt. `FS_OFS` definiert hier eine neue Nummer für das Overlay-Dateisystem und `OFS_FS` welches Dateisystem bzw. Gerät überlagert wird. Bisher sind 4 Dateisysteme im Kernel definiert, weshalb `FS_OFS` auf `4` gesetzt wird [2, S. 410].

⟨*constants* 54a⟩≡ [54a]
```
#define FS_OFS 4
#define OFS_FS DEV_FD1
```
Defines:
`FS_OFS`, used in chunks 55–57.
`OFS_FS`, used in chunks 56 and 77–83.

Weiterhin wird der Name „ofs“ für das Overlay-Dateisystem in `fs_names` eingetragen, wofür der bisherige Code-Chunk `410b` nun wie folgt aussieht [2, S. 410]:

⟨*global variables* 54b⟩≡ 54c▷ [54b]
```
char *fs_names[] = { "ERROR", "minix", "fat", "dev", "ofs" };
```
Defines:
`fs_names`, never used.

Ebenfalls wird eine globale Variable `ofs_init_complete` angelegt. Diese wird später durch die Funktion `initialize_module()` auf `1` gesetzt, um eine erfolgreiche Initialisierung des Overlay-Dateisystems anzuzeigen.

⟨*global variables* 54b⟩+≡ ◁54b [54c]
```
int ofs_init_complete = 0;
```
Defines:
`ofs_init_complete`, used in chunks 16b, 56, 73, and 77–83.

Da für den Kernel keine eigene Header-Datei existiert und beim Inkludieren der `module.h`-Datei aus dieser Arbeit doppelte Typdefinitionen entstehen würden, werden die Funktionsprototypen mit „extern“ gekennzeichnet und in `<function prototypes>` eingefügt. Exemplarisch ist das an dieser Stelle mit `ofs_open()` aufgezeigt. Eine vollständige Liste befindet sich in Anhang A.

⟨*function prototypes* 54d⟩≡ 72▷ [54d]
```
extern int ofs_open(const char *path, int mode);
```

Uses ofs_open 37b.

## 4.2. Erweiterung der Funktionslogik im Kernel

Damit die definierten Funktionen tatsächlich zum Einsatz kommen, ist es erforderlich, die entsprechenden Funktionen aus dem Virtuellen-Dateisystem von Ulix zu modifizieren. Dabei sind zwei Typen von Funktionen zu unterscheiden. Bei Funktionen, die einen Filedeskriptor entgegen nehmen, erfolgt die Integration durch Erweiterung des `switch`-Statements, was später noch genauer dargestellt wird. Der andere Typ von Funktion nimmt statt des Filedeskriptors eine Pfadangabe entgegen, was die Integration etwas umfangreicher macht und am Beispiel von `u_open()` demonstriert wird.

### 4.2.1. Funktionen mit Filedeskriptor

Da alle Funktionen mit Dateizugriff über einen Fildeskriptor über ein fast identisches `switch`-Statement verfügen, wird der Wechsel in das Overlay-Dateisystem am Beispiel von `u_close()` demonstriert. Alles was hierbei zu tun ist, ist eine neue `case FS_OFS` Bedingung einzufügen, die dann die Funktion `ofs_close()` mit dem durchgereichten Parameter aufruft. Das Resultat sieht folgendermaßen aus:

⟨*function implementations* 55⟩≡ 59b ▷ [55]

```
int u_close (int fd) {
  ⟨ VFS functions: turn fd into (fs, localfd) pair or fail ⟩
  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
    case FS_OFS:   return ofs_close (localfd);
    default:       return -1;
  }
}
```

Defines:
  u_close, used in chunks 57b, 73, 77a, 78a, and 81a.
Uses error 73, FS_OFS 54a, and ofs_close 38d.

An welcher Stelle `case FS_OFS` eingefügt wird, spielt hierbei keine Rolle. Weitere Funktionen, die auf diese Weise erweitert wurden sind `u_read()`, `u_write()`, `u_lseek()` und `u_ftruncate()`.

### 4.2.2. Funktionen mit Pfadangabe

In diesem Abschnitt folgt die nun etwas kompliziertere Art der Integration bei Funktionen, die statt eines Filedeskriptors eine Pfadangabe verwenden. Die zugrunde liegende Logik, wie in das Overlay-Dateisystem gesprungen wird, folgt dabei immer folgendem Schema: Zuerst wird anhand des Gerätes und dem Zustand von `ofs_init_complete` entschieden, ob die Datei im Overlay-Dateisystem vorhanden sein müsste. Anschließend wird mit `ofs_find_pathname()` geprüft, ob dem so ist und anhand des Rückgabewertes entschieden, wie weiter vorgegangen wird.

Den kompliziertesten Fall stellt hierbei `u_open()` dar, da diese Funktion zum einen dafür zuständig ist, die Dateien ins Overlay-Dateisystem zu kopieren und zum anderen einen gülti-

gen Filedesriptor zu erzeugen, der dann bei den im Abschnitt vorher genannten Funktionen Verwendung findet.

Da ebenfalls der Modulcharakter des Overlay-Dateisystems erhalten bleiben soll, werden die Eingriffe in bestehende Funktionen darauf beschränkt, dass nur ein Erweiterungs-Code-Chunk in diese eingefügt werden soll, der bei Bedarf wieder entfernt werden kann. Bietet Ulix irgendwann die Funktionalität zur Laufzeit Kernelmodule nachzuladen, wäre so auch der Weg bereitet, das Overlay-Dateisystem zu aktivieren.

Abbildung 4.1 auf Seite 58 schafft einen Überblick, was bei einer Erweiterung alles zu beachten ist. Anhand der blauen Abfrage wird überprüft, ob in das Overlay-Dateisystem gesprungen werden muss. Ist das nicht der Fall, so wird mit der Standardprozedur fortgefahren. Muss in das Overlay-Dateisystem gesprungen werden, so werden dort noch Entscheidungen getroffen, wie weiter zu verfahren ist. Aufgrund des oben genannten Modulcharakters wird vom grünen Zweig nicht mehr zurück in den Gelben gewechselt.

Nachfolgend wird nun dieser Erweiterungs-Chunk für die Funktion `u_open()` entwickelt. Durch bisherigen Aufruf der Funktion sind bereits die Variablen `abspath`, `device`, `localpath` und `oflag` definiert und mit den entsprechenden Werten belegt.

Wie bereits beschrieben beginnt der Chunk damit, dass geprüft wird, ob die Datei im Overlay-Dateisystem sein müsste oder nicht und ob dieses bereit zur Verwendung ist. Da der Dateiname im Overlay-Dateisystem nur maximal 248 Zeichen inklusive `'\0'`-Terminierung lang sein darf, wird dieser von `abspath` nach `ofs_path` kopiert. Ein etwaiges Abschneiden von Zeichen am Ende wird hierbei in Kauf genommen.

Existiert die Datei allerdings nicht, so werden Maßnahmen ergriffen, um diesen Missstand zu korrigieren. Nach dieser Abfrage ist jedenfalls davon auszugehen, dass die Datei existiert und es wird versucht diese zu öffnen. Sollte dies auch wieder fehlschlagen, so liegt ein größerer Fehler vor und es wird der Funktionsaufruf mit `return -1` beendet. Ist das Öffnen erfolgreich, so wird ein Filedeskriptor, bestehend aus `FS_OFS` und `ofs_idx` erzeugt, der in bereits genannten Funktionen mit Filedeskriptor weiter verwendet werden kann.

⟨*extend u_ open switch to ofs* 56⟩≡ (59b) [56]

```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path[248] = {0};
    strncpy(ofs_path, abspath, 247);
    int file_exists = ofs_find_pathname(ofs_path);
    if(file_exists < 0) {
        ⟨extend u_ open file doesn't exists 57a⟩
    }
    int ofs_idx = ofs_open(ofs_path, oflag);
    int ofs_fs = FS_OFS;
    if( ofs_idx == -1) return -1;
    else return (ofs_fs « 8) + ofs_idx;
}
```

Uses `FS_OFS` 54a, `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_fs` 57b, `ofs_idx` 57b, `ofs_init_complete` 54c, and `ofs_open` 37b.

Nun wird damit fortgefahren die Datei ins Overlay-Dateisystem zu übertragen. Dazu wird versucht, diese auf dem Minix-Dateisystem zu öffnen. Schlägt dies nicht fehl, so werden zuerst die Metainformationen in `mx_stat` gespeichert. Nun gilt es weitere Fallunterscheidungen zu treffen. Existiert die Inode-Nummer bereits im Overlay-Dateisystem, so wird versucht einen Hardlink auf eine bereites existierende Datei zu öffnen. Hier muss nun im Overlay-Dateisystem nur ein neuer Hardlink erzeugt werden und die Datei kann geöffnet werden. Nur für den Fall, dass es sich nicht um einen Hardlink handelt, muss der Dateiinhalt kopiert werden.

⟨*extend u_ open file doesn't exists* 57a⟩≡ (56) [57a]

```
int mx_fd = mx_open (device, localpath, O_RDONLY);
int mx_fs = FS_MINIX;
if (mx_fd != -1) {
    //File exists -> copy
    mx_fd = (mx_fs « 8) + mx_fd;

    struct stat mx_stat = {0};
    u_stat(abspath, &mx_stat);

    int inode_exists = ofs_find_inode(mx_stat.st_ino);

    if(inode_exists < 0) {
        ⟨extend u_ open copy file data 57b⟩
    } else {
        ⟨extend u_ open create hardlink 59a⟩
    }
}
```

Defines:
`mx_fd`, used in chunk 57b.
`mx_fs`, never used.
Uses `O_RDONLY` 73, `ofs_find_inode` 27a, `st_ino` 73, and `u_stat` 79b.

Der Kopiervorgang selbst ist nicht sonderlich kompliziert. Am Anfang wird eine Datei im Overlay-Dateisystem angelegt, in die der Inhalt kopiert werden kann. Da dort mit ganzen Speicherseiten gearbeitet wird, ist es naheliegend als Zwischenspeicher `buffer` eine ganze Speicherseite zu verwenden. Anschließend wird in einer Schleife der Dateiinhalt kopiert. Ist dies abgeschlossen, wird die Speicherseite von `buffer` wieder freigegeben, die Dateien geschlossen und durch das Schreiben von `mx_stat` die Metainformationen in das Overlay-Dateisystem übertragen.

⟨*extend u_ open copy file data* 57b⟩≡ (57a) [57b]

```
int ofs_idx = ofs_open(ofs_path, O_RDWR | O_CREAT);
int ofs_fs = FS_OFS;
int ofs_fd = 0;
if( ofs_idx == -1) return -1;
else ofs_fd = (ofs_fs « 8) + ofs_idx;

char *buffer = (char *) request_new_page();

int bytes_to_copy = mx_stat.st_size;
while(bytes_to_copy > 0) {
    int bytes_in_loop = PAGE_SIZE;
    if(bytes_to_copy < PAGE_SIZE) {
        bytes_in_loop = bytes_to_copy;
    }
    u_read(mx_fd, buffer, bytes_in_loop);
    u_write(ofs_fd, buffer, bytes_in_loop);
    bytes_to_copy -= bytes_in_loop;
}
release_page((unsigned int) buffer / PAGE_SIZE);
u_close(mx_fd); u_close(ofs_fd);
ofs_idx = ofs_find_pathname(ofs_path);
ofs_write_stat(ofs_idx, &mx_stat);
u_stat(abspath, &mx_stat);
```

Defines:

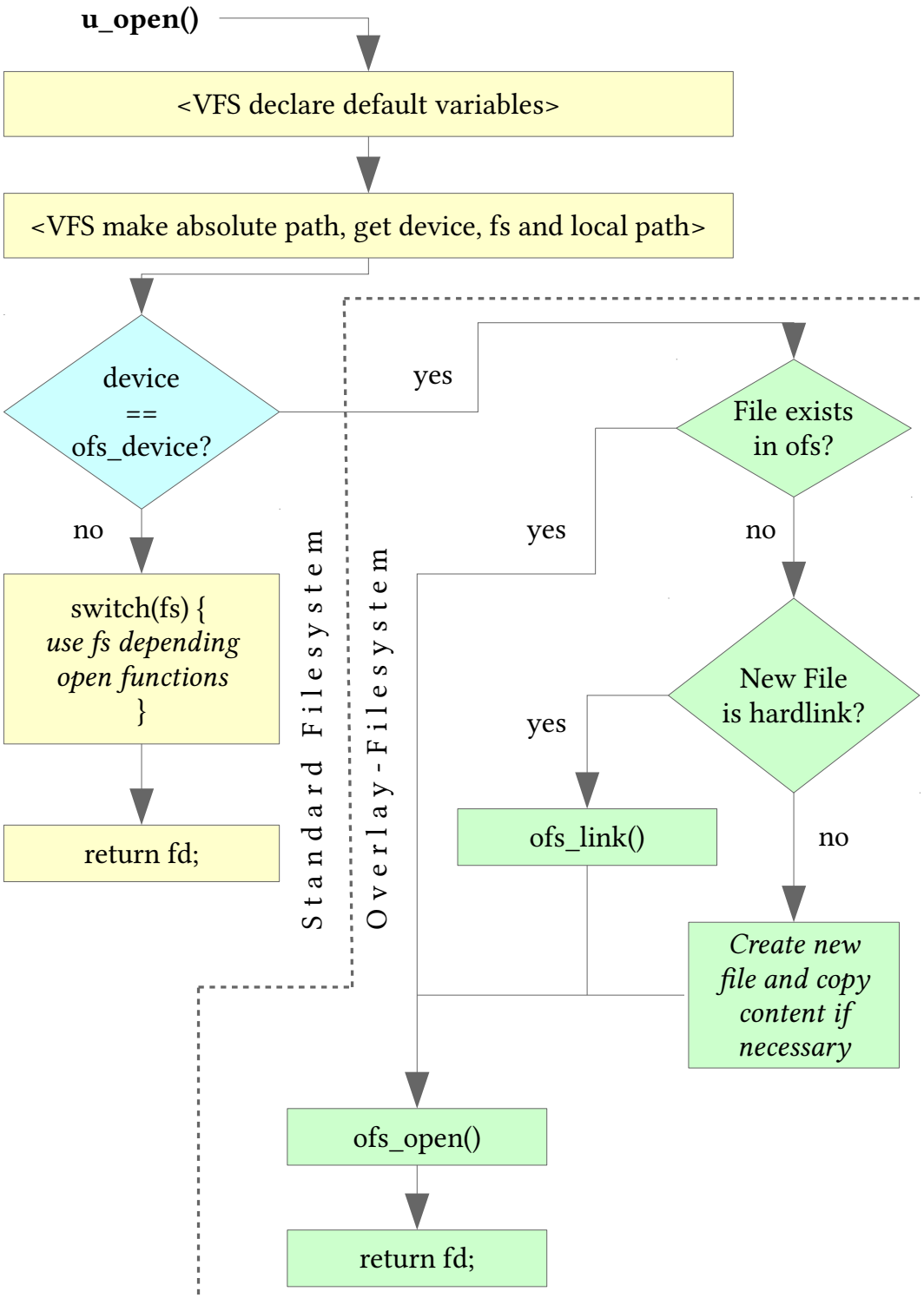


Abbildung 4.1.: Öffnen einer Datei mit Wechsel in das Overlay-Dateisystem

`buffer`, never used.
`bytes_to_copy`, never used.
`ofs_fd`, never used.
`ofs_fs`, used in chunk 56.
`ofs_idx`, used in chunk 56.
Uses `FS_OFS` 54a, `mx_fd` 57a, `O_CREAT` 73, `O_RDWR` 73, `ofs_find_pathname` 26b, `ofs_open` 37b, `ofs_write_stat` 30a, `PAGE_SIZE` 73, `st_size` 73, `u_close` 55, and `u_stat` 79b.

Das Erzeugen des Hardlinks ist vergleichsweise einfach. Es muss nur der bereits existierende Name `path_old` ausgelesen und mit `ofs_link` der Verweis erzeugt werden. Im Anschluss wird durch den Aufruf von `ofs_write_stat()` sichergestellt, dass sich die Inode-Nummer im gerade erzeugten Hardlink befindet.

⟨*extend u_ open create hardlink* 59a⟩≡ (57a) [59a]
```
char path_old[249] = {0};
ofs_get_path_from_index(inode_exists, path_old);
ofs_link(path_old, ofs_path);
struct stat stat_buf = {0};
ofs_stat(path_old, &stat_buf);
ofs_write_stat(inode_exists, &stat_buf);
```
Defines:
`path_old`, used in chunks 46, 47, and 72.
`stat_buf`, never used.
Uses `ofs_get_path_from_index` 25d, `ofs_link` 46d, `ofs_stat` 49d, and `ofs_write_stat` 30a.

Der eben beschriebene Erweiterungs-Chunk kann nun in die `u_open()`- Funktion eingefügt werden, sodass diese nun wie folgt aussieht. Hierbei ist zu erkennen, dass die Funktionalität des Overlay-Dateisystems durch Hinzufügen oder Entfernen des Erweiterungs-Chunks gesteuert werden kann.

⟨*function implementations* 55⟩+≡ ◁55 77b▷ [59b]
```
int u_open (char *path, int oflag, int open_link) {
  ⟨VFS functions: declare default variables ⟩
  ⟨VFS functions: make absolute path, get device, fs and local path ⟩
  ⟨u_open: handle symlink ⟩
  if (scheduler_is_active) {
    ⟨u_open: check permissions ⟩              // see user/group chapter
  }
  int fd;
  ⟨extend u_ open switch to ofs 56⟩
  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 73, 77a, 78a, and 81a.
Uses `error` 73.

Damit wäre die Integration in Ulix abgeschlossen. Alle anderen Erweiterungs-Chunks, die dem groben Schema des Beispiels `u_open()` folgen, sind in Anhang C zu finden.

## 4.3. Funktionstest

Um die korrekte Arbeitsweise des Overlay-Dateisystems zu testen, werden ein paar einfache Kommandos in der Shell eingeben. Zur Vermeidung von weiteren Einflüssen wird hierfür nicht das Wurzeldateisystem überlagert, sondern ein Floppy-Image, das in `/mnt` gemountet wurde. Überprüft wird das Testergebnis mit dem Befehl `ls()`, der eine korrekte Auflistung ausgeben soll. Getestet wird zuerst das Verhalten in Verzeichnissen, die bereits auf dem Floppy-Image existieren. Anschließend folgt eine ähnliche Überprüfung für Dateien und Verzeichnisse, die nur im Overlay-Dateisystem existieren. Abschließend folgt noch ein Test von Verknüpfungen.

**Test in gemischten Verzeichnissen** Für diesen Test wird nun in /mnt eine neue Datei „ofs.txt“ angelegt. Da /mnt bereits existiert, kann im Overlay-Dateisystem die neue Datei angelegt werden, obwohl das zugehörige Verzeichnis dort noch nicht angelegt ist. Anschließend wird mit dem Editor die Zeichenkette „Hello World!“ in die Datei geschrieben und mit `cat` in der Shell ausgegeben. In `/mnt` befinden sich bereits drei Dateien mit Inhalt und der Größe 1680 Byte. Erwartet wird für einen erfolgreichen Test, dass bei einem Aufruf von `ls` in diesem Verzeichnis alle Dateien aufgelistet werden sowie eine korrekte Angabe der Dateigrößen. Die neu erzeugte `ofs.txt` sollte 12 Byte groß sein. Da der Editor aber ein Steuerzeichen an das Ende der Zeichenkette in die Datei schreibt, beträgt die Größe hier 13 Byte.

Das Listing 4.1 zeigt den Ablauf der einzelnen Befehle. Die Bedienung des Editors `vi` ist sinngemäß dargestellt und die Ausgabe von `ls` aus Ulix übernommen. Wie erwartet zeigt die Ausgabe alle Dateien mit deren korrekter Größe an, weshalb der Test als erfolgreich angesehen wird. Aufgrund der virtualisierten Umgebung ist die Systemzeit in Ulix nicht korrekt.

Listing 4.1: Testeingaben in ULIX mit Ausgabe von ls in gemischten Verzeichnissen

```
$ touch /mnt/ofs.txt
$ vi /mnt/ofs.txt
    write "Hello World!" to file
$ cat /mnt/ofs.txt
Hello World!
$ ls -l /mnt/ofs
  1 drwxr-xr-x  6 1000 1000     396 17 Aug 13:00 .
  1 drwxr-xr-x  2 1000 1000     160 17 Aug 13:00 ..
  2 -rw-r--r--  1 1000  100   1680 17 Aug 15:01 test.txt
  3 -rw-r--r--  1 1000  100   1680 17 Aug 15:01 test2.txt
  4 -rw-r--r--  1 1000  100   1680 17 Aug 16:36 aaa.txt
  1 -rw-------  1 1000  100     13 22 Jul  2016 ofs.txt
```

**Test nur im Overlay-Dateisystem** Im Gegensatz zum vorherigen Test wird nun ein Verzeichnis „ofs_dir“ innerhalb des Overlay-Dateisystems erstellt. Dort wird wieder die Datei „ofs.txt“ mit dem Inhalt „Hello World!“ erstellt und mit `cat` ausgegeben. Anschließend wird `ls` aufgerufen. Es wird erwartet, dass nun im Verzeichnis `/mnt/ofs_dir` eine Datei `ofs.txt` mit 13 Byte Größe existiert. Danach wird versucht, das Verzeichnis `/mnt/ofs_dir` zu löschen, was fehlschlagen sollte, da das Verzeichnis nicht leer ist. Erst nach dem Löschen von `ofs.txt` darf `ofs_dir` gelöscht werden, was durch weitere Aufrufe von `ls` überprüft wird. Der Test ist als erfolgreich anzusehen, wenn keine unerwarteten Fehler auftreten und die angezeigten

Auflistungen von `ls` korrekt sind. Hierbei ist im Speziellen darauf zu achten, dass die Statusinformationen für das übergeordnete Verzeichnis /mnt korrekt ausgegeben werden.

Im Listing 4.2 ist nun zu erkennen, dass keine unerwarteten Fehler auftraten und die Ausgabe der Statusinformationen zum übergeordneten Verzeichnis korrekt angezeigt werden. Der Test ist damit ebenfalls als erfolgreich zu betrachten.

Listing 4.2: Testeingaben in ULIX mit Ausgabe von ls nur im Overlay-Dateisysten

```
$ mkdir /mnt/ofs_dir
$ touch /mnt/ofs_dir/ofs.txt
$ vi /mnt/ofs_dir/ofs.txt
     write "Hello World!" to file
$ cat /mnt/ofs_dir/ofs.txt
Hello World!
$ ls -l /mnt/ofs_dir
  1 drwxrwxrwx  3 1000  100       198 22 Jul  2016 .
  1 drwxr-xr-x  2 1000 1000       160 17 Aug 13:00 ..
  1 -rw-------  1 1000  100        13 22 Jul  2016 ofs.txt
$ rmdir /mnt/ofs_dir
rmdir returns -1
$ rm /mnt/ofs_dir/ofs.txt
unlink returns 0
$ ls -l /mnt
  1 drwxr-xr-x  6 1000 1000       396 17 Aug 13:00 .
  1 drwxr-xr-x  2 1000 1000       160 17 Aug 13:00 ..
  2 -rw-r--r--  1 1000  100      1680 17 Aug 15:01 test.txt
  3 -rw-r--r--  1 1000  100      1680 17 Aug 15:01 test2.txt
  4 -rw-r--r--  1 1000  100      1680 17 Aug 16:36 aaa.txt
  1 drwxrwxrwx  2 1000  100       198  6 Aug  2016 ofs_dir
$ rmdir /mnt/ofs_dir
rmdir returns 0
$ ls -l /mnt
  1 drwxr-xr-x  5 1000 1000       396 17 Aug 13:00 .
  1 drwxr-xr-x  2 1000 1000       160 17 Aug 13:00 ..
  2 -rw-r--r--  1 1000  100      1680 17 Aug 15:01 test.txt
  3 -rw-r--r--  1 1000  100      1680 17 Aug 15:01 test2.txt
  4 -rw-r--r--  1 1000  100      1680 17 Aug 16:36 aaa.txt
```

**Test von Verknüpfungen** Da eine wichtige Funktionalität eines Dateisystems die Verfügbarkeit von Verknüpfungen darstellt, wird dies anhand eines kurzen Beispiels getestet. Auf die vorhandene Datei `aaa.txt` wird ein Hardlink erzeugt und anschließend dessen Zugriffmodus auf 0700 geändert. Es wird erwartet, dass sowohl der Zugriffsmodus des Hardlinks als auch der der ursprünglichen Datei geändert wird. Weiterhin sollte die Link-Anzahl bei beiden nun 2 betragen. Zur Überprüfung dient hier wieder `ls`. Abschließend wird noch ein Symlink auf den zuvor erstellten Hardlink erzeugt und der Dateiinhalt mit `cat symlink` ausgegeben. Es wird erwartet, dass hier der Inhalt des Hardlinks angezeigt wird.

Das Listing 4.3 zeigt, dass Verknüpfungen wie erwartet funktionieren. Die Dateiberechtigungen wurden sowohl bei aaa.txt als auch bei hardlink erfolgreich geändert, die Anzahl der

Links ist 2 und die Ausgabe von `symlink` ist ebenfalls wie erwartet. Da die Datei allerdings 1680 Bytes enthält, wurde die Ausgabe von `cat` für das Listing abgeschnitten.

Listing 4.3: Test von Links im Overlay-Dateisystem

```
$ ln /mnt/aaa.txt /mnt/hardlink
$ ls -l /mnt
  1 drwxr-xr-x  6 1000 1000      396 17 Aug 13:00 .
  1 drwxr-xr-x  2 1000 1000      160 17 Aug 13:00 ..
  2 -rw-r--r--  1 1000  100     1680 17 Aug 15:01 test.txt
  3 -rw-r--r--  1 1000  100     1680 17 Aug 15:01 test2.txt
  4 -rwx------  2 1000  100     1680 17 Aug 16:36 aaa.txt
  1 -rwx------  2 1000  100     1680 17 Aug 16:36 hardlink
$ ln -s /mnt/hardlink /mnt/symlink
$ cat /mnt/symlink
01 11111112222222221111111111222222222211111111112222222222
02 11111112222222221111111111222222222211111111112222222222
03 11111112222222221111111111222222222211111111112222222222
04 11111112222222221111111111222222222211111111112222222222
05 11111112222222221111111111222222222211111111112222222222
```

# 5. Zusammenfassung

Diese Arbeit zeigt, dass es auch mit Literate Programming möglich ist, bisherige Programme, die ebenfalls mit dieser Methode geschrieben wurden, zu erweitern. Zunächst erscheint dies nicht offensichtlich, da ein solches Programm in sich als abgeschlossen zu betrachten ist. Eine Erweiterung kann allerdings mit wenig Eingriff in den bisherigen Programm-Code erfolgen, wie diese Arbeit zeigt. Diese Änderungen bisherigen Codes können durch Literate Programming an der jeweiligen Stelle auch ausführlich beschrieben werden, wo die „klassische Programmierung“ nicht so viele Möglichkeiten der Kommentierung und Beschreibung bietet. Zwar können Kommentare im Quelltext helfen die Änderungen verständlich zu machen, allerdings ist trotzdem einiges an geistiger Arbeit nötig, um bei der Implementierung eines Anderen alle Details zu verstehen. Dieser Vergleich bietet sich dem Leser dieser Arbeit ebenfalls, da die Integration in den Ulix -Kernel nur am Beispiel von `u_open()` durchgeführt wurde und die anderen Erweiterungen in Anhang C zu finden sind – welche dort klassisch programmiert, mit Kommentaren im Quelltext, vorliegen.

## 5.1. Kritik

Die Implementierung in dieser Arbeit ist nicht in allen Punkten perfekt. Bei genauerer Betrachtung fällt auf, dass Verzeichnisse zwar mit zunehmender Anzahl an Einträgen wachsen, aber beim Löschen dieser Einträge nicht mehr verkleinert werden. Die Funktion `ofs_get_free_dir_entry()` versucht dies zu kompensieren, indem mögliche Lücken aufgefüllt werden. Im Rahmen dieser Arbeit ist dieses Verhalten allerdings vertretbar, da eine Funktion zur Verkleinerung von Verzeichnissen einiges an Umfang bedarf, aber nicht weiter zum Verständnis von Verzeichnissen beiträgt.

Bei der Wahl der Datenhaltung ist ebenfallls noch Potenzial zur Verbesserung vorhanden. So befindet sich der Block mit den Metadaten am Anfang einer jeden Datei und benötigt 512 Byte in dieser Speicherseite. Dies hat zur Folge, dass bei Speicherzugriffen immer dieser Offset mit einberechnet werden muss und sich keine klare Blockgröße für das Dateisystem ergibt. Zu lösen wäre dies dadurch, dass dieser Meta-Block in eine eigene Tabelle ausgelagert wird, die nur diese Blöcke enthält. Dadurch entsteht zwar neuer Verwaltungsaufwand für genannte Tabelle, allerdings mit dem Vorteil, dass dann eine klare Blockgröße von 4 KiB existiert. Diese feste Blockgröße ist dann gewinnbringend, wenn Speicherseiten in kleinere Einheiten, wie 1 KiB, unterteilt werden und nicht nur den Inhalt einer Datei enthalten. Dadurch kann der Speicher für kleinere Dateien effizienter genutzt werden. Für die Anzahl der Speicherzugriffe bei einen Zugriff auf die Datei entsteht dadurch kein Nachteil. Diese optimierte Lösung wurde allerdings nicht für diese Arbeit in Betracht gezogen, da auch hier der zusätzliche Aufwand in keinem Verhältnis zum Verständnis für Dateisysteme steht.

Auf das Setzen der Fehlervariable `errno` wird in dieser Arbeit komplett verzichtet. Für die meisten Systemcalls in Ulix ist dies ebenfalls so umgesetzt [2, S. 207]. Da diese Fehlervariable später durch Benutzerprogramme ausgelesen werden soll, ist es für diese erforderlich die entsprechenden Fehlernummern zu kennen. Bei einer Analyse der Datei `ulixlib.h` wurde

allerdings klar, dass dort nur acht Fehlernummern hinterlegt sind. Damit besteht unter anderem keine Möglichkeit den Benutzer über zu wenig Speicher zu informieren. Da es nicht zum Ziel dieser Arbeit gehört die Bibliothek von Ulix zu erweitern, wurde `errno` nicht vergessen sondern bewusst nicht implementiert.

Die Anzeige des freien Speichers des Overlay-Dateisystems mit dem Kommando `df` wurde nicht vergessen. Allerdings stellte sich auch hier durch eine Analyse der Datei `df.c` heraus, dass dies nicht möglich ist. Die Implementierung in dieser Datei arbeitet mit einer festen Liste an Geräten, für die nacheinander die Funktion `diskfree` aufgerufen wird. Selbst mit einer Anpassung der `diskfree`-Funktion in Ulix würde hier aber nie das Gerät für das Overlay-Dateisystem übergeben werden. Abgesehen davon, dass ein solches nicht existiert, müsste trotzdem `df.c` angepasst werden, was nicht zum Umfang dieser Arbeit gehört. Ulix lässt den Benutzer trotzdem nicht im Unklaren über den Speicherfüllstand und blendet zu diesem Zweck in der Statusleiste die Anzahl an freien Speicherrahmen ein. Diese Zahl informiert den Benutzer also permanent über den Speicher des Overlay-Dateisystems, weshalb die fehlende Funktionalität über `df` keinen Minuspunkt darstellt.

## 5.2. Ausblick

Die Umsetzung des Overlays über das bisherige Dateisystem ist in dieser Arbeit sehr spezifisch gehalten und auf ein Gerät limitiert. Soll dies auf mehrere Geräte ausgedehnt werden, ist die `ofs_inode`-Struktur um ein Attribut für die Gerätenummer zu erweitern. Die Abfrage, ob in das Overlay-Dateisystem gesprungen wird oder nicht, kann nicht mehr mit einem simplen Vergleich erfolgen, sondern muss eine Liste mit Geräten durchlaufen. Listing 5.1 zeigt diese Änderung mit Pseudocode.

Listing 5.1: Pseudocode-Darstellung für mehrere Geräte

```
ofs_devices = [DEV_1, DEV_2, DEV_3];
if(device is in ofs_devices) {
    switch to ofs-function
)
```

Sollen andere Dateisysteme als Minix unterstützt werden, so muss in der aktuellen Implementierung in zwei Code-Chunks eine Unterscheidung getroffen werden. Einmal in der Erweiterung für `u_open()` wo Daten in das Overlay-Dateisystem kopiert werden und in der Erweiterung für `u_getdent()`. Dort werden Verzeichniseinträge mit `mx_getdent` in ein Overlay-Verzeichnis geschrieben. Alle anderen Erweiterungen nutzen zum Kopieren `u_open()`, sodass hier nicht eingegriffen werden muss.

Mit dem Konzept eines Overlay-Dateisystems lassen sich noch andere praktische Funktionen realisieren. Bietet man dem Benutzer die Möglichkeit am Ende einer Sitzung die Daten zurück auf das Dateisystem zu schreiben, so schafft dies die Möglichkeit von Snapshots. Wurden beispielsweise Änderungen am System getätigt, die eine Fehlfunktion hervorrufen, so kann durch einen Neustart der Ursprungszustand wiederhergestellt werden. Waren die Änderungen erfolgreich, so kann der Inhalt des Overlay-Dateisystems auf das bestehende Dateisystem geschrieben werden und die Änderungen bleiben erhalten.

# Identifier Index

# Abbildungsverzeichnis

# Listings

# Literaturverzeichnis


[1] A. S. Tanenbaum and H. Bos, *Moderne Betriebssysteme =: Modern operating systems*. Always learning, Hallbergmoos: Pearson, 4., aktualisierte auflage ed., 2016. OCLC: 945726741.

[2] Eßer, Hans-Georg and Freiling, Felix, *The Design and Implementation of the ULIX Operating System, University of Erlangen-Nuremberg*. Sept. 2015.

[3] Eßer, Hans-Georg, "Ulix OS – The Literate Operating System." http://ulixos.org/.

[4] S. Brugger, "Implementation eines FAT-Treibers für das Lehrbetriebssystem ULIX," Jan. 2016.

[5] Beraru, Liviu, "Implementation eines Dateisystems und einer RAM-Disk für das Betriebssystem ULIX," Feb. 2013. US Patent 6,308,265.

[6] D. P. Quigley, J. Sipek, C. P. Wright, and E. Zadok, "UnionFS: User- and Community-oriented Development of a Unification Filesystem," in *Proceedings of the 2006 Linux Symposium*, vol. 2, (Ottawa, Canada), pp. 349–362, July 2006.

[7] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[8] M. Teßmer, "Literate Programming zur Dokumentation in der Systemadministration," vol. P-239, (Dresden), pp. 433–445, GI e.V., 2015.

[9] N. Ramsey, *Literate Programming Tools Need Not Be Complex*. Princeton University, Department of Computer Science, 1991.

[10] W. Stallings, *Betriebssysteme: Prinzipien und Umsetzung*. Informatik, München: Pearson Studium, 4., überarb. aufl., "bafög-ausg.-ed., 2005. OCLC: 76760078.

[11] Vodafone Kabel Deutschland GmbH, "Preisliste und Leistungsbeschreibung," Aug. 2016.

[12] Dornig, Cliff, "Implementierung eines SLIP-Moduls für das Lehrbetriebssystem ULIX," July 2014.

# A. Funktionsprototypen für den Ulix-Kernel

Nachfolgend die kompletten Funktionsprototypen im Ulix-Kernel vom Overlay-Dateisystem

⟨*function prototypes* 54d⟩+≡ ◁54d [72]

```
extern int ofs_get_free_fd();
extern int ofs_get_free_inode();
extern void ofs_get_path_from_index(int index, char *buf);
extern int ofs_find_pathname(const char* path);
extern int ofs_find_inode(int mx_inode);
extern int ofs_new_inode_file(int index);
extern int ofs_create_empty_file(const char *path, int mode);
extern void ofs_write_stat(int idx, struct stat *buf);
extern int ofs_write_dir_entry(const char *path, struct dir_entry *entry);
extern void ofs_append_files_to_dir(const char *path);
extern int ofs_position_to_data_index(int position);
extern int ofs_relative_data_position(int page_index, int position);
extern int ofs_absolute_data_position(int page_index, int position);
extern int ofs_bytes_in_page(int page_index, int position);
extern int ofs_open(const char *path, int mode);
extern int ofs_close(int fd);
extern int ofs_lseek(int fd, int offset, int whence);
extern int ofs_write(int fd, const void *buf, int count);
extern int ofs_read(int fd, void *buf, int count);
extern int ofs_ftruncate(int fd, int length);
extern int ofs_stat(const char *path, struct stat *buf);
extern int ofs_getdent(const char *path, int index, struct dir_entry *buf);
extern int ofs_mkdir(const char *path, int mode);
extern int ofs_unlink(const char *path);
extern int ofs_link(const char *path_old, const char *path_new);
extern int ofs_symlink(const char *path_old, const char *path_new);
extern int ofs_rmdir(const char *path);

extern int ofs_chown(const char *path, short owner, short group);
extern int ofs_chmod(const char *path, short mode);
```

Uses `entry` 47a, `idx` 53a, `index` 47d, `ofs_absolute_data_position` 35d, `ofs_append_files_to_dir` 34b, `ofs_bytes_in_page` 36b, `ofs_chmod` 51b, `ofs_chown` 50c, `ofs_close` 38d, `ofs_create_empty_file` 28d, `ofs_find_inode` 27a, `ofs_find_pathname` 26b, `ofs_ftruncate` 45b, `ofs_get_free_fd` 23b, `ofs_get_free_inode` 24a, `ofs_get_path_from_index` 25d, `ofs_getdent` 51d, `ofs_link` 46d, `ofs_lseek` 44b, `ofs_mkdir` 52c, `ofs_new_inode_file` 28b, `ofs_open` 37b, `ofs_position_to_data_index` 34d, `ofs_read` 42b, `ofs_relative_data_position` 35b, `ofs_rmdir` 53c, `ofs_stat` 49d, `ofs_symlink` 47c, `ofs_unlink` 48b, `ofs_write` 39c, `ofs_write_dir_entry` 31e, `ofs_write_stat` 30a, and `path_old` 59a.

# B. Typdefinitionen aus Ulix

Typdefinitionen, Funktionsprototypen und andere Datentypen

⟨*public elementary type definitions* 73⟩≡ (16a) [73]

```
#define PAGE_SIZE 4096

#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

#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

#define SEEK_SET 0    // absolute offset
#define SEEK_CUR 1    // relative offset
#define SEEK_END 2    // EOF plus offset

#define MAX_THREADS 1024

#define CMDLINE_LENGTH 50   // how long can a process name be?
#define MAX_PFD 16   // up to 16 open_ files per process

typedef unsigned char           byte;
typedef unsigned char        boolean;
typedef unsigned short          word;
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)
typedef unsigned int       memaddress;
typedef unsigned int addr_space_id;
typedef unsigned int thread_id;
typedef void (*sighandler_t)(int);

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;

extern unsigned int system_time;
extern thread_id current_task;
extern int ofs_init_complete;

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

addr_space_id addr_space;
thread_id next;       // id of the ``next'' thread
thread_id prev;       // id of the ``previous'' thread
boolean used;
int error;
int exitcode;
int waitfor;    // pid of the child that this process waits for
char cmdline[CMDLINE_LENGTH];
boolean   new;                    // is this thread new?
void      *top_of_thread_kstack;  // extra kernel stack for this thread
int terminal;
int files[MAX_PFD];
char cwd[256];
  sighandler_t sighandlers[32];
  unsigned long sig_pending;
  unsigned long sig_blocked;
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
} TCB;

extern TCB thread_table[MAX_THREADS];
```

```
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
};

extern void *request_new_page ();
extern void release_page (unsigned int pageno);
extern int printf(const char *format, ...);
extern void install_syscall_handler (int syscallno, void *syscall_handler);
extern int strcmp (const char *str1, const char *str2);
extern void *memcpy (void *dest, const void *src, size_t count);
extern void *memset  (void *dest, char val, size_t count);
extern char *strcpy(char *dest, const char *src);
extern char *strncpy(char *dest, const char *src, size_t n);
extern size_t strlen (const char *str);

extern int u_open (char *path, int oflag, int open_link);
extern int u_read      (int fd, void *buf, int nbyte);
extern int u_write     (int fd, void *buf, int nbyte);
extern int u_close     (int fd);
```

Defines:
addr_space_id, never used.
boolean, never used.
byte, used in chunk 20b.
cmdline, never used.
CMDLINE_LENGTH, never used.
context_t, used in chunk 16b.
cwd, never used.
error, used in chunks 55, 59b, and 77–84.
exitcode, never used.
files, never used.
MAX_PFD, never used.
MAX_THREADS, never used.
memaddress, never used.
O_APPEND, used in chunk 38b.
O_CREAT, used in chunks 37c, 47c, 52c, and 57b.
O_RDONLY, used in chunks 24c, 42c, 51d, 57a, 77a, 78a, and 81a.
O_RDWR, used in chunks 31e, 32d, 40a, 42c, 52c, and 57b.
O_WRONLY, used in chunks 40a and 47c.
PAGE_SIZE, used in chunks 34–36, 46b, 49b, and 57b.
S_IFBLK, never used.
S_IFCHR, never used.
S_IFDIR, used in chunk 53a.
S_IFIFO, never used.
S_IFLNK, used in chunk 47d.
S_IFMT, never used.
S_IFREG, used in chunk 29b.
S_IFSOCK, never used.
S_IRGRP, never used.
S_IROTH, never used.
S_IRUSR, used in chunk 37c.
S_IRWXG, never used.
S_IRWXO, never used.
S_IRWXU, never used.

`S_ISGID`, never used.
`S_ISUID`, never used.
`S_ISVTX`, never used.
`S_IWGRP`, never used.
`S_IWOTH`, never used.
`S_IWUSR`, used in chunk 37c.
`S_IXGRP`, never used.
`S_IXOTH`, never used.
`S_IXUSR`, never used.
`SEEK_CUR`, used in chunk 44c.
`SEEK_END`, used in chunk 44c.
`SEEK_SET`, used in chunks 24c, 31–33, 44c, 51d, and 52a.
`sighandler_t`, never used.
`size_t`, used in chunks 41 and 43b.
`st_atime`, used in chunks 30a and 50a.
`st_ctime`, used in chunks 30a and 50a.
`st_dev`, never used.
`st_gid`, used in chunks 30a and 50a.
`st_ino`, used in chunks 30a and 57a.
`st_mode`, used in chunks 30a, 50a, and 83b.
`st_mtime`, used in chunks 30a and 50a.
`st_nlink`, used in chunk 50a.
`st_rdev`, never used.
`st_size`, used in chunks 30a, 50a, and 57b.
`st_uid`, used in chunks 30a and 50a.
`TCB`, never used.
`terminal`, never used.
`thread_id`, never used.
`top_of_thread_kstack`, never used.
`uint`, never used.
`uint16_t`, used in chunk 19d.
`uint32_t`, used in chunk 19d.
`uint64_t`, never used.
`uint8_t`, never used.
`ulong`, never used.
`ulonglong`, never used.
`waitfor`, never used.
`word`, used in chunks 20b and 77b.
Uses `file` 32b 33c 38b 44c 47d 50a 53a, `memcpy` 41a 43b, `ofs_init_complete` 54c, `u_close` 55, and `u_open` 59b.

# C. Erweiterungs-Chunks für Ulix

**chmod** Erweiterung für `u_chmod`. Existiert die Datei im Overlay-Dateisystem, kann direkt `ofs_chmod` ausgeführt werden, andernfalls wird die Datei erst kopiert und dann `ofs_chmod` ausgeführt.

⟨*extend u_ chmod switch to ofs* 77a⟩≡ (77b) [77a]
```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path[248] = {0};
    strncpy(ofs_path, abspath, 247);
    int file_exists = ofs_find_pathname(ofs_path);
    if(file_exists > -1) {
        // File in OFS
        return ofs_chmod(ofs_path, mode);
    } else {
        // File not in OFS -> Copy and chmod
        int fd = u_open(ofs_path, O_RDONLY, FOLLOW_LINK);
        u_close(fd);
        file_exists = ofs_find_pathname(ofs_path);
        // Error while copying?
        if(file_exists > -1) {
            return ofs_chmod(ofs_path, mode);
        }
    }
}
```
Uses `O_RDONLY` 73, `ofs_chmod` 51b, `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_init_complete` 54c, `u_close` 55, and `u_open` 59b.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁59b 78b▷ [77b]
```
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 ⟩    // see user/group chapter
  }

  get_dev_and_path (abspath, &device, &fs, (char*)&localpath);

  ⟨extend u_ chmod switch to ofs 77a⟩

  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, never used.
Uses error 73 and word 73.

**chown** Erweiterung für u_chown. Existiert die Datei im Overlay-Dateisystem, kann direkt ofs_chown ausgeführt werden, andernfalls wird die Datei erst kopiert und dann ofs_chown ausgeführt.

⟨*extend u_ chown switch to ofs* 78a⟩≡ (78b) [78a]

```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path[248] = {0};
    strncpy(ofs_path, abspath, 247);
    int file_exists = ofs_find_pathname(ofs_path);
    if(file_exists > -1) {
        // File in OFS
        return ofs_chown(ofs_path, owner, group);
    } else {
        // File not in OFS -> Copy and chown
        int fd = u_open(ofs_path, O_RDONLY, FOLLOW_LINK);
        u_close(fd);
        file_exists = ofs_find_pathname(ofs_path);
        // Error while copying?
        if(file_exists > -1) {
            return ofs_chown(ofs_path, owner, group);
        }
    }
}
```

Uses O_RDONLY 73, ofs_chown 50c, ofs_find_pathname 26b, OFS_FS 54a, ofs_init_complete 54c, u_close 55, and u_open 59b.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁77b 79b▷ [78b]

```
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);

  ⟨extend u_ chown switch to ofs 78a⟩

  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, never used.
Uses error 73 and file 32b 33c 38b 44c 47d 50a 53a.

**stat** Erweiterung für `u_stat`. Hier wird die Pfadangabe modifiziert, falls diese mit `/`, `/.` oder `/..` endet. Im letzten Fall wird dann das übergeordnete Verzeichnis in den Pfad geschrieben.

⟨*extend u_stat switch to ofs* 79a⟩≡ (79b) [79a]

```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    int parent = 0;
    char ofs_path[248] = {0};
    char *ptr_path = &ofs_path[247];
    strncpy(ofs_path, abspath, 247);
    while(*ptr_path == '\0') ptr_path-;
    // cut off / at the end
    if(*ptr_path == '/') { *ptr_path = '\0'; }
    // filter current and parent directory
    else if(*ptr_path == '.' && *(ptr_path-1) == '/') {*(ptr_path-1) = '\0'; }
    else if(*ptr_path == '.' && *(ptr_path-1) == '.'
            && *(ptr_path-2) == '/') {
                parent = 1; // folder is parent folder
                ptr_path-=2; *ptr_path = '\0';
                while(*ptr_path != '/') ptr_path-;
                *(ptr_path+1) = '\0';
                if(*ptr_path == '/' && ptr_path != ofs_path) { *ptr_path = '\0'; }
                // now ofs_path contains "one directory up"-path
            }
    int file_exists = ofs_find_pathname(ofs_path);
    if(file_exists > -1) {
        // File is in OFS
        return ofs_stat(ofs_path, buf);
    } else if(parent == 1) {
        // File not in OFS -> point localpath to "one directory up" and pass
        // on to existing stat-functions.
        int last_idx = strlen(localpath) - 3;
        if(last_idx > 0) {
            ptr_path = &localpath[last_idx-1];
            while(*ptr_path != '/') ptr_path-;
            *(ptr_path+1) = '\0';
        }
    }
}
```

Uses `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_init_complete` 54c, and `ofs_stat` 49d.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁78b 80b▷ [79b]

```
int u_stat (const char *path, struct stat *buf) {
  ⟨VFS functions: declare default variables ⟩
  ⟨VFS functions: make absolute path, get device, fs and local path ⟩

  ⟨extend u_stat switch to ofs 79a⟩

  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 57 and 83b.
Uses `error` 73.

**mkdir** Erweiterung für `u_mkdir`. Für den Fall, dass das Verzeichnis noch nicht existiert, wird eine neues angelegt.

⟨*extend u_ mkdir switch to ofs* 80a⟩≡ (80b) [80a]
```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path[248] = {0};
    strncpy(ofs_path, abspath, 247);
    int file_exists = ofs_find_pathname(ofs_path);
    if(file_exists < 0) {
        // Dir not in OFS
        return ofs_mkdir(ofs_path, mode);
    }
}
```
Uses `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_init_complete` 54c, and `ofs_mkdir` 52c.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁79b 80d▷ [80b]
```
int u_mkdir (const char *path, int mode) {
  ⟨VFS functions: declare default variables ⟩
  ⟨VFS functions: make absolute path, get device, fs and local path ⟩
  ⟨extend u_ mkdir switch to ofs 80a⟩
  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;
  }
}
```
Defines:
`u_mkdir`, never used.
Uses `error` 73.

**rmdir** Erweiterung für `u_rmdir`. Löscht eventuell existierendes Verzeichnis aus dem Overlay-Dateisystem.

⟨*extend u_ rmdir switch to ofs* 80c⟩≡ (80d) [80c]
```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path[248] = {0};
    strncpy(ofs_path, abspath, 247);
    int file_exists = ofs_find_pathname(ofs_path);
    if(file_exists > -1) {
        // Dir is in OFS
        return ofs_rmdir(ofs_path);
    }
}
```
Uses `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_init_complete` 54c, and `ofs_rmdir` 53c.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁80b 81b▷ [80d]
```
int u_rmdir (const char *path) {
  ⟨VFS functions: declare default variables ⟩
  ⟨VFS functions: make absolute path, get device, fs and local path ⟩
  ⟨extend u_ rmdir switch to ofs 80c⟩
  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_rmdir`, never used.
Uses `error` 73.

**link** Erweiterung für `u_link`. Wenn Target `ofs_path_old` bereits im Overlay-Dateisystem existiert, wird `ofs_link` aufgerufen, wenn nicht, wird diese Datei erst ins Overlay-Dateisystem kopiert und dann verlinkt.

⟨*extend u_link switch to ofs* 81a⟩≡ (81b) [81a]
```
  if(device == OFS_FS && ofs_init_complete == 1) {
      // File should be on OFS
      char ofs_path_old[248] = {0};          char ofs_path_new[248] = {0};
      strncpy(ofs_path_old, abspath, 247);   strncpy(ofs_path_new, abspath2, 247);
      int file_exists = ofs_find_pathname(ofs_path_old);
      if(file_exists > -1) {
          // File in OFS
          return ofs_link(ofs_path_old, ofs_path_new);
      } else {
          // File not in OFS
          int fd = u_open(ofs_path_old, O_RDONLY, FOLLOW_LINK);
          u_close(fd);
          file_exists = ofs_find_pathname(ofs_path_old);
          // error while copying?
          if(file_exists > -1) {
              return ofs_link(ofs_path_old, ofs_path_new);
          }
      }
  }
```

Uses `error` 73, `O_RDONLY` 73, `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_init_complete` 54c, `ofs_link` 46d, `u_close` 55, and `u_open` 59b.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁80d 82b▷ [81b]
```
  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

    ⟨extend u_link switch to ofs 81a⟩

    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
  // printf ("DEBUG: u_link, fs = %d, fs2 = %d, dev = %d, "    // REMOVE_DEBUGGING_CODE
  //         "dev2 = %d,\nlocalpath = %s, localpath2 = %s\n",  // REMOVE_DEBUGGING_CODE
  //         fs, fs2, device, device2, localpath, localpath2); // REMOVE_DEBUGGING_CODE
  return mx_link (device, localpath, localpath2);
}
```

Defines:
  `u_link`, never used.
Uses `dirname` 37d 47a 49a and `error` 73.

**symlink** Erweiterung für `u_symlink`. Hier wird das Target nicht geprüft oder kopiert, da ein Symlink auch falsche Links enthalten kann.

⟨*extend u_ symlink switch to ofs* 82a⟩≡ (82b) [82a]

```
if(device2 == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path_old[248] = {0};      char ofs_path_new[248] = {0};
    strncpy(ofs_path_old, path, 247);  strncpy(ofs_path_new, abspath2, 247);
    int file_exists = ofs_find_pathname(ofs_path_new);
    if(file_exists < 0) {
        // File not in OFS
        return ofs_symlink(ofs_path_old, ofs_path_new);
    } else {
        return -1;
    }
}
```

Uses `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_init_complete` 54c, and `ofs_symlink` 47c.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁81b 83a▷ [82b]

```
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);
  ⟨extend u_ symlink switch to ofs 82a⟩
  if (fs2 != FS_MINIX)  return -1;              // error: not Minix
  return mx_symlink (device2, (char*)path, localpath2);
}
```

Defines:
  `u_symlink`, never used.
Uses `error` 73.

**unlink** Erweiterung für `u_unlink` löscht für den Fall, dass die Datei im Overlay-Dateisystem existiert.

⟨*extend u_ unlink switch to ofs* 82c⟩≡ (83a) [82c]

```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path[248] = {0};
    strncpy(ofs_path, abspath, 247);
    int file_exists = ofs_find_pathname(ofs_path);
    if(file_exists > -1) {
        // File in OFS
        return ofs_unlink(ofs_path);
    }
}
```

Uses `ofs_find_pathname` 26b, `OFS_FS` 54a, `ofs_init_complete` 54c, and `ofs_unlink` 48b.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁82b 84▷ [83a]

```
int u_unlink (const char *path) {
  ⟨VFS functions: declare default variables ⟩
  ⟨VFS functions: make absolute path, get device, fs and local path ⟩
  ⟨extend u_ unlink switch to ofs 82c⟩
  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`, never used.
Uses `error` 73.

**getdent** Erweiterung für `u_getdent`. Entfernt ebenfalls einen eventuell vorhandenen `/` am Ende der Pfadangabe und „wechselt“ für `..` in das übergeordnete Verzeichnis. Wenn dieses existiert, wird `ofs_getdent` aufgerufen. Ist das nicht der Fall, wird das Verzeichnis ins OFS kopiert. Diese Überprüfung wird nur für `index==0` getroffen, da angenommen wird, dass `u_getdent` nur von `ls` aufgerufen wird, was den Index von `0` an hochzählt. Für das übergeordnete Verzeichnis, wird einfach nur `..` in den Dateinamen geschrieben – den Rest erledigt `u_stat`.

⟨*extend u_ getdent switch to ofs* 83b⟩≡ (84) [83b]

```
if(device == OFS_FS && ofs_init_complete == 1) {
    // File should be on OFS
    char ofs_path[248] = {0};
    char *ptr_path = &ofs_path[247];
    strncpy(ofs_path, abspath, 247);
    while(*ptr_path == '\0') ptr_path-;
    if(*ptr_path == '/') *ptr_path = '\0';
    if(index == 1) {
        while(*ptr_path != '/' && (ptr_path - ofs_path) > 0) ptr_path-;
        *ptr_path = '\0';
    }
    int dir_exists = ofs_find_pathname(ofs_path);
    if(dir_exists > -1 && ofs_path[0] != '\0') {
        // File is in OFS
        return ofs_getdent(ofs_path, index, buf);
    } else if (index == 0) {
        //Copy dir to OFS

        // copy mx entries
        struct stat mx_stat = {0};
        u_stat(abspath, &mx_stat);
        ofs_mkdir(ofs_path, mx_stat.st_mode);
        dir_exists = ofs_find_pathname(ofs_path);
        ofs_write_stat(dir_exists, &mx_stat);
        struct dir_entry mx_entry = {0};
        for(int idx_mx_dir = 2;
                mx_getdent (device, localpath, idx_mx_dir, &mx_entry) > -1;
                idx_mx_dir++) {
                    ofs_write_dir_entry(ofs_path, &mx_entry);
        }
```

```
        // copy existing entries from OFS to the created directory
        ofs_append_files_to_dir(ofs_path);
    } else if (index == 1) {
        // Set '..' and return 0
        buf->inode = 1;
        buf->filename[0] = '.'; buf->filename[1] = '.';
        return 0;
    }
}
```

Uses dir_exists 37e, index 47d, ofs_append_files_to_dir 34b, ofs_find_pathname 26b, OFS_FS 54a, ofs_getdent 51d, ofs_init_complete 54c, ofs_mkdir 52c, ofs_write_dir_entry 31e, ofs_write_stat 30a, st_mode 73, and u_stat 79b.

Eingefügt in Ulix :

⟨*function implementations* 55⟩+≡ ◁83a [84]

```
int u_getdent (const char *path, int index, struct dir_entry *buf) {
  ⟨VFS functions: declare default variables ⟩
  ⟨VFS functions: make absolute path, get device, fs and local path ⟩
  ⟨extend u_ getdent switch to ofs 83b⟩
  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;
  }
}
```

Defines:
u_getdent, never used.
Uses error 73 and index 47d.