FOM Hochschule für Oekonomie & Management

Studienzentrum München

Bachelor-Thesis

zur Erlangung des Grades eines

Bachelor of Science (B.Sc.)

über das Thema

Implementation eines Schedulers für das Lehrbetriebssystem Ulix

von

Markus Felsner

Erstgutachter: Dipl.-Math., Dipl.-Inform. Hans-Georg Eßer

Matrikelnummer: 245368

Abgabedatum: 2013-08-26

Inhaltsverzeichnis

Abkürzungsverzeichnis 7

1. Einleitung **8**
1.1. Problemstellung 8
1.2. Forschungsmethodik 9
1.3. Übersicht über angewandte Methoden 10
1.3.1. Methodik zur Literaturrecherche 11
1.3.2. Literate Programming 11
1.3.3. Die Expertenbefragung 12
1.3.4. Das Vorgehensmodell der Software-Entwicklung 12
1.4. Aufbau der Arbeit 13

2. Grundlagen **14**
2.1. Einführung 14
2.2. Kooperative und preemptive Verfahren 14
2.3. CPU-lastige und I/O-lastige Prozesse 15
2.4. Scheduling-Ziele 15
2.5. Betriebsformen 17
2.6. Prozess-Zustände 17
2.7. Zustandsübergänge 18
2.8. Scheduling-Zeitpunkte 19
2.9. Scheduling-Algorithmen 20
2.9.1. First-Come-First-Served (FCFS) 20
2.9.2. Shortest-Job-First (SJF) 20
2.9.3. Shortest-Remaining-Time-Next (SRTN) 21
2.9.4. Round-Robin-Scheduling (RR) 21
2.9.5. Prioritäten-Scheduling 21
2.9.6. Multilevel-Scheduling (ML) 22
2.9.7. Feedback-Scheduling 23
2.9.8. Multilevel-Feedback-Queue-Scheduling (MLFQ) 23
2.9.9. Proportional-Share-Scheduling 24
2.9.10. Guaranteed-Scheduling 24
2.9.11. Rate-Monotonic-Scheduling, Deadline-Monotonic-Scheduling, Earliest-Deadline-First und Least-Laxity-First 25

3. Scheduling-Verfahren in Betriebssystemen **26**
3.1. Lehrbetriebssysteme 26
3.1.1. xv6 26

3.1.2. PintOS . . . 27
3.1.3. Xinu . . . 27
3.1.4. GeekOS . . . 28
3.1.5. Minix 2 . . . 28
3.1.6. Minix 3 . . . 29
3.2. Für den produktiven Einsatz geeignete Betriebssysteme . . . 29
3.2.1. Unix . . . 29
3.2.2. FreeBSD . . . 30
3.2.3. Windows . . . 32
3.2.4. Linux . . . 33
3.3. Zusammenfassung der Betriebssystem-Untersuchung . . . 34

4. Entwurf **36**
4.1. Auswertung des Expertenfragebogens . . . 36
4.2. Ergebnis der Expertenbefragung . . . 36
4.3. Wahl des Algorithmus . . . 37
4.3.1. Nicht geeignete Verfahren . . . 37
4.3.2. Geeignete Verfahren . . . 37
4.3.3. Das gewählte Verfahren: MLFQ . . . 38
4.4. Gestaltung des MLFQ-Verfahrens . . . 38
4.4.1. MLFQ-Verfahren: Das Basispinzip . . . 38
4.4.2. Ausschluss . . . 39
4.5. Spezifikation . . . 39

5. Implementierung **41**
5.1. Das chronologische Vorgehen . . . 41
5.2. Einführung von Prozess-Prioritäten . . . 41
5.3. Die Implementation mittels Literate Programming . . . 42
5.3.1. Die neue Ready-Queue . . . 42
5.3.2. Einfügen eines Prozesses in die Ready-Queue . . . 44
5.3.3. Die Auswahl eines neuen Prozesses . . . 46
5.3.4. Das erste Programm: Die Funktion start_program_from_disk() . . . 46
5.3.5. Anpassen der Funktion ulix_fork() . . . 47
5.3.6. Die zentrale Scheduling-Funktion scheduler() . . . 48
5.3.7. Dynamische Quanten . . . 53
5.3.8. System Call setpriority() . . . 54
5.3.9. System Call getpriority() . . . 54
5.3.10. System Call nice() . . . 55

6. Evaluation **57**
6.1. Testumgebung . . . 57
6.2. Testphasen . . . 57
6.2.1. Regressionstest . . . 57
6.2.2. Systemtest . . . 58

6.3. Fallstudien . . . 58
6.3.1. Fallstudie zu Anforderung 1a: . . . 58
6.3.2. Fallstudie zu Anforderung 1b . . . 58
6.3.3. Fallstudie zu Anforderung 1c . . . 58
6.3.4. Fallstudie zu Anforderung 2 . . . 59
6.3.5. Fallstudie zu Anforderung 3 . . . 59
6.3.6. Fallstudie zu Anforderung 4 . . . 59
6.3.7. Fallstudie zu Anforderung 5 . . . 60
6.3.8. Fallstudie zu Anforderung 6 . . . 61
6.3.9. Fallstudie zu Anforderung 7 . . . 61
6.3.10. Fallstudie zu Anforderung 8 . . . 62
6.3.11. Fallstudie zu Anforderung 9 . . . 63
6.3.12. Fallstudie zu Anforderung 10 . . . 63

7. Schluss **65**
7.1. Zusammenfassung . . . 65
7.2. Fazit . . . 65
7.3. Kritik an der Arbeit . . . 66
7.4. Ausblick . . . 67

A. Fragebogen der Expertenbefragung **68**

B. Testprogramme **69**

C. Simulator **73**

Identifier Index **79**

Index **80**

Literaturverzeichnis **80**

Abbildungsverzeichnis

1.1. Design-Science-Forschungsrichtlinien. Quelle: Entnommen aus Hevner et al., 2004, S. 83 10

2.1. Prozess-Zustandsmodell. Darstellung in Anlehnung an Glatz, 2006, S. 111 . 18

5.1. Die konzeptuelle Organisation einer doppelt verketteten Liste die Prozesse 4 und 2 mit priokey 25 bzw. 14 enthält. Darstellung in Anlehnung an Comer, 2011, S. 50. 42
5.2. Programmablaufplan der Funktion insert() 45
5.3. Programmablaufplan der Funktion dequeue() 47
5.4. Schematische Darstellung der Funktionsaufrufe der Funktion scheduler() durch andere Funktionen 48
5.5. Programmablaufplan der Funktion scheduler() 50
5.6. Programmablaufplan der Funktion timer_handler() 51

Tabellenverzeichnis

1.1. Literatursuchergebnisse in Meta-Such-Maschinen 11

3.1. Scheduling-Verfahren der untersuchten Betriebssysteme 35

4.1. Zusammenfassung des Ergebnisses der Expertenbefragung 37

5.1. Zuordnung Prozess-Prioritäten zu Quanten 53

Abkürzungsverzeichnis

RR Round-Robin

SJF Shortest-Job-First

SRTN Shortest-Remaining-Time-Next

ML Multi-Level

MLFQ Multi-Level-Feedback-Queue

CPU Central-Processing-Unit

I/O Input/Output

RMS Rate-Monotonic-Scheduling

DMS Deadline-Monotonic-Scheduling

EDF Earliest-Deadline-First

LLF Least-Laxity-First

1. Einleitung

Betriebssysteme haben die Aufgabe den Benutzerprogrammen ein einfaches, klares Modell des Computers zur Verfügung zu stellen und außerdem Ressourcen wie Prozessoren, Arbeitsspeicher, Festplatten, Drucker und viele weitere Geräte, zu steuern (vgl. Tanenbaum, 2009, S. 30). Betriebssysteme wie Linux oder Windows sind sehr komplex. Der Umfang des Quellcodes dieser liegt im Bereich von fünf Millionen Codezeilen (vgl. Tanenbaum, 2009, S. 31). Um interessierten Gruppen, wie Studenten, den Aufbau von Betriebssystemen zu vermitteln, wurden schlankere, eigens für Ausbildungswecke geeignete Lehrbetriebssysteme entworfen. Anhand eines Lehrbetriebssystems (*instructional* oder *educational operating system*) können Betriebssystem-Abläufe gut veranschaulicht werden. Einer dieser Abläufe ist die Steuerung der Ressource Prozessor, welches das zentrale Thema dieser Arbeit darstellt.

In diesem Kapitel wird zuerst auf die Problemstellung eingegangen. Diese beinhaltet die Beschreibung des Untersuchungsgegenstandes, der Untersuchungsperspektive, sowie die Darstellung des Zieles der Arbeit. Anschließend wird die Forschungsmethodik, sowie die in dieser Bachelor-Thesis angewandten Methoden erläutert.

1.1. Problemstellung

Ulix-i386 ist ein Lehrbetriebssystem, das von Dipl.-Math., Dipl.-Inform. Hans-Georg Eßer und Prof. Dr. Felix Freiling an der Universität Erlangen-Nürnberg entwickelt wird. Dieses Unix-ähnliche Betriebssystem ist, wie kein anderes Lehrbetriebssystem, vollständig mittels der Programmiermethode *Literate Programming* implementiert. Da sich Ulix noch in der Entwicklung befindet, fehlen noch einige Betriebssystem-Komponenten. In der Ulix-Version 0.05, auf der diese Arbeit ursprünglich aufsetzt, ist ein primitiver Round-Robin-Scheduler vorhanden. Hieraus entsteht der Bedarf, einen CPU-Scheduler zu implementieren, der den Ansprüchen an ein Lehrbetriebssystem genügt.

So kann als *Untersuchungsgegenstand* die Betriebssystem-Komponente *CPU-Scheduler* definiert werden. Um zu ermitteln, welches Verfahren sich für Ulix eignet müssen folgende grundlegende Fragen beantwortet werden:

- Welche Funktion erfüllt ein CPU-Scheduler?
- Welche Kategorien von CPU-Schedulern gibt es?
- Welche Arten von von Betriebssystemen existieren?
- Welche Arten von Prozessen gibt es und in welchen Zuständen kann sich ein Prozess in einem Betriebssystem befinden?

- Wann muss ein CPU-Scheduler eine Entscheidung treffen?
- Welche CPU-Scheduling-Verfahren gibt es und welches Verfahren eignet sich für welche Art von Betriebssystem?

Neben diesen Fragen zu den Grundlagen des CPU-Schedulings werden für die Ermittlung der Gestaltung des Verfahrens auch andere, in der Praxis eingesetzte Betriebssysteme untersucht. So wird ermittelt welche Scheduling-Verfahren in anderen Unix-ähnlichen Betriebssystemen im Einsatz sind. Hierzu werden Lehrbetriebssysteme und für den produktiven Einsatz geeignete Betriebssysteme dieser Kategorie untersucht. Die *Untersuchungsperspektive* hierbei ist die Umsetzung der Implementierung der Scheduling-Verfahren in diesen Betriebssystemen. Auf Basis dieser Informationen soll das *Ziel der Arbeit* erreicht werden:
Es soll ein funktionaler CPU-Scheduler in Ulix implementiert werden. Dies bildet die Voraussetzung für das *übergeordnete Ziel*:
Es soll Studenten das Themengebiet CPU-Scheduling leicht verständlich vermittelt werden können.

1.2. Forschungsmethodik

Um der Forderung nach einer ausreichend strukturierten und wissenschaftlichen Arbeitsweise gerecht zu werden, wurde eine Methode verwendet, welcher dieser Bachelor-Thesis als leitender Rahmen dient. Die Forschungsmethodik ergibt sich aus der Problemstellung. Da diese Arbeit zum Ziel hat, das *Artefakt* CPU-Scheduler zu entwickeln, eignet sich die *Design-Science-Methodik*. Sie ist ein anerkanntes Verfahren, um gezielt zu wissenschaftlichen Erkenntnissen im Sinne der Gestaltung von neuartigen IT-Artefakten zu gelangen. Allgemein wird zwischen materiellen und abstrakten Artefakten unterschieden. Zu abstrakten Artefakten zählen Theorien, Konstrukte, Modelle und Methoden. Instanzen in Form von Implementationen und Methoden werden als materielle Artefakte bezeichnet (vgl. Hevner et al., 2004, S. 77). Letzteres ist auch der Ergebnistyp dieser Arbeit.

Die Anwendung der Design-Science-Methodik erlaubt es, die Ergebnisse dieser Bachelor-Thesis überprüfbar und reproduzierbar zu machen (vgl. Hevner et al., 2004, S. 76). Sie definiert sieben Richtlinien, an denen sich diese Arbeit orientiert (siehe Abbildung 1.1). Die Richtlinien wurden wie folgt in dieser Arbeit umgesetzt:

1. *Entwicklung von Artefakten:* Das Ergebnis des Design-Science ist ein materielles Artefakt. Das Ergebnis ist die Implementierung des identifizierten Scheduling-Verfahrens in Ulix (vgl. Hevner et al., 2004, S. 82). Dies ist Inhalt des Kapitels 5.

2. *Problemrelevanz:* Das identifizierte Problem ist relevant, da in Ulix nur ein provisorischer CPU-Scheduler vorhanden ist und der Bedarf an der Lösung dieses Problems besteht. Die Relevanz ist, wie in der Problemstellung (Kapitel 1.1) beschrieben, gegeben (vgl. Hevner et al., 2004, S. 86).

3. *Evaluierung des Artefaktes:* Die Qualität und Nützlichkeit des entwickelten Artefaktes werden vor dem Hintergrund des identifizierten Problems gezeigt (vgl. Hevner et al., 2004, S. 87). Die Evaluation erfolgt in Kapitel 6.

Design-Science Research Guidelines	
Guideline	**Description**
Guideline 1: Design as an Artifact	*Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.*
Guideline 2: Problem Relevance	*The objective of design-science research is to develop technology-based solutions to important and relevant business problems.*
Guideline 3: Design Evaluation	*The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.*
Guideline 4: Research Contributions	*Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.*
Guideline 5: Research Rigor	*Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.*
Guideline 6: Design as a Search Process	*The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.*
Guideline 7: Communication of Research	*Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.*

Abbildung 1.1.: Design-Science-Forschungsrichtlinien. Quelle: Entnommen aus Hevner et al., 2004, S. 83

4. *Beitrag zur Forschung:* Der Beitrag zur Forschung wird geleistet. Dieser Beitrag ist das entwickelte Artefakt (vgl. Hevner et al., 2004, S. 88). Dieses Artefakt wird mittels anerkannter Methoden erstellt (Kapitel 1.3). Die Implementierung des Scheduling-Verfahrens wird zum ersten Mal mittels Literate Programming umgesetzt.

5. *Korrektheit der Vorgehensweise:* Sowohl die Entwicklung als auch die Evaluierung des Artefaktes wird sorgfältig und exakt durchgeführt. Bei der Entwicklung wird darauf geachtet, dass das Artefakt, für interessierte Gruppen, wie Studenten, nützlich ist. Der Fokus liegt darauf, die Güte des Artefaktes in Bezug auf die Lösung des anfangs identifizierten Problems darzulegen (vgl. Hevner et al., 2004, S. 89).

6. *Design-Science als Suchprozess:* Es werden im Rahmen der Migration mehrere Entwicklung- und Evaluierungszyklen durchlaufen um eine befriedigende Lösung für auftretende Probleme zu finden (vgl. Hevner et al., 2004, S. 91). Die Entwicklungszyklen werden im Kapitel „Implementation“ (Kapitel 5) dargestellt.

7. *Präsentation:* Der Fortschritt der Forschung, sowie die Forschungsergebnisse werden im Kolloquium einem wissenschaftlichen, als auch einem wirtschaftlich ausgerichteten Publikum präsentiert (vgl. Hevner et al., 2004, S. 92).

1.3. Übersicht über angewandte Methoden

Innerhalb des in dieser Bachelor-Thesis verwendeten Design-Science-Forschungsprozesses finden verschiedene wissenschaftliche Methoden Anwendung:

1.3.1. Methodik zur Literaturrecherche

Die Methodik der Literaturrecherche wurde auf Basis der Arbeit von vom Brocke et al. vorgenommen (vom Brocke et al., 2009).

- Zur Grundlagenforschung wurden Betriebssystem-Fachbücher herangezogen, die das Thema CPU-Scheduling detailliert beschreiben. Die Anzahl (größer 1000) der Google-Scholar-Zitationen dienten hier als Indikator für relevante Literatur.

- Die in Bearbeitung befindliche Dissertation von H.-G. Eßer und der bestehende Ulix-Sourcecode diente der Einarbeitung in das Betriebssystem Ulix.

- Es wurden Meta-Such-Maschinen mit den identifizierten Suchbegriffen nach relevanten Artikeln durchsucht. Die MIS Journal Rankings (Saunders, 2005) und die VHB-Liste (vhbonline, 2011) wurden hierzu zur Eingrenzung der State-of-the-Art-Artikel herangezogen. In letzterem waren nur Artikel der Ranking-Klassen A-D relevant.

  Eine Übersicht findet sich in Tabelle 1.1. Die Spalte *Suche* gibt an, ob in Titel, Zusammenfassungen oder Veröffentlichungen nach dem Suchbegriff gesucht wird, *Abdeckung* den Zeitraum, in dem gesucht wird, *Treffer* die Anzahl der erzielten Treffer mit dem eingegebenen Suchbegriff und *Gesichtet* die Anzahl der Treffer die gelesen wurden.

  Es wurden folgende Suchwörter verwendet: *Scheduler*, *Echtzeit*, *Interaktiv*, *Multiprozessor*, *Education*, *Betriebssystem*, *Scheduling*, *Lehrbetriebssystem*, *Literate Programming*, *Priorität*, sowie Namen der verschiedenen Schedulingalgorithmen und deren englischen Bezeichnungen.

Tabelle 1.1.: Literatursuchergebnisse in Meta-Such-Maschinen

Meta-Such-Maschine	**Suche**	**Abdeckung**	**Treffer**	**Gesichtet**
ACM Digital Library	all fields	1956+	650	125
IEEEXplore	all fields	1956+	750	45
ScienceDirect	all fields	1956+	150	10
SpringerLink	all fields	1956+	100	15
CiteSeer	all fields	1956+	240	35

- Die Literaturverzeichnisse der gefundenen Journals dienten als Basis für Vorwärts- und Rückwärts-Suche nach Webster (vgl. Webster & Watson, 2002, S. 16).

1.3.2. Literate Programming

Literate Programming nach Knuth (vgl. Knuth, 1984, S.97 - S. 111) ist eine Programmiermethode, die es ermöglicht, Programme so zu gestalten, dass sie von Menschen gut lesbar und gut zu verstehen sind. Der Grund, Literate Programming zu verwenden, resultiert daraus, dass Quellcode häufig nicht oder unzureichend dokumentiert ist. Das gesamte Betriebssystem Ulix ist mittels dieser Methode implementiert. Sie wurde Anfang der 80er Jahre von Donald E. Knuth ins Leben gerufen. Das Prinzip ist folgendes: Es existiert eine Datei, aus welcher zum einen der Sourcecode und zum anderen die Beschreibung

extrahiert wird. Auf diese Arbeit angewandt bedeutet das: Sie wurde mit LaTeX verfasst. In diesem LaTeX-Dokument befindet sich auch der gesamte Quellcode dieses Projektes. Mittels dem Literate-Programming-Werkzeug *Noweb* ist es möglich, aus diesem Dokument ausführbare Ulix-Code-Dateien zu generieren. Gleichzeitig wird die PDF-Datei erzeugt die Sie gerade lesen. Es ist zudem eine n-zu-m-Verbindung möglich, d. h. n LaTeX-Dokumente und m Sourcecode-Dateien. So wird, vereinfacht dargestellt, aus diesem Dokument (bachelorarbeit.nw) und dem bestehenden Dokument der Dissertation von H.-G. Eßer (diss-hgesser-ulix.tex) der Sourcecode in gemeinsame Sourcecode-Dateien extrahiert (ulix.c, ulix.h) (Smith, 1996).

1.3.3. Die Expertenbefragung

Zur Ermittlung der Gestaltung des zu implementierenden CPU-Schedulers wurde eine Expertenbefragung durchgeführt. Hierzu wurde ein halbstandardisierter Fragebogen erstellt (Anhang A), welcher mehrere ordinal-skalierte und eine offene Fragestellung beinhaltet. Die Fragen gestalten sich auf Basis der Kriterien die ein Scheduler erreichen kann nach Tanenbaum (vgl. Tanenbaum, 2009, S. 197). Das Vorgehen zur Auswertung und das Ergebnis der Befragung finden sich in Kapitel 4.1. Die Auswahl des Experten erfolgte unter berücksichtigung der fachlichen Kompetenz als auch der Erfahrung mit Ulix. Es wurde ein Experte befragt.

1.3.4. Das Vorgehensmodell der Software-Entwicklung

Um der Forderung einer professionellen Entwicklung der Software gerecht zu werden, bedarf es eines Vorgehensmodells. Für diese Arbeit eignet sich das iterativ-inkrementelle Vorgehensmodell. Durch *iteratives* Vorgehen wird ein vorgegebenes Problem durch wiederholte Bearbeitung gelöst. Der Zusatz *inkrementell* bedeutet, dass bei jedem Durchlauf nicht nur das existierende Ergebnis verfeinert wird, sondern dass neue Funktionalität hinzukommt. Dies bedeutet, dass zuerst ein Inkrement der Kernfunktionalität (auch Basisinkrement) entwickelt wird, welches dann mit weiteren Inkrementen um neue Funktionalität ergänzt werden kann (vgl. Kleuker, 2010, S. 30). Dieses Modell wurde aus folgenden Gründen gewählt:

- Es ist flexibel, was die Änderungen von Anforderungen während der Entwicklungszeit betrifft. Dieses Kriterium ist von Bedeutung, da Ulix während der Erstellung dieser Arbeit weiterentwickelt wurde und dies auch Komponenten betrifft, wie z. B. Dispatcher-Funktionalität, die in direktem Zusammenhang mit der Arbeit stehen.
- Die Implementierung und Migration eines neuen Scheduling-Verfahrens sind komplex, da in vielen Bereichen des Ulix-Quellcodes Änderungen erforderlich sind. Zudem erweisen sich die Änderungen am Betriebssystem-Kernel als heikel. Zu viele Modifikationen auf einmal ziehen im Fehlerfall eine aufwendige Fehlersuche nach sich. Diese Faktoren erfordern es, die Entwicklung in mehreren, aufeinander aufbauenden, Inkrementen durchzuführen.

1.4. Aufbau der Arbeit

Nachdem dieses Kapitel die Schwerpunkte Problemstellung und Forschungsmethodik behandelt, wird in Kapitel 2 auf die Grundlagen des CPU-Schedulings eingegangen. In Kapitel 3 werden in der Praxis eingesetzte Lehrbetriebssysteme und kommerzielle Betriebssysteme untersucht. Die Anforderungsanalyse, die in Kapitel 4 beleuchtet wird, bildet die Basis für die in Kapitel 4 beschriebene Implementation des Schedulers. Schließlich wird in Kapitel 5 eine Zusammenfassung beschrieben, Fazit gezogen, Kritik an der Arbeit geübt und Ausblick für die weitere Forschung gegeben.

2. Grundlagen

Dieses Kapitel erläutert die Grundlagen des CPU-Schedulings. In der Einführung werden eine Definition und Abgrenzung des Begriffs Scheduler vorgenommen. Neben den unterschiedlichen Scheduler-Kategorien werden die verschiedenen Prozessarten und Scheduling-Ziele vorgestellt. Die möglichen Betriebsformen werden erläutert und verschiedene Prozess-Zustände und Zustandsübergänge dargestellt. Anschließend werden die Zeitpunkte, an denen Scheduler-Entscheidungen getroffen werden, untersucht und die verschiedenen Scheduling-Verfahren erörtert. Innerhalb der Kapitel wird versucht, dem Leser einen Einblick in den Ist-Zustand von Ulix zu vermitteln.

2.1. Einführung

Immer wenn eine unteilbare Ressource, wie eine I/O-Komponente oder der Prozessor (CPU) bzw. der Prozessorkern, effizient vergeben werden soll, bedarf es einer Komponente die das steuert. Diese Komponente wird als Scheduler (englisch *scheduler* = Arbeitsplaner; *to schedule* = einteilen) bezeichnet (vgl. Stallings, 2001, S. 394).

In einem Betriebssystem gibt es verschiedene Typen von Schedulern. Neben dem I/O-Scheduler, welcher die Lese- und Schreibzugriffe auf blockorientierte Geräte steuert (vgl. Tanenbaum, 2009, S. 895), existieren drei weitere, welche dem Typ des Prozess-Schedulings zugeordnet werden können. Diese werden als Long-Term-Scheduler (langfristiger Scheduler), Medium-Term-Scheduler (mittelfristiger Scheduler) und Short-Term-Scheduler (kurzfristiger Scheduler) bezeichnet. Das Long-Term-Scheduling war in früheren Stapelverarbeitungssystemen verbreitet und befasst sich mit der Verwaltung der im Betriebssystem ankommenden Jobs. Das Medium-Term-Scheduling befasst sich mit der Vergabe des Speichers, wenn das System Swapping unterstützt (siehe hierzu auch Kapitel 2.6) (vgl. Mandl, 2013, S. 111). In modernen Betriebssystemen existiert meist nur noch der Short-Term-Scheduler. Dieser, auch CPU-Scheduler (kurz Scheduler) genannt, hat die Aufgabe, in einem Betriebssystem, in dem mehrere Prozesse um eine oder mehrere CPUs konkurrieren, zu entscheiden, welchem Prozess bzw. Thread die CPU als nächstes zugeteilt wird.

2.2. Kooperative und preemptive Verfahren

Welcher Prozess als nächstes gewählt wird hängt von der Scheduling-Strategie ab. Es werden zwei Kategorien von Scheduling-Strategien unterschieden; die *kooperative* (nicht verdrängende, nicht unterbrechende, non preemptive) und die *preemptive* (verdrängende oder unterbrechende) Strategie:

- Bei der kooperativen Strategie wählt der Scheduler einen Prozess. Dieser läuft so lang, bis er blockiert, sich beendet oder die CPU freiwillig freigibt (vgl. Tanenbaum,

2009, S. 195). Der Name *kooperativ* bezieht sich auf die Prozesse, welche sich untereinander kooperativ verhalten müssen. Hier kann das Betriebssystem keinem Prozess die CPU entziehen (vgl. Hansen, 1973, S. 195). Tanenbaum weist daher bei dieser Strategie auf die Gefahr hin, dass ein Prozess unter Umständen ewig laufen und somit das System zum Absturz bringen kann (vgl. Tanenbaum, 2009, S. 195).

- Bei der preemptiven Strategie, die in Ulix zum Einsatz kommt, kann der laufende Prozess bei jedem Timer-Interrupt unterbrochen werden und die CPU einem anderen Prozess zugeteilt werden. Der unterbrochene Prozess wird dann weiter ausgeführt sobald der Scheduler entscheidet, dass ihm wieder die CPU zugeteilt wird (vgl. Tanenbaum, 2009, S. 196). So entsteht der Eindruck, als würden die Prozesse gleichzeitig laufen. Man spricht hier von *Quasiparallelität* (vgl. Tanenbaum, 2009, S. 124). Die Technik nennt sich *Multiprogrammierung* (vgl. Comer, 2011, S. 14). In Multiprozessor- oder Multikern Rechnerarchitekturen spricht man von echter Parallelität, wenn so viele Prozesse ausgeführt werden können, wie CPUs zur Verfügung stehen. Die Systematik der abwechselnd laufenden Prozesse ist aber die gleiche wie bei Einprozessor-Maschinen (vgl. Mandl, 2013, S. 109). Preemptives Scheduling kann schnell auf dringende Anfragen reagieren. Allerdings ist der Preis hierfür erhöhte Komplexität und Aufwand (vgl. Hansen, 1973, S. 195).

2.3. CPU-lastige und I/O-lastige Prozesse

Die meisten Prozesse wechseln ständig zwischen zwei Zuständen: Entweder sie rechnen (auch CPU-Burst) oder sie warten auf I/O (auch I/O Burst) – beispielsweise weil ein Systemaufruf erfolgt, wenn aus einer Datei gelesen wird. Die Prozessausführung beginnt mit einem CPU-Burst, darauf folgt ein I/O-Burst, darauf folgt wieder ein CPU-Burst und so weiter. Schließlich endet der letzte CPU-Burst mit einem `exit`-Systemaufruf. Prozesse, die überwiegend rechnen, werden als CPU-lastige Prozesse bezeichnet. Prozesse, die viel Zeit mit dem Warten auf ein anderes Betriebsmittel verbringen, nennt man I/O-lastig (vgl. Silberschatz, 2010, S. 184).

2.4. Scheduling-Ziele

Es gibt unterschiedliche Scheduling-Ziele (auch *Scheduling-Kriterien*) (vgl. Stallings, 2001, S. 400) die ein Scheduling-Verfahren erreichen kann. Je nach Umgebung – der Form des Betriebssystems bzw. dem Bereich der Anwendung können diese Ziele variieren. Grundsätzlich werden mehrere Ziele unterschieden (vgl. Tanenbaum, 2009, S. 197 und Stallings, 2001, S. 400):

- *Fairness*: Fairness kann unterschiedlich interpretiert werden. Zum einen gibt es Fairness zwischen den Prozessen, d. h. jeder Prozess soll einen gleichen Anteil der Rechenzeit bekommen (vgl. Tanenbaum, 2009, S. 197). Zum anderen gibt es Fairness zwischen den Benutzern. Hiermit ist gemeint, dass die CPU-Nutzung, im Gegensatz zur gleichen Verteilung zwischen Prozessen, gleichmäßig zwischen den Systembenut-

zern oder Gruppen verteilt wird (vgl. Larmouth, 1975, S. 29 und Kay & Lauder, 1988, S. 44).

- *Prioritäteneinsatz*: Wenn Prozessen Prioritäten zugewiesen werden, muss das Scheduling-Verfahren höher priorisierte Prozesse gegenüber niedriger priorisierten Prozessen bevorzugen. Zudem muss sichergestellt werden, dass Prozesse nicht „verhungern“ können (siehe hierzu Kapitel 2.9.5).

- *Ressourcen-Balance*: Alle Hardware-Komponenten sollen ausgelastet werden. Beispielsweise ist es sinnvoll, gleichzeitig CPU-lastige und I/O-lastige Prozesse im Speicher zu halten. So befindet sich weder die CPU noch andere Hardware-Komponenten im Leerlauf (vgl. Tanenbaum, 2009, S. 198).

- *Durchsatz*: Maximieren der fertiggestellten Jobs pro Zeiteinheit. Gemessen wird dies in der Anzahl der Jobs, die in einer bestimmten Zeit erledigt werden (vgl. Tanenbaum, 2009, S. 198).

- *Durchlaufzeit*: Minimieren der Zeit vom Start bis zur Beendigung eines Jobs. Hierzu zählen die Zeit für die Ausführung sowie die Zeit für das Warten auf Ressourcen (vgl. Tanenbaum, 2009, S. 198).

- *CPU-Ausnutzung*: Die CPU soll möglichst häufig durch Prozesse belegt sein und soll sich möglichst wenig im Idle-Zustand befinden (was beispielsweise auch passiert wenn alle Prozesse blockiert sind). Gemessen wird dies am Prozentsatz der Zeit, in der die CPU beschäftigt ist (vgl. Tanenbaum, 2009, S. 199).

- *Reaktionszeit oder Antwortzeit*: Die Zeit, die zwischen einer Anfrage und der ersten Reaktion vergeht (vgl. Tanenbaum, 2009, S. 199).

- *Proportionalität*: Die intuitiven Erwartungen des Benutzers sollen erfüllt werden (vgl. Tanenbaum, 2009, S. 199). Dies soll unabhängig von der Auslastung des Systems sichergestellt sein. Dies ist ein weiches Kriterium.

- *Deadlines*: Aufgaben müssen in vorgegebener Zeit ausgeführt werden. Deadlines müssen eingehalten werden (vgl. Tanenbaum, 2009, S. 199).

- *Vorhersagbarkeit*: Das Scheduling muss gut vorhersagbar arbeiten. Am Beispiel Multimedia-Systeme kann dies erläutert werden: Hier können Qualitätseinbußen, die dadurch entstehen, dass der Audioprozess zu unregelmäßig läuft, eine Verschlechterung der Tonqualität nach sich ziehen (vgl. Tanenbaum, 2009, S. 199).

Die Ziele, die eine Scheduling-Strategie optimieren soll, sind oft gegensätzlich. Beispielsweise zieht eine schnelle Durchlaufzeit eines Prozesses eine Verschlechterung der Antwortzeit nach sich. Es müssen somit Ziele, die ein Scheduler erfüllen soll, priorisiert werden, um das gewünschte Verhalten des Betriebssystems zu erreichen (vgl. Tanenbaum, 2009, S. 197).

2.5. Betriebsformen

Die Schedulingziele werden durch die (geplante) Betriebsform und die gegebenen Hardwarebedingungen (z. B. Prozessoranzahl) bestimmt. Für alle Formen gelten die drei Ziele *Fairness*, *Prioritäteneinsatz* und *Ressourcen-Balance* (vgl. Tanenbaum, 2009, S. 197).

Die Übersicht der einzelnen Betriebsformen und ihrer zugehörigen Ziele orientiert sich an Nehmer & Sturm (vgl. Nehmer & Sturm, 1998, S. 104):

- *Dialogbetrieb*: Diese Betriebsform (auch *Time-sharing* gennant) ist weit verbreitet. Primäres Scheduling-Ziel ist hier die besondere Unterstützung interaktiver Anwendung. Dieser Betriebsform werden die Ziele *Reaktionszeit* und *Proportionalität* zugeordnet.

- *Stapelverarbeitungs- oder Batchbetrieb*: Diese Betriebsform war in den 60er und 70er Jahren weit verbreitet. Die Scheduling-Ziele sind für diesen Betrieb *Durchsatz*, *CPU-Ausnutzung* und *Durchlaufzeit*.

- *Echtzeitbetrieb*: Hier ist die Einhaltung von *Deadlines* das primäre Ziel. Ein weiteres Ziel dieser Betriebsform ist *Vorhersagbarkeit*. Die Kategorie des Echtzeit-Schedulings wird in harte und weiche Echtzeitbedingungen unterteilt. *Weiche* Echtzeitbedingungen zeichnen sich dadurch aus, dass das Überschreiten einer Deadline zwar unerwünscht, aber nicht fatal ist. Es können Anwendungen wie digitale Multimedia-Anwendungen hierzu gezählt werden. Bei *harten* Echtzeitbedingungen dürfen Deadlines keinesfalls überschritten werden (vgl. Goyal et al., 1996, S. 107). Diese Betriebssystemform findet sich dort, wo zeitkritische Operationen innerhalb absoluter Fristen abgearbeitet werden müssen. Solche Anforderungen finden sich beispielsweise in Luft- und Raumfahrt, militärischen Anwendungen, im Automobilbereich oder in Telekommunikationssystemen (vgl. Davis & Burns, 2011, S. 35).

- *Hintergrundbetrieb*: Bei dieser Betriebsform sind die Fertigstellungszeitpunkte der Prozesse völlig unkritisch. Das primäre Ziel ist, dass die Ausführung solcher Prozesse den eigentlichen Rechenbetrieb minimal bis gar nicht beeinflusst. Diese Betriebsform wird zumeist in Kombination mit einem Dialog- oder Echtzeitbetriebssystem genutzt.

In den sogenannten Universalbetriebssystemen (engl. *General-purpose-systems*) wie Linux oder Windows, kommt häufig eine Mischung aus den Betriebsformen Dialogbetrieb, Stapelverarbeitungsbetrieb, Hintergrundbetrieb und weichem Echtzeit-Betrieb zum Einsatz (vgl. Suranauwarat & Taniguchi, 2001, S. 42).

2.6. Prozess-Zustände

In einem modernen Betriebssystem befinden sich in der Regel Prozesse in unterschiedlichen Zuständen. Jeder Prozess hat zu jeder Zeit immer nur einen einzigen Zustand. In der Literatur findet sich häufig das einfache Zustandsmodell, welches zwischen drei Zuständen unterscheidet (vgl. Tanenbaum, 2009, S. 132 und Nehmer & Sturm, 1998, S. 109 und Glatz, 2006, S. 138).

1. *Rechnend (running)*: Einem Prozess in diesem Zustand ist die CPU zugeteilt.
2. *Blockiert (blocked)*: Blockierte Prozesse sind nicht lauffähig und warten darauf, dass ein bestimmtes Ereignis eintritt (z. B. Beendigung einer I/O-Operation oder einer bestimmten Synchronisationsbedingung).
3. *Bereit (ready)*: Die Prozesse in diesem Zustand befinden sich in der Ready-Warteschlange (auch *Ready-Queue* genannt) und warten darauf, (wieder) der CPU zugeteilt zu werden.

Silberschatz erläutert in seiner Darstellung fünf Grund-Zustände – Stallings nennt dieses Modell *Five State Model* (vgl. Silberschatz, 2010, S. 103 und Stallings, 2001, S. 115). In diesem werden die drei eben erwähnten um die Zustände *Neu* (*new*) und *Beendet* (*terminated*) (siehe Abbildung 2.1) erweitert.

4. *Neu (new)*: Der Prozess wird gerade erstellt.
5. *Beendet (terminated)*: Der Prozess wurde beendet (weil er fertig ist, aufgrund eines Fehlers oder weil er durch einen anderen Prozess beendet wurde) (vgl. Tanenbaum, 2009, S. 129).

Ein Betriebssystem, dass die Erzeugung von Prozessen unterstützt, die aber nicht gestartet werden, besitzt zudem den Zustand *Inaktiv* (vgl. Glatz, 2006, S. 139). Ulix unterstützt dies nicht. Ein System, das Swapping unterstützt, besitzt zusätzlich den Zustand *Ausgelagert (Swap)* (vgl. Nehmer & Sturm, 1998, S. 110). Auch Swapping, was bei den meisten aktuellen Betriebssystemen durch Anwendung des Paging-Verfahrens überflüssig geworden ist (vgl. Tanenbaum, 2009, S. 235), kommt in Ulix nicht zum Einsatz.

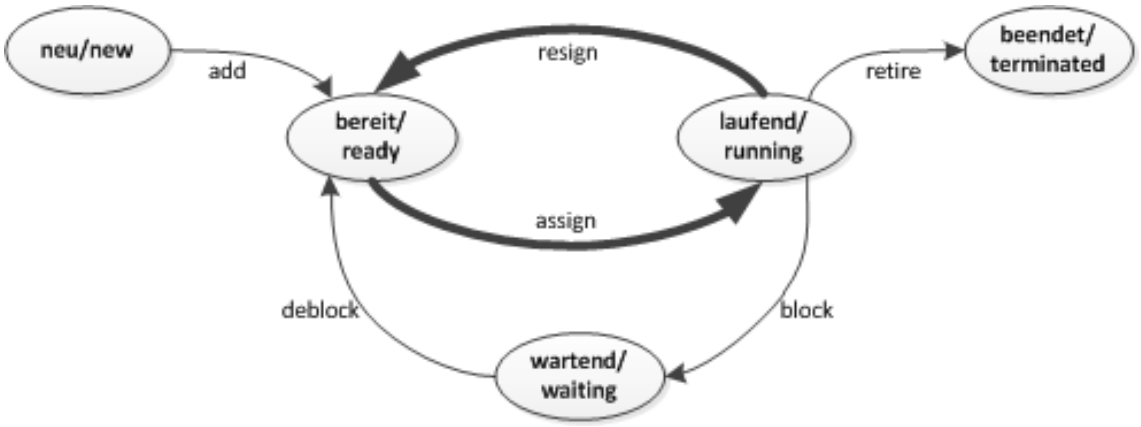

Abbildung 2.1.: Prozess-Zustandsmodell. Darstellung in Anlehnung an Glatz, 2006, S. 111

2.7. Zustandsübergänge

Im Zustandsmodell aus Abbildung 2.1 sind auch die möglichen Übergänge zwischen den Zuständen aufgeführt (vgl. Glatz, 2006, S. 111). Die Übergänge zwischen den Zuständen werden (teilweise) durch Kernel-Funktionen, die der Komponente *Dispatcher* zugeordnet werden, durchgeführt:

- *add*: Ein neu erzeugter Prozess wird in die Menge der rechenbereiten Prozesse aufgenommen. In ULIX wird ein Prozess in die Ready-Queue mittels der Funktion `add_to_ready_queue()` aufgenommen.

- *assign*: Nach einer Scheduling-Entscheidung wird die CPU dem Prozess zugeteilt. Er befindet sich dann im Zustand *Rechnend*.

- *resign*: Einem aktuell rechnenden Prozess wird die CPU entzogen, und er kommt wieder in die Liste der rechenbereiten Prozesse (Zustand *Bereit*).

- *block*: Einem Prozess wird aufgrund einer I/O-Operation oder eines Seitenfehlers der Prozessor entzogen. Er wechselt in den Zustand *Blockiert*. In ULIX wird ein Prozess hierfür in eine separate Queue mittels der Funktion `add_to_blocked_queue()` verschoben.

- *ready* oder *deblock*: Ein Prozess, der blockiert war, wechselt, nachdem das Ereignis auf das er gewartet hat, eingetreten ist, wieder in in den Zustand *Bereit*.

- *retire*: Ein der CPU zugeteilter Prozess endet und wechselt in den Zustand *Beendet*. Belegte Ressourcen werden wieder freigegeben.

Für das (Short-term) Scheduling sind nur die Zustände *Bereit* und *Rechnend* sowie die Zustandsübergänge *assign* und *resign* relevant (vgl. Glatz, 2006, S. 112).

2.8. Scheduling-Zeitpunkte

Tanenbaum nennt verschiedene Zeitpunkte, an denen ein Scheduler eine Entscheidung treffen muss, welchem Prozess als nächstes die CPU zugeteilt wird (vgl. Tanenbaum, 2009, S. 195):

1. Bei Erzeugung eines neuen Prozesses befinden sich der Elternprozess und der Kindprozess rechenbereitem Zustand. Hier muss eine Entscheidung getroffen werden, welcher der beiden rechnen darf. ULIX nutzt, wie auch UNIX, zur Erzeugung eines neuen Prozesses den System Call `fork()`.

2. Wenn ein Prozess beendet wird, muss eine Scheduling-Entscheidung getroffen werden. Da er nicht mehr vohanden ist, muss ein rechenbereiter Prozess gewählt werden. Befindet sich kein Prozess mehr in lauffähigem Zustand, läuft im Normalfall ein vom System bereitgestellter Leerlaufprozess. ULIX nutzt zur Prozessbeendigung den System Call `exit()`.

3. Eine Entscheidung, welcher Prozess als nächstes rechnen darf, muss getroffen werden, wenn der bisher rechnende Prozess blockiert wird. Dies kann z. B. aufgrund einer `wait()`- Anweisung, bei welcher auf Beendigung eines Kindprozesses gewartet wird, auftreten.

4. Bei Auftreten eines I/O-Interrupts. Wurde dieser Interrupt von einem I/O-Gerät ausgelöst, wird ein bisher blockierter Prozess, der auf dieses Signal wartet, wieder

rechenbereit. Wurde durch den I/O-Handler der Scheduler aufgerufen, muss dieser jetzt entscheiden welcher Prozess rechnen darf – der Prozess, der eben rechenbereit geworden ist, der zum Zeitpunkt des Interrupts rechnende Prozess oder irgend ein anderer. Bei Ulix wird der Scheduler nicht vom I/O Handler aufgerufen.

Ein Prozess kann in Ulix die CPU freiwillig mittels des System Calls `yield()` abgeben. Bei preemptiven Systemen kann zu jedem Timer-Interrupt-Zeitpunkt eine Entscheidung getroffen werden.

2.9. Scheduling-Algorithmen

Es gibt viele bekannte und weit verbreitete Scheduling-Algorithmen (auch *Scheduling-Verfahren* oder *Scheduling-Strategien*). Auch Ulix nutzt ab Version 0.06 mit Round-Robin eines dieser Verfahren. Die meisten, aktuell in Betriebssystemen eingesetzten Strategien (siehe Kapitel 3) beruhen auf diesen klassischen Verfahren. Zuerst werden die Verfahren, die für den Stapelverarbeitungsbetrieb optimiert sind (First-Come-First-Served, Shortest-Job-First und Shortest-Remaining-Time-Next), vorgestellt, dann folgen mit Round-Robin, Multilevel-Queue, Multilevel-Feedback-Queue, Proportional-Share und Guaranteed-Scheduling, einige Verfahren die in interaktiven Systemen eingesetzt werden. Schließlich werden noch die Echtzeitbetrieb-Verfahren Rate-Monotonic-Scheduling (RMS), Deadline-Monotonic-Scheduling (DMS), Earliest Deadline-First (EDF) und Least-Laxity-First (LLF) vorgestellt.

2.9.1. First-Come-First-Served (FCFS)

Die wohl einfachste Strategie, das FCFS-Verfahren, ist ein einfach zu verstehendes, nicht preemptives Verfahren. Es gibt hier eine Warteschlange (*Queue*), in der sich die rechenbereiten Prozesse befinden (*Ready-Queue*). Der Prozess, der als erstes rechenbereit ist, wird gestartet und darf so lange rechnen wie er will. Jeder weitere wird ans Ende der Queue angereiht. Blockiert der laufende Prozess, darf der nächste Prozess in der Queue rechnen. Ist dieser Task wieder rechenbereit, wird er wieder an das Ende der Queue gehängt (vgl. Tanenbaum, 2009, S. 200).

2.9.2. Shortest-Job-First (SJF)

Dieses nicht unterbrechende Scheduling-Verfahren wählt stets den Prozess aus, der den kleinsten zu erwartenden Rechenzeitbedarf besitzt. Die Umsetzung kann auf zwei Wegen erfolgen: Zum einen kann diese Angabe bei Stapelaufträgen mit angegeben werden. Zum anderen besteht die Möglichkeit, auf Basis von Vergangenheitswerten eine Vorhersage zu schätzen. Mandl erwähnt, dass dieses Verfahren das optimale Verfahren ist, jedoch schwer zu realisieren und in heutigen Betriebssystemen nicht vorzufinden ist (vgl. Mandl, 2013, S. 109).

2.9.3. Shortest-Remaining-Time-Next (SRTN)

Diese Strategie stellt eine Abwandlung von Shortest-Job-First dar. Es ist die unterbrechende Variante von SJF. Es wählt als nächsten Prozess immer den, der den geringsten verbleibenden Rechenzeitbedarf hat. Es gelten zudem die gleichen Einschränkungen wie bei SJF (vgl. Nehmer & Sturm, 1998, S. 147). Einem rechnenden Prozess wird die CPU entzogen, sobald ein Prozess lauffähig wird, der eine kleinere Rechenzeit benötigt (vgl. Nehmer & Sturm, 1998, S. 147). Dieses Modell ist für kurzlebige Prozesse sehr vorteilhaft (vgl. Tanenbaum, 2009, S. 202). Für langlebige Prozesse eignen sich sowohl SJF als auch SRTN nicht – hier besteht sogar die Möglichkeit, dass Prozesse verhungern (siehe hierzu *Starvation* in Kapitel 2.9.5).

2.9.4. Round-Robin-Scheduling (RR)

Round-Robin ist eine verdrängende Strategie. Es ist eines der ältesten Scheduling-Verfahren. Es ist einfach zu implementieren und sieht alle Prozesse zu jedem Zeitpunkt als gleich wichtig an (vgl. Tanenbaum, 2009, S. 202).

Jeder Prozess bekommt ein *Quantum* (auch *Zeitscheibe* oder *Timeslice*) zugewiesen. Ein Quantum ist die Zeitspanne, die ein Prozess maximal die CPU nutzen darf. Nach Ablauf wird diesem Prozess die CPU entzogen und er wird, wie beim FCFS-Verfahren, in die Ready-Queue als hinterstes Element angefügt. Die CPU wird dann, falls ein anderer Prozess laufbereit ist, einem anderen Prozess zugeteilt. Da der Wechsel von einem Prozess zum nächsten (Kontextwechsel) sehr aufwändig ist, spielt die Wahl der Größe des Quantums eine wichtige Rolle.

Wird das Quantum sehr groß angesetzt, leidet die Antwortzeit. Stallings merkt hier an, sobald das Quantum so groß wie der längste Prozess angesetzt wird, verkommt RR zu FCFS (vgl. Stallings, 2001, S. 406). Wird das Quantum zu klein angesetzt, wird zu viel Zeit durch den Kontextwechsel verschwendet. Silberschatz nennt eine Länge von 10 ms bis 100 ms für das übliche Quantum, Tanenbaum spricht von 20 ms bis 50 ms (vgl. Silberschatz, 2010, S. 194 und Tanenbaum, 2009, S. 203).

Es gibt darüber hinaus einige weitere RR-Varianten. Eines davon beschreibt Kleinrock: Beim *selfish-Round-Robin-Verfahren* (sRR) existieren zwei Queues, eine *active*-Queue und eine *holding*-Queue. In der active-Warteschlange dürfen sich nur eine bestimmte Anzahl, inklusive dem laufenden Prozess, befinden. Neue Prozesse werden in die holding-Queue eingefügt und verbleiben dort, bis ihre Priorität größer ist als die der Prozesse in der active-Queue. Im Zeitverlauf werden die Prioritäten der Prozesse in den Queues dekrementiert. Diese Variante bevorzugt ältere Prozesse (vgl. Kleinrock, 1970, S. 455).

2.9.5. Prioritäten-Scheduling

Das am häufigsten eingesetzte Verfahren in heutigen Betriebssystemen ist das gerade angesprochene RR-Verfahren ergänzt um eine Prioritätssteuerung (vgl. Mandl, 2013, S. 109). Im Gegensatz zum RR-Verfahren, bei dem jeder Prozess die gleiche Wichtigkeit besitzt, werden beim Prioritätsscheduling allen Prozessen Prioritäten zugeteilt. Durch diese Priorisierung wird entweder die Scheduling-Entscheidung beeinflusst oder/und die Größe der

Quanten (vgl. Nehmer & Sturm, 1998, S. 117 und Nieh et al., 2001, S. 4). In preemptiven Systemen wird beim klassischen Prioritäten-Verfahren, sobald ein Prozess rechenbereit geworden ist, geprüft, ob er eine höhere Priorität besitzt als der, der gerade die CPU belegt. Im Fall, dass der neue bzw. der jetzt nicht mehr blockierte Prozess eine höhere Priorität besitzt, wird dem laufenden Prozess die CPU entzogen und der höher priorisierte Prozess darf rechnen. Es werden hier *statische* und *dynamische* Verfahren unterschieden (vgl. Nehmer & Sturm, 1998, S. 117):

- *Statische Verfahren*: Diese Verfahren kommen, unter anderen, in Echtzeitsystemen zur Anwendung. Die Priorität jedes Prozesses wird hier zum Erzeugungszeitpunkt festgelegt und kann danach nicht mehr verändert werden.
- *Dynamische Verfahren*: Bei diesen Verfahren kann die Priorität auch nachträglich vom System oder vom Benutzer verändert werden.

Silberschatz unterscheidet des Weiteren zwischen *intern* und *extern* definierten Prioritäten (vgl. Silberschatz, 2010, S. 192):

- *Intern* definierte Prioritäten werden bestimmt durch im System messbare Größen, wie Anzahl der geöffneten Dateien oder Speicheranforderungen.
- *Extern* definierte Prioritäten werden durch Kriterien bestimmt die sich außerhalb des Betriebssystems befinden, wie z. B. die Wichtigkeit eines Prozesses oder die Höhe der für die Rechenzeit bezahlten Geldsumme.

Beim Prioritätsscheduling existieren die zwei Problematiken *Starvation* und *Priority inversion* (vgl. Silberschatz, 2010, S. 193):

- *Starvation* (oder *Indefinite blocking*): Im Fall, dass ständig Prozesse lauffähig werden oder ablaufen, die eine hohe Priorität besitzen, kann es dazu kommen, dass niedrig priorisierte Prozesse *verhungern*. D. h., sie kommen nie in den Besitz der CPU, da ständig Prozesse laufen, die eine höhere Priorität besitzen als sie.
- *Priority inversion* (Prioritätsumkehr): Bei dieser Problematik sind mehrere Prozesse mit unterschiedlicher Priorität und eine Ressource mit wechselseitigem Ausschluss beteiligt. Im Fall, dass ein hoch priorisierter Prozess auf eine Ressource zugreifen will, die von einem niedrig priorisiertem Prozess belegt ist, muss dieser warten, bis die Ressource wieder freigegeben wird. Existiert nun aber ein dritter Prozess (der die Ressource nicht verwendet) mit mittlerer Priorität, der den niedrigeren Prozess verdrängt, wird diese Ressource nie freigegeben, und der hoch priorisierte Prozess kann nie laufen. Somit verdrängt der mittel priorisierte Prozess den hoch priorisierten.

2.9.6. Multilevel-Scheduling (ML)

In der Praxis finden sich überwiegend Scheduler die verschiedene Verfahren wie RR oder FCFS kombinieren. Gängige Kombinationen sind der Betrieb von Dialog- und Hintergrundbetrieb oder die gleichzeitige Unterstützung von Echtzeitaufträgen und zeitunkritischen Aufträgen (vgl. Nehmer & Sturm, 1998, S. 118). Auch beim prioritätsbasierten

Scheduling kommen in der Praxis Multilevel-Techniken (mehrere Queues für rechenbereite Prozesse) zum Einsatz, da beispielsweise die Reihenfolge von Prozessen mit gleicher Priorität festgelegt werden muss. Im Gegensatz zu Nehmer & Sturm rechnet Tanenbaum diese Variante dem Prioritätsscheduling zu (vgl. Tanenbaum, 2009, S. 203).

Eine Form der Realisierung von Multilevel-Scheduling kann durch die Unterteilung der Ready-Queue in mehrere Teil-Warteschlangen erfolgen. Diese werden dann mit einem oder unterschiedlichen Verfahren pro Teil-Warteschlange abgearbeitet (z. B. RR, FCFS). Silberschatz erwähnt eine Ausprägung von Multilevel-Scheduling die den Prozessen jeder Teilliste unterschiedlich lange Zeitscheiben einräumt (vgl. Silberschatz, 2010, S. 196).

2.9.7. Feedback-Scheduling

Dieses Scheduling-Verfahren berücksichtigt das bisherige Verhalten von Prozessen. So werden Parameter, wie z. B. die Priorität eines Prozesses beim Auftreten bestimmter Ereignisse, angepasst. Ein solches Ereignis tritt auf wenn beispielsweise ein Prozess erneut rechenbereit wird oder vom User-Mode in den Kernel-Mode wechselt (siehe Kapitel 3.2.1).

Wie auch in Kapitel 4.4 beschrieben, kann auf die Länge von Prozessen reagiert werden. So können z. B. CPU-lastige Prozesse bestraft werden, sobald sie ihr Quantum verbrauchen.

Ein Feedback-Mechanismus ist beispielsweise *Aging*, welches zur Vermeidung von Prozessinversion oder Starvation eingesetzt wird. Hier wird die Priorität eines Prozesses, abhängig von der Wartezeit auf die CPU, erhöht (vgl. Nehmer & Sturm, 1998, S. 119).

2.9.8. Multilevel-Feedback-Queue-Scheduling (MLFQ)

Dieses Verfahren kombiniert die Eigenschaften von Multilevel- und Feedback-Scheduling. Die Ursprünge dieses Verfahrens gehen zurück auf Corbato et al., 1962. Während beim Multilevel-Scheduling Prozesse bei Erstellung den Klassen fix zugeordnet werden, können die Prozesse beim MLFQ-Scheduling die Prozessklassen wechseln. Epema beschreibt zwei MLFQ-Basisvarianten (vgl. Epema, 1998, S. 368):

- *Head-of-line-policy*: Jeder neue Prozess kommt in die Queue mit der höchsten Priorität. Jedes Mal, wenn dieser Prozess sein Quantum verbraucht, wird er in die nächstniedriger priorisierte Queue verschoben. Dies wiederholt sich, bis er die niedrigste Prioritätsstufe erreicht hat. In dieser verbleibt er.

- *Decay-usage* (auch *Processor-usage-aging* oder *Priority-aging*): Diese Scheduler passen die Priorität ständig dynamisch, auf Basis der vom Prozess zuletzt verbrauchten CPU-Zeit, an. Hierzu werden diese dynamischen Prioritäten alle x Timer-Interrupts neu berechnet. Prozesse, die viel CPU-Zeit verbrauchen, bekommen so eine niedrigere Priorität; Prozesse, die wenig CPU-Zeit verbrauchen, eine höhere Priorität. Es gibt verschiedene Ausprägungen dieser Variante. Diese werden in den Kapiteln 3.1.6, 3.2.1, 3.2.2 und 3.2.4 ausführlich betrachtet.

2.9.9. Proportional-Share-Scheduling

Proportional-Share steht für eine Klasse von Verfahren, die jedem Prozess bzw. User einen Anteil an der verfügbaren Rechenzeit zuweisen, der proportional zu seiner Wichtigkeit ist. Nieh et al. zählen neben dem Round-Robin-Verfahren auch die folgenden drei Verfahren zu dieser Klasse (vgl. Nieh et al., 2001, S. 4):

- *Lottery-Scheduling*: Hier erfolgt die Scheduling-Entscheidung zufällig, wie bei Lotterielosen. Jedem Prozess werden ein oder mehrere (je nach Wichtigkeit des Prozesses) Lotterielose zugeteilt. Muss eine Scheduling-Entscheidung getroffen werden, wird ein Lotterielos zufällig gewählt und dem Prozess der das Los besitzt, wird als nächstes die CPU zugeteilt (vgl. Waldspurger & Weihl, 1994, S. 2 und Waldspurger & Weihl, 1995, S. 18).

- *Fair-Queuing-Scheduling*: Ein Vertreter dieser Klasse ist das sogenannte *Stride-Scheduling*. Hier wird zu jedem Prozess ein *stride* bestimmt. Ein *stride* ist die Zeit die ein Prozess warten muss bevor er wieder der CPU zugeteilt wird. Der Prozess mit dem kleinsten *stride* wird der CPU am häufigsten zugeteilt. *Strides* werden durch virtuelle Zeiteinheiten, den sogenannten *passes*, repräsentiert. Es gibt die drei Stati *pass* und *stride* und *ticket* die jedem Prozess zugeordnet sind. Die Variable *ticket* stellt die CPU-Zuteilung relativ zu anderen Prozessen dar. *Stride* ist umgekehrt proportional zu *tickets* und repräsentiert die Zeit zwischen der Ausführung eines Prozesses gemessen in *passes*. Der Prozess mit dem niedrigsten *pass*-Wert wird ausgeführt. *Pass* ist hierbei der virtuelle Zeitindex für die Wahl des nächsten Prozesses. *Pass* wird jedesmal, wenn ein Prozess gerechnet hat, neu berechnet (vgl. Waldspurger & Weihl, 1995, S. 49).

- *Fair-Share-Scheduling*: Bei diesem prioritätsbasierten Verfahren werden die Prozessbesitzer berücksichtigt (vgl. Kay & Lauder, 1988, S. 49). Das Modell sieht vor, dass jeder Nutzer bzw. jede Gruppe einen Teil der CPU belegt. Der Scheduler wählt die Prozesse so aus, dass genau dieser Teil ausgenutzt wird. Dieses Verfahren hat das Ziel, Fairness nicht zwischen Prozessen sondern zwischen den Nutzern bzw. Gruppen herzustellen (vgl. Kay & Lauder, 1988, S. 44).

2.9.10. Guaranteed-Scheduling

Beim *Garantierten Scheduling* wird jedem Prozess der gleiche Anteil der CPU-Zeit zugeteilt. Bei n Prozessen im System wird jedem Prozess $1/n$ der CPU-Leistung zugeteilt. Bei diesem Verfahren muss das System festhalten wie viel CPU-Zeit jeder Prozess seit seiner Erzeugung konsumiert hat. So kann der Anteil der CPU-Zeit berechnet werden zu dem ein Prozess berechtigt ist. Dann wird das Verhältnis der tatsächlich verbrauchten CPU-Zeit zur berechtigten CPU-Zeit berechnet. Die Prioritätsberechnung richtet sich dann danach, ob der Prozess so viel CPU-Zeit bekommen hat, wie ihm zustand. Die Strategie teilt dem Prozess, der die *niedrigste* Priorität besitzt, die CPU so lange zu, bis sein Verhältnis das seines nächsten Konkurrenten überstiegen hat (vgl. Tanenbaum, 2009, S. 206 - S. 207).

2.9.11. Rate-Monotonic-Scheduling, Deadline-Monotonic-Scheduling, Earliest-Deadline-First und Least-Laxity-First

Dieser Abschnitt beschreibt bekannte Verfahren, die in Echtzeitbetriebssystemen zum Einsatz kommen. Es gibt hier *statische* und *dynamische* Verfahren. Zuerst werden statische Verfahren beleuchtet:

- *Rate-Monotonic-Scheduling (RMS)* wird eingesetzt, wenn Prozesse periodisch auftreten, d. h. ein Prozess wird alle x Zeiteinheiten wiederholt ausgeführt. Je kürzer die Perioden sind desto höhere Prioritäten werden zugewiesen. Es werden auch aperiodische Prozesse (unregelmäßige und unvorhersagbare Prozesse) unterstützt (vgl. Liu & Layland, 1973, S. 50).

- *Deadline-Monotonic-Scheduling (DMS)* funktioniert ähnlich wie RMS. Es weist dem Prozess mit der kürzesten relativen Deadline die höchste Priorität zu (vgl. Liu & Layland, 1973, S. 51).

Zu den dynamischen Verfahren zählen die beiden folgenden:

- *Earliest-Deadline-First (EDF)*: Hier werden Prioritäten nach der Länge der Deadlines vergeben. Es hat immer derjenige Prozess die höchste Priorität, dessen Deadline als nächstes abläuft (vgl. Liu & Layland, 1973, S. 55).

- *Least-Laxity-First (LLF)* ist EDF sehr ähnlich. Hier spielt die Bedienzeit ein wichtige Rolle, welche die Zeit repäsentiert, die ein Prozess Zugriff auf die CPU hat. Während bei EDF die Deadline betrachtet wird, wird bei LLF die restliche Bedienzeit betrachtet. Somit erhalten die Prozesse eine Priorität die umgekehrt proportional zu ihrer verbleibenden Zeit bis zum Eintreffen der Deadline ist. Dieser Zeitraum wird *slacktime* oder auch Spielraum genannt. Somit werden Prozessen mit kurzen Slacktimes hohe Prioritäten zugewiesen (vgl. Lee et al., 2012, S. 71).

3. Scheduling-Verfahren in Betriebssystemen

Dieses Kapitel beleuchtet die Umsetzung der Scheduling-Verfahren in verschiedenen Unix-ähnlichen Lehrbetriebssystemen und produktiv eingesetzten Betriebssystemen. Da die Umsetzung der Verfahren in den meisten Betriebssystemen zu komplex ist, um sie in dieser Arbeit vollständig zu beleuchten, werden nur die Besonderheiten der jeweiligen Verfahren dargestellt. Der Fokus der Untersuchung liegt hier auf den Kriterien *Prinzip der Scheduling-Verfahren* und auf der *Prozessverwaltung* in den Systemen.

3.1. Lehrbetriebssysteme

Es gibt eine Vielzahl an Lehrbetriebssystemen. Für diese Arbeit wurden einige der bekanntesten Systeme ausgewählt. Zu diesen zählen Minix 2, Xinu, xv6 und NachOS (vgl. Anderson & Nguyen, 2005, S. 187 und Aviv et al., 2012, S. 80). Da PintOS als Nachfolger von NachOS zählt wird anstelle NachOS das Betriebssystem PintOS untersucht. Minix 3 wird auch produktiv eingesetzt, eignet sich aber auch noch zu Lehrzwecken und wird der Gruppe der Lehrbetriebssysteme zugeordnet (vgl. Tanenbaum, 2009, S. 839).

3.1.1. xv6

xv6 ist ein Unix-ähnliches Lehrbetriebssystem. Es ist in ANSI C implementiert und auf der Prozessorarchitektur Intel x86 lauffähig. Die Prozesstabelle besteht aus einer Liste die maximal 64 Prozesse enthalten kann. Das Scheduling-Verfahren ist Round-Robin. Das Verfahren wird mittels den beiden Funktionen `sched()` und `scheduler()` realisiert.

Wird ein Prozesswechsel erforderlich, was beispielsweise bei jedem Timer-Interrupt der Fall ist, wird die Funktion `sched()` aufgerufen. Deren Hauptfunktion besteht darin, einen Kontextwechsel vom Prozess-Kontext des gerade laufenden Prozesses zum Scheduler-Kontext durchzuführen.

Die Aufgabe der Funktion `scheduler()`, die bereits bei Betriebssystemstart aufgerufen wird, ist die Suche nach einem lauffähigen Prozess und dem anschließenden Kontextwechsel vom Scheduler-Kontext zum Prozess-Kontext des gerade ausgewählten Prozesses. Die Funktion `scheduler()` enthält, vereinfacht dargestellt, zwei Schleifen. Aus der ersten (der äußeren) Schleife wird niemals hinausgesprungen. Diese umschließt eine (die innere) Schleife, die die Prozessliste, angefangen bei Prozess 0, durchläuft. Der erste Prozess der den Status `RUNNABLE` hat, wird ausgewählt, in den Zustand `RUNNING` versetzt und ein Kontextwechsel zu diesem Prozess vorgenommen. Gibt dieser Prozess z. B. durch `yield()` die CPU freiwillig ab, wird durch die von `yield()` aufgerufene Funktion `sched()` ein Kontextwechsel zum Scheduler-Kontext vollzogen und die (innere) Schleife wieder nach der Stelle, an dem der vorherige Kontext-Wechsel erfolgte, fortgesetzt. Der Ausgangspunkt der Suche

wird auf Prozess 0 zurückgesetzt. Dann beginnt alles wieder von vorne. Die innere Schleife wird ein weiteres Mal solange durchlaufen bis ein Prozess den Status `RUNNABLE` hat (vgl. Cox et al., 2012, S. 55 - S. 56).

3.1.2. PintOS

Auch PintOS, ein von NachOS inspiriertes Lehrbetriebssystem, ist wie xv6 in ANSI C geschrieben und auf x86-Prozessorarchitekturen lauffähig (vgl. Pfaff et al., 2009, S. 453). Das System ist grundlegend funktional und ist auf die Erweiterung um zusätzliche Komponenten durch Studenten ausgelegt. Es ist standardmäßig ein RR-Verfahren implementiert. Es sind Funktionen vorbereitet, um innerhalb eines Studentenprojektes einen statischen Prioritäten-Scheduler implementieren zu können. Auf dieser Basis kann in einem weiteren Projekt ein MLFQ-Verfahren vom Typ Decay-usage-Verfahren erarbeitet (vgl. Pfaff et al., 2009, S. 455).

PintOS verwaltet seine Prozesse in doppelt verketteten Listen. Diese Listen werden von einem Head- und einem Tail-Element begrenzt. Sobald ein Prozess in PintOS lauffähig wird, wird er am Ende der Ready-Queue, vor dem Tail-Element, eingefügt. Muss der nächste Prozess gewählt werden, wird der Prozess nach dem Head-Element der Ready-Queue gewählt und entfernt. Die Quanten können dynamisch verwaltet werden. Bei jedem Timer-Interrupt wird die Variable `thread_ticks` inkrementiert. Ist diese Variable gleich oder größer der Variable `TIME_SLICE` muss der Prozess die CPU abgeben. Nachdem der Kontextwechsel zum nächsten Prozess stattgefunden hat, wird `TIME_SLICE` wieder auf 0 gesetzt.

3.1.3. Xinu

Das Lehrbetriebssystem Xinu (Comer, 1984) wurde ursprünglich für MIPS-Prozessoren entwickelt und wurde im Laufe der Zeit auf viele andere Plattformen portiert. Die Scheduling-Einheit in Xinu sind Prozesse. Xinu verwendet einen statischen Prioritäten-Scheduler. Jedem Prozess wird eine Priorität zugewiesen. Es wird immer der Prozess mit der höchsten Priorität gewählt. Verbraucht ein Prozess sein Quantum und befindet sich kein Prozess mit höherer Priorität in der Ready-Queue, so darf der aktuelle Prozess ein weiteres Quantum laufen. Innerhalb jeder Prioritätsstufe gilt das RR-Prinzip.

Die Prozessverwaltung ist der von PintOS sehr ähnlich. Auch hier werden die Listen von einem Head- und einem Tail-Element begrenzt (vgl. Comer, 2011, S. 50). Allerdings wird ein Prozess der laufbereit geworden ist, so in die Ready-Queue eingefügt das eine, nach Priorität sortierte Liste entsteht. Prozesse mit gleicher Priorität werden so eingefügt das sie im RR-Verfahrens abgearbeitet werden. Hierzu ein Beispiel: Befindet sich ein Prozess mit Priorität 50 in der Ready-Liste, wird ein Prozess der der Liste neu hinzugefügt wird und ebenfalls die Priorität 50 besitzt, hinter dem bestehenden Prio 50 Prozess einsortiert.

Wenn ein neuer Prozess gewählt werden muss, gilt das gleiche wie bei PintOS. Der laufende Prozess befindet sich nicht in der Ready-Liste (vgl. Comer, 2011, S. 74).

3.1.4. GeekOS

GeekOS ist ein weiteres Lehrbetriebssystem welches auch in ANSI C implementiert und auf der Prozessorarchitektur Intel x86 lauffähig ist (vgl. Hovemeyer et al., 2004, S. 315). Wie auch PintOS ist es auf die Erweiterung in Studenten-Projekten ausgelegt. GeekOS arbeitet standardmäßig mit einem statischen Prioritätsscheduler, der dem von Xinu stark ähnelt. Die Funktion, um den nächsten lauffähigen Prozess bei einer Scheduling-Entscheidung zu finden ist, anders als bei den restlichen Lehrbetriebssystemen, in Assembler implementiert. In einem Projekt wird ein MLFQ-Verfahren implementiert (vgl. Hovemeyer et al., 2004, S. 317).

Ein Kontextwechsel wird in GeekOS erforderlich wenn ein Prozess freiwillig die CPU freigibt oder durch einen Timer-Interrupt unterbrochen wird. Ersteres geschieht entweder mittels `yield()` oder `wait()`. Bei `yield()` wird der Prozess wieder ans Ende der Ready-Queue gestellt. Bei `wait()` wird der Prozess ans Ende der Wait-Queue gestellt. Die Umsetzung der (Vorbereitung auf) dynamischen Quanten ist ähnlich umgesetzt wie in PintOS (vgl. Hovemeyer, 2001, S. 7).

3.1.5. Minix 2

Minix ist wohl das bekannteste Lehrbetriebsystem. Es ist Unix-ähnlich, basiert auf einem Mikrokernel, und der Quellcode ist frei verfügbar. Die Scheduling-Einheit in Minix 2.0 sind Prozesse.

Minix 2 nutzt einen statischen Prioritätsscheduler (vgl. Tanenbaum, 1990, S. 140). Hier verwaltet der Scheduler drei Queues, die lauffähige Prozesse enthalten – eine für I/O-Aufgaben (Task-Prozesse), eine für Server-Prozesse und eine für Anwender-Prozesse. Server-Prozesse unterscheiden sich von Task-Prozessen dadurch, dass sie in einem niedriger privilegierten Level laufen als Task-Prozesse und nicht direkt auf I/O-Ports zugreifen können.

Die Wahl des Prozesses, der als nächstes gewählt wird, fällt immer auf den ersten Prozess der Queue mit der höchsten Priorität. Ist diese leer, wird die Queue mit nächsthöchster Priorität auf lauffähige Prozesse getestet. Sind alle Queues leer, läuft der Leerlaufprozess (Idle routine). Server-Prozesse werden nie unterbrochen (vgl. Tanenbaum, 1990, S. 141). Sie dürfen solange laufen bis sie selbst blockieren. Die Anwender-Prozesse werden im RR-Verfahren behandelt. Sobald ihr Quantum abgelaufen ist, werden sie ans Ende ihrer jeweiligen Queue (`User_Q`) gesetzt. Auch Prozesse, die blockiert waren und wieder frei sind, werden ans Ende ihrer Queue gesetzt. Es befinden sich immer nur lauffähige Prozesse in den Queues. Prozesse, die blockieren oder durch ein Signal terminieren, werden aus den Queues entfernt. Ein Prozess der aktuell läuft, wird nicht von der Queue entfernt. In Minix 2 werden Scheduling-Entscheidungen getroffen sobald ein Prozess blockiert oder wieder lauffähig (unblock) wird und sobald das Quantum eines Benutzer-Prozesses abgelaufen ist.

Die Implementierung des Verfahrens wird mittels zweier Arrays, `rdy_head` und `rdy_tail` realisiert. `rdy_head` enthält für jede der Queues einen Zeiger, welcher auf den Kopf der jeweiligen Queue zeigt. Das Array `rdy_tail` arbeitet identisch, bis auf den Unterschied, dass die Einträge auf das Ende der jeweiligen Queue zeigen. Die Funktion kann so effizient

Prozesse wieder ans Ende ihrer Queue einreihen.

Die Auswahl der Queue mit der höchsten Priorität wird durch die Funktion `pick_proc` erledigt, welche hierfür den Zeiger `proc_ptr` verwendet. Hier wird, angefangen bei der Queue, welche die höchst priorisierten Prozesse enthält, jede Queue untersucht, ob ein Kopf-Element vorhanden ist. Sind alle Queues leer wird der Zeiger auf den Leerlauf-Prozess gesetzt. Die Funktion `pic_proc()` wird zum einen immer dann ausgeführt wenn sich an der Queue etwas ändert, was Auswirkungen auf die Wahl des nächsten Prozesses haben kann. Zum anderen kommt sie zum Einsatz sobald der Prozess, dem aktuell die CPU zugewiesen ist, blockiert wird.

Um einen lauffähigen Prozess am Ende seiner Warteschlange einzusortieren wird die Funktion `ready()` aufgerufen. Mittels der Funktion `unready()` wird ein nicht mehr lauffähiger Prozess vom Anfang der Queue gelöscht. Im Normalfall findet sich der Prozess der entfernt werden soll am Anfang der Queue, da ein Prozess laufen muss um blockiert werden zu können. Im Falle das z.B. ein User-Prozess nicht läuft, kann es sein das die `User_Q`-Queue nach ihm durchsucht wird und er gelöscht wird.

3.1.6. Minix 3

In Minix 3 kommt ein Head-of-line-MLFQ-Scheduler mit 16 Queues zum Einsatz. Die Systemprozesse werden in den Queues 0 bis 6, Userprozesse in 7 bis 14 und der Idle-Prozess in Queue 15 verwaltet. Jede Queue wird in einem modifizierten RR-Verfahren behandelt. Die Quanten unterscheiden sich für jede Queue – niedrigpriorisierte Prozesse, wie User-Prozesse, erhalten ein kleines Quantum, hochpriorisierte Prozesse erhalten ein großes Quantum. Verbraucht ein Prozess sein Quantum, wird er in der nächstniedrigeren Queue am Ende einsortiert. Verbraucht er sein Quantum nicht, weil er unterbrochen wurde, wird er, wenn er wieder lauffähig ist, an den *Kopf* der Ready-Queue gesetzt. Er bekommt dann aber nur das Quantum, das er beim letzten Lauf nicht genutzt hat. Es wird bei jedem Timer-Interrupt geprüft, ob der aktuelle Prozess noch berechtigt ist zu laufen (vgl. Minix3, 2013). Die Auswahl des nächsten Prozesses bei einer Scheduling-Entscheidung ähnelt stark der bei Minix 2.

3.2. Für den produktiven Einsatz geeignete Betriebssysteme

Es werden in diesem Kapitel die Scheduling-Verfahren von Betriebssystemen, die für den produktiven Einsatz geeignet sind, untersucht. Die Auswahl der Systeme umfasst aktuelle Betriebssysteme wie Linux, FreeBSD *ab* 5.1, als auch klassische Betriebssysteme wie Unix V Rel. 3 und FreeBSD *bis* 5.1. Darüber hinaus wird ein Einblick in das Scheduling-Verfahren von Windows ab XP gegeben.

3.2.1. Unix

Der klassische Unix-Scheduler, der in System V Rel. 3 und BSD 4.3 arbeitet, wurde dafür ausgelegt, dass sowohl I/O-lastige als auch CPU-lastige Prozesse möglichst optimal bedient werden (vgl. Nehmer & Sturm, 1998, S. 124). Diese Version unterstützt Echtzeitaspekte

noch nicht. Das angewandte Verfahren ist ein MLFQ-Verfahren der Klasse Decay-usage-Scheduler (vgl. Epema, 1998, S. 367). Wie bei Minix 3 existieren mehrere Queues, die lauffähige Prozesse aufnehmen können. Es wird immer der Prozess am Kopf der höchst priorisierten Queue gewählt. Die Quanten haben eine Länge zwischen 100 ms und 1 s. Diese Version unterstützt auch Swapping.

Das Unix-Scheduling unterscheidet Kernmodusprioritäten und Benutzermodusprioritäten und passt, vereinfacht dargestellt, die Priorität von Anwenderprozessen bei Wechseln zwischen den Modis, an. Prozesse die im Kernmodus blockieren, bekommen eine höhere Priorität zugewiesen. Somit können diese Prozesse den Kernmodus nach Aufheben der Blockade schnell wieder verlassen. Des weiteren wird die Priorität aller Prozesse im Benutzermodus einmal pro Sekunde angepasst. Diese Neuberechnung geschieht nach der Formel:

$$Priority\_Value = Threshold\_priority + Nice\_value + Recent\_CPU\_Usage/2$$

Hierbei ist $Priority\_Value$ die neuberechnete Priorität und $Threshold\_priority$ ein gleichbleibender Wert, der systemabhängig Werte zwischen 40 und 60 annehmen kann. $Nice\_value$ besitzt standardmäßig den Wert 20 und kann mit dem Befehl `nice` bzw. `renice` angepasst werden (Wertebereich von -19 bis 20). $Recent\_CPU\_Usage$ ist die bereits konsumierte Rechenzeit des Prozesses. Diese wird nach Ablauf des Quantums um 1 erhöht und zum Zweck von Aging jede Sekunde halbiert.

3.2.2. FreeBSD

FreeBSD als weiteres Unix-ähnliches Betriebssystem besitzt seit Version 5.0 zwei verschiedene Scheduler (vgl. McKusick & Neville-Neil, 2004, S. 98). Zwischen diesen beiden Scheduling-Varianten wird aufgrund von Performancesteigerung nicht dynamisch gewechselt, sondern eine Variante muss bereits beim Kompilieren des Kernels gewählt werden. Bis Version 5.1 war der *4.4BSD Scheduler* Standard. Seit Version 5.2 ist der *ULE-Scheduler* der Standard-Scheduler. Die Scheduling Einheit sind bei beiden Varianten Threads.

4.4BSD-Scheduler

Der 4.4BSD-Scheduler von FreeBSD ist vom Prinzip dem Unix-4.3-Scheduler sehr ähnlich. Es ist ebenfalls ein MLFQ-Scheduler vom Typ Decay-usage, welcher prioritätenbasiert arbeitet und I/O-lastige Threads bevorzugt. Die Größe des Quantums beträgt in FreeBSD 100 ms (vgl. McKusick & Neville-Neil, 2004, S. 99). Auch hier werden die dynamischen Prioritäten angepasst. Die Neuberechnung für Usermode-Threads findet alle 40 ms statt (vgl. McKusick & Neville-Neil, 2004, S. 100):

$$kg\_user\_pri = PRI\_MIN\_TIMESHARE + [kg\_est\_cpu/4] + 2 \times kg\_nice$$

Werte niedriger als `PRI_MIN_TIMESHARE` (160) werden auf 160 gesetzt, Werte größer als 223 werden auf `PRI_MAX_TIMESHARE` (223) gesetzt. *kg_user_pri* entspricht der Bezeichnung von `Priority_Value` aus BSD 4.3, *kg_nice* entspricht `Nice_value` (hier von -20 bis 20). *kg_est_cpu* liefert eine Schätzung der bisherigen CPU Nutzung des Threads. Durch diese Form der ständigen Neuberechnung vermindert sich die Priorität durch vergangene CPU-Nutzung linear.

Auch bei FreeBSD werden die Prioritäten der Usermode-Threads angepasst. Hierzu wird die Ready-Queue jede Sekunde durchlaufen. Dies ist die Aufgabe der Funktion `schedcpu()`. Die Funktion `RoundRobin()` ist für die Prüfung auf Neuzuteilung der CPU zuständig. Diese wird 10 mal in der Sekunde aufgerufen (=Quantum 100ms). Eine weitere mit dem Scheduling in Verbindung stehende Funktionen ist `hardclock()`. Sie wird alle 10 ms ausgeführt und aktualisiert bei jedem vierten Aufruf `kg_est_cpu`. Für die Neuberechnung wird dort die Funktion `resetpriority()` aufgerufen. Neue Threads werden durch die Funktion `setrunnable()` in die Liste der lauffähigen Threads platziert. Sie ruft des Weiteren `updatepri()` auf, welche `kg_est_cpu` neu berechnet; anschließend wird auch hier `reset-priority()` aufgerufen.

Es existieren 256 Prioritätsstufen aufgeteilt auf fünf Thread Klassen (0-63 Bottom-half kernel (interrupt), 64-127 Top-half kernel, 128-159 Real-time user, 160-223 Time-sharing user, 224-255 Idle user). Der FreeBSD-Kernel kann logisch in Bottom-half und Top-half geteilt werden. Der Top-half Teil stellt Dienste zur Verfügung die in Verbindung mit System Calls und Traps stehen. Der Bottom-half-Teil enthält Routinen die aufgerufen werden wenn Hardware-Interrupts behandelt werden müssen (vgl. McKusick & Neville-Neil, 2004, S. 103). Es existiert eine globale Ready-Queue. Diese ist aufgeteilt in 64 Teillisten. Die Auswahl des nächsten Threads wird durch ein Array realisiert, welches alle Kopfelemente der 64 Warteschlangen enthält. Ein mit dem Array in Verbindung stehender Bit-Vektor `rq_status` dient dem finden der höchst priorisierten nicht-leeren Teilliste. Die Funktion `runq_add()` fügt Threads am Ende der Teillisten ein, die Funktion `runq_remove()` entfernt den Eintrag am Kopf der Teilliste. Die Funktion `runq_choose()` ist dafür verantwortlich, den nächsten lauffähigen Prozess zu finden: Vereinfacht wird hierzu eine nicht-leere Liste gesucht, indem der Eintrag des ersten Bits in `rq_status` gefunden wird, der ungleich 0 ist – ist `rq_status` =0, wird der `idle loop`-thread gewählt. Wird ein Thread gefunden, wird er vom Kopf der Teilliste entfernt und aktiviert.

ULE Scheduler

Der ULE-Scheduler wurde in erster Linie zur Unterstützung von Mehrprozessor- bzw. Mehrkern-Rechnerarchitekturen entworfen. Hier werden SMP-Systeme und Symmetric Multithreading-(SMT) Prozessoren unterstützt (vgl. McKusick & Neville-Neil, 2004, S. 101). Es existieren drei Arrays für jede CPU; die *idle queue*, in der sich die idle-Threads befinden und die beiden Warteschlangen *current* und *next*. Die Suche des nächsten Threads läuft wie folgt: Threads der *current queue* werden nach ihrer Priorität geordnet abgearbeitet. Ist die Liste leer, werden die *current*- und *next*-Arrays getauscht und das Scheduling wird erneut gestartet. Erst wenn sich keine Threads in den beiden Warteschlangen mehr befinden, wird die *idle queue* abgearbeitet. *Real-time*, *interrupt* und *interactive* Threads

werden stets in die *current queue* eingefügt – *noninteractive* Threads in die *next queue*. Interaktive Threads werden anhand von Heuristiken identifiziert, welche auf den Kriterien *sleep* (Wartezeit auf I/O) und *run* (Nutzungszeit der CPU) basieren.

Es existieren zwei Load-Balancing-Mechanismen zur Unterstützung der Migration von Threads zwischen mehreren Prozessoren: Jedes Mal, wenn ein Prozessor einen Thread auswählt, prüft er vorher ob sich ein anderer Prozessor im Idle-Status befindet. Dann wird der Thread diesem Prozessor zugeordnet. Der zweite Mechanismus, genannt *push migration*, wählt zwei Prozessoren aus – den mit der stärksten Auslastung und den mit der geringsten. Dann werden die Warteschlangen der beiden Prozessoren angeglichen.

3.2.3. Windows

Der Scheduler von Windows XP und höher ist ebenfalls ein preemptiver, prioritätsbasierter MLFQ-Scheduler (Microsoft, 2012). Die Scheduling-Einheit sind unter Windows Threads. Alle Threads mit dem selben Prioritätslevel werden gleich behandelt. Zuerst werden den Threads der höchsten Priorität Zeitscheiben zugewiesen. Hier wird nach dem RR-Prinzip vorgegangen. Gibt es keine lauffähigen Threads dieses Prioritätslevels mehr, werden die Threads der nächst niedrigeren Prioritätslevel mittels RR-Verfahren abgearbeitet. Wird ein Prozess lauffähig, der eine höhere Priorität besitzt als der Thread, der aktuell läuft, wird der aktuell rechnende Thread sofort unterbrochen. Dem höher priorisierten Thread wird dann ein komplettes Quantum zugewiesen.

Windows unterscheidet *interne Prioritäten auf Kernelebene* und *Prioritäten auf Laufzeitsystemebene*. Es gibt 32 Prioritätsstufen. Die niedrigste Priorität ist 0 und die höchste 31. Die Prioritäten 16–31 sind Echtzeitprioritäten, 1–15 werden an die Benutzer-Threads vergeben und 0 ist für Systemzwecke reserviert. Die Priorität von Benutzer-Threads kann, im Gegensatz zu den Kernel-Threads, vom System angepasst werden. Hierzu besitzen die Benutzer-Threads eine dynamische Priorität. Diese kann vom System erhöht oder verringert werden, allerdings nur bei Threads mit einem Prioritätenwert 0–15. Deren Priorität wird um maximal 15 Prioritätsstufen erhöht (*Priority Boost*). Es gibt drei Situationen, in der die dynamische Priorität erhöht wird:

1. Wird ein Fenster, das der Thread-Klasse `NORMAL_PRIORITY_CLASS` angehört, in den Vordergrund gebracht, wird die Prioritätsklasse des Threads der zu diesem Fenster gehört, erhöht.

2. Immer wenn in einem Fenster Eingaben gemacht werden, z. B. Tastatureingaben, wird die Priorität des Threads dem das Fenster angehört, erhöht.

3. Wenn das Ereignis, auf das ein blockierter Thread gewartet hat, eingetroffen ist, erfährt dieser Thread einen *Priority Boost*.

Nachdem ein Thread einen solchen Boost erfahren hat, dekrementiert der Scheduler die Priorität jedes Mal um eine Stufe sobald der Thread ein komplettes Quantum gelaufen ist. Zur Vermeidung von Prioritätsinversion wird einer zufälligen Auswahl an lauffähigen Prozessen eine höhere Priorität zugewiesen. Auch ein Mechanismus, der Starvation verhindern

soll, ist vorhanden: Alle ca. vier Sekunden prüft der Scheduler, ob lauffähige Threads existieren, die seit der letzten Prüfung nicht mehr im Besitz der CPU waren. Ist dies der Fall, wird die Priorität der Threads um 15 Stufen angehoben und deren Quantum verlängert.

3.2.4. Linux

Dieser Abschnitt befasst sich zuerst mit dem O(1)-Scheduler, der ab Linux-Kernel-Version 2.6 eingesetzt wurde. Der ab Linux-Kernel 2.6.22 eingesetzte Completely-Fair-Scheduler (CFS) wird anschließend behandelt.

O(1) Scheduler

Hier handelt es sich um ein MLFQ-Verfahren der Gruppe Decay-usage-Scheduler. Linux unterscheidet 140 verschiedene Prioritäten. Diese sind unterteilt in drei Klassen von Threads: Die beiden Echtzeit-Thread-Klassen (1) Echtzeit-FIFO und (2) Echtzeit-Round-Robin mit Prioritätsspektrum von 0–99 und die dritte Klasse (3) Timesharing, die Nicht-Echtzeit-Threads betrifft und den Prioritäten 100–139 zugeordnet werden (vgl. Tanenbaum, 2009, S. 869 – S. 871).

Die (1)-Klasse wird in FIFO-Manier abgearbeitet, die Klassen (2) und (3) in RR. Es gibt auch hier viele Ähnlichkeiten zu den bereits vorgestellten Schedulern. Wie bei FreeBSD ULE basiert das Finden des nächsten Threads auf einer Bit-Map. Es existiert eine Runqueue pro CPU, die mehrere Queues enthält. Hier heißen die Arrays *active* und *expired*. Es existiert jedoch kein *idle*-Array wie bei FreeBSD ULE. Diese beiden Arrays *active* und *expired* sind jeweils Zeiger auf ein Array von 140 Listenköpfen. Das Prinzip ist ähnlich – neue Threads werden in das Array *active* eingefügt. Threads die ihr Quantum verbrauchen kommen in das Array *expired*. Die die das nicht tun, weil sie vorher blockieren, werden wieder ans Ende der jeweiligen Queue im *active*-Array einsortiert. Sobald sich keine Threads mehr in den Queues des *active*-Arrays befinden, werden die Zeiger getauscht, und das *expired*-Array wird zum *active*-Array. Auch Linux arbeitet mit einem Nice-Befehl (Werte von -20 bis +19), welcher die statische Priorität eines Threads festlegt. Die Startpriorität eines Prozesses beträgt 0 und kann nur durch den Nice-Befehl geändert werden.

Neben der statischen Priorität besitzt jeder Thread noch eine dynamische Priorität. Diese wird ständig neu berechnet. So werden Threads, die I/O-lastig sind, mit einem Prioritätsbonus von bis zu +5 belohnt. CPU-lastige Threads werden mit einer Prioritätsanpassung von maximal -5 bestraft. Diese Boni und Mali werden zur statischen Priorität addiert bzw. subtrahiert. Auch hier werden, wie beim FreeBSD ULE-Verfahren, zur Unterscheidung von I/O-lastigen und CPU-lastigen Threads, Statistiken geführt.

Linux weist Threads mit einem hohen Prioritätswert hohe Quanten zu und Threads mit niedriger Priorität niedrige Quanten. Folgend werden die Formeln zur Berechnung des Quantums dargestellt (vgl. Bovet & Cesati, 2008, S. 263). Falls die statische Priorität kleiner 120 ist, gilt:

$$Quantum \text{ in ms } = (140\text{–}statische\ Priorität) \times 20$$

Falls die statische Priorität größer oder gleich 120 ist, gilt:

$$Quantum \text{ in ms } = (140\text{–}statische\ Priorität) \times 5$$

Completely-Fair-Scheduler

Der CFS-Scheduler ist ein *Distributed-Weighted-Round-Robin*-Verfahren (vgl. Li et al., 2009, S. 65) vom Typ *Fair-Queuing*. Er unterscheidet sich vollkommen von den bisher in diesem Kapitel untersuchten Schedulern. Er ist darauf ausgelegt die CPU-Zeit möglichst fair auf alle Threads zu verteilen (vgl. Mandl, 2013, S.133). Im Unterschied zu seinem Vorgänger unterhält der CFS keine Statistiken über laufende Threads. Es existieren auch keine Runqueues und Quanten. Threads werden keine Prioritäten zugeordnet und es gibt nicht das Prinzip der active- und expired-Queues. Die Laufzeitkomplexität der wichtigsten Operationen wie Suchen, Einfügen und Löschen beträgt O(log n).

Jeder Thread soll einen gerechten Anteil der CPU zugewiesen bekommen. Wenn es beispielsweise zwei lauffähige Threads gibt, wird jedem Thread 50 % der CPU zugeteilt. Da auf realer Hardware nur ein Thread die CPU nutzen kann, werden die nichtrechnenden Threads gegenüber dem rechnenden Thread, benachteiligt. Daher ist jedem Thread ein Schlüssel (`wait_runtime`) zugeordnet. Dieser Schlüssel besitzt eine Genauigkeit im Nanosekunden-Bereich. Er reflektiert wie lange der Thread bereits auf seine Ausführung wartet. Bei Threads, die aktuell nicht der CPU zugeteilt sind, wird `wait_runtime` stetig erhöht. Während die CPU einem Thread zugeteilt ist, wird die Zeit die er rechnet, von seinem `wait_runtime`-Wert abgezogen. Des weiteren gibt es einen Wert `fair_clock`. Dieser Wert reflektiert, wieviel Zeit einem Thread die CPU hätte zugewiesen werden sollen, wenn er lauffähig gewesen wäre. Die Umsetzung des Verfahrens, erfolgt mittels eines *Rot-Schwarz-Baumes* (auch R/B-Baum) pro CPU. Jeder Thread stellt einen Knoten im Baum dar. Die Threads sind nach dem „`wait_runtime - fair_clock`“-Wert sortiert. Es wird immer der Thread der sich am weitesten *links* im Baum befindet der CPU zugeteilt (vgl. Molnar, 2008). Der `Nice`-Befehl wird umgesetzt, indem durch Erhöhung bzw. Verminderung des Nice-levels, die Zeit, die er die CPU nutzen darf, prozentual angepasst wird. Beispielsweise wird, wenn das Nice-level eines Threads um +1 angehoben wird, seine CPU-Nutzungszeit um 10 % reduziert.

3.3. Zusammenfassung der Betriebssystem-Untersuchung

In diesem Kapitel erfolgte eine Untersuchung verschiedener Scheduling-Verfahren in Lehrbetriebssystemen und für den produktiven Einsatz geeignete Betriebssystemen. So konnten die Verfahren RR, statische Prioritätsscheduler, MLFQ in verschiedenen Ausprägungen, sowie Fair-Queuing am praktischen Beispiel erörtert werden. Die beleuchteten Lehrbetriebssysteme, mit der Ausnahme von xv6, arbeiten bzw. sind ausgelegt, Scheduling mittels Prioritäten umzusetzen. In xv6 und Minix 2 sind statische Prioritätsscheduler bzw.

ML-Scheduler im Einsatz. Während in Minix 3 bereits ein MLFQ-Verfahren eingesetzt wird, sind PintOS und GeekOS auf dieses Scheduling-Verfahren ausgelegt, d. h. es gibt hier bereits vordefinierte Funktionen, die noch nicht vollständig implementiert sind. Diese sollen von Studenten implementiert werden.

Die klassischen hier erwähnten Betriebssysteme, wie Unix, FreeBSD bis Kernel-Version 5.1 und Linux bis Kernel-Version 2.6.22 nutzen prioritätenbasierte MLFQ-Verfahren. In den hier untersuchten aktuellen Betriebssystemen zeichnen sich zwei Richtungen des Schedulings ab: Windows und FreeBSD ab Kernel-Version 5.1 verwenden prioritätenbasierte MLFQ-Verfahren. Linux ab Kernel-Version 2.6.23 nutzt ein Fair-Queuing-Verfahren. Auch Solaris, welches in dieser Arbeit nicht ausführlich untersucht wird, nutzt in der aktuellen Version standardmäßig einen MLFQ-Scheduler. Dieser unterstützt auch Fair-Share-Scheduling (vgl. Mauro & McDougall, 2006, S. 22).

Keines der untersuchten Betriebssysteme unterstützt harte Echtzeitanforderungen. Hingegen werden weiche Echtzeitanforderungen von allen für den produktiven Einsatz geeigneten Betriebssystemen außer Unix V4.3 unterstützt. Auch Minix 3 unterstützt weiche Echtzeitanforderungen. Anhand von FreeBSD ULE und Linux wurde ein Einblick in Scheduling-Verfahren gegeben, die speziell auf Multiprozessor bzw. Multikern-Systeme ausgelegt sind.

Tabelle 3.1.: Scheduling-Verfahren der untersuchten Betriebssysteme

Betriebssystem	**Verfahren**	**Komplexität**
xv6	RR	niedrig
PintOS	RR	niedrig
Xinu	statische Prioritäten	mittel
GeekOS	statische Prioritäten	mittel
Minix 2	statische Prioritäten	mittel
Minix 3	MLFQ	hoch
Unix	MLFQ	hoch
FreeBSD	MLFQ	hoch
Linux bis 2.6.22	MLFQ	hoch
Linux ab 2.6.23	Fair-Queuing	hoch
Solaris ab V9	Standard MLFQ	hoch
Windows XP und höher	MLFQ	hoch

4. Entwurf

Die Wahl eines geeigneten Scheduling-Verfahrens gestaltet sich als schwierige Aufgabe. Ein Betriebssystem-Designer muss eine Vorstellung davon haben, welche Programme auf dem System laufen werden bzw. wie sich die Prozesse verhalten werden (vgl. Silberschatz, 2010, S. 213). Hieraus kann er ein geeignetes Scheduling-Verfahren ableiten. Im Fall von Ulix werden die Anforderungen an den Scheduling-Algorithmus durch eine Expertenbefragung ermittelt. Hierzu wird in diesem Kapitel die Auswertung des Fragebogens erläutert und das Ergebnis vorgestellt. Auf dieser Basis wird das Verfahren gewählt. Nach der Vorstellung der Gestaltung des Verfahrens erfolgt die Darstellung der Spezifikation.

4.1. Auswertung des Expertenfragebogens

Wie in Kapitel 1.3.3 erwähnt, wurde eine Expertenbefragung durchgeführt. Dies wurde durch einen Fragebogen realisiert (Anhang A), welcher mehrere ordinalskalierte und eine offene Fragestellung beinhaltet. Die Antworten der Fragen des Fragebogens fließen, wie im Folgenden beschrieben, in die Wahl des Scheduling-Verfahrens und dessen Umsetzung ein:

- Die Kriterien, welche die Experten als *sehr wichtig* erachten sind maßgeblich für die Umsetzung. Diese bilden die *Soll-Kriterien*.
- Das Ergebnis der offenen Frage ist maßgeblich für die Umsetzung und dient als zusätzliches *Soll-Kriterium*.
- Die Kriterien, welche die Experten als *wichtig* erachten werden bei der Umsetzung berücksichtigt. Diese bilden die *Kann-Kriterien*.
- Die Kriterien, welche die Experten als *unwichtig* erachten werden bei der Umsetzung nicht berücksichtigt.

4.2. Ergebnis der Expertenbefragung

Die Rücklaufquote der Expertenbefragung lag bei 100 %. Das Ergebnis der Expertenbefragung ist in Tabelle 4.1 dargestellt. Die geforderten Kriterien Reaktionszeit und Proportionalität sprechen für ein Dialogbetrieb-orientiertes Verfahren. Es soll, in für den produktiven Einsatz geeigneten, Unix-ähnlichen Betriebssystemen im Einsatz sein und auf bewährten Algorithmen beruhen. Es soll didaktisch gut vermittelbar implementiert werden und kann eine niedrige Komplexität besitzen. Weiter kann das Verfahren weiche Echtzeitanforderungen unterstützen. Es werden weder *Ausschluss-* noch *Anschluss-Kriterien* definiert.

Tabelle 4.1.: Zusammenfassung des Ergebnisses der Expertenbefragung

Kriterium	Soll-Kriterium	Kann-Kriterium
Verfahren wird in kommerziellen OS eingesetzt	X	
bewährte Algorithmen/Konzepte einsetzen	X	
Auf gute didaktische Vermittelbarkeit achten	X	
Reaktion/Antwortzeit	X	
Proportionalität		X
Unterstützung weicher Echtzeitanforderungen		X
Niedrige Komplexität des Verfahrens		X

4.3. Wahl des Algorithmus

Da die Anforderungen an das zu implementierende Verfahren definiert sind, kann nun der Scheduling-Algorithmus gewählt werden. Hierzu werden die bereits in Kapitel 2.9 vorgestellten Verfahren hinsichtlich ihrer Eignung untersucht. Zuerst werden die nicht geeigneten Verfahren dargestellt und begründet warum sie nicht geeignet sind. Anschließend werden die geeigneten Verfahren dargestellt und das gewählte Verfahren präsentiert.

4.3.1. Nicht geeignete Verfahren

- *FIFO, SJF, SRTN* sind für den Stapelverarbeitungsbetrieb ausgelegt. FIFO und SJF sind nicht unterbrechende Verfahren. Darüber hinaus beruhen SJF und SRTN auf Vorhersagen über die Laufzeit der Prozesse. Dies erfordert eine spezielle Systemumgebung, die in Ulix nicht gegeben ist.
- *RMS, DMS, EDF* und *LLF* basieren auf Deadlines und Periodizität. Dies gibt es bei Ulix nicht. Sie kommen ausschließlich in Echtzeitbetriebssystemen zum Einsatz.
- *Fair-Share-Scheduling* ist nur sinnvoll, wenn das Betriebssystem mehrbenutzerfähig ist. Da Ulix nicht mehrbenutzerfähig ist wird es als nicht geeignet eingestuft.

4.3.2. Geeignete Verfahren

- *Lottery-Scheduling*: Dieses Verfahren eignet sich für Ulix. Die Implementation basiert auf Zufallszahlen. Hierfür müsste eine Funktion implementiert werden, die diese Zufallszahlen generiert. Dazu könnte der für die MIPS-Rechnerarchitektur konzipierte *Fast Random Number Generator* als Vorlage verwendet werden (vgl. Waldspurger & Weihl, 1995, S. 95). Der Sourcecode des Verfahrens ist verfügbar und in ANSI C implementiert (vgl. Waldspurger & Weihl, 1995, S. 38).
- *Fair-Queuing*: Auch dieses Verfahren eignet sich für Ulix. Ein Vertreter dieses Verfahrens ist das komplexe *Distributed-Weighted-Round-Robin-Verfahren*, wie es in Linux der CFS-Scheduler umsetzt. Eine weniger komplexe Variante beschreiben Waldspurger et al. als *Stride Scheduling*. Auch hierfür ist der Sourcecode verfügbar und in ANSI C geschrieben (vgl. Waldspurger & Weihl, 1995, S. 52).
- *Guaranteed-Scheduling*: Dieses Verfahren eignet sich für Ulix. Es basiert auf Prioritäten.

- *Round-Robin-Scheduling* wurde in der ULIX-Version 0.06 von H.G.-Eßer implementiert.
- *Multilevel-Queue-Scheduling / Prioritätsscheduling*: Diese Klasse von Verfahren eignen sich für ULIX. Die Komplexität ist hier nicht zu hoch.

4.3.3. Das gewählte Verfahren: MLFQ

Neben dem MLFQ-Verfahren ist nur das *Distributed-Weighted-Round-Robin-Verfahren* in einem aktuellen Betriebssystem als Standardverfahren im Einsatz. Alle anderen geeigneten Verfahren finden keine Anwendung in einem aktuellen System. Der CFS ist zu komplex für die Umsetzung in dieser Arbeit: Das Linux-Modul fair.c der Linux-Kernel-Version 3.3.7 enthält die Funktionalität dieses Verfahrens und besteht aus ca. 4400 Code-Zeilen (vgl. Mandl, 2013, S. 134). Dieser Algorithmus wird daher nicht gewählt.

MLFQ-Scheduler sind (immer noch) in den meisten aktuellen Betriebssystemen, wie in Tabelle 3.1 zu sehen ist, im Einsatz. Die Kriterien der Anforderungsanalyse werden durch dieses Verfahren vollständig erfüllt. Es gibt komplexe und weniger komplexe MLFQ-Varianten. Die Gruppe der Head-of-line-MLFQ-Verfahren (siehe Abschnitt 2.9.8) gehört zu den weniger komplexen Verfahren. Dieser Algorithmus wird für diese Arbeit gewählt.

4.4. Gestaltung des MLFQ-Verfahrens

Das in dieser Arbeit umgesetzte Verfahren orientiert sich an der Arbeit von Arpaci-Dusseau & Arpaci-Dusseau (vgl. Arpaci-Dusseau & Arpaci-Dusseau, 2012). Diese beschreiben das Basisprinzip des MLFQ-Verfahrens. Es folgt eine Zusammenfassung dieses Prinzips. Anschließend werden die Themen abgegrenzt, die, obwohl sie im Basis-Prinzip enthalten sind, in dieser Arbeit nicht umgesetzt werden.

4.4.1. MLFQ-Verfahren: Das Basispinzip

Bei diesem Verfahren existieren mehrere Prioritätsstufen. Die Prioritäten dienen der Entscheidung welchem Prozess als nächstes die CPU zugeteilt wird. Wird ein Prozess, der eine höhere Priorität besitzt als der gerade rechnende Prozess, rechenbereit, muss dem gerade rechnenden Prozess die CPU entzogen werden. Sobald die Priorität eines Prozesses erhöht oder verringert wird, muss geprüft werden welcher Prozess laufen darf. Prozesse mit gleicher Priorität werden im Round-Robin-Verfahren abgearbeitet.

Es sollen interaktive Prozesse bevorzugt werden. Ein I/O-lastiger Prozess gibt die CPU meist ab, bevor sein Quantum abgelaufen ist. In diesem Fall ändert sich seine Priorität nicht. Ein CPU-lastiger Prozess nutzt sein Quantum meist komplett. Jedes Mal, wenn ein Prozess sein Quantum verbraucht, wird er in die nächst niedriger priorisierte Stufe verschoben. In dieser Art wird MLFQ versuchen, aus der Vergangenheit von Prozessen zu lernen. Aufgrund dieser Historie soll sein zukünftiges Verhalten prognostiziert werden.

Prozesse mit hoher Priorität bekommen das größte Quantum. Prozesse mit niedriger Priorität bekommen das kleinste Quantum. Es soll möglich sein, mittels System Calls

die Priorität eines Prozesses absolut festzulegen, die Priorität eines Prozesses abzufragen sowie einen Nice-Wert für einen Prozess festzulegen.

4.4.2. Ausschluss

Das Basisprinzip ist an die Gegebenheiten von Ulix sowie an die Begrenzung der Arbeit angepasst:

- Verhinderung von *Gaming* wird nicht implementiert. Unter Gaming wird verstanden, dass Entwickler ihre Programme so anpassen, dass sie sich einen Vorteil bei Schedulingentscheidungen verschaffen. Dies ist für ein Lehrbetriebssystem nicht in erster Linie relevant.
- Ein neuer Prozess erhält nicht die höchste zu vergebende Priorität. Neue Prozesse bekommen die Standardpriorität 50 bzw. erben ihre Priorität vom Vaterprozess. Dies erweist sich bei Experimenten mit Prioritäten als nützlich. Auch können Prioritäten über 50, für weiche Echtzeitanforderungen genutzt werden.
- Verhinderung von *Priority-Inversion* oder *Starvation* bringt eine Erhöhung der Komplexität und wird aufgrund der Begrenzung der Arbeit nicht umgesetzt.

4.5. Spezifikation

Ein wichtiger Teil des Entwurfsprozesses ist, die gewünschten Eigenschaften und Randbedingungen, die durch die Anforderungsanalyse ermittelt wurden, zu dokumentieren (vgl. Liggesmeyer, 2009, S. 442). Dies ist für die Implementierung und Evaluierung erforderlich. Hierzu wird in diesem Kapitel eine Spezifikation erstellt. Der Spezifikation liegen das von Arpaci-Dusseau & Arpaci-Dusseau dargestellte Basis-Verfahren sowie die vom Autor gewählten Rahmenbedingungen zu Grunde.

- Anforderung 1a: Die Priorität eines Prozesses soll im User-Mode abgefragt werden können.
- Anforderung 1b: Die Priorität eines Prozesses soll im User-Mode absolut festgelegt werden können.
- Anforderung 1c: Die Priorität eines Prozesses soll im User-Mode relativ festgelegt werden können.
- Anforderung 2: Neue Prozesse erhalten die Standardpriorität (50).
- Anforderung 3: Ein Elternprozess vererbt seine Priorität an seine Kindprozesse.
- Anforderung 4: Prozesse mit gleicher Priorität werden im Round-Robin-Verfahren abgearbeitet.
- Anforderung 5: Sobald sich die Priorität eines Prozesses ändert, muss geprüft werden, welcher Prozess laufen darf.

- Anforderung 6: Prozesse die ihr Quantum verbrauchen, werden um eine Prioritätsstufe herabgestuft.
- Anforderung 7: Wird ein Prozess rechenbereit, der eine höhere Priorität besitzt als der gerade rechnende Prozess, muss dem rechnenden Prozess die CPU entzogen werden.
- Anforderung 8: Prozesse mit hoher Priorität erhalten das größte Quantum. Prozesse mit niedriger Priorität erhalten das kleinste Quantum (siehe Tabelle 5.1).
- Anforderung 9: Die niedrigste Prioritätsstufe, die an einen Prozess vergeben bzw. in die er herabgestuft werden kann, beträgt 2.
- Anforderung 10: Prozesse die ihr Quantum nicht verbrauchen, dürfen ihre Prioritätsstufe behalten.
- Anforderung 11: Es gibt 99 Prioritätsstufen für reguläre Prozesse: 100 ist die höchste Priorität und 2 die niedrigste Priorität.
- Anforderung 12: Die Priorität des Idle-Prozesses beträgt 1.

5. Implementierung

Dieses Kapitel beschreibt zuerst das chronologische Vorgehen bei der Durchführung der Implementation. Anschließend wird die Umsetzung der Implementation durch *Literate Programming* dargestellt. Die Implementierung erfolgt auf Basis der Spezifikation aus Kapitel 4.5.

5.1. Das chronologische Vorgehen

Dieses Kapitel erläutert den zeitlichen Verlauf der Implementation. Das Vorgehen wird in drei Phasen gegliedert. Wie bereits in Kapitel 1.3.4 erwähnt, erfolgte die Umsetzung der Implementation auf Basis des inkrementell-iterativen Vorgehensmodells.

1. *Einführung von Prozessprioritäten*: Es wurde zu Beginn ein Simulator erstellt (Anhang 3). Dieser Simulator wurde in der Entwicklungsumgebung Microsoft Visual Studio 2012 implementiert und basiert auf bestehenden Funktionen von Ulix sowie auf Funktionen von XINU. Mit diesem Simulator konnte eine Ready-Queue simuliert werden. Funktionen die das Hinzufügen und Entfernen von Elementen zur Ready-Queue ermöglichen, konnten so implementiert werden. Diese stellten das Basisinkrement dar. Anschließend wurden die Funktionen in Ulix integriert. Das Ergebnis dieser Phase war ein statischer Prioritäts-Scheduler (Anforderungen 2, 3, 4, 5, 7, 9, 11 und 12).

2. *System Calls*: Um den Scheduler innerhalb des User-Modes parametrisieren zu können wurden mehrere System Calls (`getpriority()`, `setpriority()` und `nice()`) implementiert (Anforderungen 1a, 1b und 1c).

3. *Feedback-Funktionalität*: Im letzten Schritt wurden dynamische Quanten (Anforderung 8) sowie Funktionalität, die eine Reaktion auf das Prozessverhalten ermöglicht (Anforderungen 6 und 10), eingeführt.

5.2. Einführung von Prozess-Prioritäten

Es wurden in Kapitel 3 bereits verschiedene Varianten zur Umsetzung von prioritätsbasiertem Scheduling erläutert. Es können zwei Varianten abgegrenzt werden: Die Variante, die auch in Minix 2 (siehe Kapitel 3.1.5) im Einsatz ist und die Variante in Xinu (siehe Kapitel 3.1.3). Die erste Variante teilt die Ready-Queue in mehrere Queues. Sie ist sehr performant bei Operationen wie Hinzufügen, Entfernen oder Verschieben von Prozessen (logarithmische Laufzeitkomplexität). Die Implementierung ist jedoch komplex. So fällt die Wahl auf die zweite Variante (Kapitel 3.1.3). Diese realisiert die Ready-Queue mittels

einer doppelt verketteten Liste in der die Prozesse ihrer Priorität nach einsortiert werden. Hier überwiegen die Vorteile der einfachen Implementierung und der guten Skalierbarkeit gegenüber den Nachteilen der schlechteren Performance (lineare Laufzeitkomplexität). Gerade die Skalierbarkeit spielt eine wichtige Rolle für das Experimentieren mit der Anzahl der Prioritätsstufen (maximal 32767 wenn die Variable `priokey` den Typ `int` zugewiesen bekommt). In Abbildung 5.1 ist die konzeptuelle Organisation der Ready-Queue dargestellt.

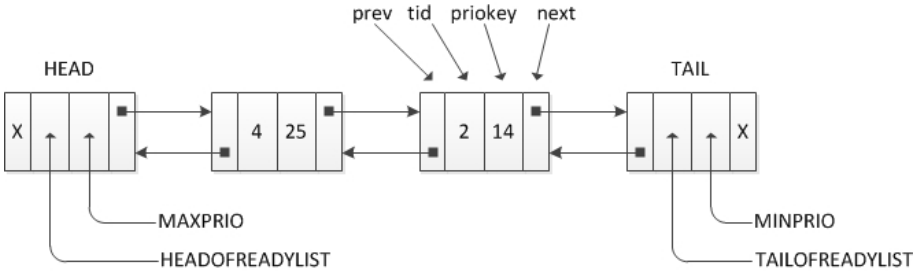

Abbildung 5.1.: Die konzeptuelle Organisation einer doppelt verketteten Liste die Prozesse 4 und 2 mit priokey 25 bzw. 14 enthält. Darstellung in Anlehnung an Comer, 2011, S. 50.

5.3. Die Implementation mittels Literate Programming

Dieser Abschnitt beschreibt zuerst die Implementation der Funktionen `insert()` und `dequeue()` die die Basis für das prioritätsbasierte Scheduling bilden. Anschließend werden Modifikationen an bestehenden Funktionen `start_program_from_disk()`, `ulix_fork()` und `scheduler()` dargestellt. Es folgt die Implementation der Feedback-Funktionalität sowie der System Calls. Es wurden drei System Calls implementiert, um innerhalb des User-Modes Prozessprioritäten auslesen und setzen zu können. Mit dem System Call `setpriority()` kann die Priorität eines Prozesses absolut festgelegt werden. Mittels `getpriority()` kann die Priorität eines Prozesses ausgelesen werden. Mittels `nice()` kann die Priorität, relativ zu der Priorität, die der Prozess aktuell besitzt, festgelegt werden. Durch diese System Calls kann das Verfahren vom Benutzer parametrisiert werden und somit eine Trennung von Policy und Mechanismus erfolgen (vgl. Levin et al., 1975, S. 134).

5.3.1. Die neue Ready-Queue

In Ulix werden Prozesse in dem Array `thread_table`, welches auf 1024 Einträge dimensioniert ist, verwaltet. Dieses Array nimmt `TCB`-Elemente vom Typ `struct TCB` auf. Diese `struct`-Elemente enthalten unterschiedliche Datentypen. Jeder Prozess wird in einem solchen `TCB-struct` abgebildet. So enthält ein `TCB-struct`, neben anderen Elementen, die Prozess-ID `tid` vom Typ `thread_id`. Weiterhin sind die Variablen `prev` und `next` vom Typ `thread_id` enthalten, mit welchen eine doppelt verkettete Liste realisiert wird. Für die Nutzung von Prioritäten wird ein zusätzlicher Schlüssel benötigt, der vom jeweiligen Prozess den Wert seiner Priorität speichert. Hierfür wird der `struct TCB` um die Variable `priokey` vom Typ `int` erweitert.

43a ⟨*more TCB entries* 43a⟩≡

```
int priokey;
```

Es muss die Anzahl der möglichen Prioritätsstufen definiert werden. Da alle Prozesse zwischen MINPRIO und MAXPRIO einsortiert werden, sind bei folgender Konfiguration 100 (`MAXPRIO`-1; Die Stufen 0 und 101 werden nicht für Prozesse genutzt) Prioritätsstufen möglich.

43b ⟨*kernel declarations* 43b⟩≡ 43c ▷

```
#define MAXPRIO                 101
#define MINPRIO                   0
```

Defines:
`MAXPRIO`, used in chunks 44a and 55c.
`MINPRIO`, used in chunks 44a and 55c.

Um mit mehr Prioritätsstufen zu experimentieren, muss lediglich `MAXPRIO` angepasst werden.

ULIX 0.07 nutzt einen `TCB`-Eintrag, um den Anfang und das Ende der Ready-Queue zu kennzeichnen. Diesem ist die `tid` 0 zugewiesen. Er kann nicht zum Speichern von Prozess-Informationen genutzt werden und dient lediglich als sogenannter *Anker*. Die prioritätsbasierte Ready-Queue benötigt zwei dieser Anker.

43c ⟨*kernel declarations* 43b⟩+≡ ◁ 43b 43d ▷

```
#define HEADOFREADYLIST         0
#define TAILOFREADYLIST         1000
```

Defines:
`HEADOFREADYLIST`, used in chunks 44–46 and 52.
`TAILOFREADYLIST`, used in chunks 44a and 52.

Der Konstante `HEADOFREADYLIST` wird der Wert 0 zugewiesen. Dies ist die `tid` des Head-Elements. Die `tid` des Tail-Elements kann, innerhalb der 1024 Einträge, bis auf den Wert 1, beliebig gesetzt werden. Grund hierfür ist der Idle-Prozess, der in ULIX auf `tid` 1 festgelegt ist. Es wurde hierfür die `tid` 1000 gewählt und der Konstante `TAILOFREADYLIST` zugewiesen. Zudem wird die Konstante `EMPTY` auf -1 gesetzt. Diese kommt zum Einsatz, wenn ein `prev` oder `next`-Wert eines Prozesses als leer gekennzeichnet werden soll, weil er keinen Vorgänger bzw. Nachfolger mehr hat.

43d ⟨*kernel declarations* 43b⟩+≡ ◁ 43c 44c ▷

```
#define EMPTY                   -1
```

Defines:
`EMPTY`, used in chunks 44a and 46c.

In der `main()` Funktion findet die Initialisierung der Ready-Queue mit den gerade definierten Konstanten statt. Das Head-Element besitzt keinen Vorgänger. Somit wird sein `prev`-Wert als leer markiert. Da die Liste zu Beginn leer ist, ist der Nachfolger des Head-Elements das Tail-Element und der Vorgänger des Tail-Elements das Head-Element. Dem Head-Element wird der Prioritätswert 101 zugewiesen. Er bekommt somit Priorität zugewiesen, die um eine Stufe höher ist, als die Prioritätsstufe, die der höchstpriorisierte Prozess annehmen kann. Dem Tail-Element wird der Prioritätswert 0 zugewiesen. Dies ist eine Stufe niedriger als die Stufe, die der Prozess mit der niedrigsten Priorität annehmen kann. Das Tail-Element besitzt keinen Nachfolger, und daher wird sein `next`-Wert als leer

markiert. Der Anker `TAILOFREADYLIST` wird zudem noch als verwendent gekennzeichnet, um nicht mehr an einen anderen Prozess vergeben zu werden.

44a ⟨*Felsner:main:init ready list* 44a⟩≡

```
thread_table[HEADOFREADYLIST].prev = EMPTY;
thread_table[HEADOFREADYLIST].tid = HEADOFREADYLIST;
thread_table[HEADOFREADYLIST].next = TAILOFREADYLIST;
thread_table[HEADOFREADYLIST].priokey = MAXPRIO;

thread_table[TAILOFREADYLIST].prev = HEADOFREADYLIST;
thread_table[TAILOFREADYLIST].tid = TAILOFREADYLIST;
thread_table[TAILOFREADYLIST].next = EMPTY;
thread_table[TAILOFREADYLIST].priokey = MINPRIO;
thread_table[TAILOFREADYLIST].used = true;
```

Uses `EMPTY` 43d, `HEADOFREADYLIST` 43c, `MAXPRIO` 43b, `MINPRIO` 43b, and `TAILOFREADYLIST` 43c.

5.3.2. Einfügen eines Prozesses in die Ready-Queue

Die Funktion `insert()` hat die Aufgabe, einen Prozess der Ready-Queue hinzuzufügen. Die bestehende Funktion `add_to_ready_queue()` wird von einigen anderen Funktionen aufgerufen. Um nicht all diese Funktionen modifizieren zu müssen, wird die Funktion `add_to_ready_queue()` beibehalten, ihr Inhalt aber so verändert, dass nur noch der Funktionsaufruf `insert()` vorhanden ist.

44b ⟨*Felsner:add to ready queue* 44b⟩≡

```
insert(t, thread_table[t].priokey);
```

Uses `insert` 45a.

Die Funktion `insert()` enthält die Logik, um Prozesse ihrem Prioritätswert nach in die Ready-Queue einzusortieren (siehe auch Programmablaufplan 5.2).

Beim Aufruf werden die beiden Parameter `proc` und `key` übergeben. Mittels der Variable `proc` vom Typ `thread_id` wird die `tid` des Prozesses, der in die Ready-Queue aufgenommen werden soll, übergeben. Die Variable `key` vom Typ `int` beinhaltet die Priorität des Prozesses. Es muss unterbunden werden, dass durch einen Scheduler-Aufruf von `timer_handler()` die Funktion unterbrochen wird. Dies wird mit dem Aufruf `DISABLE_SCHEDULER` bewirkt. Es folgt das Durchsuchen der Ready-Queue und die Einsortierung des Prozesses. Bevor aus der Funktion zurückgesprungen wird, werden Scheduler-Aufrufe durch den `timer_handler()` mit `DISABLE_SCHEDULER` wieder aktiviert.

Mittels der Variablen `lprev` vom Typ `thread_id` kann die Ready-Queue durchlaufen werden. Zu Beginn der Funktion wird ihr die `tid` des ersten Prozesses der Ready-Queue zugewiesen.

44c ⟨*kernel declarations* 43b⟩+≡ ◁43d 46b▷

```
void insert(thread_id proc, int key);
```

Uses `insert` 45a.

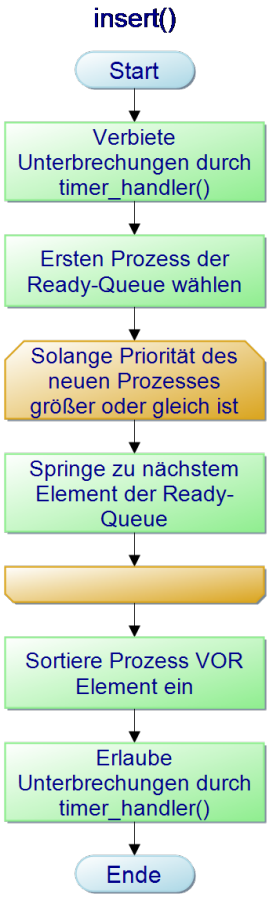

Abbildung 5.2.: Programmablaufplan der Funktion insert()

45a ⟨*kernel functions* 45a⟩≡ 46c ▷

```
void insert(thread_id proc, int key){
 thread_id lnext;
 thread_id lprev;
 DISABLE_SCHEDULER;
 lnext = thread_table[HEADOFREADYLIST].next;
 ⟨Felsner:kernelfunction:insert:schleife 45b⟩
 ⟨Felsner:kernelfunction:insert:einsortieren 46a⟩
 ENABLE_SCHEDULER;
 return;
}
```

Defines:
`insert`, used in chunks 44, 49a, 54c, and 55c.
Uses `HEADOFREADYLIST` 43c.

Die Ready-Queue wird dann, angefangen beim ersten Element der Ready-Queue, solange durchlaufen, wie der `priokey`-Wert den das Element mit der `tid lnext` besitzt, größer oder gleich dem Wert der Variablen `key` ist.

45b ⟨*Felsner:kernelfunction:insert:schleife* 45b⟩≡ (45a)

```
while (thread_table[lnext].priokey >= key){
 lnext = thread_table[lnext].next;
}
```

Sobald dies nicht mehr der Fall ist, wird er in die Ready-Queue vor `lnext` einsortiert, indem die Zeiger auf Vorgänger und Nachfolger entsprechend gesetzt werden.

46a ⟨*Felsner:kernelfunction:insert:einsortieren* 46a⟩≡ (45a)

```
lprev = thread_table[lnext].prev;
thread_table[proc].next = lnext;
thread_table[proc].prev = lprev;
thread_table[proc].priokey  = key;
thread_table[lprev].next = proc;
thread_table[lnext].prev = proc;
```

5.3.3. Die Auswahl eines neuen Prozesses

Der Scheduler wählt, wenn eine Scheduling-Entscheidung getroffen werden muss, stets den Prozess nach dem Head-Element der Ready-Queue. Dies ist Aufgabe der Funktion `dequeue()` (siehe auch Programmablaufplan 5.3). Zu Beginn muss auch hier mit `DISABLE_SCHEDULER` unterbunden werden, dass durch einen Scheduler-Aufruf der Funktion `timer_handler()` die Funktion unterbrochen wird. Dann wird der Variable `pid` vom Typ `thread_id` die `tid` des Prozesses zugewiesen, der sich am Anfang der Ready-Queue befindet. Anschließend wird die bestehende Funktion `remove_from_ready_queue()` aufgerufen. Diese entfernt diesen ersten Prozess aus der Ready-Queue indem die Zeiger auf Vor- und Nachfolger entsprechend gesetzt werden. Dies geschieht aus dem Grund, dass der laufende Prozess nicht in der Ready-Liste vorhanden sein darf, da er einen anderen Status besitzt. Schließlich wird die `tid` des entfernten Prozesses zurückgegeben, und Scheduler-Aufrufe durch den `timer_handler()` werden wieder ermöglicht.

46b ⟨*kernel declarations* 43b⟩+≡ ◁44c 47a▷

```
thread_id dequeue();
```

46c ⟨*kernel functions* 45a⟩+≡ ◁45a 48b▷

```
thread_id dequeue(){
 DISABLE_SCHEDULER;
 thread_id pid;
 pid=thread_table[HEADOFREADYLIST].next;
 remove_from_ready_queue(pid);
 thread_table[pid].next = EMPTY;
 thread_table[pid].prev= EMPTY;
 ENABLE_SCHEDULER;
 return pid;
}
```

Uses `EMPTY` 43d and `HEADOFREADYLIST` 43c.

5.3.4. Das erste Programm: Die Funktion start_program_from_disk()

Die Funktion `start_program_from_disk()` wird in der `main()`-Funktion aufgerufen. `start_program_from_disk()` startet, vereinfacht dargestellt, das Programm `testprog.c`. Dieses Programm läuft im User-Mode, enthält die User-Shell und bekommt immer den `tid`-Wert

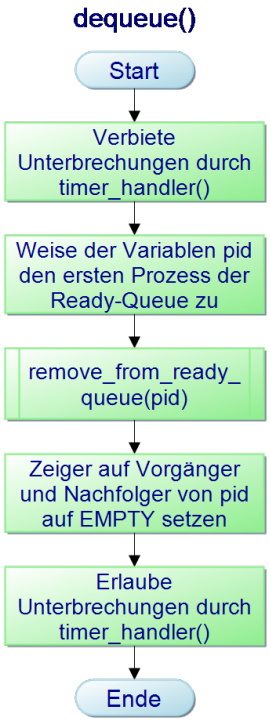

Abbildung 5.3.: Programmablaufplan der Funktion dequeue()

1 zugewiesen. Es dient als Idle-Prozess und ihm wird Priorität 1 (`IDLEPRIO`) zugewiesen. Dies ist der niedrigste Prioritätswert den ein Prozess besitzen kann. Der Idle-Prozess befindet sich stets, wenn weitere lauffähige Prozesse vorhanden sind, in der Ready-Queue an letzter Stelle.

47a ⟨*kernel declarations* 43b⟩+≡ ◁46b 47c▷

```
#define IDLEPRIO                                    1
```

Defines:
`IDLEPRIO`, used in chunk 47b.

47b ⟨*Felsner: start program from disk: set Prio* 47b⟩≡

```
thread_table[tid].priokey = IDLEPRIO;
```

Uses `IDLEPRIO` 47a.

5.3.5. Anpassen der Funktion ulix_fork()

Die Funktion `ulix_fork()` hat, wie in Unix auch, die Aufgabe, aus dem aktuellen Prozess eine Kopie zu erzeugen, welcher dann als Kindprozess des erzeugenden Prozesses läuft (Vaterprozess). Dieser Kindprozess soll die Priorität des Vaterprozesses erben, außer er stammt vom Init-Prozess (`start_program_from_disk`) ab. Dann wird ihm nicht die Idle-Priorität zugewiesen, sondern die Standardpriorität (50). Dies kann zu Experimentierzwecken genutzt werden. So werden alle Prozesse einer Shell mit deren Prioritätswert gestartet.

47c ⟨*kernel declarations* 43b⟩+≡ ◁47a 53b▷

```
#define STANDARDPRIO                                50
```

Defines:
STANDARDPRIO, used in chunk 48a.

48a ⟨*Felsner:ulix fork:set parent prio* 48a⟩≡
```
if (ppid == 1){
 thread_table[new_tid].priokey = STANDARDPRIO;
}
else
{
 thread_table[new_tid].priokey = thread_table[ppid].priokey;
}
```
Uses STANDARDPRIO 47c.

5.3.6. Die zentrale Scheduling-Funktion scheduler()

Sie wird von vielen anderen Funktionen aufgerufen (siehe Darstellung 5.4) und hat die Aufgabe, den ersten Prozess der Ready-Queue zu wählen. Wenn sich dieser nicht vom aktuell laufenden (`current_task`) unterscheidet, wird ihm die CPU zugeteilt (siehe auch Programmablaufplan 5.5).

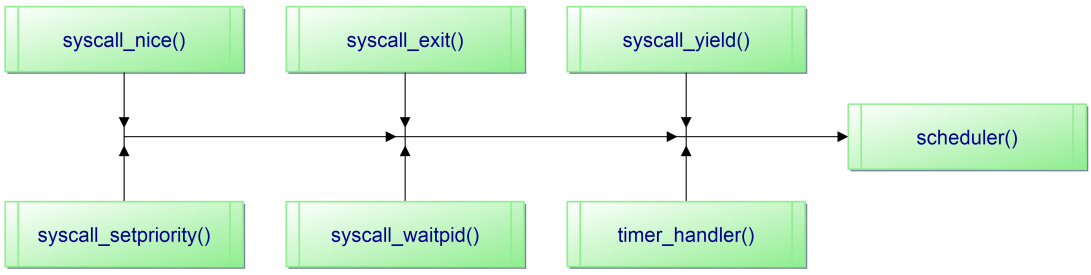

Abbildung 5.4.: Schematische Darstellung der Funktionsaufrufe der Funktion scheduler() durch andere Funktionen

48b ⟨*kernel functions* 45a⟩+≡ ◁46c 54c▷
```
void scheduler (struct regs *r, int source) {
 debug_printf ("*");
 inside_yield = false;

 ⟨check for zombies (never defined)⟩

 if (!scheduler_is_active)  return;
 if (!thread_table[2].used) return;    // are there already two threads?

 int tid = current_task; t_old = &thread_table[tid];

 ⟨Felsner:Scheduler:CheckState 49a⟩
 ⟨Felsner:Scheduler:FindNextProcessAndSet_new 49b⟩
 ⟨Felsner:Scheduler:SetStateCurr 49c⟩
 ⟨Felsner:Scheduler:SetQuantum 53e⟩
 ⟨Felsner:Scheduler:Resetquantumticks 53c⟩
```

```
⟨Felsner:Scheduler:ContextSwitch 50⟩
return;
}
```

Defines:
scheduler, used in chunks 52, 54c, and 55c.

Im Detail hat die Funktion folgende Aufgaben:

- Behandlung von Zombie-Prozessen. Dies ist bereits Bestandteil von ULIX.
- Prüfen, ob aus der `Scheduler`-Funktion unter Umständen vorzeitig zurückgesprungen werden kann. Dies ist der Fall, wenn durch den Aufruf `DISABLE_SCHEDULER` die Konstante `scheduler_is_active` auf den Wert 0 gesetzt wurde oder nur der Idle-Prozess existiert.
- Wenn der Prozess den Status `TSTATE_CURR` besitzt, wenn ihm also die CPU zugeteilt ist, wird der Status auf `TSTATE_READY` gesetzt und er anschließend in die Ready-Queue einsortiert.

49a ⟨*Felsner:Scheduler:CheckState* 49a⟩≡ (48b)

```
if (t_old->state == TSTATE_CURR) {
 t_old->state = TSTATE_READY;
 insert(tid, t_old->priokey);
}
```

Uses `insert` 45a and `TSTATE_CURR` 49d.

- Den nächstberechtigten Prozess mittels der Funktion `dequeue()` ermitteln und `t_new` setzen.

49b ⟨*Felsner:Scheduler:FindNextProcessAndSet*`t_new` 49b⟩≡ (48b)

```
tid = dequeue();
t_new = &thread_table[tid];
```

- Diesem Prozess den Status `TSTATE_CURR` zuweisen.

49c ⟨*Felsner:Scheduler:SetStateCurr* 49c⟩≡ (48b)

```
 thread_table[tid].state = TSTATE_CURR;
```

Uses `TSTATE_CURR` 49d.

49d ⟨*Felsner:state current* 49d⟩≡

```
#define TSTATE_CURR   9
```

Defines:
`TSTATE_CURR`, used in chunk 49.

- Jetzt finden die Berechnung des Quantums und die Rücksetzung der globalen Variable `quantum_ticks` statt. Dies wird in Kapitel 5.3.7 behandelt.

- Es folgt der Kontext-Wechsel, der durchgeführt wird, wenn sich neuer und alter Prozess unterscheiden. Dies ist bereits Bestandteil von ULIX.

50 ⟨*Felsner:Scheduler:ContextSwitch* 50⟩≡ (48b)

```
if (t_new != t_old) {
 current_task = tid;
 ⟨context switch (never defined)⟩
}
```

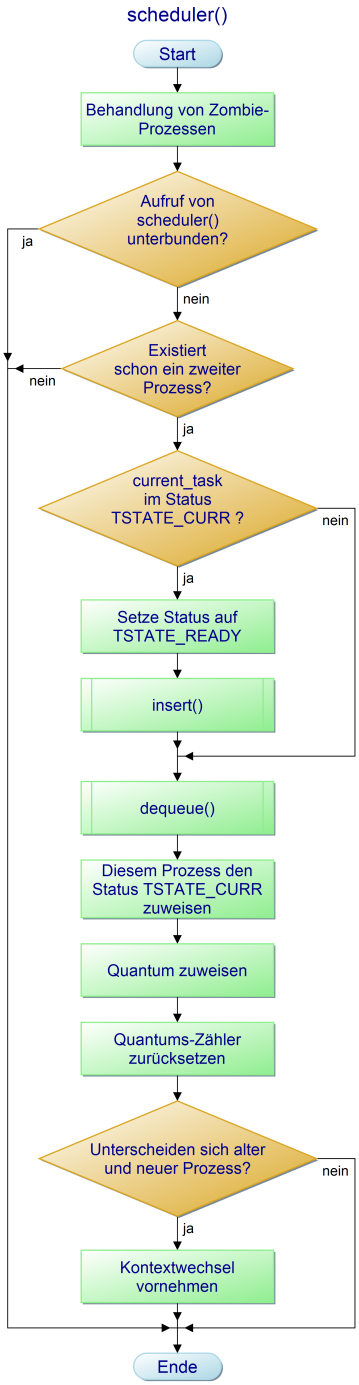

Abbildung 5.5.: Programmablaufplan der Funktion scheduler()

Preemptives Scheduling: Die Funktion timer_handler()

Die Funktion `timer_handler` wird bei jedem Timer-Interrupt aufgerufen (siehe auch Programmablaufplan 5.6).

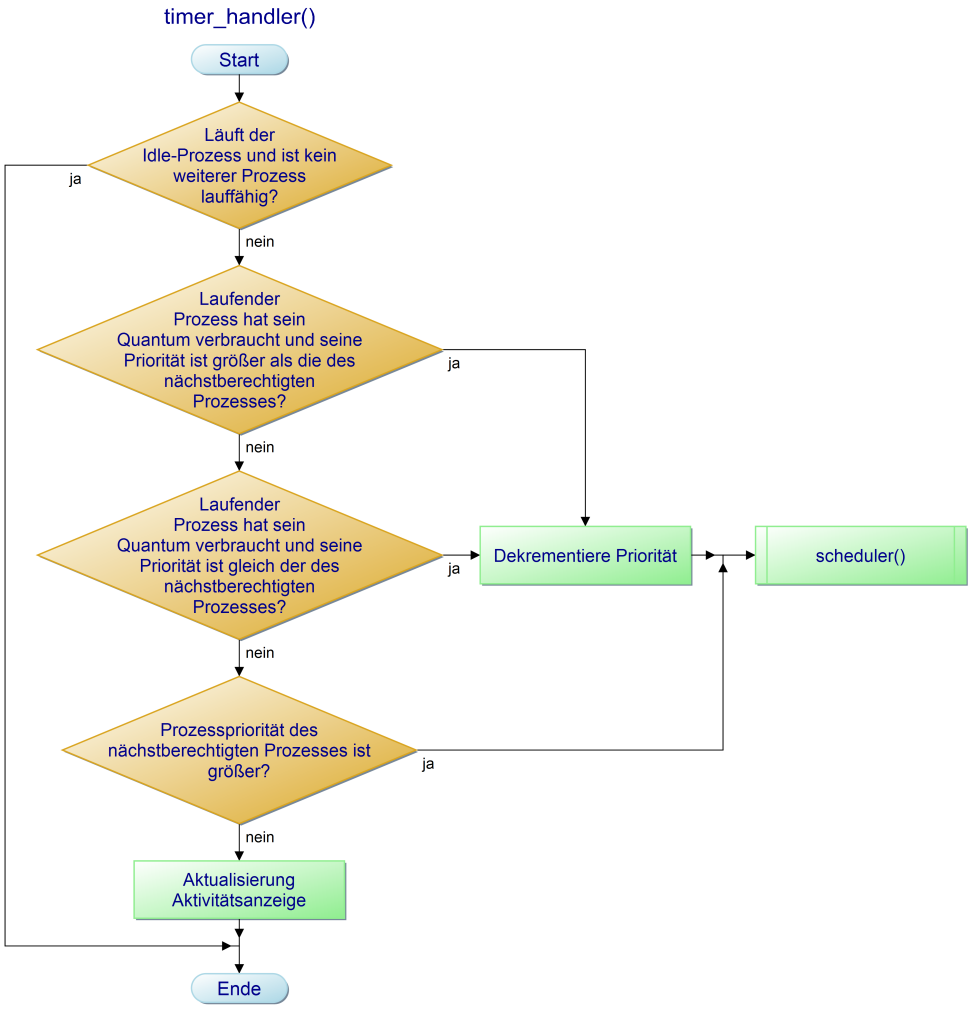

Abbildung 5.6.: Programmablaufplan der Funktion timer_handler()

Bei *jedem* `timer_handler`-Aufruf wird geprüft, ob der aktuell der CPU zugeteilte Prozess die CPU freigeben muss (um sie unter Umständen sofort wieder zu bekommen). Dies ist der Fall, wenn ein anderer Prozess berechtigt ist die CPU zu bekommen oder wenn das Quantum des laufenden Prozesses abgelaufen ist. Im letzten Fall wird die CPU dem Prozess sofort wieder zugeteilt, falls kein anderer Prozess berechtigt ist zu laufen. Im Detail wird `scheduler()` aufgerufen wenn:

1. die Priorität des laufenden Prozesses größer als die des nächstberechtigten Prozesses (dem ersten Prozess in der Ready-Queue) ist und sein Quantum abgelaufen ist.

2. die Priorität des laufenden Prozesses gleich groß wie die des nächstberechtigten Prozesses ist und sein Quantum abgelaufen ist.

3. die Priorität des laufenden Prozesses kleiner wie die des nächstberechtigten Prozesses ist.

In Fall *1* und *2* wird die Priorität des Prozesses eine Stufe herabgesetzt. In Fall *3* wird die Priorität nicht vermindert, da hier der laufende Prozess unterbrochen wird und er damit in seiner Prioritätsstufe verbleiben darf.
In allen drei Fällen wird anschließend die Funktion `scheduler()` aufgerufen.

Zuerst wird geprüft, ob der aktuell der CPU zugeteilte Prozess der Idle-Prozess ist und sich sonst kein Prozess in der Ready-Queue befindet. Dann kann sofort zur `end`-Sprungmarke gesprungen werden.

52 ⟨*felsner:timerhandler:scheduleraufruf* 52⟩≡

```
if (system_ticks % 1 == 0) {
 quantum_ticks++;
 if ((thread_table[current_task].tid == 1) &&
 (thread_table[thread_table[HEADOFREADYLIST].next].tid == TAILOFREADYLIST)){
  goto end;
 }
 if ((thread_table[current_task].priokey >
 thread_table[thread_table[HEADOFREADYLIST].next].priokey) &&
 (quantum_ticks > quantum)){
  if (scheduler_is_active){
   if ((current_task > 1) && (thread_table[current_task].priokey > 2)){
    thread_table[current_task].priokey--;
    scheduler(r);
    goto end;
   }
  }
 }
 if ((thread_table[current_task].priokey ==
 thread_table[thread_table[HEADOFREADYLIST].next].priokey) &&
 (quantum_ticks > quantum)){
  if (scheduler_is_active){
   if ((current_task > 1) &&
   (thread_table[current_task].priokey > 2)){
    thread_table[current_task].priokey--;
    scheduler(r);
    goto end;
   }
  }
 }
 if (thread_table[current_task].priokey
 < thread_table[thread_table[HEADOFREADYLIST].next].priokey){
  scheduler(r);
  goto end;
 }
 end:;
}

if (system_ticks % quantum == 0) {
 ⟨felsner:timerhandler:originalSchedAnzeige 53a⟩
}
```

Uses HEADOFREADYLIST 43c, scheduler 48b, and TAILOFREADYLIST 43c.

Jeden quantum-ten Timer-Aufruf wird die Aktivitätsanzeige aktualisiert. Die Realisierung der Anzeige ist bereits Bestandteil von ULIX.

53a ⟨*felsner:timerhandler:originalSchedAnzeige* 53a⟩≡ (52)

```
uint videoaddress = VIDEORAM + 79*2;
*((char*)videoaddress) = sched_chars[sched_c];
sched_c++; sched_c %= 4;
```

5.3.7. Dynamische Quanten

Die globale Variable quantum_ticks wird bei jedem ersten timer_handler-Aufruf um 1 erhöht. So ist es möglich, dynamische Quanten zu unterstützen. Sobald quantum_ticks einen größeren Wert als die Variable quantum besitzt, ist sichergestellt, dass der gerade rechnende Prozess sein Quantum aufgebraucht hat. Da in der Funktion scheduler() die Variable quantum_ticks auf 0 zurückgesetzt wird, fängt damit das Spiel von vorne an.

53b ⟨*kernel declarations* 43b⟩+≡ ◁47c 53d▷

```
uint quantum_ticks = 0;
```

53c ⟨*Felsner:Scheduler:Resetquantumticks* 53c⟩≡ (48b)

```
quantum_ticks = 0;
```

Die globale Variable quantum enthält den Wert des Quantums von current_task. Ein neuer Prozess erhält ein Quantum auf Basis seiner Priorität zugewiesen. So erhält ein Prozess, der eine Priorität von 99 hat, ein Quantum von 10. Ein Prozess, der beispielsweise eine Priorität von 20 besitzt darf maximal drei Quanten lang laufen (siehe Tabelle 5.1).

Tabelle 5.1.: Zuordnung Prozess-Prioritäten zu Quanten

Prioritätsstufen	**Quantum**
2-9	1
10-19	2
20-29	3
30-39	4
40-49	5
50-59	6
60-69	7
70-79	8
80-89	9
90-100	10

53d ⟨*kernel declarations* 43b⟩+≡ ◁53b 54a▷

```
int quantum = 1;
```

53e ⟨*Felsner:Scheduler:SetQuantum* 53e⟩≡ (48b)

```
quantum = ((thread_table[tid].priokey +10) / 10);

if (quantum < 1) quantum = 1;
if (quantum > 10) quantum = 10;
```

5.3.8. System Call setpriority()

`syscall_setpriority()` erhält beim Aufruf zum einen die `tid` des Prozesses im Register `ecx` und zum anderen die Priorität, die diesem zugewiesen werden soll, im Register `ebx`. Nachdem die Zuweisung der Priorität an den Prozess erfolgt, muss dieser, falls er sich im Status `TSTATE_Ready` befindet, aus der Ready-Queue entfernt und neu einsortiert werden. Am Ende wird `scheduler()` aufgerufen, da geprüft werden muss, ob dieser Prozess durch die Prioritätsveränderung jetzt laufen oder eventuell auch nicht mehr laufen darf.

54a ⟨*kernel declarations* 43b⟩+≡ ◁53d 55a▷

```
#define SCHED_SRC_SETPRIORITY 5
```

Defines:
`SCHED_SRC_SETPRIORITY`, used in chunk 54c.

54b ⟨*initialize syscalls* 54b⟩≡ 54d▷

```
insert_syscall (__NR_setpriority,  syscall_setpriority);
```

Uses `syscall_setpriority` 54c.

54c ⟨*kernel functions* 45a⟩+≡ ◁48b 54e▷

```
void syscall_setpriority (struct regs *r) {
 //tid =  ecx
 //prio=  ebx
 thread_table[(int)r->ecx].priokey = (int)r->ebx;

 if (thread_table[(int)r->ecx].state == TSTATE_READY) {
  remove_from_ready_queue ((int)r->ecx);
  insert((int)r->ecx, (int)r->ebx);
 }
 scheduler (r, SCHED_SRC_SETPRIORITY);
 return;
};
```

Defines:
`syscall_setpriority`, used in chunk 54b.
Uses `insert` 45a, `SCHED_SRC_SETPRIORITY` 54a, and `scheduler` 48b.

5.3.9. System Call getpriority()

Dieser System Call ermöglicht es, aus dem User-Mode die Priorität eines Prozesses auszulesen. Die Funktion erhält beim Aufruf die `tid` des Prozesses im Register `ebx`. Der Wert von `priokey` dieses Prozesses wird in das Register `eax` geschrieben und anschließend aus der Funktion gesprungen.

54d ⟨*initialize syscalls* 54b⟩+≡ ◁54b 55b▷

```
insert_syscall (__NR_getpriority,  syscall_getpriority);
```

Uses `syscall_getpriority` 54e.

54e ⟨*kernel functions* 45a⟩+≡ ◁54c 55c▷

```
void syscall_getpriority(struct regs *r) {
 //tid =(int)r->ebx
 r->eax = thread_table[(int)r->ebx].priokey;
```

```
 return;
};
```

Defines:
syscall_getpriority, used in chunk 54d.

5.3.10. System Call nice()

Der System Call `syscall_nice()` ändert die Priorität eines Prozesses relativ zu seinem aktuellen Prioritätswert. Umso höher der Nice-Wert, umso *netter* ist er zu anderen Prozessen. So wirkt sich ein hoher Nice-Wert negativ auf seine Priorität aus und andersherum. Der System Call `syscall_nice` ist nicht wie der Nice-Befehl in Unix konzipert. So ist es bei dieser Variante möglich die Priorität aller regulärer Prozesse zu ändern. Die Funktion erhält hierfür beim Aufruf die `tid` des Prozesses im Register `ecx` und den Nice-Wert `nicelevel`, der diesem zugewiesen werden soll im Register `ebx`. Der Wertebereich reicht von -5 bis +5. Dieser Wert kann experimentell verändert werden. Ist der Nice-Wert oder die neue Priorität außerhalb des gültigen Bereiches, wird eine Fehlermeldung ausgegeben und aus der Funktion gesprungen. Ist alles in Ordnung, erfolgt die Zuweisung der neuen Priorität an den Prozess. Falls er sich im Status `TSTATE_READY` befindet wird aus der Ready-Queue entfernt und neu einsortiert. Wie auch bei `setpriority()` wird am Ende `scheduler()` aufgerufen.

55a ⟨*kernel declarations* 43b⟩+≡ ◁54a

```
#define SCHED_SRC_NICE 4
```

Defines:
SCHED_SRC_NICE, used in chunk 55c.

55b ⟨*initialize syscalls* 54b⟩+≡ ◁54d

```
insert_syscall (__NR_nice,  syscall_nice);
```

Uses syscall_nice 55c.

55c ⟨*kernel functions* 45a⟩+≡ ◁54e

```
void syscall_nice (struct regs *r) {
 //nicelevel =(int)r->ebx
 //tid=(int)r->ecx
 int newprio;
 int nicelevel = (int)r->ebx;
 if ((nicelevel < (-5)) || (nicelevel > (5))){
  debug_printf ("Nice-Level ausserhalb Bereich. Nicelevel: %d \n",nicelevel);
  return;
 }
 newprio = (thread_table[(int)r->ecx].priokey - nicelevel);
 if ((newprio < MINPRIO) || (newprio > MAXPRIO)){
  debug_printf ("Prio ausserhalb Bereich Newprio: %d \n",newprio);
  return;
 }
 thread_table[(int)r->ecx].priokey = newprio;
 if (thread_table[(int)r->ecx].state == TSTATE_READY) {
  remove_from_ready_queue ((int)r->ecx);
```

```
    insert((int)r->ecx, newprio);
   }
   scheduler (r, SCHED_SRC_NICE);
   return;
  };
```

Defines:

`syscall_nice`, used in chunk 55b.

Uses `insert` 45a, `MAXPRIO` 43b, `MINPRIO` 43b, `SCHED_SRC_NICE` 55a, and `scheduler` 48b.

6. Evaluation

Nach Abschluss der Implementierung des Verfahrens widmet sich dieses Kapitel der Evaluation des Schedulers. Zuerst werden die unterschiedlichen Phasen des Projektes und die in diesen Phassen angewendeten Testmethoden vorgestellt. Anschließend werden die Testfälle beschrieben und bewertet.

6.1. Testumgebung

Die Tests werden mit Debian 11 durchgeführt. ULIX wird innerhalb einer *Qemu*-Virtualisierungsumgebung getestet. Der Qemu-PC-Emulator ist in Version 0.12.5 installiert.

6.2. Testphasen

Im Laufe der Entwicklung wurden mehrere Phasen durchlaufen (siehe die chronologische Abfolge der Implementierung aus Kapitel 5.1), die unterschiedliche Testverfahren erforderlich machten. So wurde während der Implementation mit Regressionstests gearbeitet. Nach Fertigstellung der Implementation wurde ein Systemtest durchgeführt.

6.2.1. Regressionstest

Die Erstellung des Simulators zu Beginn der Arbeit und die anschließende Integration erfolgten in mehreren Inkrementen. Jedes Inkrement wurde integriert und getestet. Hierzu wurde mit Regressionstests gearbeitet. Das Ziel eines Regressionstests ist es nachzuweisen, dass durch eine Modifikationen am Programm-Code keine unerwünschten Auswirkungen der Funktionalität der mit dieser Änderung in Verbindung stehenden Teile des Systems erfolgt sind (vgl. Liggesmeyer, 2009, S. 192). Ein Regressionstest besteht aus der Wiederholung von bereits durchgeführten Testläufen. Es werden identische Eingaben durchgeführt und die Ausgaben mit der Vorläuferversion verglichen. Falls keine Unterschiede auftreten oder die Unterschiede gewünscht sind, ist der Regressionstest erfolgreich absolviert (vgl. Liggesmeyer, 2009, S. 192).

Die Regressionstests wurden teils *manuell*, teils *automatisiert* durchgeführt (vgl. Liggesmeyer, 2009, S. 193). *Manuell* bedeutet, dass beispielsweise ein oder mehrere Prozesse mittels Testprogrammen in der Shell gestartet werden und das Verhalten anhand der Ausgabe am Bildschirm ausgewertet wird. Hierzu wird der Code an relevanten Stellen mittels `printf`-Kommandos instrumentiert. Eine *Automatisierung* der Regressionstests erfolgt mittels Testprogrammen, welche das Ergebnis eines Tests selbst validieren und auswerten. Mit Zunahme der Funktionalität des Scheduling-Verfahrens wurden nach und nach Testprogramme entwickelt bzw. weiterentwickelt.

6.2.2. Systemtest

Beim abschließenden Systemtest wird das Betriebssystem gegen die gesamten Anforderungen getestet. Im Gegensatz zum Regressionstest wird beim Systemtest gegen die Spezifikation (Kapitel 4.5) getestet. D. h., bei der Durchführung dieser Testfälle entstehen Reaktionen der Software. Die Korrektheit dieser Reaktionen wird anhand der Spezifikation beurteilt. Das Testende ist dann erreicht, wenn die Spezifikation mit Testfällen vollständig abgedeckt ist (vgl. Liggesmeyer, 2009, S. 50).

6.3. Fallstudien

Die Basis der Fallstudien bildet die Spezifikation aus Kapitel 4.5. Die in ihr definierten Anforderungen werden in Testfälle übersetzt. In jedem Testfall wird zu Beginn beschrieben, welche Funktionalität getestet wird und wie der jeweilige Test durchgeführt wird. Anschließend wird die Bildschirmausgabe dargestellt. Hierzu wurden, wie bereits erwähnt, an relevanten Stellen `printf`-Kommandos im Code verwendet.

6.3.1. Fallstudie zu Anforderung 1a:

Beschreibung: Die Priorität eines Prozesses soll vom User-Mode aus abgefragt werden können. Dies ist durch das Testprogramm `TESTgetpriority()` (siehe Seite 69) realisiert. Die Ausführung innerhalb der ersten erzeugten Shell liefert folgende Ausgabe:

```
Prioritaet des aktuellen Prozesses. Prozess ID:2, Prioritaet:50
```

Beurteilung: Der Prioritätswert wurde korrekt ausgelesen.

Ergebnis: Test erfolgreich.

6.3.2. Fallstudie zu Anforderung 1b

Beschreibung: Die Priorität eines Prozesses soll vom User-Mode aus absolut festlegt werden können. Dies soll sowohl für den laufenden als auch für rechenbereite Prozesse möglich sein. Dies ist durch das Testprogramm `TESTsetpriority()` (siehe Seite 69) realisiert. Die Ausführung liefert folgende Ausgabe:

```
Prioritaet des aktuellen Prozesses. Prozess ID:2, Prioritaet:49
Prioritaet wird auf 40 gesetzt
Prioritaet des aktuellen Prozesses. Prozess ID:2, Prioritaet:40
```

Beurteilung: Die Priorität wurde auf den vorgegebenen Wert angepasst.

Ergebnis: Test erfolgreich.

6.3.3. Fallstudie zu Anforderung 1c

Beschreibung: Es soll eine relative Änderung der Priorität möglich sein. Dies soll sowohl für den laufenden als auch für rechenbereite Prozesse möglich sein. Dies ist durch das Testprogramm `TESTnice()` (siehe Seite 69) realisiert. Die Ausführung liefert folgende Ausgabe:

```
Prioritaet des aktuellen Prozesses. Prozess ID:2, Prioritaet:40
nice-Wert -5 wird gesetzt
Prioritaet des aktuellen Prozesses. Prozess ID:2, Prioritaet:45
nice-Wert +3 wird gesetzt
Prioritaet des aktuellen Prozesses. Prozess ID:2, Prioritaet:42
```

Beurteilung: Der Aufruf der `nice`-Funktion mit Nice-Wert -5 hatte eine Änderung der Priorität von 40 auf 45 zur Folge. Der Aufruf der `nice`-Funktion mit Nice-Wert +3 hatte eine Änderung der Priorität von 45 auf 42 zur Folge.

Ergebnis: Test erfolgreich.

6.3.4. Fallstudie zu Anforderung 2

Neue Prozesse bekommen die Standardpriorität 50. Die Funktion `r5` (siehe Seite 72) liest die Priorität der ersten erzeugten Shell (läuft als Prozess 2) aus:

```
Prioritaet des aktuellen Prozesses. Prozess ID:5, Prioritaet:50
```

Beurteilung: Die Priorität dieser Shell hat einen Wert von 50.

Ergebnis: Test erfolgreich.

6.3.5. Fallstudie zu Anforderung 3

Beschreibung: Ein Elternprozess vererbt seine Priorität an seine Kindprozesse. Die Shell (4) läuft im Prozess (6) mit der Priorität 70. Die Funktion `lproc()`(siehe Seite 71) wird gestartet. Sie erzeugt einen CPU-lastigen Prozess, der eine längere Zeit läuft. Die Ausführung liefert folgende Ausgabe:

```
*timer_handler:((p curr>next)&q abg).pid:7,prio:70,sticks:2761,quantum:8
*timer_handler:((p curr>next)&q abg).pid:7,prio:69,sticks:2769,quantum:7
*timer_handler:((p curr>next)&q abg).pid:7,prio:68,sticks:2777,quantum:7
...
```

Beurteilung: Der neue Prozess hat als Vaterprozess die Shell (4). Er hat die Priorität der Shell geerbt.

Ergebnis: Test erfolgreich.

6.3.6. Fallstudie zu Anforderung 4

Beschreibung: Prozesse mit gleicher Priorität werden im Round-Robin-Verfahren abgearbeitet. Die Fallstudie erfolgt anhand zweier CPU-lastiger Prozesse. Beide werden mit Priorität 50 gestartet. Diese müssen in Round-Robin-Verfahren laufen. Folgendes Ergebnis liefert der Aufruf von der Funktion `r2()` (siehe Seite 71):

```
*pid=7, fork() returned: 0 time:1352 prio: 50
timer_handler:((p curr=next)&q abg).pid:7,prio:50,sticks:68 quantum:6
*pid=8, fork() returned: 0 time:1352 prio: 50
timer_handler:((p curr>next)&q abg).pid:8,prio:50,sticks:75,quantum:6
*timer_handler:((p curr=next)&q abg).pid:7,prio:49,sticks:81,quantum:5
*timer_handler:((p curr>next)&q abg).pid:8,prio:49,sticks:87,quantum:5
*timer_handler:((p curr=next)&q abg).pid:7,prio:48,sticks:93,quantum:5
*timer_handler:((p curr>next)&q abg).pid:8,prio:48,sticks:99,quantum:5
...
*timer_handler:((p curr=next)&q abg).pid:7,prio:38,sticks:210,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:38,sticks:215,quantum:4
*timer_handler:((p curr=next)&q abg).pid:7,prio:37,sticks:220,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:37,sticks:225,quantum:4
*...
```

Beurteilung: Die beiden Prozesse wechseln sich nach Ablauf ihres Quantums ab. Dies ist aus dem Wert *sticks*, der die Variable `system_ticks` darstellt, ersichtlich.

Ergebnis: Test erfolgreich.

6.3.7. Fallstudie zu Anforderung 5

Beschreibung: Sobald sich die Priorität eines Prozesses ändert, muss geprüft werden, welcher Prozess laufen darf. Dies geschieht durch die System Calls `setpriority()` und `nice()`. Hier wird, nachdem die Priorität eines Prozesses verändert wird, stets die Funktion `scheduler()` aufgerufen. Die Fallstudie baut auf den vorherigen Testfall auf. Die beiden Prozesse laufen im Round-Robin-Verfahren. Vom Benutzer wird, sobald die Priorität der Prozesse auf 36 abgesunken ist, die Priorität des zweiten Prozesses mittels `setpriority()` auf 40 erhöht. Dies geschieht mit der Funktion `r3()` (siehe Seite 72). Jetzt muss sofort der zweite Prozess alleine laufen, bis seine Priorität wieder so hoch wie die des ersten Prozesses ist. Dann werden die beiden Prozesse wieder im Round-Robin-Verfahren behandelt. Die Ausführung liefert folgende Ausgabe:

```
...
*timer_handler:((p curr=next)&q abg).pid:7,prio:38,sticks:215,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:38,sticks:220,quantum:4
*timer_handler:((p curr=next)&q abg).pid:7,prio:37,sticks:225,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:37,sticks:230,quantum:4
*timer_handler:((p curr=next)&q abg).pid:7,prio:36,sticks:235,quantum:4
*timer_handler:(p curr<next).pid:8,prio:36,sticks:239,quantum:4
*
```

An dieser Stelle erfolgt der Aufruf der Funktion `r3()`.

```
*Prioritaet des aktuellen Prozesses. Prozess ID:2, Prioritaet:50
*timer_handler:((p curr>next)&q abg).pid:8,prio:40,sticks:245,quantum:5
```

```
*timer_handler:((p curr>next)&q abg).pid:8,prio:39,sticks:250,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:38,sticks:255,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:37,sticks:260,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:36,sticks:265,quantum:4
*timer_handler:((p curr=next)&q abg).pid:7,prio:35,sticks:270,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:35,sticks:275,quantum:4
*timer_handler:((p curr=next)&q abg).pid:7,prio:34,sticks:280,quantum:4
*timer_handler:((p curr>next)&q abg).pid:8,prio:34,sticks:285,quantum:4
...
```

Beurteilung: Nach Aufruf der Funktion `r3` wechselt die Ausführung sofort zum Prozess 8. Nachdem die Priorität von Prozess 8 auf den Wert 36 abgesunken ist, wechseln sich die Prozesse in Round-Robin-Verfahren wieder ab.

Ergebnis: Test erfolgreich.

6.3.8. Fallstudie zu Anforderung 6

Beschreibung: Ein langer Prozess verbraucht sein Quantum. Jedes Mal wenn ein Prozess seine Quantum verbraucht, wird seine Priorität um 1 herabgestuft. Dies wurde anhand des letzten Testfalls geprüft.
Beurteilung: Jedes Mal wenn ein Prozess sein Quantum verbraucht hat, wird sein Prioritätswert um eine Stufe vermindert.

Ergebnis: Test erfolgreich.

6.3.9. Fallstudie zu Anforderung 7

Beschreibung: Wird ein Prozess rechenbereit, der eine höhere Priorität besitzt als der gerade rechnende Prozess, muss dem gerade rechnenden Prozess die CPU entzogen werden. Hierzu wird mit über die Funktion `p40` ein Prozess mit Priorität 40 gestartet. Anschließend wird über die Funktion `r4` (siehe Seite 72) ein Prozess gestartet, der eine Priorität von 99 besitzt.

```
*Prioritaet des aktuellen Prozesses. Prozess ID:7, Prioritaet:40
timer_handler:(p curr<next). pid:7, prio:40, sticks:181 quantum: 5
*rsyscall_yield(): 2, prio: 50, systemticks: 181 quantum: 6
*timer_handler:(p curr<next). pid:7, prio:40, sticks:183 quantum: 5
*4syscall_yield(): 2, prio: 50, systemticks: 183 quantum: 6
*timer_handler:(p curr<next). pid:7, prio:40, sticks:189 quantum: 5
*
```

An dieser Stelle erfolgt der Aufruf von `r4()`.

```
*timer_handler:((p curr>next)&q abg).pid:8,prio:99,sticks:200,quantum:10
*timer_handler:((p curr>next)&q abg).pid:8,prio:98,sticks:211,quantum:10
*timer_handler:((p curr>next)&q abg).pid:8,prio:97,sticks:222,quantum:10
```

```
*timer_handler:((p curr>next)&q abg).pid:8,prio:96,sticks:233,quantum:10
*timer_handler:((p curr>next)&q abg).pid:8,prio:95,sticks:244,quantum:10
...
```

Beurteilung: Sobald der Prozess mit Priorität 99 gestartet wird, wird dem aktuell laufenden Prozess mit Priorität 40 die CPU entzogen.

Ergebnis: Test erfolgreich.

6.3.10. Fallstudie zu Anforderung 8

Beschreibung: Prozesse mit hoher Priorität erhalten das größte Quantum. Prozesse mit niedriger Priorität erhalten das kleinste Quantum. Prozesse der Priorität 2–9 bekommen ein Quantum von einem `system_tick` zugewiesen, Prozesse der Priorität 10–19 ein Quantum von zwei `system_ticks`. Dieses Muster geht weiter bis Priorität 99. Prozesse mit Priorität 90–99 erhalten ein Quantum von 10 `system_ticks`. Dies wird mit der Funktion `r4()` getestet: Hier wird ein langer, CPU-intensiver Prozess ausgeführt. Das Ergebnis des Tests wurde auf relevante Stellen begrenzt:

```
**timer_handler:((p curr>next)&q abg).pid:7,prio:99,sticks:53,quantum:10
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:90,sticks:152,quantum:10
*timer_handler:((p curr>next)&q abg).pid:7,prio:89,sticks:162,quantum:9
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:80,sticks:252,quantum:9
*timer_handler:((p curr>next)&q abg).pid:7,prio:79,sticks:261,quantum:8
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:70,sticks:342,quantum:8
*timer_handler:((p curr>next)&q abg).pid:7,prio:69,sticks:350,quantum:7
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:60,sticks:422,quantum:7
*timer_handler:((p curr>next)&q abg).pid:7,prio:59,sticks:429,quantum:6
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:50,sticks:492,quantum:6
*timer_handler:((p curr>next)&q abg).pid:7,prio:49,sticks:498,quantum:5
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:40,sticks:552,quantum:5
*timer_handler:((p curr>next)&q abg).pid:7,prio:39,sticks:557,quantum:4
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:30,sticks:602,quantum:4
*timer_handler:((p curr>next)&q abg).pid:7,prio:29,sticks:606,quantum:3
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:20,sticks:642,quantum:3
*timer_handler:((p curr>next)&q abg).pid:7,prio:19,sticks:645,quantum:2
...
```

```
*timer_handler:((p curr>next)&q abg).pid:7,prio:10,sticks:672,quantum:2
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:688,quantum:1
...
```

Beurteilung: Die Quanten werden korrekt auf Basis der Priorität vergeben. Der Prozess hat sein jeweils zugewiesenes Quantum korrekt verbraucht.

Ergebnis: Test erfolgreich.

6.3.11. Fallstudie zu Anforderung 9

Die niedrigste Prioritätsstufe, in die ein Prozess herabgestuft werden kann, beträgt 2. Dies kann anhand des vorhergehenden Testfalls beurteilt werden:

```
...
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:690,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:692,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:694,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:696,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:698,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:700,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:702,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:704,quantum:1
*timer_handler:((p curr>next)&q abg).pid:7,prio:2,sticks:706,quantum:1
...
```

Beurteilung: Sobald ein Prozess auf Prioritätsstufe 2 abgesunken ist, vermindert sich seine Prioritätsstufe nicht mehr. Sie bleibt anschließend dauerhaft auf 2.

Ergebnis: Test erfolgreich.

6.3.12. Fallstudie zu Anforderung 10

Beschreibung: Ein I/O-lastiger Prozess gibt die CPU ab, bevor sein Quantum abgelaufen ist. In diesem Fall ändert sich seine Priorität nicht. Hierzu gibt es eine Funktion, die einen Prozess erzeugt. Dieser rechnet einige Zeit. Aber nur solange, dass sein Quantum nicht verbraucht wird. Dann gibt er die CPU freiwillig mittels `yield()` auf.

```
*syscall_yield():2,prio:50,systemticks:98,quantum:6
*syscall_yield():2,prio:50,systemticks:98,quantum:6
...
*syscall_yield():2,prio:50,systemticks:102,quantum:6
*syscall_yield():2,prio:50,systemticks:103,quantum:6
...
*syscall_yield():2,prio:50,systemticks:106,quantum:6
*syscall_yield():2,prio:50,systemticks:106,quantum:6
```

```
...
*syscall_yield():2,prio:50,systemticks:112,quantum:6
*syscall_yield():2,prio:50,systemticks:112,quantum:6
...
*syscall_yield():2,prio:50,systemticks:118,quantum:6
*syscall_yield():2,prio:50,systemticks:119,quantum:6
...
*syscall_yield():2,prio:50,systemticks:170,quantum:6
...
```

Beurteilung: Der Prozess gibt jedes Mal die CPU freiwillig ab. Seine Priorität bleibt konstant bei der ursprünglichen Priorität.

Ergebnis: Test erfolgreich.

7. Schluss

Abschließend wird in diesem Kapitel die Arbeit zusammengefasst und ein Fazit gezogen. Es folgen die Kritik an der Arbeit und ein Ausblick für die weitere Forschung.

7.1. Zusammenfassung

Zu Beginn dieser Arbeit bestand die Herausforderung, den bestehenden Ulix-Code und in diesem Zusammenhang die zukünftige Entwicklung von Ulix zu verstehen. Es folgte die Auseinandersetzung mit der Thematik des Schedulings. Hierzu wurden die Grundlagen, sowie moderne und klassische Algorithmen, die für die verschiedensten Anwendungen ausgelegt waren, beleuchtet. Es wurden Lehrbetriebssysteme und für den produktiven Einsatz geeignete Betriebssysteme untersucht. Eine weitere Herausforderung bestand in der Umsetzung der Implementation mittels *Literate Programming*. Nach der Einarbeitung in diese Themen erfolgte die Implementation.

Anfangs war der Entwicklungsstand von Ulix noch nicht so weit, dass ein komplexer Scheduler hätte implementiert werden können. Es gab zu Beginn keine `start_program_from_disk`-Funktion. Kontextwechsel waren nicht möglich, und Befehle wie `wait()` oder `yield()` funktionierten noch nicht. In Versionsstand Ulix 0.06 gab es die Funktion `start_program_from_disk`. Jetzt war auch möglich, Testprogramme im User-Mode ablaufen zu lassen. Kontextwechsel-Funktionalität war jetzt vorhanden, lief aber noch nicht stabil. In diesem Entwicklungsstand wurde ein statisches Prioritätsscheduling-Verfahren durch den Autor implementiert. Somit konnte die Basis für weitere komplexere Algorithmen geschaffen werden. Mit Ulix-Version 0.07 waren die Funktionen `wait()` und `yield()` funktionstüchtig und Kontextwechsel liefen stabil. Jetzt war es möglich, ein MLFQ-Verfahren zu implementieren.

7.2. Fazit

Im Rahmen der in dieser Arbeit vorgenommenen Untersuchung wurde kein Betriebssystem bzw. keine Betriebssystemkomponente gefunden, die mit der Programmiermethode *Literate Programming* umgesetzt ist. Dies bestätigt sowohl die Einzigartigkeit dieser Arbeit als auch die des gesamten Ulix-Systems. Die Umsetzung des Verfahrens in dieser Arbeit mittels Literate Programming erwies sich als sehr nützlich. Man wird dadurch gezwungen, alle Gedanken zum Source-Code aufzuschreiben. Mit zunehmendem Umfang des Source-Codes hilft das auch nach längerer Zeit den Code noch zu verstehen.

Jedes der untersuchten *Lehrbetriebssysteme* beschäftigt sich intensiv mit dem Thema Scheduling. Alle, mit Ausnahme von xv6, arbeiten mit prioritätenbasierten Scheduling-Verfahren. In xv6 und Minix 2 sind statische Prioritätsscheduler bzw. ML-Scheduler im Einsatz. Während in Minix 3 bereits ein MLFQ-Verfahren eingesetzt wird, sind in PintOS und GeekOS die Funktionen hierfür vorbereitet und sollen von Studenten ausgearbeitet werden.

Die *Betriebssysteme* Unix, FreeBSD 4.4, Linux bis Kernel-Version 2.6.22, Windows und FreeBSD ULE nutzen prioritätenbasierte MLFQ-Verfahren. Linux ab Kernel-Version 2.6.23 nutzt ein Fair-Queuing-Verfahren. Auch Solaris nutzt in der aktuellen Version standardmäßig einen MLFQ-Scheduler. Dieser unterstützt auch Fair-Share-Scheduling. So kann festgehalten werden, dass die Scheduling-Verfahren in den meisten aktuellen Betriebssystemen auf Algorithmen beruhen, die Anfang der 80iger Jahre entwickelt wurden. Eine neue Richtung, weg von Multilevel-Queue-Feedback-Verfahren, hin zu Proportional-Share-Verfahren, wurde beim CFS in der aktuellen Linux-Version eingeschlagen.

Keines der untersuchten Betriebssysteme unterstützt harte *Echtzeitanforderungen*. Hingegen werden weiche Echtzeitanforderungen von allen für den produktiven Einsatz geeigneten Betriebssystemen, außer Unix V4.3, unterstützt. Auch das Lehrbetriebssystem Minix 3 unterstützt weiche Echtzeitanforderungen.

Die Untersuchung ergab weiter, dass nur zwei Varianten zur Umsetzung von prioritätsbasiertem Scheduling im Einsatz sind.

Das Forschungsthema Scheduling scheint nach wie vor ein *aktives Forschungsgebiet* zu sein. So sind Themengebiete wie beispielsweise *Energieeffizientes Scheduling auf mobilen Endgeräten* (Min et al., 2012) ein aktiver Zweig. Auch das hier umgesetzte MLFQ-Verfahren wird aktuell immer noch optimiert (Hoganson, 2009). Tanenbaum sieht dieses Forschungsgebiet als eher *von den Forschern getrieben* als *von der Nachfrage gezogen* und behauptet: „Alles in allem sind Prozesse, Threads und Scheduling keine so heißen Forschungsthemen mehr, wie sie es einmal waren. Die Forschung ist weitergezogen.“ (Tanenbaum, 2009, S. 217). Diese Meinung kann aufgrund des Richtungswechsels von Linux Richtung Proportional-Share-Scheduling nur begrenzt geteilt werden.

7.3. Kritik an der Arbeit

Der realisierte Scheduler besitzt keine Funktionalität zur Vermeidung von *Prioritäten-Inversion*. Auch ist es möglich, dass lange Prozesse *verhungern* können, da keine Funktionalität implementiert wurde, die das verhindert. Auch *Gaming* ist bei der jetzt implementierten Variante möglich.
Der Fokus musste nicht auf Fehlerbehandlung gesetzt werden. Aus diesem Grund wurde im Kapitel 6.2.2 auf Äquivalenzklassenbildung verzichtet.
Der `nice`-Befehl kann für einen Prozess mehrfach aufgerufen, und somit kann die Begren-

zung des Wertebereiches umgangen werden.
Als weiterer Kritikpunkt kann angeführt werden, dass der Expertenfragebogen nur mit einem Experten durchgeführt wurde.

7.4. Ausblick

In dieser Arbeit wurde prioritätsbasiertes Scheduling in Ulix eingeführt. Dies kann als Basis für weitere prioritätenbasierte Verfahren, wie beispielsweise Guaranteed-Scheduling oder die komplexere Decay-usage-MLFQ-Variante, dienen.

Weiterer Forschungsbedarf ist vorhanden, um die im vorherigen Kapitel fehlenden Funktionalitäten zur Verhinderung von *Priority-Inversion*, *Starvation* (von langen Prozessen) und *Gaming* zu implementieren.

Eine Erweiterung des Scheduling-Verfahrens von Ulix könnte durch Mehrprozessor bzw. Mehrkernunterstützung erfolgen. Die Variante des in *FreeBSD ULE* oder *Linux O(1)* beleuchteteten Verfahrens als natürliche Erweiterung für Ulix umgesetzt werden.

A. Fragebogen der Expertenbefragung

Expertenbefragung im Rahmen der Bachelor-Thesis: Implementation eines CPU-Schedulers für das Lehrbetriebssystem Ulix

1. Welche Kriterien soll der Scheduler erfüllen (bitte ankreuzen):

Kriterium	Sehr wichtig	Wichtig	Unwichtig
1.Fairness*			
2.Balance*			X
3.Antwortzeit*		X	
4.Proportionalität*		X	
5.Niedrige Durchlaufzeit*			X
6.Unterstützung harter Echtzeitanforderungen			X
7.Unterstützung weicher Echtzeitanforderungen		X	
8.Vorhersagbarkeit*			
9.bewährte Algorithmen/Konzepte einsetzen	X		
10.Neuartigen Algorithmus entwickeln			X
11.Besonderer Fokus auf Fehlerbehandlung			X
12.Auf gute Didaktische Vermittelbarkeit achten	X		
13.Hohe Komplexität des Verfahrens			X
14.Niedrige Komplexität des Verfahrens		X	
15.Verfahren wird in kommerziellen BS eingesetzt	X		

*Kriterien nach Tanenbaum2009 S. 197

Anmerkungen zu den Auswahlkriterien von 1:

---

2. Gibt es darüber hinaus noch Anforderungen an den Scheduler?

`keine`---

B. Testprogramme

```
void alotwork(){
 int outer; int inner;
 for (outer=0; outer<200000; outer++) {
  for (inner=0; inner<40000; inner++) {
  getpid();
  }
 }
 return;
}
void littlework(){
 int outer;
 for (outer=0; outer<200; outer++) {
  getpid();
 }
 return;
}


void TESTgetpriority(){
 printf ("\nTESTgetpriority:\n");
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d, Prioritaet:%d\n"
 , getpid(), getpriority(getpid()));
 return;
}


void TESTsetpriority(){
 printf ("\nTESTsetpriority:\n");
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d, Prioritaet:%d\n"
 , getpid(), getpriority(getpid()));
 printf ("Prioritaet wird auf 40 gesetzt\n");
 setpriority(40, getpid());
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d, Prioritaet:%d\n"
 , getpid(), getpriority(getpid()));
 return;
}
```

```
void TESTnice(){
 printf ("\nTESTnice:\n");
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d, Prioritaet:%d\n"
 , getpid(), getpriority(getpid()));
 printf ("nice-Wert -5 wird gesetzt\n");
 nice(-5,getpid());
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d, Prioritaet:%d\n"
 , getpid(), getpriority(getpid()));
 printf ("nice-Wert +3 wird gesetzt\n");
 nice(3,getpid());
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d, Prioritaet:%d\n"
 , getpid(), getpriority(getpid()));
 return;
}
void TESThigherPriority(){
 printf ("\nTESThigherPriority:\n");
 printf ("Prozess wird gestartet. Prozess ID:%d, Prioritaet:%d\n"
 , getpid(), getpriority(getpid()));
 p20();
 return;
}


void p20(){
 int pid;
 if ((pid = fork()) < 0) {
  printf ("Error");
 }
 else if (pid == 0) {
  setpriority(20, getpid());
  printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d
  , Prioritaet:%d\n", getpid(), getpriority(getpid()));
  alotwork();
  exit(0);
 }
}


void p40(){
 int pid;
 if ((pid = fork()) < 0) {
  printf ("Error");
 }
 else if (pid == 0) {
  setpriority(40, getpid());
```

```
  printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d
  , Prioritaet:%d\n", getpid(), getpriority(getpid()));
  alotwork();
  exit(0);
 }
}

void r2 () {
 int pids[2];
 int i;
 int n = 2;
 int k;
 int p;
 for (i = 0; i < n; ++i) {
  if ((pids[i] = fork()) < 0) {
   printf ("error");
  } else if (pids[i] == 0)
  {
   printf ("pid=%d, fork() returned: %d time:%d prio: %d\n"
   , getpid(), pids[i], p, getpriority(getpid()));
   alotwork();
   exit(0);
  }
 }
}

void sproc(){
 int outer;
 for (outer=0; outer<200; outer++) {
  littlework();
  yield();
 }
 return;
}

void lproc(){
 int pid;
 if ((pid = fork()) < 0) {
  printf ("Error");
 }
 else if (pid == 0) {
  alotwork();
  exit(0);
```

```
 }
}

void r3(){
 setpriority(40, 8);
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d
 , Prioritaet:%d\n", getpid(), getpriority(getpid()));
 exit(0);
}

void r4(){
 int pid;
 if ((pid = fork()) < 0) {
  printf ("Error");
 }
 else if (pid == 0) {
  setpriority(99, getpid());
  alotwork();
  exit(0);
 }
}

void r5(){
 printf ("Prioritaet des aktuellen Prozesses. Prozess ID:%d
 , Prioritaet:%d\n", getpid(), getpriority(getpid()));
}
```

C. Simulator

```
#include <stdio.h>
#include <stdlib.h>
#define INITPRIO        20
#define EMPTY   -1
#define boolean unsigned int
#define true 1
#define false 0
#define null 0
#define MAX_THREADS 1024
#define TSTATE_READY     1
#define TSTATE_FORK      3
#define TSTATE_EXIT      4
#define TSTATE_WAITFOR   5
#define ENABLE_SCHEDULER scheduler_is_active = true
#define DISABLE_SCHEDULER scheduler_is_active = false
#define SCHED   1
#define HEADOFREADYLIST         0
#define TAILOFREADYLIST         30 // nach belieben
#define MAXPRIO                         101
#define MINPRIO                         0

typedef unsigned char         uint8_t;
typedef unsigned short       uint16_t;
typedef unsigned int         uint32_t;
typedef unsigned long long   uint64_t;
typedef unsigned char byte;
typedef unsigned int thread_id;
typedef unsigned short u16int;

typedef struct {
 thread_id tid;
 thread_id ppid;
 int state;
 int exitcode;
 int waitfor;
 thread_id next;
 thread_id prev;
```

```
 boolean used;
 short int fixstack;
 int priokey;
 char *name;
} TCB;

typedef struct {
 thread_id next;
 thread_id prev;
} blocked_queue;

void add_prio(char* n);
void assign(int cpuid);
void block(int cpuid, blocked_queue* q);
void deblock(thread_id t, blocked_queue* q);
void resign(int cpuid);
void retire(thread_id t);
volatile int current_task;
TCB thread_table[MAX_THREADS];
thread_id schedule();
void remove_from_ready_queue(); // Änderung in thread_id
void add_to_blocked_queue(thread_id t, blocked_queue* bq);
void remove_from_blocked_queue(thread_id t, blocked_queue* bq);
thread_id front_of_blocked_queue(blocked_queue bq);
int register_new_tcb (int as_id);

thread_id front_of_blocked_queue(blocked_queue bq) {
 return bq.next;
}

void add_to_blocked_queue(thread_id t, blocked_queue* bq) {
 thread_id last = bq->prev;
 bq->prev = t;
 thread_table[t].next = 0; // {\Tt{}t\nwendquote} is ‘‘last’’ thread
 thread_table[t].prev = last;
 if (last == 0) {
  bq->next = t;
 } else {
 thread_table[last].next = t;
 }
}

void remove_from_blocked_queue(thread_id t, blocked_queue* bq) {
```

```
 thread_id prev_thread = thread_table[t].prev;
 thread_id next_thread = thread_table[t].next;
 if (prev_thread == 0) {
   bq->next = next_thread;
 } else {
  thread_table[prev_thread].next = next_thread;
 }
 if (next_thread == 0) {
  bq->prev = prev_thread;
 } else {
  thread_table[next_thread].prev = prev_thread;
 }
}

char* state_names[6] = { "---", "READY", "---", "FORK", "EXIT", "WAIT4" };
void print_thread_table();
boolean used;
void scheduler ();
int scheduler_is_active = false;

int register_new_tcb () {
 boolean tcbfound = false;
 int tcbid;
 for ( tcbid=1; ((tcbid<1024) && (!tcbfound)); tcbid++ ) {
  if (thread_table[tcbid].used == false) {
   ltcbfound = true;
   break;
  }
 };
 if (!tcbfound) {
  return -1; // no free TCB!
 };
 thread_table[tcbid].used = true;
 return tcbid;
}

void create_init_process() {
 int tid = 1;
 int tcbid = 1;
 tid = tcbid = register_new_tcb() ;
 thread_table[tcbid].tid = tid;
 thread_table[tcbid].name = "init prozess";
 current_task = tid;
```

```
 insert(tcbid, 0, INITPRIO);
 ENABLE_SCHEDULER;
};

void remove_from_ready_queue(thread_id t) {
 thread_id prev_thread = thread_table[t].prev;
 thread_id next_thread = thread_table[t].next;
 thread_table[prev_thread].next = next_thread;
 thread_table[next_thread].prev = prev_thread;
 return(t);
}

thread_id getfirst(thread_id head){
 thread_id      proc;
 if ((proc=thread_table[head].next) < 1024){
  proc=thread_table[head].next;
  remove_from_ready_queue(proc);
  return(proc);
 }
 else
 {
  return(EMPTY);
 }
}

int     insert(thread_id proc, thread_id head, int key){
 thread_id      lnext;
 thread_id      lprev;
 lnext = thread_table[head].next;
 while (thread_table[lnext].priokey >= key)
  lnext = thread_table[lnext].next;
  thread_table[proc].next = lnext;
  thread_table[proc].prev = lprev = thread_table[lnext].prev;
  thread_table[proc].priokey  = key;
  thread_table[lprev].next = proc;
  thread_table[lnext].prev = proc;
  return;
}

void init_thread_table(){
 thread_id bereinigung;
 for (bereinigung = 1; bereinigung < 1024; bereinigung++) {
  thread_table[bereinigung].tid = 0;
```

```
  thread_table[bereinigung].priokey = 0;
 }

 thread_table[HEADOFREADYLIST].prev = TAILOFREADYLIST;
 thread_table[HEADOFREADYLIST].tid = HEADOFREADYLIST;
 thread_table[HEADOFREADYLIST].next = TAILOFREADYLIST;
 thread_table[HEADOFREADYLIST].priokey = MAXPRIO;

 thread_table[TAILOFREADYLIST].prev = HEADOFREADYLIST;
 thread_table[TAILOFREADYLIST].tid = TAILOFREADYLIST;
 thread_table[TAILOFREADYLIST].next = HEADOFREADYLIST;
 thread_table[TAILOFREADYLIST].priokey = MINPRIO;
 thread_table[TAILOFREADYLIST].used = true;
}

void Ausgabe_der_Testeinträge_sortiert_nach_TCB_ID(){
 int counterausgabe;
 printf("Ausgabe_der_Testeinträge_sortiert_nach_TCB_ID\n");
 printf("tcbid\tprev\tnext\tprio\n");
 for (counterausgabe = 0; counterausgabe < 50; counterausgabe++) {
  printf("%d\t%d\t%d\t%d\t%s\n",
  thread_table[counterausgabe].tid,
  thread_table[counterausgabe].prev,
  thread_table[counterausgabe].next,
  thread_table[counterausgabe].priokey,
  thread_table[counterausgabe].name);
 }
 printf("\n");
}

void Aufbau_der_Testeinträge(){
 int counteraufbau;
 thread_id free;
 int lprio;
 printf("Aufbau_der_Testeinträge\n");
 for (counteraufbau = 1; counteraufbau < 31; counteraufbau++) {
  free = register_new_tcb ();
  thread_table[free].tid = free;
  lprio = thread_table[free].priokey = 1 + ( rand() % 100 ) ;
  thread_table[free].name = "Prozess ";
  insert(free, 0, lprio);
  printf("Aufbau_der_Testeinträge%d\n", free);
  }
```

```
 printf("\n");
}

void Head_entfernen_und_ausgeben(){
 thread_id chosenprocess;
 int counterdequeue;
 thread_id fTCB;
 printf("Head_entfernen_und_ausgeben:\n");
 for (counterdequeue = 1; counterdequeue < 100; counterdequeue++) {
  chosenprocess = getfirst(0);
  printf("%d:\t%d\t%s\n", chosenprocess
  , thread_table[chosenprocess].priokey, thread_table[chosenprocess].name);
 }
 printf("\n");
}
void add_prio(char* lname){
 thread_id lfreeTCB;
 printf("Prozess_einfügen\n");
 lfreeTCB = register_new_tcb();
 thread_table[lfreeTCB].name = lname;
 thread_table[lfreeTCB].tid = lfreeTCB;
 insert(lfreeTCB, 0, 50);
}

void main(){
 init_thread_table();
 create_init_process();
 Aufbau_der_Testeinträge();
 Ausgabe_der_Testeinträge_sortiert_nach_TCB_ID();
 add_prio("neu1");
 add_prio("neu2");
 add_prio("neu3");
 add_prio("neu4");
 Ausgabe_der_Testeinträge_sortiert_nach_TCB_ID();
 Head_entfernen_und_ausgeben();
}
```

Identifier Index

EMPTY: 43d, 44a, 46c
HEADOFREADYLIST: 43c, 44a, 45a, 46c, 52
IDLEPRIO: 47a, 47b
insert: 44b, 44c, 45a, 49a, 54c, 55c
MAXPRIO: 43b, 44a, 55c
MINPRIO: 43b, 44a, 55c
SCHED_SRC_NICE: 55a, 55c
SCHED_SRC_SETPRIORITY: 54a, 54c
scheduler: 48b, 52, 54c, 55c
STANDARDPRIO: 47c, 48a
syscall_getpriority: 54d, 54e
syscall_nice: 55b, 55c
syscall_setpriority: 54b, 54c
TAILOFREADYLIST: 43c, 44a, 52
TSTATE_CURR: 49a, 49c, 49d

Literaturverzeichnis

Anderson, C. L. & Nguyen, M. (2005): *A survey of contemporary instructional operating systems for use in undergraduate courses.* In: J. Comput. Sci. Coll., **21**, 1: 183–190.

Arpaci-Dusseau, R. H. & Arpaci-Dusseau, A. C. (2012): *Scheduling The Multi-Level Feedback Queue.* http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf, abgerufen am: 12.12.2012.

Aviv, A. J., Mannino, V., Owlarn, T., Shannin, S., Xu, K. & Loo, B. T. (2012): *Experiences in teaching an educational user-level operating systems implementation project.* In: SIGOPS Oper. Syst. Rev., **46**, 2: 80–86.

Bovet, D. & Cesati, M. (2008): *Understanding the Linux Kernel.* O'Reilly Media.

Comer, D. (1984): *Operating System Design: The Xinu approach.* Operating System Design. Prentice-Hall.

Comer, D. (2011): *Operating System Design: The Xinu Approach Linksys Version.* Prentice-Hall software series. CRC Press.

Corbato, F. J., Merwin-Daggett, M. & Daley, R. C. (1962): *An experimental time-sharing system.* In: *Proceedings of the May 1-3, 1962, spring joint computer conference*, AIEE-IRE '62 (Spring), 335–344. ACM, New York, NY, USA.

Cox, R., Kaashoek, F. & Morris, R. (2012): *xv6 a simple, Unix-like teaching operating system.* http://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf, abgerufen am: 14.12.2012.

Davis, R. I. & Burns, A. (2011): *A survey of hard real-time scheduling for multiprocessor systems.* In: ACM Comput. Surv., **43**, 4: 35:1–35:44.

Epema, D. H. J. (1998): *Decay-usage scheduling in multiprocessors.* In: ACM Trans. Comput. Syst., **16**, 4: 367–415.

Glatz, E. (2006): *Betriebssysteme: Grundlagen, Konzepte, Systemprogrammierung.* Dpunkt.Verlag GmbH.

Goyal, P., Guo, X. & Vin, H. M. (1996): *A Hierarchical CPU Scheduler for Multimedia Operating Systems.* 107–121.

Hansen, P. B. (1973): *Operating system principles.* Prentice-Hall, Inc.

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004): *Design Science in Information Systems Research.* In: MIS Quarterly, **28**, 1: 75–105.

Hoganson, K. (2009): *Reducing MLFQ scheduling starvation with feedback and exponential averaging.* In: J. Comput. Sci. Coll., **25**, 2: 196–202.

Hovemeyer, D. (2001): *GeekOS: An Instructional Operating System for Real Hardware.* http://geekos.sourceforge.net/docs/geekos-paper.pdf, abgerufen am: 12.02.2013.

Hovemeyer, D., Hollingsworth, J. K. & Bhattacharjee, B. (2004): *Running on the bare metal with GeekOS.* In: *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, 315–319. ACM, New York, NY, USA.

Kay, J. & Lauder, P. (1988): *A fair share scheduler.* In: Commun. ACM, **31**, 1: 44–55.

Kleinrock, L. (1970): *A continuum of time-sharing scheduling algorithms.* In: *Proceedings of the May 5-7, 1970, spring joint computer conference*, AFIPS '70 (Spring), 453–458. ACM, New York, NY, USA.

Kleuker, S. (2010): *Grundkurs Software-Engineering mit UML.* Vieweg Verlag.

Knuth, D. E. (1984): *Literate Programming.* In: The Computer Journal, **27**, 2: 97–111.

Larmouth, J. (1975): *Scheduling for a share of the machine.* In: Software: Practice and Experience, **5**, 1: 29–49.

Lee, J., Easwaran, A. & Shin, I. (2012): *Laxity dynamics and LLF schedulability analysis on multiprocessor platforms.* In: Real-Time Syst., **48**, 6: 716–749.

Levin, R., Cohen, E., Corwin, W., Pollack, F. & Wulf, W. (1975): *Policy/mechanism separation in Hydra.* In: *Proceedings of the fifth ACM symposium on Operating systems principles*, SOSP 75, 132–140. ACM, New York, NY, USA.

Li, T., Baumberger, D. & Hahn, S. (2009): *Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin.* In: SIGPLAN Not., **44**, 4: 65–74.

Liggesmeyer, P. (2009): *Software-Qualität: Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag.

Liu, C. L. & Layland, J. W. (1973): *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* In: J. ACM, **20**, 1: 46–61.

Mandl, P. (2013): *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation.* Springer Fachmedien Wiesbaden.

Mauro, J. & McDougall, R. (2006): *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture.* Pearson Education.

McKusick, M. & Neville-Neil, G. (2004): *The Design and Implementation of the FreeBSD Operating System.* Pearson Education.

Microsoft (2012): *Scheduling.* http://msdn.microsoft.com/en-us/library/windows/desktop/ms685096%28v=vs.85%29.aspx, abgerufen am: 14.10.2012.

Min, A. W., Wang, R., Tsai, J., Ergin, M. A. & Tai, T.-Y. C. (2012): *Improving energy efficiency for mobile platforms by exploiting low-power sleep states.* In: *Proceedings of the 9th conference on Computing Frontiers*, CF '12, 133–142. ACM, New York, NY, USA.

Minix3 (2013): *Minix 3 Source Code.* http://www.minix3.org/download/index.html, abgerufen am: 14.02.2013.

Molnar, I. (2008): *This is the CFS Scheduler.* http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt, abgerufen am: 01.08.2013.

Nehmer, J. & Sturm, P. (1998): *Systemsoftware.* Dpunkt-Lehrbuch. dpunkt -Verlag für digitale Technologie.

Nieh, J., Vaill, C. & Zhong, H. (2001): *Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler.*

Pfaff, B., Romano, A. & Back, G. (2009): *The pintos instructional operating system kernel.* In: *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, 453–457. ACM, New York, NY, USA.

Saunders, C. (2005): *Journal Rankings page (edited by Carol Saunders).* http://ais.affiniscape.com/displaycommon.cfm?an=1&subarticlenbr=432, abgerufen am: 14.10.2012.

Silberschatz, A. (2010): *Operating System Concepts 8th Edition International Student Version with WileyPlus Set.* Wiley Plus Products Series. John Wiley & Sons Canada, Limited.

Smith, C. (1996): *Noweb file inclusion.* http://www.literateprogramming.com/best/nowebfi.html, abgerufen am: 05.05.2013.

Stallings, W. (2001): *Operating systems - internals and design principles (4th ed.).* Prentice Hall.

Suranauwarat, S. & Taniguchi, H. (2001): *The design, implementation and initial evaluation of an advanced knowledge-based process scheduler.* In: SIGOPS Oper. Syst. Rev., **35**, 4: 61–81.

Tanenbaum, A. S. (1990): *Betriebssysteme, Entwurf und Realisierung.* Carl Hanser.

Tanenbaum, A. S. (2009): *Moderne Betriebssysteme (3. Auflage).* Pearson Studium.

vhbonline (2011): *VHB-JOURQUAL 2.1 (2011) Verband der Hochschullehrer für Betriebswirtschaft e.V.* http://vhbonline.org/service/jourqual/vhb-jourqual-21-2011/, abgerufen am: 12.09.2012.

vom Brocke, J., Simons, A., Niehaves, B., Riemer, K., Plattfaut, R. & Cleven, A. (2009): *Reconstructing the Giant: On the Importance of Rigour in Documenting the Literature*

Search Process. In: *Proceedings of the European Conference on Information Systems (ECIS)*. Verona, Italy.

Waldspurger, C. A. & Weihl, W. E. (1994): *Lottery scheduling: flexible proportional-share resource management*. In: *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94. USENIX Association.

Waldspurger, C. A. & Weihl, W. E. (1995): *Stride Scheduling: Deterministic Proportional- Share Resource Management*. Techn. Ber., Cambridge, MA, USA.

Webster, J. & Watson, R. (2002): *Analyzing the past to prepare for the future: Writing a literature review*. In: MIS Quarterly, **26**, 2: 13–23.

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen hat. Ich erkläre mich damit einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

Erding, den 5. August 2013