Diplomarbeit

Look-Ahead Scheduling

Stefan Wächtler

18. Dezember 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter: Dipl.-Inf. Michael Roitzsch
Dipl.-Inf. Björn Döbel

Technische Universität Dresden
Fakultät Informatik

Dresden, 20.06.2012

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten: Wächtler, Stefan

Studiengang: Informatik

Immatrikulationsnummer: 3391319

Thema: *Look-Ahead Scheduling*

Scheduling (Einplanung) ist eine zentrale Tätigkeit in Rechnersystemen zur Verteilung von Ressourcen zwischen mehreren konkurrierenden Konsumenten. Besondere Aufmerksamkeit genießt dabei das Scheduling des Hauptprozessors (CPU), da dieser für Software die Schnittstelle zu allen weiteren Geräten im System darstellt.

Typische CPU-Scheduling-Strategien sind die faire Verteilung der CPU-Zeit zwischen allen bereiten Aktivitäten oder die Nutzung von statischen Prioritäten. Beide decken jedoch das Verhalten von Echtzeitanforderungen in interaktiven Desktop-Systemen nur unzureichend ab. Alternative Ansätze wie das am Lehrstuhl Betriebssysteme entwickelte ATLAS-Konzept (Auto-Training Look-Ahead Scheduler) setzen auf Deadlines (Zeitschranken) und Angaben der Abarbeitungszeit als zentrales Entscheidungsmerkmal.

Ziel dieser Arbeit ist die Implementierung und Evaluierung eines CPU-Schedulers auf Basis des ATLAS-Konzepts. Der Scheduler ist direkt in den Linux-Betriebssystemkern zu implementieren und bietet eine Schnittstelle an, welche die Anmeldung von CPU-Aufträgen (Jobs) unter der Angabe von Deadline und Abarbeitungszeit zulässt. Die Angabe der Abarbeitungszeit ist optional und der Scheduler soll diese nur als Schätzung betrachten. In der Arbeit wird ein unter diesen Randbedingungen geeignetes Systemverhalten definiert und durch eine entsprechende Scheduling-Strategie implementiert.

Arbeitslast des Schedulers sollen gewöhnliche Desktop-Anwendungen mit weichen Echtzeitanforderungen sein. Systeme mit harten Echtzeitanforderungen sind ausdrücklich ausgenommen. Die Implementierung des Schedulers kann sich auf ein Einprozessorsystem beschränken, in der schriftlichen Arbeit sollen Ansätze für eine Mehrprozessor-Umsetzung diskutiert werden.

Die Auswertung erfolgt mit Mikro-Benchmarks und ausgewählten Belastungstests. Je nach Zeitaufwand für die Implementierung des Schedulers ist hier in Absprache mit dem Betreuer ein Anwendungsszenario zu entwerfen und mit dessen Hilfe der Scheduler zu evaluieren.

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Michael Roitzsch
Institut:	Systemarchitektur
Beginn:	22. 06. 2012
Einzureichen:	21. 12. 2012

Unterschrift des betreuenden *Hochschullehrers*

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 18. Dezember 2012

Stefan Wächtler

Danksagung

Zu Beginn danke ich Herrn Professor Härtig dafür, dass er es mir ermöglicht hat, meine Diplomarbeit unter der Betreuung von Michael und Björn am Lehrstuhl für Betriebssysteme zu erarbeiten. Letzteren gebührt für die hervorragende Betreuung während der Arbeit ein ganz besonderes Dankeschön.

Ich möchte mich an dieser Stelle auch bei meinen Eltern Ingrid und Frank Wächtler bedanken. Ohne deren Verständnis, Hilfsbereitschaft und finanzielle Unterstützung wäre mein Studium an der Technischen Universität Dresden nur schwierig umsetzbar gewesen.

Bedanken möchte ich mich auch bei meinen Kommilitonen im Studentenlabor für die zahlreichen Diskussionen, die glücklicherweise auch von Zeit zu Zeit über den Horizont der Informatik hinaus reichten und für die notwendige Abwechslung gesorgt haben.

Für das Korrekturlesen der Arbeit bedanke ich mich besonders bei meinen Freunden Dr. Christian Ott und Melanie Titze.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Prozesse und Threads, Tasks und Jobs 3
2.2 Harte und weiche Echtzeitsysteme 4
2.3 Scheduler 5
2.4 Scheduler in Linux 7
2.4.1 Schichtenarchitektur 7
2.4.2 Scheduling-Klassen 8
2.4.3 Aufruf des Schedulers unter Linux 9
2.5 Scheduling-Strategien 10
2.5.1 Earliest Deadline First (EDF) 10
2.5.2 Latest Release Time (LRT) 10
2.5.3 Akkumulierte Slack-Zeit 11
2.5.4 Vergleich EDF und LRT 11
2.6 ATLAS 12
2.6.1 Komponenten von ATLAS 12
2.6.2 Primitive des ATLAS-Schedulers 13
2.6.3 Anwendungsbeispiele für ATLAS 14
2.7 Verwandte Arbeiten 15
2.7.1 Einplanung basierend auf Reservierungen 15
2.7.2 Einplanung basierend auf gerechter Rechenzeitverteilung 16
2.7.3 Vergleich mit ATLAS 17

3 Design **19**
3.1 Anforderungen/Annahmen des ATLAS-Schedulers 19
3.1.1 Anforderungen 19
3.1.2 Limitierungen innerhalb der Arbeit 20
3.2 Einordung des ATLAS-Schedulers 21
3.3 Scheduling-Strategie 22
3.3.1 LRT als grundlegende Scheduling-Strategie 22
3.3.2 Erster Ansatz einer Scheduling-Strategie 22
3.3.3 Lösungsansatz bei Unterschätzung der Ausführungszeit 24
3.3.4 Überschreitung der Ausführungszeit eines Jobs trotz Vorlauf . . 24
3.3.5 Blockierende Threads 25
3.4 Ablaufplan auf Basis von LRT 27
3.4.1 Bedeutung 27

3.4.2 Grundlagen . . . 28
3.4.3 Einfügen . . . 28
3.4.4 Aktualisieren . . . 29
3.4.5 Löschen . . . 30
3.5 Auswahl des nächsten Threads in ATLAS-LRT . . . 30
3.6 Überlastsituationen bei Job-Einplanungen im Ablaufplan . . . 32
3.6.1 Erkennung von Überlast . . . 32
3.6.2 Behandlung von Überlastsituationen . . . 32
3.7 Ansatz für Scheduling auf Mehrprozessorsystemen . . . 35
3.8 Absolute und relative Zeitangaben . . . 37
3.9 Verhalten bei `fork` beziehungsweise `clone` . . . 38

4 Implementierung **39**
4.1 Umsetzung der Primitive des ATLAS-Konzepts . . . 39
4.1.1 Betriebsystemaufrufe . . . 39
4.1.2 `/proc`-Dateisystem . . . 40
4.2 Implementierung neuer Scheduling-Klassen in Linux . . . 41
4.3 Wahl der Zeitbasis . . . 43
4.4 Lost-Wakeup-Problem bei der Implementierung von `next` . . . 44
4.5 Probleme bei der Implementierung . . . 46
4.5.1 Verwendung von printk innerhalb des Schedulers . . . 46
4.5.2 Verwendung von Timern innerhalb des Schedulers . . . 46

5 Evaluierung **49**
5.1 Testumgebung und -system . . . 49
5.2 Überprüfung der Funktionalität . . . 49
5.2.1 Linux Trace Toolkit – next generation . . . 50
5.2.2 Darstellung . . . 51
5.3 Mikrobenchmarks . . . 52
5.3.1 Vergleich von angeforderter und zugewiesener Rechenzeit . . . 53
5.3.2 Vergleich von zugewiesener und tatsächlicher Rechenzeit . . . 54
5.3.3 Kosten für die Einplanung eines Jobs . . . 56
5.3.4 Kosten für den Start eines neuen Jobs . . . 58
5.4 FFplay . . . 59
5.4.1 Funktionsweise des angepassten FFplays . . . 60
5.4.2 Durchführung der Messung . . . 61
5.4.3 Auswertung . . . 61

6 Zusammenfassung **67**

Literaturverzeichnis **69**

Abbildungsverzeichnis

2.1 Zustandsübergänge eines Threads 3
2.2 Schichten-Design der Scheduler innerhalb des Linux-Kerns 7
2.3 Beispielhafter Ablaufplan nach EDF und LRT 11
2.4 Komponenten von ATLAS 13

3.1 Einordnung des ATLAS-Schedulers in die Schichtenarchitektur von Schedulern des Linux-Kerns 21
3.2 Alternatives Schichten-Design von Schedulern des Linux-Kerns zur Umsetzung des ATLAS-Schedulers 23
3.3 Einteilung der Slack-Zeit in Perioden 23
3.4 Aufgestellter Ablaufplan gegenüber tatsächlichen Ablaufplan beim Blockieren eines Jobs 25
3.5 Anordnung der Scheduling-Klassen in der Schichtenarchitektur von Schedulern des Linux-Kerns 27
3.6 Veränderung des Ablaufplans durch Einfügen eines neuen Jobs 28
3.7 Aktualisierung des Ablaufplans nach Ausführung eines Jobs 29
3.8 Löschen eines Jobs aus dem Ablaufplan 30
3.9 Ablaufplan zur Auswahl des nächsten Threads 31
3.10 Einplanung eines Jobs mit verkürzter Ausführungszeit 33
3.11 Einplanung eines Jobs mit Kooperation der Anwendungen bei Überlast 34
3.12 Ungeordneter konsistenter Ablaufplan 35

4.1 Lost-Wakeup-Problem 45

5.1 Anteil zugewiesener Rechenzeit bei 50% Reservierung 54
5.2 Verpasste Deadlines in Abhängigkeit des Faktors k zur Anpassung der angeforderten Rechenzeit 56
5.3 Kosten für die Einplanung eines neuen Jobs 57
5.4 Cache-Misses bei der Übermittlung eines Jobs 58
5.5 Kosten für den Aufruf von `next` 59
5.6 Abarbeitungsstufen bei der Anzeige eines Videos mit angepasstem FFplay 60
5.7 Zeit zwischen zwei aufeinanderfolgenden Frames beim Abspielen eines Videos mit FFplay 62

1 Einleitung

Der wachsende Einfluss interaktiver Systeme ist allgegenwärtig und beeinflusst schon heute unser aller Leben. Diente der heimische Rechner vor einigen Jahrzehnten noch vorwiegend der Text- und Datenverarbeitung, wird er heute für weit anspruchsvollere Aufgaben eingesetzt. Beispiele sind multimediale Anwendungen wie die Audio- und Videowiedergabe oder die Darstellung komplexer Webseiten mit eingebetteten Multimedia-Inhalten. Längst werden solche Aufgaben aber auch schon von Tablets und Smartphones umgesetzt, die Desktop-Systemen in ihrem Funktionsumfang ebenbürtig sind. Die zusätzliche Mobilität dieser Geräte ermöglicht neue Funktionen, welche sich immer größerer Beliebtheit erfreuen.

Der rasante technische Fortschritt erfordert auch Weiterentwicklungen auf niedrigeren Systemebenen, um neuen Ansprüchen gerecht zu werden. Benutzer interaktiver Anwendungen erwarten eine zeitnahe Reaktion auf ihre Eingaben und tolerieren je nach Anwendungsgebiet kaum lange Wartezeiten. Ein Video-Player hat beim Anzeigen der Bilder eines Videos ähnlich strikte Zeitanforderungen: Um eine flüssige Videowiedergabe zu gewährleisten benötigt die Echtzeitanwendung einerseits ausreichend viel Rechenzeit, andererseits muss diese auch im richtigen Moment zur Verfügung stehen. Dies ist die Aufgabe des CPU-Schedulers, der die Zuweisung von Rechenzeit steuert. Gerade bei Anwendungen mit zeitkritischem Verhalten kommt dem CPU-Scheduler eine besondere Bedeutung zu: Die Einplanung muss unter Beachtung der Zeitanforderungen durchgeführt werden, andernfalls können Echtzeitanwendungen ihre Aufgaben nicht wie vom Benutzer gefordert umsetzen.

Obwohl der Linux-Kern verschiedene CPU-Scheduler bereitstellt, erfolgt in Desktop-Systemen bei der Einplanung keine Unterscheidung zwischen Echtzeitanwendungen und regulären Anwendungen. Folglich werden beide Klassen identisch behandelt. Dies funktioniert einwandfrei, solange im System ausreichend freie Rechenressourcen vorhanden sind. Erst wenn dies nicht mehr der Fall ist, erhalten Echtzeitanwendungen nicht die benötigten Ressourcen und können ihre Aufgaben somit nicht rechtzeitig abschließen: Für den Betrachter eines Videos ist dies ärgerlich, da beispielsweise eine rechenintensive Anwendung ohne Zeitanforderungen das Abspielen eines Videos nicht beeinträchtigen sollte.

Ein CPU-Scheduler, der zwischen den verschiedenen Arten von Anwendungen unterscheidet und die Zeitanforderungen von Echtzeitanwendungen beachtet, ermöglicht in ausgelasteten Systemen eine bessere Einplanung. Diese Unterscheidung erfordert eine geeignete Schnittstelle, die es einer Anwendung erlaubt ihre Rechenzeitanforderungen dem System mitzuteilen. An dieser Stelle setzt das am Lehrstuhl entwickelte *ATLAS*-Konzept [RWH13] an. ATLAS steht für *Auto Training Look Ahead Scheduler* und ermöglicht Anwendungen die Übermittlung von Rechenzeitanforderungen. Basierend auf diesen Informationen optimiert der ATLAS-Scheduler die Einplanung von Anwendun-

gen auf der CPU und ermöglicht auch in ausgelasteten Systemen die Ausführung von Echtzeitanwendungen mit der Einhaltung von Garantien.

Das Ziel meiner Diplomarbeit ist die Analyse, Implementierung und Evaluierung eines Schedulers unter Linux, der auf dem ATLAS-Konzept basiert.

Die Arbeit ist wie folgt aufgebaut: Im Kapitel 2 werden die Grundlagen für das Verständnis der Arbeit gelegt. Im Vordergrund steht dabei die Definition der verwendeten Begriffe sowie die Beschreibung verschiedener Aspekte eines Schedulers. Außerdem wird ATLAS detailliert beschrieben. Kapitel 3 beschreibt die Umsetzung des ATLAS-Schedulers sowie Strategien für den Umgang mit Überlast. Details zur Implementierung und Probleme bei der Umsetzung des Entwurfs beschreibt Kapitel 4. Eine Evaluierung des Schedulers erfolgt durch verschiedene Mikrobenchmarks und durch ein komplexeres Beispiel. Details dazu beschreibe ich in Kapitel 5. Den Abschluss der Arbeit bildet Kapitel 6: Neben einem Ausblick über mögliche weiterführende Arbeiten fasse ich die gesammelten Ergebnisse zusammen.

2 Grundlagen

In diesem Kapitel werden die notwendigen Grundlagen für das Verständnis der Arbeit erläutert. Dazu werden im ersten Abschnitt 2.1 die Begriffe *Prozesse* und *Threads* sowie *Tasks* und *Jobs* eingeführt. Die Abgrenzung *weicher* und *harter Echtzeitsysteme* ist Gegenstand des Abschnitts 2.2.

Die Bedeutung eines *Schedulers* für das System wird im Abschnitt 2.3 hervorgehoben. Darauf aufbauend gehe ich im Abschnitt 2.4 auf die schon vorhandenen Scheduler des Linux-Kerns ein und betrachte zwei allgemein gebräuchliche Scheduling-Strategien im Abschnitt 2.5 näher.

Abschnitt 2.6 beschreibt das *ATLAS-Konzept* und macht damit deutlich, welche Aufgabe die Arbeit innerhalb dieses Konzepts erfüllt. Abschnitt 2.7 dieses Kapitels befasst sich mit verwandten Arbeiten.

2.1 Prozesse und Threads, Tasks und Jobs

Die Repräsentation einer laufenden Anwendung auf Betriebssytemebene ist der *Prozess*. Ein Prozess besteht aus genau einem *Adressraum*, einer Menge zugewiesener *Ressourcen* sowie aus mindestens einem *Thread*. Ein Thread ist eine Ausführungseinheit mit eigenem *Stack*, die sich zu jedem Zeitpunkt in genau einem der Zustände *blockiert*, *bereit* oder *laufend* befindet. Übergänge zwischen den Zuständen sind wie in Abbildung 2.1 gezeigt möglich.

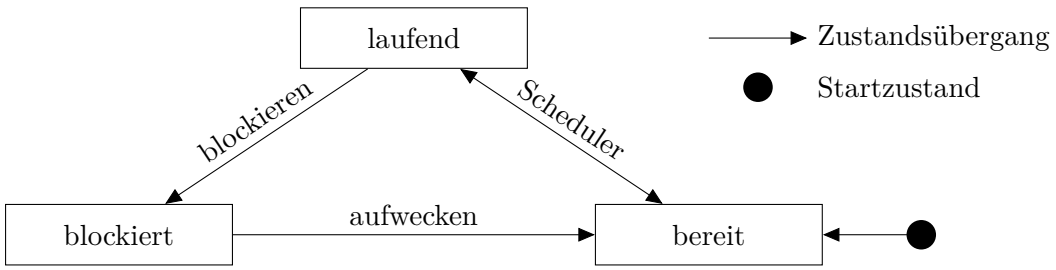


Abbildung 2.1: Zustandsübergänge eines Threads

Neu erzeugte Threads befinden sich im Zustand bereit und warten auf die Zuweisung einer CPU. Dies ist die Aufgabe des *CPU-Schedulers*, einem Algorithmus, der für die Auswahl des nächsten Threads aus der Menge der bereiten Threads zuständig ist. Die Auswahl wird als *Scheduling-Entscheidung* bezeichnet und basiert dabei auf einer Strategie, für welche die Bezeichnung *Scheduling-Strategie* gebräuchlich ist.

Wird ein Thread vom CPU-Scheduler ausgewählt, dann wechselt sein Zustand zu laufend und der Thread arbeitet die Instruktionen des abzuarbeitenden Programms ab.

Blockiert der Thread bei einer Operation, beispielsweise beim Lesen von Daten von der Festplatte, dann wechselt der Zustand des Threads zu blockiert. Der Thread kann nicht weiterlaufen, bis das Ereignis, auf welches der Thread wartet, eingetreten ist und dieser wieder aufgeweckt wird. Erst dann wechselt sein Zustand zurück zu bereit.

Der Übergang von laufend zu bereit und somit das Entziehen der CPU mit anschließender Scheduling-Entscheidung kann auch eingeleitet werden, wenn der Thread den Prozessor schon zu lange genutzt hat. Der Aufruf des CPU-Schedulers wird dann wie im Abschnitt 2.4.3 näher beschrieben durch einen Timer-Interrupt erzwungen.

Im Linux-Kern werden Threads als *leichtgewichtige Prozesse* implementiert. Demzufolge gibt es in *nebenläufigen Anwendungen*, dies sind Anwendungen mit mehr als einem Thread, mehrere Prozesse, die den gleichen Adressraum benutzen. In dieser Arbeit trenne ich Prozess und Thread strikt voneinander ab, jedoch werden die Begriffe in Bezug auf Linux in anderen Quellen oft als Synonyme verwendet.

Im Kontext des CPU-Schedulings werden die Begriffe *Task* sowie *Job* verwendet. Eine Task ist dabei eine nicht weiter spezifizierte Aufgabe bestehend aus Jobs. Damit kann ein System jederzeit durch die Angabe der Menge $T = \{\tau_i \mid i = 1, ..., n\}$ von Tasks τ_i spezifiziert werden. Jede Task τ_i besteht dabei aus einer Menge von im allgemeinsten Fall aperiodischen Jobs $\tau_i = \{J_{i,j} \mid j = 1, ..., m\}$. Innerhalb dieser Arbeit wird angenommen, dass Jobs jederzeit unterbrechbar sind und auch zu beliebigen Zeitpunkten bei der Abarbeitung blockieren können. Zwischen den Jobs der gleichen Task bestehen Abhängigkeiten: Die Ausführung von Job $J_{i,j+1}$ kann erst gestartet werden, wenn die Ausführung von Job $J_{i,j}$ abgeschlossen ist. Die Abarbeitung eines festen Jobs aus Sicht des Betriebssystems erfolgt immer von genau einem Thread, eine parallele Abarbeitung des gleichen Jobs von mehreren Threads ist unzulässig.

Innerhalb der Arbeit nutze ich zur Vereinfachung die für Threads gebräuchlichen Bezeichnungen auch für Jobs: Ein *blockierender Job* in diesem Kontext ist ein Job, dessen abarbeitender Thread blockiert. Das *Starten eines Jobs* in diesem Sinne bezeichnet die Auswahl des zugehörigen Threads durch den CPU-Scheduler für die weitere Abarbeitung.

2.2 Harte und weiche Echtzeitsysteme

Systeme werden als *Echtzeitsysteme* bezeichnet, wenn deren Rechenergebnisse neben der Korrektheit zusätzlichen Zeitanforderungen unterliegen. Der Nutzen der Berechnungen ist abhängig von der rechtzeitigen Fertigstellung innerhalb vorgegebener Zeitschranken. Dabei werden Zeitangaben, welche die späteste Fertigstellung einer Berechnung beschreiben, auch als *Deadlines* bezeichnet. Solche zeitlichen Abhängigkeiten werden von der Umgebung des Systems vorgegeben. Je nach Nutzwert eines erst nach Ablauf der Deadline verfügbaren Rechenergebnisses wird zwischen *harten* und *weichen* Echtzeitsystemen unterschieden [Liu00]. Eine dritte Art von Echtzeitsystemen, die sogenannten *firmen* Echtzeitsysteme, haben in dieser Arbeit keine Bedeutung und werden nicht gesondert betrachtet.

In *harten Echtzeitsystemen* führt das Verpassen einer Deadline zu schwerwiegenden bis katastrophalen Folgen. Das heißt, dass ein nach der Deadline zur Verfügung stehendes Ergebnis keinerlei Nutzen mehr hat. Die Angabe der Abarbeitungszeit von *harten*

Echtzeitproblemen erfolgt dabei durch eine maximale Ausführungszeit, die auch als *worst case execution time* (WCET) bezeichnet wird. Die Festlegung der WCET ist pessimistisch und beinhaltet den Fall, bei dem der Job einer Task zur Lösung des Problems die maximale Zeit beansprucht.

Beim Start einer Anwendung mit harten Echtzeitanforderungen erfolgt auf Basis der WCET eine Zulassungsprüfung, bei der festgestellt wird, ob die Einplanung der Task möglich ist, ohne bereits zugesicherte Ausführungszeiten anderer Tasks zu verletzen. Bei erfolgreicher Prüfung wird die Task angenommen, andernfalls wird diese abgewiesen. Aufgrund der pessimistischen Festlegung der WCET verwenden Anwendungen im Durchschnitt nur einen kleinen Bruchteil der WCET für Berechnungen, so dass Systeme mit ausschließlich harten Echtzeitanwendungen sehr schlecht ausgelastet sind.

Ein Beispiel für ein System mit harten Echtzeitanforderungen ist die Steuerung des Zündvorgangs in Motoren: Das Benzin-Luft-Gemisch muss genau im richtigen Moment gezündet werden, andernfalls sind Motorschäden wahrscheinlich.

Für viele Anwendungsgebiete sind die Anforderungen harter Echtzeitsysteme zu streng. Das Verpassen einer Deadline hat beispielsweise beim Abspielen eines Videos keine katastrophalen Auswirkungen, sondern geht lediglich mit einer Verminderung der Qualität einher. Die Qualität eines Videos sinkt für den Benutzer dabei mit steigender Anzahl verpasster Deadlines. Es ist jedoch nicht notwendig, das Abspielen eines Videos zu unterbrechen, wenn ein Bild nicht rechtzeitig angezeigt werden konnte. Systeme mit Echtzeitanforderungen dieser Art werden auch als *weiche Echtzeitsysteme* bezeichnet.

Auch die Einplanung von Anwendungen mit weichen Echtzeitanforderungen erfolgt durch die Angabe einer Deadline sowie einer Ausführungszeit. Im Gegensatz zur WCET bei harten Echtzeitanwendungen kann die Spezifikation einer weichen Echtzeitanwendung durch die Angabe einer Ausführungszeit in Verbindung mit einer Wahrscheinlichkeit erfolgen. Diese gibt an, wie wahrscheinlich es ist, dass ein Job innerhalb der angegebenen Ausführungszeit abgeschlossen wird.

Für Anwendungen ohne Echtzeitanforderungen wird auch der Begriff *Best-Effort* verwendet.

2.3 Scheduler

Die zur Verfügung stehende Rechenzeit innerhalb eines Systems wird durch den *CPU-Scheduler* verwaltet. Dieser muss in einem aus n Prozessorkernen bestehenden System genau n bereite Threads für die Ausführung auswählen. Der Entwurf des Systems muss dabei sicherstellen, dass zu jedem Zeitpunkt mindestens n bereite Threads existieren. Gibt es mehr bereite Threads als verfügbare Rechenkerne, dann muss die Auswahl gemäß einer Scheduling-Strategie erfolgen. Innerhalb dieser Arbeit verwende ich die Begriffe CPU-Scheduler und Scheduler als Synonyme.

Die Anforderungen an den Scheduler eines Systems sind je nach Anwendungsgebiet vielseitig und unterschiedlich. Im Folgenden sollen verschiedene Anforderungen näher erläutert werden.

Aus Sicht des Schedulers haben in harten Echtzeitsystemen alle bereits eingeplanten Tasks und deren Jobs zugewiesene Garantien, die in jedem Fall einzuhalten sind. Deshalb

muss der Scheduler eines harten Echtzeitsystems vor der Annahme einer neuen Task prüfen, ob weiterhin alle bereits zugesicherten Garantien anderer Tasks eingehalten werden. Ist dies der Fall, wird die neue Task angenommen, andernfalls wird diese abgelehnt. Der Scheduler eines Systems muss demzufolge eine *Zulassungsprüfung* durchführen, um über die Annahme oder Abweisung einer Task zu entscheiden.

Die Benutzer von Desktop-Systemen erwarten die parallele Ausführung von Anwendungen, um zeitgleich mehrere Anwendungen auszuführen: Arbeitet der Nutzer an einem Textdokument, dann soll beispielsweise ein im Hintergrund laufender Download nicht unterbrochen werden. Übersteigt die Anzahl bereiter Threads die Anzahl verfügbarer Prozessorkerne, dann kann eine parallele Ausführung durch die regelmäßige Umschaltung der laufenden Threads simuliert werden. Die Verteilung der Rechenzeit zwischen verschiedenen Anwendungen kann durch die Zuweisung von *Prioritäten* beeinflusst werden, wobei gilt: Je höher die Priorität einer Anwendung ist, desto größer ist das Verhältnis der zugewiesenen Rechenzeit im Vergleich zu einer Anwendung mit niedrigerer Priorität. Arbeiten in einem System zeitgleich mehrere Nutzer, dann ist es die Aufgabe des Schedulers, die Rechenzeit einerseits gleichmäßig zwischen verschiedenen Nutzern und andererseits zwischen den Tasks desselben Benutzers unter Beachtung der *Prioritätszuweisungen* zu verteilen. Diese Anforderung wird als *Fairness* bezeichnet.

In Systemen, die direkt mit den Eingaben des Nutzers arbeiten, gibt es Anwendungen, die abhängig von den Eingaben des Nutzers regelmäßig zwischen den Zuständen bereit, laufend und blockiert wechseln. Als Beispiel dient hier ein Texteditor, der die meiste Zeit blockiert ist. Bei einer Tastatureingabe durch den Benutzer wechselt der Zustand des verarbeitenden Threads, getriggert durch einen Interrupt, in den Zustand bereit. Nach Auswahl des Threads durch den Scheduler wird der eingegebene Buchstabe angezeigt und der Thread blockiert erneut, um auf weitere Eingaben zu warten. Solche *interaktiven Threads* sollten gegenüber *nicht-interaktiven Threads*, die beispielsweise über längere Zeit rechenintensive Aufgaben ausführen, bevorzugt werden. Diese Eigenschaft des Schedulers wird als *Interaktivität* bezeichnet.

Wird die Interaktivität nicht berücksichtigt, führt dies zu einer Verzögerung zwischen dem Auftreten eines Ereignisses, der Eingabe eines Buchstabens über die Tastatur im konkreten Beispiel, und der folgenden Reaktion, der Anzeige im Texteditor. Ist diese Verzögerung in einer vom Benutzer spürbaren Größenordnung, Card et al. [CRM91] spezifizieren eine Zeit von nur 100 ms für die Wahrnehmungsverarbeitung des Menschen, entsteht der Eindruck eines trägen Systemverhaltens. Dies wird vom Benutzer nicht akzeptiert.

Steigende Energiepreise sowie eine immer größer werdende Anzahl von Rechnersystemen führen dazu, dass auch Scheduler optimiert werden, um den Energieverbrauch von Rechnersystemen zu reduzieren. Die Ausführung von Threads auf Mehrprozessorsystemen wird hierbei so beeinflusst, dass die Taktfrequenz bestimmter Prozessoren beziehungsweise Kerne reduziert wird oder diese komplett abgeschaltet werden. Dieses Vorgehen erlaubt die *Reduzierung des Energieverbrauchs* [YN].

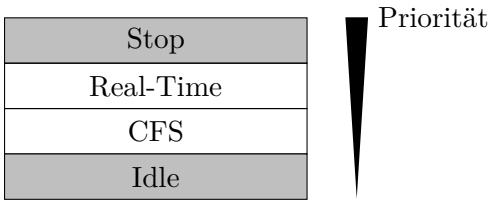


Abbildung 2.2: Schichten-Design der Scheduler innerhalb des Linux-Kerns

2.4 Scheduler in Linux

Der Linux-Kern der Version 3.5 besitzt zwei verschiedene Scheduler: den *Completely Fair Scheduler* (CFS) und den *Real-Time-Scheduler*.

Mit der Freigabe des Linux-Kerns der Version 2.6.23 wurde der bis dahin verwendete *O(1)-Scheduler* durch CFS ersetzt [Jon09]. Beide Scheduler wurden von Ingo Molnar entworfen und implementiert. Der verwendete Real-Time-Scheduler des Linux-Kerns wurde im Zuge dessen ebenfalls überarbeitet und vereinfacht.

Im Folgenden wird der im Linux-Kern verwendete Entwurf näher erläutert und die einzelnen Scheduler beschrieben. Wenn es nicht explizit anders erwähnt wird, dann beziehen sich die Betrachtungen auf den Linux-Kern der Version 3.5.

2.4.1 Schichtenarchitektur

Ab Version 2.6.23 wurde ein Entwurf gewählt, der es erlaubt, im Linux-Kern mit Hilfe eines mehrschichtigen Designs [Mol07] beliebig viele Scheduler zu verwenden. Abbildung 2.2 zeigt die Anordnung der bereits vorhandenen Scheduler des Linux-Kerns. Die Schichten und deren Anordnung sind dabei fest im Quellcode definiert und eine Änderung zur Laufzeit ist nicht vorgesehen. Untereinander sind die Schichten voneinander isoliert, trotzdem ist ein Datenaustausch zwischen ihnen möglich. Zur Gewährleistung der Skalierbarkeit läuft auf jeder CPU eine eigene Instanz der jeweiligen *Scheduling-Klasse* mit CPU-lokalen Variablen.

Jede Scheduling-Klasse verwaltet eine eigene Menge von Threads. Zwei beliebige Mengen sind dabei paarweise disjunkt, so dass zu jedem Zeitpunkt ein Thread innerhalb des Systems genau einer Scheduling-Klasse zugeordnet ist. Beim Aufruf des Schedulers fragt dieser beginnend bei der Klasse Stop nach einem bereiten Thread. Liefert die Klasse keinen bereiten Thread zurück, wird zur nächsten Klasse übergegangen bis ein bereiter Thread ermittelt wurde. Der Entwurf der Klasse *Idle* stellt sicher, dass es einen solchen Thread immer gibt.

Die strikte Ordnung der vorhandenen Klassen führt dazu, dass es zwischen den verschiedenen Klassen eine feste Prioritätszuordnung gibt. Diese *Klassen-Priorität* darf nicht mit der Priorität eines Threads innerhalb der Klasse selbst verwechselt werden. Letztere beeinflusst die Auswahl des nächsten Threads aus der Menge der bereiten Threads innerhalb der Scheduling-Klasse.

Der von Molnar verwendete modulare Entwurf erlaubt, dass Scheduling-Klassen mit höherer Klassen-Priorität die Rechenzeit komplett verwenden, wenn bei jeder Abfrage

durch das *Scheduling-Framework* ein bereiter Thread zurück gegeben wird. In diesem Fall werden die Klassen mit niedrigerer Klassen-Priorität bei Scheduling-Entscheidungen nicht berücksichtigt.

2.4.2 Scheduling-Klassen

In den folgenden Abschnitten werden die Scheduler der einzelnen Klassen näher vorgestellt.

Stop

Es gibt Situationen, welche die Unterbrechung sämtlicher Aktivitäten auf einer CPU erfordern. Beispielsweise müssen vor dem Abschalten einer CPU alle zugewiesenen Threads auf einen anderen Prozessor migriert werden. Zur Durchführung solcher Aufgaben dient die Scheduling-Klasse *Stop*, die pro CPU genau einen Thread mit der Bezeichnung *migration* verwaltet. Der jeweilige Thread arbeitet in einer Endlosschleife ihm übergebene Arbeit ab. Solange keine Aufgabe vorhanden ist, bleibt der Thread blockiert.

Die Scheduling-Klasse wird nur innerhalb des Kerns verwendet und hat keinerlei Bedeutung im Userland.

Real-Time-Scheduler

Linux erlaubt die Einplanung von Threads mittels des Real-Time-Schedulers, der aus Nutzersicht die höchste Klassen-Priorität besitzt. Mittels des Betriebssystemaufrufs `sched_setscheduler` kann dieser Scheduler für Threads festgelegt werden. Dabei werden eine *statische Prioritätszuordnung* sowie eine der beiden Strategien `SCHED_FIFO` beziehungsweise `SCHED_RR` konfiguriert. Insgesamt stehen Prioritätsstufen von 1 (höchste Priorität) bis 99 zur Verfügung. Die Auswahl des nächsten auszuführenden Threads erfolgt strikt nach zugewiesener Priorität.

Die Abarbeitung bereiter Threads innerhalb der gleichen Prioritätsstufe auf einem Prozessor erfolgt abhängig von der Strategie: `SCHED_FIFO` führt den Thread, der als erstes bereit war, solange aus, bis dieser blockiert beziehungsweise terminiert. Erst danach wird der nächste Thread ausgewählt. Die Ausführung des ersten Threads kann also die Ausführung anderer Threads beliebig lang verzögern. Anders ist dies bei der Strategie `SCHED_RR`: Bereite Threads der gleichen Prioritätsstufe erhalten nacheinander für eine feste Zeitdauer, auch Zeitscheibe genannt, den Prozessor und wechseln sich somit regelmäßig ab.

Auch wenn jeder Prozessor eine eigene Instanz des Real-Time-Schedulers ausführt, kann dieser als ein globaler Scheduler des Systems eingeordnet werden: Die sogenannten *Push-* und *Pop-Operationen* erzwingen auf einem System mit n Prozessoren die Ausführung der ersten n bereiten Threads mit der höchsten Priorität [Gar09].

Die Real-Time-Scheduling-Klasse ist aus Konformität des Linux-Kerns zur POSIX-API [Soc08] erforderlich und eignet sich besonders zur Implementierung von Echtzeitanwendungen in eingebetteten Systemen. In Desktop-Systemen ist die Klasse für weiche Echtzeitanwendungen aus zweierlei Hinsicht ungeeignet [AL02]: Ein Nutzer ohne besondere Rechte kann die Fairness und Sicherheit des Systems aushebeln, indem er die

Scheduling-Klasse zur Ausführung einer Anwendung nutzt, welche eine Endlosschleife ausführt und somit sämtliche zur Verfügung stehende Rechenzeit nutzt. Die Nutzung der Klasse nur Anwendern mit besonderen Rechten zu erlauben würde bedeuten, dass reguläre Nutzer keine weichen Echtzeitanwendungen wie beispielsweise Video-Player unter Einhaltung von Garantien mehr ausführen können.

Completely Fair Scheduler (CFS)

Neu erzeugte Prozesse und deren Threads werden unter Linux vom Completely Fair Scheduler (CFS) verwaltet. Folglich wird CFS im durchschnittlichen Nutzersystemen für alle Threads verwendet.

CFS definiert eine sogenannte Periode, in der alle bereiten Threads jeweils genau einmal der CPU zugeteilt werden. Die Länge einer Periode ist neben der Anzahl bereiter Threads auf einer CPU abhängig von der Anzahl der Prozessoren im System. Die Periode wird bei sehr vielen bereiten Threads verlängert, um zu verhindern, dass zu oft zwischen Threads hin und her geschaltet wird. Andernfalls sind hohe Kosten die Folge, welche ich in meiner Belegarbeit [Wä12] untersucht habe. Die Periode in einem System mit vier Prozessoren und weniger als acht bereiten Threads beträgt beispielsweise 18 ms.

In seiner Grundform ist CFS ein Scheduler, der Threads nur lokal pro CPU verwaltet. Algorithmen zum Lastausgleich werden genutzt, um die Auslastung der Prozessoren im System möglichst gleichmäßig zu halten. Dazu werden Threads bei Bedarf zwischen den Prozessoren migriert.

Idle

Hat keine der ersten drei Scheduling-Klassen einen bereiten Thread zurückgeliefert, dann gibt es derzeit keine zu erledigenden Aufgaben auf der CPU. Trotzdem muss feststehen, welche Instruktionen von der CPU als nächstes ausgeführt werden sollen. Dazu wird die Scheduling-Klasse Idle mit der niedrigsten Klassen-Priorität genutzt: Die Klasse liefert immer einen Thread zurück, der in einer Schleife die je nach Architektur möglichen Instruktionen zur Minimierung des Energieverbrauchs nutzt. Damit wird sichergestellt, dass immer ein bereiter Thread ausgewählt und der CPU zugeteilt wird.

2.4.3 Aufruf des Schedulers unter Linux

Der Scheduler eines Systems muss in regelmäßigen Abständen aufgerufen werden, um zu verhindern, dass ein bösartig oder fehlerhaft programmierter Thread den Prozessor dauerhaft für sich beansprucht. Dieser Abschnitt beschreibt in knapper Form, wann und wie der Aufruf des Schedulers unter Linux erfolgt. Dazu werden zwei unterschiedliche Möglichkeiten erläutert: der *direkte* und der *verzögerte Aufruf* [BC06].

Der direkte Aufruf des Schedulers erfolgt mittels der Funktion `schedule` im Kern und dient dem Blockieren eines Threads, meist um auf ein Ereignis zu warten. Ist das Ereignis eingetreten, dann wird der Thread wieder aufgeweckt und in die Menge der bereiten Threads eingefügt, um bei einer späteren Scheduling-Entscheidung erneut ausgewählt zu werden.

Der verzögerte Aufruf des Schedulers wird immer dann benutzt, wenn der direkte Aufruf nicht möglich oder nicht sicher ist. Beispielsweise wird in der Timer-Routine, die periodisch durch einen Timer-Interrupt aufgerufen wird, eine Prüfung durchgeführt, ob der gerade laufende Thread seine Rechenzeit ausgeschöpft hat. Ist dies der Fall, dann ist es nicht sicher, dem Thread die CPU zu entziehen. Dieser kann beispielsweise innerhalb eines kritischen Abschnitts sein und dabei Sperren halten. In diesem Fall muss der Thread seine Arbeit fortsetzen. Erst nach der Abgabe der Sperren erfolgt der Aufruf des Schedulers. Die Implementierung wird mittels eines Flags umgesetzt, welches den Thread entsprechend markiert und den zeitverzögerten Aufruf des Schedulers garantiert, sobald dies sicher ist.

2.5 Scheduling-Strategien

Unter einer Scheduling-Strategie versteht man die Auswahl des nächsten Threads in Abhängigkeit verschiedener Kriterien. In den folgenden zwei Abschnitten werde ich zunächst die Kriterien der beiden Scheduling-Strategien *EDF* und *LRT* erläutern. Außerdem gehe ich auf den Begriff der *akkumulierten Slack-Zeit* ein, der zum Vergleich beider Strategien benötigt wird.

2.5.1 Earliest Deadline First (EDF)

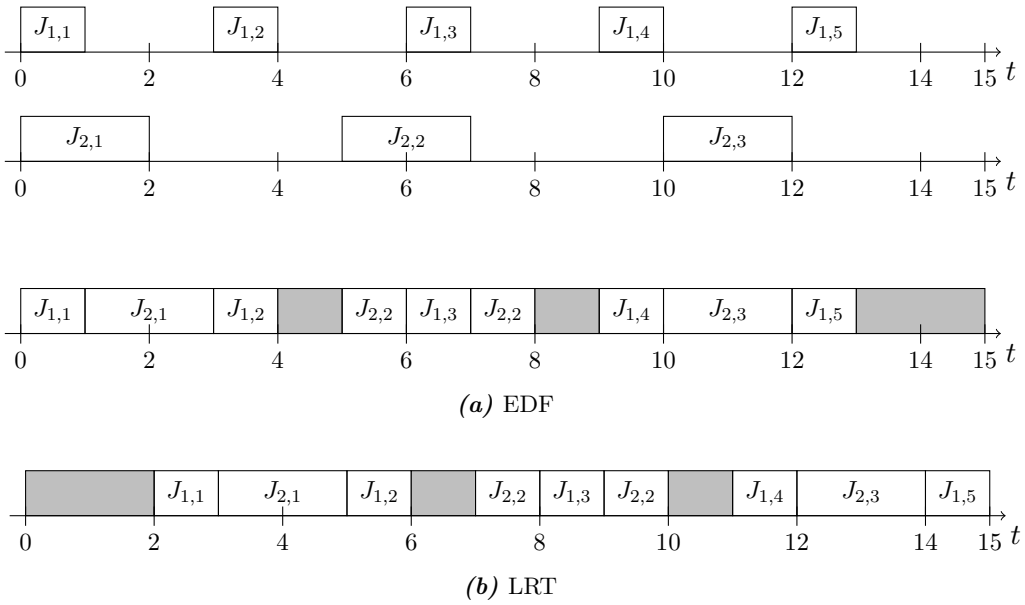
EDF ist eine ereignisgesteuerte Scheduling-Strategie. Das heißt, dass Scheduling-Entscheidungen nicht in Abhängigkeit der Zeit, sondern beim Eintreten von Ereignissen wie der Freisetzung neuer Jobs beziehungsweise der Fertigstellung laufender Jobs getroffen werden. Bei einer Scheduling-Entscheidung wählt EDF den Job aus, dessen Deadline am nächsten ist. Anstatt somit Tasks eine feste Priorität zuzuweisen, ergibt sich die Priorität eines Jobs ausschließlich anhand dessen zugewiesener Deadline dynamisch zur Laufzeit.

In Systemen mit periodischen Tasks ist es möglich, mittels EDF einen ausführbaren Ablaufplan mit einer Auslastung der CPU von bis zu 100% zu erstellen, ohne dass ein Job seine Deadline verpasst. In der Praxis muss allerdings ein Mehraufwand durch den Aufruf des Schedulers beachtet werden. Da es keinen Scheduling-Algorithmus geben kann, der den Prozessor mehr auslastet, ist EDF auf Einprozessorsystemen mit unabhängigen, unterbrechbaren Jobs optimal bezüglich der Einplanbarkeit [Der74].

2.5.2 Latest Release Time (LRT)

Wie EDF ist auch LRT eine ereignisgesteuerte Scheduling-Strategie. Im Gegensatz zu EDF erstellt LRT den Ablaufplan ausgehend von der anderen Richtung und wird deshalb auch als *reverse EDF* bezeichnet. Die Einplanung von Jobs erfolgt gemäß EDF, jedoch werden Deadlines als Freigabezeitpunkte und die Freigabezeitpunkte als Deadlines betrachtet. Durch dieses Vorgehen werden Jobs immer so spät wie möglich eingeplant.

Genau wie EDF ist LRT optimal bezüglich der Einplanbarkeit auf Einprozessorsystemen mit unabhängigen, unterbrechbaren Jobs [Liu00].



(a) EDF

(b) LRT

Abbildung 2.3: Beispiel für einen Ablaufplan

2.5.3 Akkumulierte Slack-Zeit

Bei einer Auslastung von weniger als 100% existieren Lücken im Ablaufplan, in denen kein Thread eingeplant ist. Die Summe der Lückenabschnitte zwischen zwei definierten Zeitpunkten wird dabei als *akkumlierte Slack-Zeit* bezeichnet. Im weiteren Verlauf der Arbeit nutze ich dafür die einfache Bezeichnung *Slack-Zeit*.

Periodische Tasks können durch das Tupel (e, p) beschrieben werden. Hierbei ist e die Ausführungszeit eines Jobs und p die Periode, die festlegt, wie oft neue Jobs der Task freigegeben werden. Abbildung 2.3 zeigt einen Ausschnitt des Ablaufplans zweier periodischer Tasks mit $T_1 = (1, 3)$ und $T_2 = (2, 5)$. Unabhängig von der verwendeten Scheduling Strategie ist die Slack-Zeit im Interval $[0, 15]$ gleich 4 Zeiteinheiten groß.

2.5.4 Vergleich EDF und LRT

Beim Vergleich von Abbildung 2.3(a) mit Abbildung 2.3(b) fällt auf, dass die Lücken im Ablaufplan bei LRT eher auftreten, auch wenn die Slack-Zeit unabhängig von der gewählten Strategie ist. Unter der Annahme, dass Jobs ihre angegebenen Ausführungszeiten niemals überschreiten, gibt es keine Notwendigkeit, Jobs eher auszuführen. Befinden sich im System andere Threads, die in den Lücken ausgeführt werden, kann LRT deren Antwortzeit verbessern, da Rechenzeit eher zugeteilt wird. Damit verhindert LRT im Gegensatz zu EDF die unnötig verfrühte Ausführung von Jobs.

Nachteil von LRT gegenüber EDF ist, dass Jobs, die ihre Rechenzeit auch nur minimal überschreiten, in jedem Fall nicht ihre Deadline einhalten. EDF ist hier robuster und kann je nach Situation Überschreitungen der Ausführungszeit von Jobs tolerieren.

2.6 ATLAS

Die Einplanung von Anwendungen mit weichen Echtzeitanforderungen wie beispielsweise Video-Playern erfolgt in Linux-Systemen mittels CFS zusammen mit anderen Best-Effort-Anwendungen. Solange sich das System dabei in keinem Zustand mit Überlast befindet, die nachgefragte Rechenzeit also kleiner als die zur Verfügung stehende Rechenzeit ist, funktionieren Echtzeitanwendungen auch mittels Einplanung in CFS.

Problematisch ist der Einsatz von Anwendungen mit Echtzeitanforderungen innerhalb von CFS, wenn die Rechenzeit nicht ausreichend ist und die Anwendungen ihre Jobs nicht innerhalb der Zeitschranken fertigstellen können. Im Falle des Video-Players sinkt die Qualität des Videos dann beispielsweise durch eine ruckelnde Wiedergabe.

Ein Lösungsansatz ist die Zuweisung einer höheren Priorität für Echtzeitanwendungen in CFS. Allerdings wird damit das Problem nur aufgeschoben, da eine steigende Last durch mehr rechnende Anwendungen wieder zu derselben Situation führt. Denkbar ist auch, eine bereits vorhandene Scheduling-Schicht mit einer höheren Klassen-Priorität für Anwendungen mit Echtzeitanforderungen zu verwenden. Im Standard-Kern kommt dafür nur die Real-Time-Scheduling-Klasse in Betracht. Die Klasse erfordert dabei die Abbildung von Zeitanforderungen einer Echtzeitanwendung auf statische Prioritäten. Aufgrund der beschränkten Anzahl verschiedener Prioritätsstufen sind die Möglichkeiten einer solchen Abbildung eingeschränkt und werden innerhalb der Arbeit nicht weiter untersucht.

Anwendungen mit Echtzeitanforderungen haben im Standard-Kern von Linux keine Möglichkeit, dem Scheduler explizit mitzuteilen, dass bis zu einem bestimmten Zeitpunkt eine fest zugesicherte Ausführungszeit zur Erledigung einer Aufgabe benötigt wird. Genau an diesem Punkt setzt der *Auto-Training Look-Ahead Scheduler* (ATLAS) [RWH13] an. Es handelt sich hierbei um eine Scheduling-Schnittstelle für Anwendungen speziell mit weichen Echtzeit-Anforderungen, die ihre Aufgaben in Form von Jobs mit einer festen Deadline modellieren können.

Ziel von ATLAS ist es, Anwendungen durch eine geeignete Schnittstelle die Möglichkeit zu geben, ihre auszuführenden Jobs mit den entsprechenden Deadlines zu spezifizieren und dem Scheduler mitzuteilen. Dieser kann dann den Anwendungen, die ATLAS verwenden, eine bessere Einplanung als der momentan verwendete Best-Effort-Ansatz gewährleisten.

2.6.1 Komponenten von ATLAS

Die Funktionalität von ATLAS wird durch zwei Komponenten bereitgestellt, die in Abbildung 2.4 dargestellt sind: dem *ATLAS-Scheduler* und einer Bibliothek mit der Bezeichnung *ATLAS-Lib*.

Anwendungen interagieren mit der ATLAS-Lib und nutzen deren Funktionalität zur Übermittlung von Jobs. Außerdem werden von den Threads einer *ATLAS-Anwendung* Funktionen der Bibliothek genutzt, um den Beginn eines neuen Jobs anzuzeigen.

Bei der Job-Übermittlung einer Anwendung an ATLAS wird der Job durch eine *Workloadmetric* $\underline{m}$ und eine Deadline d spezifiziert. Die Workloadmetric ist ein Vektor, der den auszuführenden Job ausschließlich mit dem internen Wissen der Anwendung beschreibt. Insbesondere enthält der Vektor keinerlei Informationen, die von der Hardware

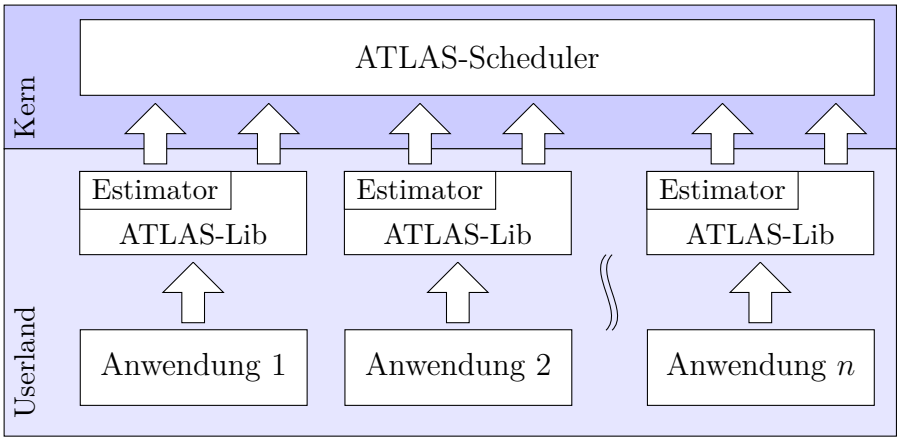


Abbildung 2.4: Komponenten von ATLAS

des Systems abhängig sind. Der *Estimator*, ein wesentlicher Bestandteil der ATLAS-Lib, schätzt mit Hilfe der Workloadmetric eine Ausführungszeit für das konkrete System ab. Dies ist für den Programmierer einer weichen Echtzeitanwendung eine enorme Vereinfachung, da eine Bestimmung der Ausführungszeit für ein System schwierig ist und mit dem Estimator entfällt. Die Schwierigkeit bei der Bestimmung der Ausführungszeit besteht zum einen darin, dass der Programmierer keine Kenntnis über alle Systeme hat, auf denen die Anwendung später ausgeführt wird. Zum anderen ist selbst bei genauem Wissen der Hardware eine Bestimmung der Ausführungszeit durch deren Komplexität schwierig [WEE+08].

Der Estimator schätzt die Ausführungszeit basierend auf der Workloadmetric sowie aus dem Vergleich der tatsächlichen Ausführungszeit früherer Jobs und deren vorher bestimmten Ausführungszeit. Durch diesen iterativen Lernprozess verbessert sich die Schätzung des Estimators mit der Anzahl der Jobs. Details zur Bestimmung der Ausführungszeit werden von Roitzsch et. al. [RWH13] beschrieben.

Nachdem der Estimator die benötigte Ausführungszeit ermittelt hat, wird diese zusammen mit der von der Anwendung spezifizierten Deadline an den *ATLAS-Scheduler* übergeben. Die Aufgabe des ATLAS-Schedulers ist dann die Einplanung der Threads auf Basis der Job-Übermittlungen. Auf jeder CPU des Systems wird dazu eine eigenständige Instanz des ATLAS-Schedulers ausgeführt, die, wie in Abbildung 2.4 gezeigt, alle auf der CPU laufenden ATLAS-Anwendungen einplant.

Der ATLAS-Scheduler ist im Gegensatz zur Anwendung und dem Estimator, die beide im Userland ausgeführt werden, direkt in den Linux-Kern integriert, da sonst keine Nutzung des Scheduling-Frameworks und der Schichtenarchitektur von Schedulern des Linux-Kerns möglich ist.

2.6.2 Primitive des ATLAS-Schedulers

Zwischen der ATLAS-Lib und dem ATLAS-Scheduler müssen Informationen ausgetauscht werden. Zur Bereitstellung der Funktionalität von ATLAS sind drei verschiedene Primitive erforderlich, welche ich in diesem Abschnitt beschreibe.

submit - Übermittlung eines Jobs

Bei der Übermittlung eines Jobs von der ATLAS-Lib an den ATLAS-Scheduler wird ein Primitiv mit der Bezeichnung `submit` verwendet. Bei jeder Übermittlung eines Jobs muss festgelegt werden, für welchen Thread der Job eingeplant werden soll. Dazu wird die *Thread-ID tid* übermittelt, die einen Thread eindeutig identifiziert. Außerdem wird die vom Estimator berechnete Ausführungszeit e sowie die von der Anwendung spezifizierte Deadline d übertragen. Aus Sicht des Schedulers besteht eine Job-Übermittlung damit aus dem Tupel (tid, e, d).

next - Ausführungsbeginn eines neuen Jobs

ATLAS-Anwendungen spezifizieren die auszuführenden Arbeiten mittels Jobs. Entsprechend gibt es Threads, die diese Jobs abarbeiten. Beginnt ein Thread mit der Abarbeitung eines neuen Jobs, dann zeigt das Primitiv `next` an, dass die Ausführung des vorherigen Jobs beendet ist. Damit kann der Estimator der ATLAS-Lib die Differenz seiner früheren Schätzung und der tatsächlich benötigten Abarbeitungszeit bestimmen und dieses Wissen für zukünftige Schätzungen verwenden.

Wird das Primitiv von einem Thread genutzt, obwohl noch keine Job-Übermittlung für diesen stattgefunden hat, dann blockiert der Thread bis ein Job verfügbar ist.

cancel - Löschen eines übermittelten Jobs

Das Primitiv `cancel` kann von der ATLAS-Lib auf Anfrage einer Anwendung dazu verwendet werden, um einen bereits an den ATLAS-Scheduler übermittelten Job zu löschen. Beispielsweise ist dieses Primitiv für einen Video-Player nützlich: Wird sowohl ein Thread für das Dekodieren von Bildern eines Videos als auch für die Anzeige der Bilder mittels des ATLAS-Schedulers eingeplant, dann kann das Löschen eines für den Anzeige-Thread übermittelten Jobs genutzt werden, um die Anzeige verspäteter Bilder auszulassen.

2.6.3 Anwendungsbeispiele für ATLAS

ATLAS kann von allen Anwendungen genutzt werden, die ihre Berechnungen durch Jobs mit einer Deadline abbilden können. Sinnvoll ist der Einsatz für Echtzeitanwendungen beispielsweise in Desktop-Systemen. Multimedia-Anwendungen wie Audio- und Video-Player können durch den Einsatz von ATLAS in Systemen profitieren, die stark durch Best-Effort-Anwendungen ausgelastet sind.

Die Integration von ATLAS in bereits vorhandene Bibliotheken zur Darstellung von Benutzeroberflächen und deren Verhalten während der Bedienung durch den Nutzer sind ein weiteres Anwendungsbeispiel: Klickt ein Benutzer auf eine Schaltfläche, dann kann ein Job für die Neuzeichnung der Schaltfläche erzeugt werden und vom ATLAS-Scheduler eingeplant werden. Es ist denkbar, mit diesem Vorgehen die Zeit zwischen Benutzereingaben und den entsprechenden Reaktionen zu optimieren.

2.7 Verwandte Arbeiten

Aus Sicht von Anwendungen gibt es unterschiedliche Scheduling-Ansätze. Im einfachsten Fall hat eine Anwendung keine Möglichkeit, dem Scheduler Zeitanforderungen mitzuteilen und kann bestenfalls eine faire Einplanung erwarten. Scheduler dieser Art erlauben meist eine zusätzliche Zuweisung von Prioritäten, welche die Scheduling-Entscheidungen beeinflussen.

Andererseits gibt es auch Scheduler, die von Anwendungen für deren Einplanung die Angabe einer Periode sowie Ausführungszeit fordern. Die Parameter werden dann verwendet, um zu überprüfen, ob eine Einplanung möglich ist, ohne die bereits zugesicherten Garantien anderer Anwendungen zu verletzen. Ist dies nicht der Fall, dann wird die Anwendung abgewiesen.

In den beiden folgenden Absätzen werden Arbeiten zu beiden Ansätzen untersucht und mit ATLAS verglichen.

2.7.1 Einplanung basierend auf Reservierungen

Der *Constant Bandwidth Server* (CBS) [ABSA98] ist ein Ansatz, der die gemeinsame Einplanung weicher und harter Echtzeitanwendungen ermöglicht. Dabei werden keine Garantien harter Echtzeitanwendungen verletzt, wenn es bei einer weichen Echtzeitanwendung zu einer Überschreitung der angegebenen Rechenzeit kommt.

Harte Echtzeitanwendungen werden mittels eines EDF-Schedulers eingeplant. Zusätzlich wird für jede Anwendung mit weichen Echtzeitanforderungen ein eigener CBS erzeugt, der vom Scheduler wie eine harte Echtzeitanwendung eingeplant wird. Der CBS selbst arbeitet die zugehörige weiche Echtzeitanwendung ab und stellt dabei sicher, dass diese bei der Überschreitung der angegebenen Ausführungszeit unterbrochen wird und erst wieder Rechenzeit in der nächsten Periode des CBS erhält. Dieses Vorgehen verhindert, dass die Garantien eingeplanter harter Echtzeitanwendungen selbst bei einer böswilligen oder fehlerhaften weichen Echtzeitanwendung verletzt werden.

Zur Nutzung des CBS müssen Anwendungen mit weichen Echtzeitanforderungen eine Periode angeben, welche die durchschnittliche Zeit zwischen zwei aufeinanderfolgenden Jobs repräsentiert. Außerdem muss die in einer Periode verwendete Rechenzeit angegeben werden. Besonders die Abschätzung dieser Zeit ist für den Entwickler schwierig, weil oft nicht bekannt ist, auf welcher Hardware die Anwendung ausgeführt wird. Ist die Abschätzung zu hoch, erfolgt die Zurückweisung des CBS bei der Prüfung der Zulassung durch den Scheduler. Ist die Abschätzung dagegen zu niedrig, werden Deadlines verpasst. Die Folge für die Anwendung ist eine niedrigere Qualität.

Der CBS erlaubt keine dynamische Anpassung der Parameter zur Laufzeit, die auf einer Änderung des Verhaltens der weichen Echtzeitanwendung basieren. Dies ist ein Nachteil gegenüber ATLAS, da die Anwendung durch Anpassung der Workloadmetric eine Veränderung der eingeplanten Rechenzeit erwirken kann, um den Scheduler frühzeitig über neue Anforderungen zu informieren.

Um den Programmierer bei der Festlegung der Ausführungszeit zu unterstützen, gibt es weitergehende Ansätze, die auf dem CBS aufbauen. Zu nennen ist hier AQuoSA [PCL08], ein Projekt welches auf der Arbeit von Abeni et. al. [ACL+05] aufbaut. Das Ziel ist

die Ausführungszeit während der Laufzeit entsprechend dem Verhalten der Anwendung anzupassen, um damit die Dienstqualität der Anwendung zu erhöhen und gleichzeitig eine hohe Auslastung der CPU zu ermöglichen. Im Gegensatz zu ATLAS ermöglicht dieser Ansatz keine dynamische Anpassung der eingeplanten Rechenzeit durch die Anwendung selbst. Die Änderung der Einplanungszeit beruht auf den Beobachtungen der genutzten Rechenzeit einer Anwendung in der Vergangenheit.

Weitergehende Ansätze wie beispielsweise von Cucinotta et. al [CCAP10] verzichten auf einen direkten Informationsfluss mit der einzuplanenden Anwendung zur Spezifikation von Periode und Ausführungszeit. Stattdessen wird das Verhalten der Anwendung zur Laufzeit untersucht und die entsprechenden Parameter für den CBS abgeleitet. Der Vorteil dieses Ansatzes ist, dass Anwendungen ohne Anpassungen eingeplant werden können. Der Nachteil ist genau wie bei den bisherigen Ansätzen, dass es keine Schnittstelle für die Anwendung gibt, um den Scheduler aktiv über Änderungen der benötigten Rechenzeit zu informieren. Entsprechend erfolgt die Änderung der eingeplanten Rechenzeit erst, nachdem zahlreiche Deadlines nicht eingehalten wurden.

Einen ähnlichen Ansatz, der auf die Anpassung von Anwendungen verzichtet, verwendet Redline [YLB+08]. Die Spezifikation der Anforderungen einer Anwendung erfolgt hier über eine zentrale Konfigurationsdatei. Die in der Datei angegebenen Rechenzeiten werden während der Laufzeit zur Optimierung der Einplanungen weiter angepasst. Da ein direkter Informationsfluss zwischen Anwendung und Scheduler nicht möglich ist, sind wieder keine kurzzeitigen dynamischen Anpassungen des Schedulers bei sich ändernden Anforderungen der Anwendung möglich. Redline erlaubt zusätzlich das Management des virtuellen Speichers sowie die Anpassung von Festplattenzugriffen. ATLAS kann derzeit nur für das CPU-Scheduling verwendet werden.

2.7.2 Einplanung basierend auf gerechter Rechenzeitverteilung

Ein anderer Scheduling-Ansatz basiert auf der Zuteilung von Rechenzeit anhand von Prioritätszuordnungen. Als Beispiel ist hier das in Linux verwendete CFS zu nennen, das auf Prinzipien der Borrowed Virtual Time [DC99] aufbaut. In Abhängigkeit der zugewiesenen Prioritäten, die in Unix auch als Nice-Levels [lin] bezeichnet werden, wird die Rechenzeit in CFS zwischen allen Threads möglichst fair verteilt. Möchte eine Anwendung die zugewiesene Rechenzeit ändern, dann erfolgt dies durch die Anpassung der Priorität. Die Rechenzeitverteilung ergibt sich immer durch die Gesamtheit aller sich im System befindlichen bereiten Threads, so dass eine globale Sicht auf das System notwendig ist, um exakte Aussagen über die Dauer der zugewiesenen Rechenzeit zu treffen.

Eine Erweiterung dieses Ansatzes ist das *Cooperative Polling* [KSSG09]. Anwendungen können Jobs mit Zeitanforderungen spezifizieren und mittels eines zusätzlichen Betriebssystemaufrufs an den Scheduler übermitteln. Anders als bei ATLAS werden Jobs durch den frühesten Freigabezeitpunkt beschrieben. Es besteht keine Möglichkeit, die Dauer der benötigten Rechenzeit anzugeben. Der Scheduler weist basierend auf den Job-Übermittlungen den Threads dann die CPU im richtigen Moment zu. Die Annahme bei der Abarbeitung der Jobs ist, dass diese die CPU nur für eine kurze Zeitdauer verwenden. Die Implementierung erlaubt außerdem nicht das Blockieren eines Threads

während der Abarbeitung eines Jobs. Andernfalls wird dieser zu einem Best-Effort-Thread herabgestuft. Im Vergleich zum Cooperative Polling erlaubt ATLAS neben der Möglichkeit der Angabe einer Rechenzeit für einen Job auch das beliebige Blockieren während der Job-Abarbeitung.

Zusammenfassend ist die Nutzung von Schedulern, die allein auf einer gerechten Rechenzeitverteilung basieren, für Echtzeitanwendungen ungeeignet, da keine Möglichkeit zur Spezifikation von Zeitanforderungen vorhanden ist. Die faire Verteilung von Rechenzeit ist für Best-Effort-Anwendungen geeignet, deren Abarbeitung nicht an zeitkritische Randbedingungen geknüpft ist.

2.7.3 Vergleich mit ATLAS

Keines der vorgestellten Systeme erlaubt die Einplanung von Aufgaben auf Basis expliziter Job-Übermittlungen, wie es mit ATLAS möglich ist. Die Festlegung der benötigten Rechenzeit vereinfacht sich auf die Spezifikation einer Workloadmetric. Diese wird vom Estimator der ATLAS-Lib zur Abschätzung der Rechenzeit verwendet und zusammen mit der von der Anwendung festgelegten Deadline an den ATLAS-Scheduler übermittelt. Der Ansatz abstrahiert damit die Hardware vollständig und vereinfacht die Anwendungsprogrammierung.

Durch die Spezifikation der Workloadmetric pro Job kann die Rechenzeit individuell festgelegt werden. Entstehende Änderungen bezüglich Rechenzeitanforderungen der Anwendung können deshalb schon *vorher* dem Scheduler mitgeteilt werden. Andere Systeme wie beispielsweise Redline erlauben die Anpassung der eingeplanten Rechenzeit erst nach der Beobachtung der Anwendung, da keine direkte Möglichkeit für den Informationsaustausch zwischen Anwendung und Scheduler besteht. Da die Anpassung der Rechenzeit erst nach der Änderung des Verhaltens der Anwendung auftritt, gibt es ein Zeitfenster, in welchem zu wenig Zeit für die Anwendung eingeplant wurde. In der Praxis bedeutet dies, dass Deadlines nicht eingehalten werden und es aus Sicht des Nutzers zu einer Qualitätsverminderung bei der Ausführung der betreffenden Anwendung kommt. Der Einsatz von ATLAS verhindert dies.

3 Design

In diesem Kapitel beschreibe ich Details zum ATLAS-Scheduler beginnend bei der Übermittlung eines Jobs bis hin zur Einplanung der Threads. Dazu fasse ich im ersten Abschnitt 3.1 die Anforderungen zusammen, welche der ATLAS-Scheduler für einen erfolgreichen Einsatz umsetzen muss. Diese Anforderungen beeinflussen zahlreiche Design-Entscheidungen, wie beispielsweise die im Abschnitt 3.2 beschriebene Einordnung des ATLAS-Schedulers in die Schichtenarchitektur von Schedulern des Linux-Kerns.

Die Beschreibung der Scheduling-Strategie im Abschnitt 3.3, die Aufstellung eines Ablaufplans im Abschnitt 3.4 und der Zusammenhang zwischen Scheduling-Strategie und Ablaufplan im Abschnitt 3.5 sind die zentralen Punkte des Kapitels.

Aufgrund von begrenzten Rechenressourcen muss das Verhalten des ATLAS-Schedulers auch in Überlastsituationen definiert werden. Eine detaillierte Betrachtung erfolgt im Abschnitt 3.6.

Bis zu diesem Punkt wurde die Abarbeitung des ATLAS-Schedulers nur lokal pro Prozessor erörtert. Abschnitt 3.7 zeigt deshalb einen möglichen Ansatz für das Scheduling auf Mehrprozessorsystemen. Einen weiteren wichtigen Aspekt beschreibt Abschnitt 3.8: Betrachtet werden die Vor- und Nachteile von relativen und absoluten Zeitangaben bei der Übermittlung von Jobs. Den Abschluss des Kapitels bildet Abschnitt 3.9 mit der Betrachtung von Threads, die während der Abarbeitung eines Jobs eine Kopie von sich selbst erzeugen.

3.1 Anforderungen/Annahmen des ATLAS-Schedulers

Der ATLAS-Scheduler kann nur erfolgreich Anwendungen mit weichen Echtzeitanforderungen einplanen, wenn eine Reihe von Anforderungen erfüllt werden. Diese Anforderungen werden im Abschnitt 3.1.1 vorgestellt. Innerhalb der Arbeit werden außerdem Annahmen vorausgesetzt, die im Abschnitt 3.1.2 zusammengefasst werden.

3.1.1 Anforderungen

A1 *Durchsetzung weicher Echtzeitgarantien.* Die Motivation zur Nutzung der ATLAS-Schnittstelle ist, einer Anwendung mit weichen Echtzeitanforderungen eine bessere Einplanung als Best-Effort in CFS zu ermöglichen. Dazu verwendet der ATLAS-Scheduler die Informationen der übertragenen Jobs für Scheduling-Entscheidungen, um Garantien für Anwendungen durchzusetzen.

A2 *Einplanung zur Laufzeit.* In der Praxis sind die auszuführenden Arbeiten einer Anwendung in der Regel abhängig von den Eingaben des Benutzers. Deshalb ergeben sich Details zu Anzahl und Freigabezeitpunkten der Jobs sowie deren Ausführungszeiten

und Deadlines erst zur Laufzeit. Der ATLAS-Scheduler soll demzufolge dynamisch zur Laufzeit neue Jobs entgegen nehmen und diese verarbeiten.

A3 *Erhaltung bestehender Klassen und deren Eigenschaften.* Der ATLAS-Scheduler soll den Linux-Kern um zusätzliche Funktionalität erweitern. Bereits bestehende Scheduling-Klassen sollen dabei erhalten bleiben und deren Eigenschaften sollen nicht durch die zusätzliche Funktionalität des ATLAS-Schedulers beeinflusst werden. Außerdem soll die Rechenzeit, die nicht vom ATLAS-Scheduler genutzt wird, anderen Scheduling-Klassen zur Verfügung stehen.

A4 *Umgang mit Überlast.* Das Verhalten des ATLAS-Schedulers muss auch in Situationen nachvollziehbar sein, in denen die angeforderte Rechenzeit größer als die noch zur Verfügung stehende Rechenzeit ist. Die Behandlung solcher Überlastsituationen wird detailliert im Abschnitt 3.6 betrachtet.

A5 *Annahme aller Jobs.* Anwendungen haben bei der Übermittlung eines Jobs kein Wissen über die noch zur Verfügung stehende einplanbare Rechenzeit. Insbesondere wenn die angeforderte Rechenzeit die noch verfügbare übersteigt, erwartet das ATLAS-Konzept trotzdem die Annahme aller Jobs. Die Abweisung eines Jobs durch den ATLAS-Scheduler ist damit unzulässig.

A6 *Umgang mit blockierenden Jobs.* Der ATLAS-Scheduler muss mit blockierenden Threads arbeiten. Dies ist erforderlich, da viele Systemaufrufe blockieren können. Der Scheduler darf an keiner Stelle annehmen, dass ein Thread während der Abarbeitung eines Jobs nicht blockiert. Andernfalls wäre der Scheduler für praktische Anwendungen nicht sinnvoll nutzbar.

A7 *Ausführungszeit als Schätzung.* Die in den Jobs übermittelte Ausführungszeit ist eine Schätzung, die sowohl unter- als auch überschätzt sein kann. Der Estimator des ATLAS-Konzepts verbessert seine Schätzungen während der Laufzeit, somit sind Schätzungen der Ausführungszeit insbesondere zu Beginn der Ausführung einer Anwendung fehlerbehaftet.

3.1.2 Limitierungen innerhalb der Arbeit

In Systemen mit mehreren Prozessoren kann ein Thread, wenn nicht durch den Programmierer beziehungsweise die Systemeinstellungen weiter beschränkt, auf einer beliebigen CPU gestartet werden. Ebenso können bereits laufende Threads zwischen CPUs migriert werden, wenn dies aus Sicht der benutzten Scheduling-Klasse aufgrund der Systemauslastung erforderlich scheint.

Die Migration von Threads zwischen CPUs ist für den ATLAS-Scheduler problematisch, da die Einplanung der Jobs lokal pro CPU erfolgt. Eine Migration eines Threads zu einer anderen CPU erfordert entsprechend, dass auch die Zuordnung der verknüpften Jobs zwischen den beteiligten Kernen angepasst wird. Eine weitergehende Analyse der entstehenden Probleme und Lösungsansätze werden im Abschnitt 3.7 vorgestellt.

Zur Vereinfachung wird innerhalb dieser Arbeit angenommen, dass Threads, die während ihres Lebenszyklus zu einem beliebigen Zeitpunkt vom ATLAS-Scheduler

eingeplant werden, vor der Übermittlung des ersten Jobs an eine feste CPU gebunden wurden. Die Einschränkung ist durch den Anwendungsprogrammierer unter Linux durch Nutzung des Systemaufrufs `sched_setaffinity` umzusetzen. Migrationen eines Threads vor der ersten Job-Übermittlung sind möglich.

3.2 Einordung des ATLAS-Schedulers

Die im Abschnitt 2.4.1 vorgestellte Schichtenarchitektur von Schedulern des Linux-Kerns erfordert eine entsprechende Einordnung des ATLAS-Schedulers, wenn dieser als eigenständige Schicht umgesetzt wird. Dieser Abschnitt beschreibt genauere Details zur Einordnung und die dadurch entstehenden Eigenschaften für das gesamte System.

Der Completely Fair Scheduler (CFS) wird, wie schon im Abschnitt 2.4.2 beschrieben, in konventionellen Desktop-Systemen für alle Nutzeranwendungen verwendet. Anforderung 1, die Durchsetzung weicher Echtzeitgarantien, ist nur möglich, wenn die vom ATLAS-Scheduler eingeplanten Threads gegenüber CFS bevorzugt werden. Aus diesem Grund ist der ATLAS-Scheduler über CFS anzuordnen.

Andererseits muss der ATLAS-Scheduler unterhalb der Real-Time-Scheduling-Klasse angeordnet werden, um nicht gegen die Annahmen bereits bestehender Anwendungen der Real-Time-Scheduling-Schicht zu verstoßen, die möglicherweise explizit für diesen Entwurf erstellt und zugelassen wurden. Damit ergibt sich der Aufbau der Scheduling-Schichten, wie er in Abbildung 3.1 dargestellt ist.

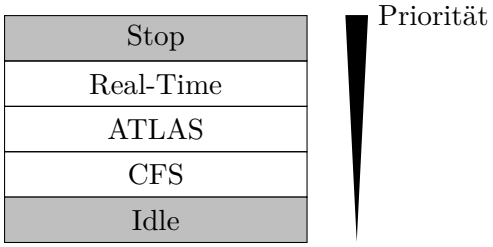


Abbildung 3.1: Einordnung des ATLAS-Schedulers in die Schichtenarchitektur von Schedulern des Linux-Kerns

Wird vom ATLAS-Scheduler kein Thread für die Abarbeitung ausgewählt, dann erhält CFS als erste Scheduling-Klasse mit niedrigerer Klassen-Priorität gemäß Anforderung 3, der Erhaltung bestehender Klassen und deren Eigenschaften, die vom ATLAS-Scheduler ungenutzte Rechenzeit. Dieser kann somit durch die verwendete Scheduling-Strategie die *Rechenzeitabgabe* an CFS steuern.

Ein alternativer Lösungsansatz ist die Erweiterung und Anpassung einer bestehenden Scheduling-Klasse um die Funktionalität des ATLAS-Schedulers ohne die Einführung einer weiteren Scheduling-Schicht. Anforderung 3 erlaubt das Hinzufügen neuer Funktionalität, wenn die bereits bestehenden Eigenschaften einer Scheduling-Klasse nicht verändert werden. Aus Entwurfssicht ist dieser Ansatz allerdings nicht von Vorteil, da die Weiterentwicklung und Wartung der Scheduling-Klasse komplizierter wird. Deshalb wird dieser Ansatz innerhalb der Arbeit nicht weiter verfolgt.

3.3 Scheduling-Strategie

ATLAS verwaltet eine Menge von bereiten Prozessen. Abhängig von den übermittelten Jobs und deren Parametern muss gemäß einer sinnvollen Strategie der nächste auszuführende Thread bestimmt werden. Im nächsten Abschnitt werden dazu zwei gebräuchliche Strategien vorgestellt und deren Vor- und Nachteile betrachtet.

3.3.1 LRT als grundlegende Scheduling-Strategie

Die im Abschnitt 2.5 beschriebenen Scheduling-Strategien EDF und LRT sind beide für die Aufstellung eines Ablaufplans geeignet, auf dem die Scheduling-Entscheidungen aufbauen. Bei Lücken im Ablaufplan erfolgt die Rechenzeitabgabe an CFS.

Die Verwendung von EDF zur Ablaufplanerstellung führt allerdings dazu, dass Jobs schon frühzeitig abgearbeitet werden, ohne dass dafür eine zwingende Notwendigkeit besteht. Dadurch ergibt sich ein Problem, welches folgendes Beispiel verdeutlicht: Befinden sich immer Jobs für einen Thread im System, welche dieser ohne zu blockieren abarbeitet, dann verhindert der Thread die Abgabe von Rechenzeit an CFS, selbst wenn die Deadlines der abzuarbeitenden Jobs weit in der Zukunft liegen und keine zwingende Notwendigkeit für deren Ausführung besteht. Folglich nutzt der ATLAS-Scheduler aufgrund seiner höheren Klassen-Priorität sämtliche zur Verfügung stehende Rechenzeit und die von CFS bereitgestellten Systemeigenschaften wie Interaktivität gehen verloren.

Bei der Verwendung von LRT besteht dieser Nachteil nicht, da Jobs erst so spät wie möglich gestartet werden. Eine minimale Überschreitung der Ausführungszeit führt dann jedoch zum Verpassen der Deadline des Jobs. Aus Sicht der Anwendung kann dann die mit dem Job verknüpfte Aufgabe nicht wie geplant rechtzeitig fertiggestellt werden. Bei der Verwendung von EDF besteht dieses Problem nur in seltenen Fällen, da die frühe Abarbeitung eines Jobs eine Unterschätzung der Ausführungszeit abhängig von der Gesamtlast erlaubt.

Sowohl EDF als auch LRT haben damit Vor- und Nachteile, wobei die Vorteile von LRT überwiegen: In nicht ausgelasteten Systemen erhält CFS früher Rechenzeit und erlaubt die Abarbeitung seiner Threads. LRT wird deshalb als grundlegende Scheduling-Strategie für die Erstellung eines Ablaufplans verwendet. Details dazu werden im Abschnitt 3.4 beschrieben.

3.3.2 Erster Ansatz einer Scheduling-Strategie

Erste Implementierungen des ATLAS-Schedulers basieren auf einem einfachen Entwurf, der in diesem Abschnitt beschrieben wird.

Abbildung 3.2 zeigt die Anordnung der verschiedenen Scheduler. Unterhalb des Real-Time-Schedulers befindet sich die Schicht ATLAS-LRT, wie sie bereits in 3.3.1 beschrieben wurde. Die unter CFS angeordnete Schicht *ATLAS-Fallback* dient dazu, die nicht benutzte Rechenzeit höherer Schichten zu verwenden, um den Threads aus ATLAS-LRT einen Vorlauf zu gewähren. Durch den Vorlauf wird das grundlegende Problem von LRT eingeschränkt, dass bei einer minimalen Überschreitung der Ausführungszeit sofort die Deadline verpasst wird.

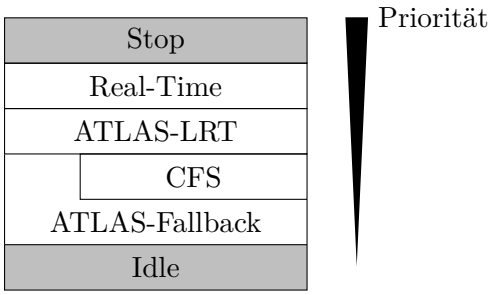


Abbildung 3.2: Alternatives Schichten-Design von Schedulern des Linux-Kerns zur Umsetzung des ATLAS-Schedulers

Die Bereitstellung eines Vorlaufs auf Basis dieses Entwurfs funktioniert, solang es in CFS keine dauerhaft bereiten Threads gibt. Diese Einschränkung ist für praktische Systeme inakzeptabel, da es immer wieder Überlastsituationen gibt. Der gewählte Lösungsansatz besteht in der Anpassung der von ATLAS-LRT ungenutzten Rechenzeit. Ist die im Abschnitt 2.5.3 beschriebene Slack-Zeit x Zeiteinheiten lang, dann wird die Rechenzeit nicht bedingungslos für die Dauer von x an Scheduling-Klassen mit niedrigerer Klassen-Priorität abgegeben. Stattdessen wird die Zeitspanne x in Perioden p_i eingeteilt.

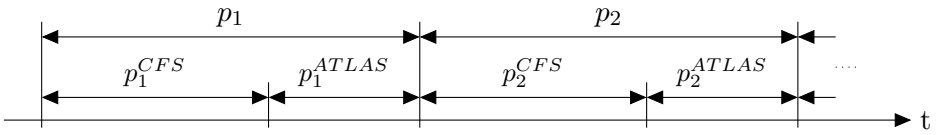


Abbildung 3.3: Einteilung der Slack-Zeit in Perioden

Eine Periode p_i besteht dabei wie in Abbildung 3.3 dargestellt aus zwei Teilen: p_i^{CFS} und p_i^{ATLAS}. Innerhalb von p_i^{CFS} erfolgt die Abgabe von Rechenzeit an CFS. Gibt es hier keine bereiten Threads, erhalten die Scheduling-Klassen mit niedrigerer Klassen-Priorität die zugewiesene Zeit. Innerhalb von p_i^{ATLAS} wird Rechenzeit gezielt an ATLAS-Fallback weitergegeben, um einen Vorlauf zu erzwingen. Somit handelt es sich nicht mehr um eine strikte Anordnung der Scheduling-Klassen mit festgelegten Klassen-Prioritäten. Durch die gezielte Zuweisung der Rechenzeit von ATLAS-LRT stehen CFS und ATLAS-Fallback deshalb in Abbildung 3.3 teilweise nebeneinander.

Das Problem des Entwurfs besteht in der Festlegung der Dauer einer Periode. Wie im Abschnitt 2.4.2 beschrieben, arbeitet CFS intern selbst mit einer Periode variabler Länge. Die Länge von p_i^{CFS} wird zu Beginn einer neuen Periode p_i deshalb mit der Dauer der CFS-Periode initialisiert, die abhängig von der Anzahl der bereiten Threads in CFS ist. Damit wird weitestgehend sichergestellt, dass jeder bereite Thread in CFS innerhalb von p_i^{CFS} die CPU einmal nutzen kann.

Schwierig ist die Festlegung der Periodenlänge für p_i^{ATLAS}. Da hierfür nur willkürlich eine Zeit festgelegt werden konnte, wurde die Periodenlänge über eine Schnittstelle in `/proc/sys/kernel/` umgesetzt, die zur Laufzeit dynamisch konfigurierbar war. Die Voreinstellung lag bei 10 ms.

Die Schwierigkeit der Festlegung der Periodenlängen ist der Hauptgrund dafür, dass dieser Entwurf nicht weiterentwickelt wurde. Ein weiterer Grund ist, dass CFS in den Zeitabschnitten p_i^{ATLAS} keine Rechenzeit nutzen kann. Damit ist die Interaktivität des Systems negativ beeinflusst, obwohl lediglich ein Vorlauf für Threads erzeugt wird.

Im folgenden Abschnitt wird ein weiterer Lösungsansatz beschrieben, der diese Limitierungen vermeidet.

3.3.3 Lösungsansatz bei Unterschätzung der Ausführungszeit

Als Ausgangspunkt dient die im Abschnitt 3.3.1 beschriebene Scheduling-Schicht basierend auf LRT. Um bei einer geringen Unterschätzung der Ausführungszeit eines Jobs zu vermeiden, dass sofort auch die Deadline des Jobs überschritten wird, können die Lücken im Ablaufplan verwendet werden: Jobs erhalten dort bereits Rechenzeit, die nicht gezählt wird. Durch diesen Ansatz erhalten Jobs einen *Vorlauf* und die angegebene Abarbeitungszeit kann in Abhängigkeit der Systemlast überschritten werden, ohne die Garantien für andere Threads entsprechend Anforderung 1 zu beeinflussen.

Neben der Nutzung der Lücken zur Erzeugung eines Vorlaufs soll die nicht verplante Zeit auch den Best-Effort-Anwendungen in CFS zur Verfügung stehen. Es gilt, diese zwei unterschiedlichen Forderungen zu vereinen.

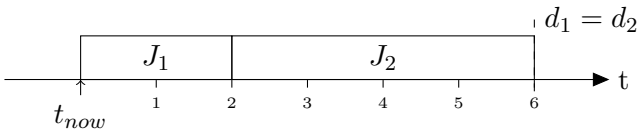
Die Summe der Lücken zwischen zwei Zeitpunkten im Ablaufplan entspricht der Slack-Zeit. Ist diese zwischen dem aktuellen Zeitpunkt und dem Job-Beginn des ersten bereiten Threads im Ablaufplan größer als 0, dann besteht keine Notwendigkeit zur Ausführung eines Threads durch den ATLAS-Scheduler. In diesem Fall wird Rechenzeit an unterliegende Scheduling-Schichten abgegeben. Um dabei einen zusätzlichen Vorlauf zu erreichen, wird aus der Menge der bereiten Threads der Thread ausgewählt, dessen Job die nächste Deadline hat. Dieser Thread wird nach CFS verschoben und erhält dort zusammen mit anderen Best-Effort-Threads für die Dauer der Slack-Zeit Rechenzeit. Ist keine weitere Slack-Zeit mehr verfügbar, wird der Aufruf des Schedulers erzwungen und der nach CFS verschobene Thread wieder vom ATLAS-Scheduler eingeplant.

Zusätzliche Probleme ergeben sich beim Blockieren des nach CFS verschobenen Threads: In diesem Fall muss der nächste bereite Thread des ATLAS-Schedulers nach CFS verschoben werden, da andernfalls kein Thread mehr Vorlauf erhält. Durch die Wiederholung dieses Vorgangs beim Blockieren verschobener Threads ist sichergestellt, dass die Lücken des Ablaufplans zur Schaffung von Vorlauf verwendet werden, solange es bereite Threads innerhalb des ATLAS-Schedulers gibt.

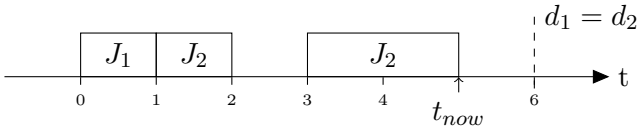
3.3.4 Überschreitung der Ausführungszeit eines Jobs trotz Vorlauf

Der im letzten Abschnitt beschriebene Ansatz zur Schaffung von Vorlauf funktioniert nur, wenn die Slack-Zeit zwischen dem aktuellen Zeitpunkt und dem Startzeitpunkt des ersten Jobs im Ablaufplan größer als 0 ist. In jedem Fall ist es möglich, dass ein Thread, sei es mit oder ohne Vorlauf, die angegebene Ausführungszeit des Jobs überschreitet.

Bei der Überschreitung der zugewiesenen Ausführungszeit muss der Job abgebrochen werden, wenn sich im Ablaufplan direkt dahinter ein weiterer Job befindet. Andernfalls



(a) aufgestellter Ablaufplan



(b) tatsächlicher Ablaufplan zum Zeitpunkt $t = 5$

Abbildung 3.4: Aufgestellter Ablaufplan gegenüber tatsächlichen Ablaufplan beim Blockieren eines Jobs

ist die Durchsetzung weicher Echtzeitgarantien gemäß Anforderung 1 nicht möglich, weil der nächste Thread nicht ausreichend viel Rechenzeit bis zu seiner Deadline erhält.

Was passiert mit dem Thread, der seine Ausführungszeit überschritten hat? Eine Möglichkeit ist der Abbruch des Jobs und somit die Beendigung des Threads. Dieser Ansatz ist nicht praktikabel, da die Ausführungszeit in Anforderung 7 explizit als eine Schätzung festgelegt wurde. Verwendet beispielsweise ein Video-Player für die Dekodierung von Bildern einen eigenen Thread und modelliert die Arbeit mittels Jobs unter Verwendung von ATLAS, dann kann die vom Estimator geschätzte Ausführungszeit für das Dekodieren eines Bildes zu kurz sein. Folglich überschreitet der Job die gemeldete Ausführungszeit. Bei einer einmaligen Unterschätzung resultiert dies im schlimmsten Fall in einem zu spät angezeigten Bild. Der Benutzer wird dies tolerieren, nicht aber den Abbruch des Videos.

Die bessere Lösungsstrategie gegenüber dem Abbruch des Threads ist die Einplanung als Best-Effort-Thread durch CFS. Durch den Wechsel des zuständigen Schedulers erhält der Thread weiterhin Rechenzeit und kann den zugewiesenen Job weiter abarbeiten und abschließen. Eine Rückkehr zum ATLAS-Scheduler ist möglich, wenn die Deadline des nächsten abzuarbeitenden Jobs nicht in der Vergangenheit liegt. In diesem Fall ist der Job im Ablaufplan eingeplant und der Thread, der vormals die angegebene Ausführungszeit eines Jobs überschritten hat, wird wieder zu einem regulär eingeplanten Thread des ATLAS-Schedulers.

3.3.5 Blockierende Threads

Anforderung 6 beschreibt, dass Threads jederzeit blockieren können. Der nach LRT aufgestellte Ablaufplan kann somit nicht 1:1 umgesetzt werden, da ein Thread zum Startzeitpunkt eines neuen Jobs im Ablaufplan blockiert sein kann.

Abbildung 3.4(a) zeigt ein Beispiel für einen nach LRT aufgestellten Ablaufplan zweier Jobs J_1 und J_2. Die Ausführungszeit von J_1 beträgt 2 Zeiteinheiten, die von J_2 4 Zeiteinheiten. Die Deadlines beider Jobs liegen bei $t = 6$. Da die Summe der Ausführungszeiten gleich der noch zur Verfügung stehenden Zeit bis zur Deadline ist, beträgt die Slack-Zeit

gleich 0. Folglich wird keine Zeit an niedrigere Scheduling-Schichten abgegeben und der J_1 zugewiesene Thread wird wie in Abbildung 3.4(b) dargestellt zum Zeitpunkt $t = 0$ gestartet. Nach einer Zeiteinheit blockiert der Thread, möglicherweise ist es für die mit dem Job verknüpfte Aufgabe noch zu früh. Der ATLAS-Scheduler muss nun gemäß Anforderung 1, der Durchsetzung weicher Echtzeitgarantien, den Thread auswählen, der Job J_2 abarbeitet. Andernfalls ist die zur Verfügung stehende Rechenzeit später nicht ausreichend, wenn die Ausführung von J_1 fortgesetzt wird. Analog zum ersten Thread blockiert auch der zweite Thread nach einer Zeiteinheit zum Zeitpunkt $t = 2$. Der ATLAS-Scheduler findet nun keine weiteren bereiten Threads mehr vor, folglich kann er nur Zeit an unterliegende Scheduling-Schichten abgeben.

Zum Zeitpunkt $t = 3$ setzt der J_2 abarbeitende Thread seine Berechnungen fort. Die verbleibende eingeplante Rechenzeit beträgt genau 3 Zeiteinheiten und eine Fertigstellung bis zur zugewiesenen Deadline ist möglich. Zum Zeitpunkt $t = 5$ wird J_1 bereit. Wenn diesem Thread jetzt sofort Rechenzeit zur Verfügung gestellt wird, kann die Deadline eingehalten werden, allerdings führt dies dazu, dass J_2 nicht fertiggestellt werden kann. Wird andererseits nicht sofort mit der Abarbeitung von J_1 begonnen, kann die Deadline des Jobs nicht eingehalten werden. Zusammenfassend bedeutet dies also aus Sicht des Schedulers zum Zeitpunkt $t = 5$, dass unabhängig von der nächsten Scheduling-Entscheidung immer ein Job die zugewiesene Deadline nicht einhalten kann.

Blockierende Threads können also zu Problemen führen, weil trotz lückenlosen Ablaufplans keine bereiten Threads vorhanden sind und deshalb Rechenzeit an niedrigere Schichten abgegeben werden muss. Als Konsequenz entsteht eine Überlastsituation, in der die noch zur Verfügung stehende Rechenzeit unzureichend ist und Threads ihre Deadline verpassen, obwohl sie sich gemäß der Spezifikation des Jobs verhalten. Auf einer CPU kann diese Situation nicht durch die Anpassung der Scheduling-Strategie aufgelöst werden, da die im Zeitabschnitt angeforderte Rechenzeit die zur Verfügung stehende Rechenzeit übersteigt.

Entstehende Überlastsituationen durch blockierende Threads auf einer einzelnen CPU können durch die Migration von Threads auf andere CPUs aufgelöst werden, auch wenn dies im Abschnitt 3.1.2 explizit ausgeschlossen wurde. Trotzdem tritt das Problem auch ohne die getroffene Vereinfachung auf: Besitzen andere Prozessoren keine freien Rechenkapazitäten, dann muss das Vorgehen des ATLAS-Schedulers definiert werden. Konkret muss festgelegt werden, was mit einem Job passiert, dessen Deadline überschritten wird, obwohl noch verbleibende Rechenzeit zur Verfügung steht.

Der gewählte Lösungsansatz besteht im Priorisieren der Threads gegenüber Best-Effort-Threads in CFS. Dieser Ansatz ist mittels einer weiteren Scheduling-Klasse mit der Bezeichnung *ATLAS-Recover* umsetzbar. Es ergibt sich das in Abbildung 3.5 dargestellte Schichten-Design von Schedulern des Linux-Kerns.

Bei der Überschreitung der Deadline eines Jobs wird der betreffende Thread an ATLAS-Recover übergeben, wenn noch Ausführungszeit für den Job reserviert ist. Befinden sich Lücken in dem für ATLAS-LRT aufgestellten Ablaufplan, so erhält ATLAS-Recover zuerst Rechenzeit. Dadurch können Threads die Abarbeitung der Jobs für die restliche Ausführungszeit fortsetzen. Wird diese überschritten, wird der Thread wie schon im Abschnitt 3.3.4 für ATLAS-LRT beschrieben, nach CFS übergeben. Gibt es keine bereiten Threads in ATLAS-Recover, dann erhält CFS Rechenzeit.

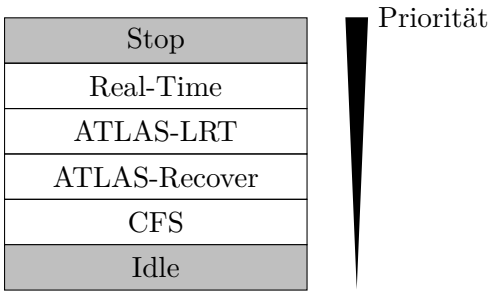


Abbildung 3.5: Anordnung der Scheduling-Klassen in der Schichtenarchitektur von Schedulern des Linux-Kerns

Da sich auch in ATLAS-Recover zu einem Zeitpunkt mehrere bereite Threads befinden können, muss eine Strategie für die Auswahl des nächsten Threads festgelegt werden. Bei einer Scheduling-Entscheidung wird dazu immer der Thread ausgewählt, dessen Job die früheste Deadline hat. Durch dieses Vorgehen erhalten Threads Vorrang, deren Deadline schon am längsten vergangen ist und die somit am dringendsten die reservierte Rechenzeit benötigen.

3.4 Ablaufplan auf Basis von LRT

Anhand der für den ATLAS-Scheduler übermittelten Jobs wird gemäß Anforderung 2 ein Ablaufplan zur Laufzeit erstellt, der die Scheduling-Entscheidungen des ATLAS-Schedulers beeinflusst. Dem Ablaufplan kommt deshalb eine wichtige Bedeutung zu, die in diesem Abschnitt genauer beschrieben wird.

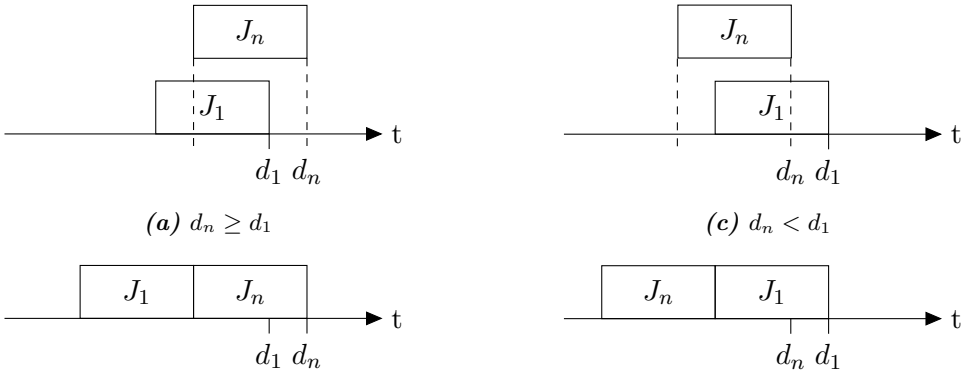
3.4.1 Bedeutung

Innerhalb des ATLAS-Schedulers wird der Ablaufplan wie folgt verwendet:

1. Die Erkennung von Überlast bei der Übermittlung neuer Jobs erfolgt mit Hilfe des Ablaufplans. Ist die verfügbare Zeit kleiner als die angeforderte Zeit, wird dies bei der Einplanung des neuen Jobs erkannt. Eine detaillierte Betrachtung der Behandlung von Überlastsituationen wird im Abschnitt 3.6 beschrieben.

2. Bei einer Scheduling-Entscheidung von ATLAS-LRT dient der Ablaufplan zur Bestimmung der Slack-Zeit wie im Abschnitt 2.5.3 beschrieben. Ist die Slack-Zeit größer als 0, dann wird Rechenzeit an unterliegende Schichten abgegeben.

3. Bei einer Slack-Zeit gleich 0 muss ein bereiter Thread der CPU zugeteilt werden, damit die in Anforderung 1 festgehaltenen weichen Echtzeitgarantien eingehalten werden können. Die Bestimmung des Threads, welcher als nächstes der CPU zugeteilt wird, basiert auf dem Ablaufplan.



(a) $d_n \geq d_1$ ***(c)*** $d_n < d_1$

(b) Ergebnis $d_n \geq d_1$ ***(d)*** Ergebnis $d_n < d_1$

Abbildung 3.6: Veränderung des Ablaufplans durch Einfügen eines neuen Jobs

3.4.2 Grundlagen

Die Erstellung des Ablaufplans erfolgt wie im Abschnitt 3.3.1 beschrieben auf Grundlage von LRT. Im Gegensatz zu LRT, welches im Abschnitt 2.5.2 des Grundlagenkapitels vorgestellt wurde, basiert die Erstellung des Ablaufplans auf nicht unterbrechbaren Jobs. Die Betrachtung der Jobs als eine *unteilbare* Einheit ergibt eine einfachere Implementierung. Außerdem ist eine Unterscheidung von teilbaren und unteilbaren Jobs bei der Erstellung des Ablaufplans aus zwei Gründen unmaßgeblich:

1. Die Berechnung der Slack-Zeit liefert dasselbe Ergebnis, da sich die Summe der Lücken zwischen zwei definierten Zeitpunkten nicht ändert.

2. Eine Umsetzung des Ablaufplans kann aufgrund von blockierenden Threads weder mit teilbaren noch mit unteilbaren Jobs 1:1 erfolgen.

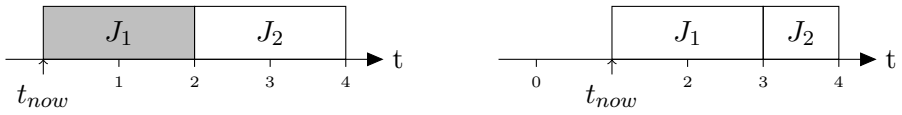
Ein Ablaufplan basierend auf unterbrechbaren Jobs erhöht zusätzlich den Aufwand für die Operationen *Einfügen*, *Aktualisieren* und *Löschen* von Jobs, die in den nächsten Abschnitten vorgestellt werden.

3.4.3 Einfügen

Bei der Übermittlung eines Jobs mit Ausführungszeit e und absoluter Deadline d an den ATLAS-Scheduler wird dieser in den Ablaufplan eingefügt. Im einfachsten Fall ist der Ablaufplan im Intervall $[d - e, d]$ noch leer und der Job kann ohne weiteren Aufwand eingefügt werden. Der Aufwand der Operation ergibt sich in diesem Fall durch das Finden der richtigen Position des Jobs innerhalb der verwendeten Datenstruktur sowie dem eigentlichen Einfügen.

Kommt es bei der Einplanung des neuen Jobs J_n zu einer Kollision, d.h. der Ablaufplan im Intervall $[d - e, d]$ ist nicht leer, dann muss die Kollision durch die Verschiebung von Jobs aufgelöst werden. Abbildung 3.6 zeigt zwei Beispiele.

Ist die Deadline des neuen Jobs größer oder gleich der Deadline eines bereits eingeplanten Jobs und besteht zwischen beiden Jobs eine Kollision wie in Abbildung 3.6(a)



(a) Aufgestellter Ablaufplan zum Zeitpunkt $t = 0$. Job J_1 ist blockiert.

(b) Ablaufplan nach Aktualisierung von J_2 zum Zeitpunkt $t = 1$.

Abbildung 3.7: Aktualisierung des Ablaufplans nach Ausführung eines Jobs

dargestellt, dann muss der bereits eingeplante Job verschoben werden. Dies geschieht, indem J_1 wie in Abbildung 3.6(b) dargestellt eher eingeplant wird.

Ist die Deadline des neuen Jobs kleiner als die Deadline eines bereits eingeplanten Jobs und besteht zwischen beiden Jobs eine Kollision wie in in Abbildung 3.6(c) dargestellt, dann wird der neue Job nach vorn verschoben, so dass die *effektive Deadline* mit dem Startzeitpunkt des bereits eingeplanten Jobs zusammenfällt. Es ergibt sich der in Abbildung 3.6(d) dargestellte Ablaufplan.

Wenn ein Job im Ablaufplan verschoben wird, dann muss geprüft werden, ob durch die Verschiebung eine neue Kollision mit weiteren Jobs entstanden ist. Durch dieses Vorgehen kann es zu weitreichenden Änderungen im gesamten Ablaufplan ähnlich einer Kettenreaktion kommen. Die Komplexität beim Einfügen eines neuen Jobs in den Ablaufplan setzt sich deshalb zusammen aus dem Finden der richtigen Position sowie dem Verschieben von möglicherweise allen n bereits eingeplanten Jobs. Die Konsequenz ist, dass die Komplexität im ungünstigsten Fall bei $O(n)$ liegt.

Die resultierende Eigenschaft durch das beschriebene Vorgehen ist, dass die Jobs innerhalb des Ablaufplans aufsteigend nach ihrer Deadline sortiert sind und es keine Kollisionen gibt. Wenn bei der Einplanung eines neuen Jobs andere Jobs nach vorn verschoben werden, dann ist es möglich, dass deren Deadlines nach der Verschiebung in der Vergangenheit liegen. In diesem Fall ist eine Überlastsituation entstanden, die im Abschnitt 3.6 genauer betrachtet wird.

3.4.4 Aktualisieren

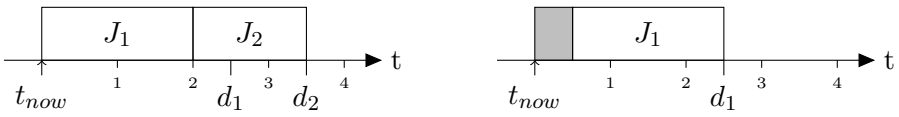
Das Aktualisieren des Ablaufplans ist bei einem Scheduling-Ereignis wie dem Blockieren oder Aufwachen eines Threads oder bei der Fertigstellung eines Jobs erforderlich. In beiden Fällen muss der Ablaufplan angepasst werden und die Zeit, die seit dem Beginn der Ausführung des laufenden Threads vergangen ist, erfasst werden.

Ein Beispiel dazu ist in Abbildung 3.7 dargestellt. Der aufgestellte Ablaufplan in Abbildung 3.7(a) zeigt zwei Jobs J_1 und J_2. Ist J_1 zum Zeitpunkt $t = 0$ blockiert, dann wird J_2 für die Abarbeitung ausgewählt. Nach einer Zeiteinheit wird J_1 bereit und unterbricht aufgrund der früheren Einplanung J_2. In dieser Situation muss die von J_2 genutzte Rechenzeit abgerechnet werden, indem der Job um eine Zeiteinheit gekürzt wird. Es ergibt sich die in Abbildung 3.7(b) dargestellt Situation.

Das Beispiel zeigt, dass auch bei der Aktualisierung eines Jobs im ungünstigsten Fall alle vorher platzierten Jobs verschoben werden müssen. Deshalb ist die Komplexität der Operation wie beim Einfügen eines Jobs gleich $O(n)$.

3.4.5 Löschen

Bei der Fertigstellung eines Jobs muss dieser wieder aus dem Ablaufplan entfernt werden. In der Regel erfolgt vorher eine Aktualisierung des Jobs entsprechend der verwendeten Rechenzeit. Auch beim Löschen eines Jobs kommt es im ungünstigsten Fall zur Verschiebung aller vorher eingeplanten Jobs, so dass die Komplexität der Operation $O(n)$ entspricht. Abbildung 3.8 zeigt ein Beispiel.



(a) Ablaufplan vorm Löschen von J_2.

(b) Angepasster Ablaufplan nach dem Löschen von J_2.

Abbildung 3.8: Löschen eines Jobs aus dem Ablaufplan

In Abbildung 3.8(a) ist zu erkennen, dass Job J_1 aufgrund von Job J_2 eher eingeplant wurde, da die effektive Deadline des Jobs nicht mit der Deadline d_1 übereinstimmt. Wie in Abbildung 3.8(b) gezeigt, wird J_1 nach der Entfernung von Job J_2 bis zur zugewiesenen Deadline d_1 verschoben. Die größtmögliche Verschiebung ist immer abhängig vom Start des nächsten Jobs und der Deadline des Jobs, der verschoben wird. Beim Löschen von Jobs aus dem Ablaufplan, entsteht eine zusätzliche Lücke, wenn die restliche Ausführungszeit des gelöschten Jobs größer als 0 ist. Die entstandene Lücke in Abbildung 3.8(b) wurde grau markiert und entspricht der Slack-Zeit zwischen den Zeitpunkten t_{now} und d_1.

3.5 Auswahl des nächsten Threads in ATLAS-LRT

Eine Scheduling-Entscheidung erfolgt durch den Aufruf der Funktion `pick_next_task` durch das Scheduling-Framework. Ist der Rückgabewert der Funktion ein `NULL`-Zeiger, dann erfolgt der entsprechende Aufruf der Funktion in der Scheduling-Schicht mit der nächstniedrigeren Klassen-Priorität. Wird ein Zeiger auf einen Thread zurück gegeben, dann wird dieser Thread für die weitere Ausführung ausgewählt.

Bei einer Scheduling-Entscheidung wird als erstes geprüft, ob es in ATLAS-LRT bereite Threads gibt. Ist dies nicht der Fall, wird ein `NULL`-Zeiger zurück gegeben und die Abarbeitung der Funktion endet.

Wenn bereite Threads vorhanden sind, wird geprüft, ob sich der ATLAS-Scheduler im *Slack-Modus* befindet. Dieser Modus kann bei einem früheren Aufruf des Schedulers aktiviert worden sein und dient zur Kennzeichnung von verbleibender Slack-Zeit. Ist der Modus aktiviert, dann besteht keine Notwendigkeit für die Ausführung eines bereiten Threads. Demzufolge liefert die Funktion einen `NULL`-Zeiger zurück.

Befindet sich der Scheduler nicht im Slack-Modus, wird die Slack-Zeit basierend auf dem nach LRT aufgestellten Ablaufplan bestimmt. Übersteigt diese einen Mindestwert, der dynamisch zur Laufzeit über die Schnittstelle `/proc/sys/kernel/min_slack_time` konfigurierbar und mit 1 ms initialisiert ist, erfolgt die Programmierung eines Timers für

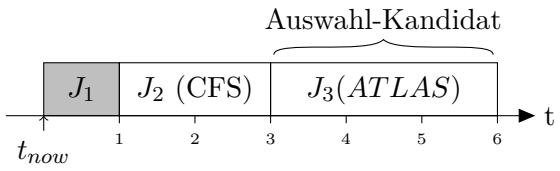


Abbildung 3.9: Ablaufplan zur Auswahl des nächsten Threads

die Dauer der Slack-Zeit. Außerdem wird der Slack-Modus aktiviert und ein `NULL`-Zeiger zurückgegeben.

Die Einführung eines Mindestwerts für die Slack-Zeit dient zur Verhinderung der Abgabe von Rechenzeit für sehr kurze Zeitabschnitte an niedrigere Schichten. Damit wird in dieser Situation das sogenannte *Over-Scheduling*, der mehrmalige Aufruf des Schedulers innerhalb sehr kurzer Zeitabschnitte, verhindert. Bei Ausführung der Timer-Routine wird der Slack-Modus deaktiviert und der der Scheduler aufgerufen.

Ist die Slack-Zeit kleiner als der gesetzte Mindestwert, wird ein zur Verwaltung der bereiten Threads genutzter Rot-Schwarz-Baum [Bay72, GS78] zur Bestimmung des Threads mit der nächsten Deadline und dessen Job verwendet. Als Sortierkriterium für den Baum dienen die Deadlines der Jobs, die von den jeweiligen Threads abgearbeitet werden. Der Rot-Schwarz-Baum wird benutzt, da eine zum Ablaufplan unabhängige Datenstruktur zur Verwaltung der bereiten Threads für die Implementierung notwendig ist.

Es ergeben sich abhängig von der Position des ermittelten Jobs im aufgestellten Ablaufplan verschiedene Möglichkeiten, weil der ATLAS-Scheduler durch unterschiedliche Scheduling-Schichten realisiert wird: Ist der bestimmte Job im Ablaufplan an vorderster Stelle, dann wird dieser ausgewählt. Andernfalls werden die vorher platzierten Jobs der Reihe nach und somit aufsteigend nach ihrer Deadline untersucht: Ist der jeweilige Job blockiert, wird zum nächsten übergegangen, bis der erste bereite Job gefunden wird. Ist der zugehörige Thread derselbe wie der mit Hilfe des Rot-Schwarz-Baums bestimmte Thread, dann wird dieser für die Ausführung ausgewählt. Handelt es sich um einen anderen Thread, dann muss dieser von einer anderen Scheduling-Klasse eingeplant sein, da sonst die Auswahl des Rot-Schwarz-Baums genau diesen Thread gewählt hätte. Da für den Job Zeit im Ablaufplan vorgesehen ist, wird der Scheduler des Threads wieder auf ATLAS-LRT gesetzt und der Thread für die Ausführung gewählt. Zu beachten ist hierbei, dass der Thread zu diesem Zeitpunkt an einem alten Job arbeitet, die Zeit im Ablaufplan aber bereits für den neuen Job abgerechnet wird.

Abbildung 3.9 zeigt ein Beispiel für die Auswahl des nächsten Jobs, wenn die Slack-Zeit gleich 0 ist. Mittels des Rot-Schwarz-Baums wurde Job J_3 als erster bereiter Job in ATLAS-LRT ausgewählt. Ausgehend vom Beginn des Ablaufplans werden die einzelnen Jobs geprüft: Job J_1 ist blockiert und somit ist die zugewiesene Scheduling-Klasse ohne Bedeutung. Job J_2 ist bereit und die zugewiesene Scheduling-Klasse ist CFS. Der abarbeitende Thread ist somit noch mit der Ausführung eines älteren Jobs beschäftigt, da er sonst von ATLAS-LRT eingeplant worden wäre. Da im Ablaufplan Zeit für J_2 reserviert ist, wird der Scheduler des abarbeitenden Threads auf ATLAS-LRT gesetzt und der Thread für die Ausführung ausgewählt.

3.6 Überlastsituationen bei Job-Einplanungen im Ablaufplan

Bei der Einplanung von neuen Jobs kommt es zu Überlastsituationen, wenn die angeforderte Rechenzeit eines Jobs größer als die noch zur Verfügung stehende Rechenzeit ist. Gemäß Anforderung 4 müssen diese Situationen erkannt und durch eine geeignete Strategie aufgelöst werden. Die Zurückweisung von Jobs ist gemäß Anforderung 5 unzulässig, da die ATLAS-Lib die bedingungslose Annahme aller Jobs durch den ATLAS-Scheduler voraussetzt.

3.6.1 Erkennung von Überlast

Die Erkennung von Überlast erfolgt wie im Abschnitt 2.5.3 beschrieben durch die Berechnung der Slack-Zeit. Dabei wird die Summe der Lücken im Ablaufplan zwischen dem aktuellen Zeitpunkt und der Deadline des neuen Jobs bestimmt. Ist die noch zur Verfügung stehende Zeit t_a größer oder gleich der angegebenen Ausführungszeit e des neuen Jobs, dann ist eine Einplanung ohne Einschränkung möglich und es liegt keine Überlastsituation vor. Ist t_a dagegen kleiner als e, dann muss durch einen geeigneten Lösungsansatz die Überlastsituation aufgelöst werden.

3.6.2 Behandlung von Überlastsituationen

Es kann zwischen zwei verschiedenen Strategien zur Behandlung von Überlastsituationen unterschieden werden. Einerseits ist eine Einplanung möglich, indem für den Job nur eine verkürzte Ausführungszeit angenommen wird. Andererseits kann der ATLAS-Scheduler, wenn die Anwendung diesem vorher geeignete Informationen zur Verfügung stellt, bereits eingeplante Jobs im Ablaufplan kürzen und damit die zur Verfügung stehende Rechenzeit erhöhen. Die Ansätze werden in den beiden folgenden Abschnitten detailliert beschrieben.

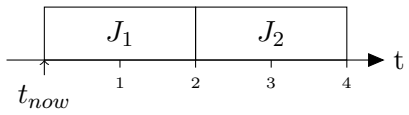
Einplanung mit verkürzter Ausführungszeit

Der einfachste Ansatz ist die Einplanung des Jobs mit einer Ausführungszeit von t_a.

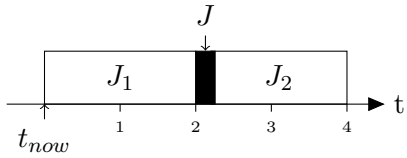
Ist t_a größer als 0, dann wird der Job mit einer kürzeren als der angegebenen Ausführungszeit eingeplant. Ist t_a gleich 0, dann wird anstelle des Jobs nur ein Platzhalter eingetragen. In beiden Fällen muss der ATLAS-Scheduler die ATLAS-Lib über die verkürzte Einplanung des Jobs J informieren, weil keine Garantie bezüglich der angeforderten Rechenzeit möglich ist. Vorstellbar ist die Weiterleitung der Information an die Anwendung, so dass diese Jobs beispielsweise wieder löscht oder je nach Anwendungsgebiet die Qualität senkt, um die benötigte Rechenzeit pro Job zu verringern.

Beendet der Job, der im Ablaufplan vor J eingeplant ist, seine Berechnungen vorzeitig, dann kann dessen restliche Ausführungszeit für J genutzt werden. Haben alle vorherigen Jobs die komplette Ausführungszeit verwendet, so dass J nur als Platzhalter im Ablaufplan vorhanden ist, dann wird J bei der Auswahl des nächsten Threads nicht berücksichtigt. Die Ausführung kann dann nur als Best-Effort-Thread durch CFS erfolgen.

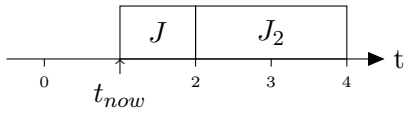
Abbildung 3.10 zeigt eine beispielhafte Abarbeitung des Ablaufplans mit einem Job, der nur als Platzhalter eingeplant ist.



(a) Ablaufplan vor Job-Einplanung



(b) Ablaufplan nach Job-Einplanung



(c) Fertigstellung von J_1

Abbildung 3.10: Einplanung eines Jobs mit verkürzter Ausführungszeit

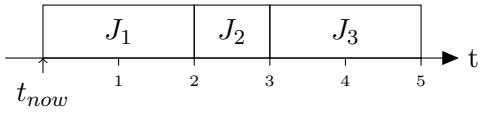
Abbildung 3.10(a) zeigt den Ablaufplan vor der Einplanung eines neuen Jobs J mit einer Ausführungszeit von 2 Zeiteinheiten und einer Deadline bei $t = 3$. Da keine Lücken im Ablaufplan enthalten sind und damit keine Rechenzeit mehr zur Verfügung steht, wird, wie in Abbildung 3.10(b) dargestellt, lediglich ein Platzhalter für J eingeplant.

Beispielsweise aufgrund einer zu hohen Schätzung der Ausführungszeit beendet J_1 bereits nach einer Zeiteinheit die Ausführung. Die entstehende Lücke wird nun nicht als Slack-Zeit, sondern für die Einplanung von J genutzt. Abbildung 3.10(c) zeigt den aktuellen Ablaufplan. Damit erhält J eine zugesicherte Ausführungszeit von einer Zeiteinheit. Dies entspricht der Hälfte der ursprünglich angeforderten Rechenzeit.

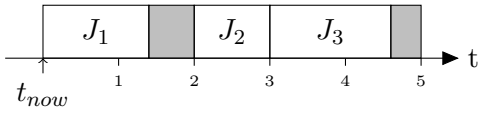
Das Vorgehen erfordert beim vorzeitigen Beenden eines Jobs die Durchsuchung des Ablaufplans, um den ersten Job beziehungsweise die ersten Jobs mit verkürzt eingeplanter Rechenzeit zu finden und entsprechend neu einzufügen. Im ungünstigsten Fall befindet sich der letzte neu einzuplanende Job genau am Ende des Ablaufplans. Die erforderliche Neueinplanung mit verlängerter Rechenzeit erfordert die Verschiebung aller vorher eingeplanten Jobs. Die resultierende Komplexität ist $O(n)$.

Einplanung mit Kooperation der Anwendungen

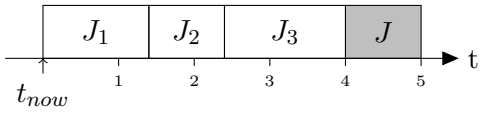
Anstatt die eingeplante Rechenzeit von Jobs sofort zu kürzen, ist ein alternativer Ansatz vielversprechend, der allerdings die Erweiterung der im Abschnitt 2.6.2 beschriebenen Primitive erfordert: Besteht für eine Anwendung beziehungsweise für die ATLAS-Lib die Möglichkeit, den ATLAS-Scheduler über mögliches Kürzungspotential der Rechenzeit von Jobs zu informieren, dann kann dieser in entsprechenden Überlastsituationen bereits eingeplante Jobs kürzen und damit die zur Verfügung stehende Rechenzeit erhöhen.



(a) Ablaufplan vor Job-Einplanung



(b) Ablaufplan nach der Kürzung von Jobs



(c) Ablaufplan nach Einplanung des neuen Jobs

Abbildung 3.11: Einplanung eines Jobs mit Kooperation der Anwendungen bei Überlast

Abbildung 3.11 zeigt ein Beispiel mit drei Jobs. J_1 besitzt ein Kürzungspotential von 50%, J_2 darf nicht gekürzt werden und J_3 darf um höchstens 20% gekürzt werden.

Der aufgestellte Ablaufplan ohne Kürzungen ist in Abbildung 3.11(a) dargestellt. Wird nun ein neuer Job J mit einer gemeldeten Ausführungszeit von einer Zeiteinheit übermittelt, dann müssen Jobs gekürzt werden, um die Einplanung zu ermöglichen. Der ATLAS-Scheduler nutzt dabei die mit dem Job verknüpften Informationen und kürzt J_1 um 30% auf eine Ausführungszeit von noch 1,4 Zeiteinheiten. J_3 wird um das erlaubte Kürzungspotential auf 1,6 Zeiteinheiten gekürzt. Es ergibt sich der in Abbildung 3.11(b) dargestellte Ablaufplan mit grau eingefärbten Lücken. Nach der Verschiebung der Jobs zur Zusammenführung der entstandenen Lücken kann J mit der gemeldeten Ausführungszeit, wie in Abbildung 3.11(c) gezeigt, erfolgreich eingeplant werden.

Die Einplanung von Jobs mit Kooperation der Anwendungen bei Überlast erlaubt eine dynamischere Einplanung. Die Vorteile bedeuten aber einen nicht zu vernachlässigenden Mehraufwand bei der Implementierung der Einplanung sowie zusätzliche Mechanismen zur Realisierung des Informationsaustauschs zwischen Anwendung beziehungsweise ATLAS-Lib und dem ATLAS-Scheduler in beide Richtungen: Einerseits muss das Primitiv zur Job-Übermittlung erweitert werden, um auch das Kürzungspotential eines Jobs zu übermitteln, andererseits muss eine Möglichkeit bereitgestellt werden, die dem ATLAS-Scheduler erlaubt, Nachrichten über durchgeführte Job-Kürzungen weiterzugeben.

3.7 Ansatz für Scheduling auf Mehrprozessorsystemen

Die Erstellung des Ablaufplans sowie die spätere Einplanung von Jobs erfolgt lokal je Prozessor. Dies bedeutet, dass in Mehrprozessorsystemen so viele *Instanzen* des ATLAS-Schedulers verwaltet werden, wie es Prozessoren im System gibt. Die einzelnen Instanzen arbeiten mit CPU-lokalen Ressourcen und sind damit voneinander unabhängig.

Wird wie im Abschnitt 3.6.1 beschrieben eine Überlastsituation bei der Erstellung des Ablaufplans erkannt, dann kann die Einplanung eines Jobs auch auf einer anderen CPU erfolgen. Dabei muss beachtet werden, dass es zu keiner *Kollision* von eingeplanten Jobs desselben Threads auf unterschiedlichen CPUs kommt. Konkret muss die folgende Bedingung vor der Einplanung gelten: Die Zeitintervalle für die Abarbeitung zweier beliebiger, paarweise verschiedener Jobs, die von demselben Thread abgearbeitet werden, sind disjunkt. Gilt diese Bedingung ohne Einschränkungen für alle Threads, die den ATLAS-Scheduler nutzen, dann bezeichne ich den Ablaufplan auch als *konsistenten Ablaufplan.*

Ein *inkonsistenter Ablaufplan* dagegen führt dazu, dass noch während der Ausführung eines Jobs auf einer CPU eine andere Instanz des ATLAS-Schedulers versucht, den entsprechenden Thread auf die eigene CPU zu migrieren, um mit der Abarbeitung des nächsten Jobs zu beginnen. Als Folge erhält der Thread nicht die angeforderte Zeit pro Job.

Erstellung des konsistenten Ablaufplans

Die Erstellung eines konsistenten Ablaufplans kann erzwungen werden, wenn bei jeder Einplanung sowie Verschiebung eines Jobs auf eine Kollisionen mit allen anderen CPUs geprüft wird. Liegt keine Kollision vor, dann ist die Einplanung des Jobs im entsprechenden Zeitintervall möglich. Um zu verhindern, dass auf Systemen mit sehr vielen Prozessoren alle möglichen CPUs bei der Einplanung berücksichtigt werden, kann die Menge der möglichen CPUs eines Threads beschränkt werden. Damit sinkt der Einplanungsaufwand pro Job, aber es kann zu der im Abschnitt 3.6 beschriebenen Überlastbehandlung kommen, obwohl möglicherweise noch freie Rechenkapazitäten zur Verfügung stehen.

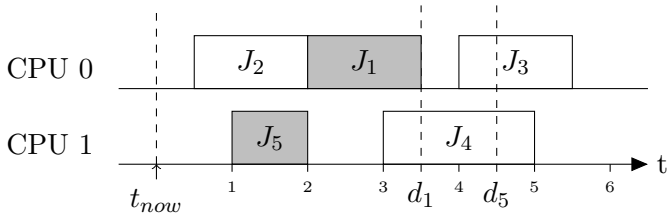


Abbildung 3.12: Ungeordneter konsistenter Ablaufplan

Innerhalb des konsistenten Ablaufplans ist eine Einordnung der Jobs in einer umgekehrten Reihenfolge möglich. Ein Beispiel zeigt Abbildung 3.12. Nacheinander werden auf CPU 0 die Jobs J_1, J_2 und J_3 eingeplant. Job J_4 wird auf CPU 1 eingetragen. Angenommen Job J_5 wird vom gleichen Thread wie Job J_1 abgearbeitet, dann ist eine Einplanung auf CPU 0 nur unter Auflösung der Überlastsitation möglich. Da jedoch auf

CPU 1 noch ausreichend viel Rechenzeit verfügbar ist, wird J_5 auf CPU 1 eingeplant. Aufgrund der früheren Deadline bei $t = 4,5$ im Vergleich zu J_4, wird der Job nach vorn verschoben und es entsteht ein inkonsistenter Ablaufplan, da J_1 und J_5 im gleichen Zeitintervall eingeplant sind. Der Übergang zum konsistenten Ablaufplan erfolgt durch die Verschiebung des Starts von J_5 auf $t = 1$.

Das Vorgehen hat dazu geführt, dass J_5 noch vor J_1 eingeplant wurde, obwohl die von der Anwendung angegebenen Deadlines eine andere Reihenfolge fordern. Folglich ist aus Sicht der Anwendung ein *ungeordneter Ablaufplan* entstanden. Um einen *geordneten Ablaufplan* zu erhalten, muss die Strategie zum Einfügen entsprechend angepasst werden. Wenn im lokalen Ablaufplan die Sortierung nach Deadlines erhalten bleiben soll, dann kann J_5 im Beispiel nicht ohne Kürzung der Deadline eingeplant werden.

Im Folgenden gehe ich davon aus, dass der konsistente Ablaufplan auch geordnet ist. Aus Sicht der Anwendung kann nur dann eine sortierte Abarbeitung der Jobs basierend auf den übermittelten Deadlines erfolgen.

Die Aufstellung eines konsistenten Ablaufplans allein löst allerdings nicht Probleme, die bei einer früheren Ausführung von Jobs entstehen: Wie im Abschnitt 3.5 beschrieben, wird ein Job ausgeführt, wenn alle vorher eingeplanten Jobs blockiert sind und es keine Slack-Zeit gibt. Diese frühere Auswahl kann wieder eine Kollision während der Ausführung hervorrufen, obwohl die Auswahl der Threads auf einen konsistenten Ablaufplan basiert. Entsprechend muss die Ausführung des konsistenten Ablaufplans angepasst werden.

Ausführung des konsistenten Ablaufplans

Der im Abschnitt 3.5 beschriebene Algorithmus zur Auswahl des nächsten Threads muss angepasst werden, um die Ausführung des aufgestellten konsistenten Ablaufplans zu ermöglichen. Im Wesentlichen muss der Algorithmus um eine Prüfung ergänzt werden, die ermittelt, ob eine Thread-Migration durchgeführt werden darf.

Genau wie bei dem schon vorgestellten Algorithmus werden nacheinander die Jobs des Ablaufplans von vorn beginnend untersucht. Blockierte Jobs werden dabei weiterhin ignoriert. Wird ein bereiter Job ermittelt, dessen abarbeitender Thread aber auf einer anderen CPU eingeplant ist, dann müssen die folgenden zwei Sachverhalte untersucht werden, die das weitere Vorgehen entscheiden:

- Wenn der Vorgänger-Job des im Ablaufplan ermittelten Jobs auf einer anderen CPU eingeplant ist, dann wird keine Thread-Migration durchgeführt, da der Thread auf der ursprünglichen CPU noch Rechenzeit erhält. Das Vorziehen des Threads kann andernfalls dazu führen, dass beide Instanzen des ATLAS-Schedulers versuchen den Thread zu starten. Dies ist unzulässig.
- Ist der gerade zu untersuchende Job nicht an vorderster Stelle im Ablaufplan – es gibt also davor blockierte Jobs mit eingeplanter Rechenzeit –, dann darf der Job nur migriert und vorzeitig ausgeführt werden, wenn der Thread gerade von CFS auf der anderen CPU eingeplant wird. Bei einer Einplanung von ATLAS-Recover würde die dort vorgesehene Rechenzeit bei der Migration verloren gehen.

Ist eine Migration aufgrund einer der beiden Punkte unzulässig, dann wird der Job als ein blockierter Job behandelt und keine Migration des Threads durchgeführt. Andernfalls

wird der Thread auf die neue CPU migriert und für die maximale Dauer des neuen Jobs ausgeführt.

Das beschriebene Vorgehen ist ein erster möglicher Ansatz für das Scheduling von Threads und deren Jobs auf mehreren CPUs. Bei einer Umsetzung müssen weitere Details beachtet werden, auf die in dieser Arbeit nicht weiter eingegangen wird.

3.8 Absolute und relative Zeitangaben

Bei der Übermittlung eines Jobs an den ATLAS-Scheduler wird die Deadline durch Angabe eines Zeitpunktes spezifiziert. Die Zeitangabe kann dabei relativ zum Zeitpunkt der Übermittlung oder absolut erfolgen. Im folgenden Abschnitt werden die Vor- und Nachteile genauer betrachtet. Dabei wird zwischen Jobs unterschieden, die unmittelbar aus eintretenden Ereignissen entstehen, und Jobs, die periodisch ausgeführt werden.

Jobs als Folge von (aperiodischen) Ereignissen erfordern die Festlegung einer Deadline in Abhängigkeit des Ereignisses. Beispielsweise kann die Einplanung eines Threads zur grafischen Darstellung von Anwendungen mittels ATLAS-Scheduler erfolgen. Klickt ein Benutzer auf eine Schaltfläche (Ereignis), dann wird ein Job für den Thread, der die eingedrückte Schaltfläche zeichnet (Reaktion), an den ATLAS-Scheduler übergeben.

Die Verwendung einer relativen Zeitangabe zur Definition der Deadline zwischen Ereignis und Reaktion ist unter Linux problematisch, da die Übergabe des Jobs an den ATLAS-Scheduler im Userland initialisiert wird und der Scheduler die Ausführung des Threads vor Übertragung des Jobs unterbrechen kann. Die Dauer der Unterbrechung muss also von der Anwendung beachtet und entsprechend von der relativen Deadline abgezogen werden. Dies ist nicht möglich, da Unterbrechungen durch den Scheduler für die Anwendung transparent sind und es keine Möglichkeit zur Bestimmung der genauen Länge einer Unterbrechung gibt.

Der bessere Ansatz ist die Verwendung absoluter Zeitangaben. Dies ist allerdings nur möglich, wenn das Auftreten des Ereignisses durch einen Zeitstempel t markiert wurde. Die Deadline d eines Jobs ergibt sich dann aus $d = t + r$, wobei r die maximale Verzögerungszeit der Reaktion in Bezug auf das Ereignis ist. Ist der Zeitpunkt des Auftretens des Ereignisses nicht bekannt, dann sind bestenfalls Schätzungen bezüglich der zu verwendenden Deadline möglich. Die Qualität der Festlegung der Deadline kann in diesem Fall weder mit absoluten noch relativen Deadlines gesteigert werden.

Die Festlegung der Deadline von *periodischen Jobs* mittels relativer Zeitangaben führt zu Ungenauigkeiten. Übermittelt beispielsweise ein Thread einige aufeinanderfolgende Jobs und erhöht die Deadline bei jeder Übermittlung um den konstanten Faktor c, dann ist der Abstand zweier aufeinanderfolgender Jobs $c + x$, wobei sich x aus der Ausführungszeit zur Übermittlung eines Jobs sowie eventuellen Unterbrechungen durch den Scheduler ergibt. Für periodische Jobs ist deshalb nur die Nutzung von absoluten Deadlines sinnvoll.

Die Umsetzung des Primitives `submit` unterstützt sowohl relative als auch absolute Zeitangaben, jedoch sollte auf die Nutzung relativer Zeitangaben aus den genannten Gründen verzichtet werden. Die Umrechnung einer relativen Zeit in eine absolute kann mittels

des Systemaufrufs `clock_gettime` mit der Uhr `CLOCK_MONOTONIC` und der Addition der relativen Zeit erfolgen.

3.9 Verhalten bei `fork` beziehungsweise `clone`

Es besteht die Möglichkeit, dass ein Thread während der Abarbeitung eines Jobs eine Kopie von sich selbst mittels `fork` beziehungsweise durch den Aufruf von `clone` mit geeigneten Parametern erstellt. Beim Aufruf von `sched_setscheduler` kann festgelegt werden, ob bei der Erzeugung einer Kopie der zugewiesene Scheduler sowie die Priorität für den neu erzeugten Thread übernommen werden sollen. Das Verhalten des ATLAS-Schedulers für Threads, die während der Ausführung eines Jobs neue Threads erzeugen, muss entsprechend definiert werden.

Wird der neue Thread vom ATLAS-Scheduler eingeplant, dann muss in jedem Fall eine neue Kopie des gerade laufenden Jobs erstellt und eingeplant werden. Andernfalls berechnen zwei Threads Rechenzeit für den gleichen Job. Dies ist weder aus Sicht des Programmierers noch aus Sicht des Schedulers sinnvoll. Zusätzlich stellt sich die Frage, ob bereits übermittelte Jobs, die noch nicht abgearbeitet werden, für den neuen Thread kopiert und eingeplant werden sollen, oder ob für diesen erst explizit neue Jobs zu übermitteln sind.

Der gewählte Lösungsansatz weist einem neuen Thread immer CFS als Scheduler zu. Dieses Vorgehen hat den Vorteil einer leichteren Implementierung. Der Programmierer kann durch die Nutzung des Primitives `next` im neu erzeugten Thread und der Übermittlung neuer Jobs für den erzeugten Thread das Verhalten des anderen Lösungsansatzes nachbilden.

4 Implementierung

In diesem Kapitel beschreibe ich ausgewählte Details zur Implementierung des ATLAS-Schedulers. Dabei steht im Abschnitt 4.1 die Umsetzung der Primitive im Mittelpunkt, welche den Informationsfluss zwischen dem ATLAS-Scheduler und der ATLAS-Lib beziehungsweise der Anwendung steuern.

Des Weiteren werde ich im Abschnitt 4.2 das Einfügen neuer Scheduling-Klassen in die Schichtenarchitektur von Schedulern des Linux-Kerns und die notwendige Synchronisation von Datenstrukturen beschreiben.

Der Linux-Kern stellt eine Reihe verschiedener Uhren mit jeweils eigenen Vor- und Nachteilen zur Verfügung, welche ich im Abschnitt 4.3 beschreiben werde. Zwei unterschiedlich schnell laufende Uhren erschwerten die Evaluierung und werden ebenfalls beschrieben.

Bei der Implementierung des Primitives `next` muss das so genannte *Lost-Wakeup-Problem* beachtet werden, welches im Abschnitt 4.4 betrachtet wird. Den Abschluss des Kapitels bildet Abschnitt 4.5 mit der Beschreibung von Schwierigkeiten und Lösungsansätzen bei der Verwendung von *Timern* sowie `printk` zur Ausgabe von Nachrichten im Linux-Kern.

4.1 Umsetzung der Primitive des ATLAS-Konzepts

Im Abschnitt 2.6.2 des Grundlagenkapitels wurden die von ATLAS benötigten Primitive erläutert. Da die ATLAS-Lib im Userland und der ATLAS-Scheduler im Linux-Kern ausgeführt werden, muss bei der Implementierung ein Mechanismus genutzt werden, der diesen Informationsaustausch erlaubt. In den beiden folgenden Abschnitten werden zwei mögliche Umsetzungen vorgestellt.

4.1.1 Umsetzung durch Betriebssystemaufrufe

Der konventionelle Weg zum Eintritt in den Linux-Kern ist die Benutzung von Betriebssystemaufrufen. Die eigentliche Umsetzung der Aufrufe variiert je nach verwendeter Systemarchitektur stark. Im Kern wird die Implementierung von neuen Systemaufrufen erleichtert, indem architekturunabhängige Makros eingeführt werden, welche die Eigenheiten der jeweiligen Architektur verbergen.

Die Integration neuer Betriebssystemaufrufe ist einfach und die Anzahl der möglichen Argumente ist ausreichend für die Implementierung der ATLAS-Primitive. Da als Argumente auch Zeiger zu beliebigen Datenstrukturen im Userland möglich sind, muss der Kern sicherstellen, dass hinter den übergebenen Zeigern korrekte Daten zu finden sind. Da hierfür im Kern bereits entsprechende Funktionen vorhanden sind, habe ich mich für die Umsetzung der Primitive durch drei zusätzliche Betriebssystemaufrufe entschieden.

Nachteilig an dieser Lösung ist, dass jeder Systemaufruf mit einer eindeutigen Nummer gekennzeichnet ist, welche sich je nach Systemarchitektur unterscheiden kann. Um den ATLAS-Scheduler somit für alle Architekturen zur Verfügung zu stellen, müssen die entsprechenden Quelldateien für die Deklaration der zur Verfügung stehenden Betriebssystemaufrufe für alle Architekturen angepasst werden. Dies ist immer erforderlich, wenn dem Linux-Kern neue Funktionalitäten durch zusätzliche Betriebssystemaufrufe hinzugefügt werden.

4.1.2 Umsetzung über das `/proc`-Dateisystem

Die Kommunikation zwischen Userland und Kern kann in Linux auch über das *proc-Dateisystem* erfolgen, welches in der Regel in `/proc` eingebunden wird. Für jeden im System laufenden Thread befindet sich ein eigenes Unterverzeichnis mit der entsprechenden Thread-ID als Kennzeichnung darin. Denkbar ist eine Umsetzung der Primitive `submit` sowie `cancel` mittels spezieller Dateien innerhalb der Verzeichnisse, auf denen dann Dateioperationen ausgeführt werden.

Möglich ist die Übermittlung eines Jobs, indem die Ausführungszeit und Deadline mit einer entsprechenden Syntax in eine Datei zur Job-Übermittlung im Verzeichnis des Threads geschrieben wird. Durch den Pfad der Datei erhält der Kern die Informationen über den entsprechenden Thread, der den Job später abarbeiten soll. Die Schreiboperation in die Datei selbst ist im Kern dann mit einem Funktionsaufruf verknüpft. Innerhalb der Funktion ist der Zugriff auf die geschriebenen Daten über die Funktionsargumente möglich und der Kern kann den neuen Job nach Auswertung der Daten in den Ablaufplan einfügen. Mit dieser Vorgehensweise entfällt die Notwendigkeit, einen neuen Betriebssystemaufruf zu implementieren.

Für jede erfolgte Job-Übermittlung besteht die Möglichkeit, innerhalb eines Verzeichnisses wie beispielsweise in `/proc/<tid>/jobs/` eine Datei einzublenden, welche als Platzhalter für den übermittelten Job dient. Damit lassen sich Informationen über einzelne Jobs durch Auslesen der jeweiligen Datei zurückgewinnen. Durch das Löschen der Datei ist die Umsetzung des Primitives `cancel` und damit das Zurückrufen eines bereits übermittelten Jobs elegant umsetzbar.

Das `/proc`-Dateisystem kann zur Umsetzung des ATLAS-Primitives `next` verwendet werden. Beispielsweise kann der Linux-Kern über den Ausführungsbeginn eines neuen Jobs informiert werden, indem eine Anwendung eine Leseoperation auf `/proc/<tid>/next_job` durchführt. Ist kein Job vorhanden, dann kann die Leseoperation entsprechend blockiert werden.

Aufgrund der deutlich einfacheren Implementierung habe ich mich für die Umsetzung der Primitive durch zusätzliche Betriebssystemaufrufe entschieden. Diese Lösung ist außerdem performanter, da keine aufwendige Prüfung der Syntax notwendig ist, um die Spezifikation des Jobs zu ermitteln. Für die Portierung von ATLAS auf andere Architekturen ist eine Umstellung durch das `/proc`-Dateisystem sinnvoll.

4.2 Implementierung neuer Scheduling-Klassen in Linux

Der modulare Aufbau des Linux-Schedulers erlaubt das einfache Hinzufügen neuer Scheduling-Klassen. Jede Klasse erstellt eine Variable vom Typ `struct sched_class`. Die Variablen der unterschiedlichen Scheduling-Klassen werden dabei in einer einfach verketteten Liste zusammengefügt und legen die Klassen-Priorität gemäß des Schichten-Entwurfs fest. Innerhalb der Struktur sind Funktionszeiger definiert, welche auf Funktionen zeigen, die von der jeweiligen Scheduling-Klasse zu implementieren sind. Die folgende Liste enthält die wichtigsten Funktionen und deren Bedeutung innerhalb des Scheduling-Frameworks.

- `enqueue_task` – Bei der Erzeugung beziehungsweise dem Aufwecken eines vorher blockierten Threads wird diese Funktion aufgerufen. Der als Argument übergebene Thread muss in die zur Verwaltung der bereiten Threads verwendete Datenstruktur eingefügt werden.
- `dequeue_task` – Die Funktion wird beim Blockieren eines Threads oder beim Abbruch eines bereiten Threads aus dem System aufgerufen. Die Funktion muss sicherstellen, dass der Thread aus der verwendeten Datenstruktur entfernt wird.
- `pick_next_task` – Diese Funktion wird bei einer Scheduling-Entscheidung aufgerufen. Aus der Menge der bereiten Threads wird der neue Thread ausgewählt, dem als nächstes die CPU zugeteilt wird.
- `put_prev_task` – Diese Funktion informiert die Scheduling-Klasse des gerade laufenden Threads darüber, dass eine neue Scheduling-Entscheidung bevorsteht und dem aktuell laufenden Thread die CPU nun entzogen wird.
- `task_tick` – Diese Funktion wird in regelmäßigen Abständen aufgerufen. Die Festlegung der Frequenz geschieht dabei bereits während der Kompilierung des Kerns. Mögliche Werte variieren je nach Systemarchitektur und können für *x86* 100 Hz, 250 Hz, 300 Hz oder 1.000 Hz sein. Die Funktion wird zur Abrechnung der Laufzeit eines Threads verwendet.
- `check_preempt_curr` – Wird ein Thread derselben Scheduling-Klasse wie der gerade laufende Thread bereit, dann wird diese Funktion aufgerufen. Anhand des Rückgabewerts wird entschieden, ob der gerade laufende Thread unterbrochen werden soll. Diese Funktion wird beispielsweise von CFS verwendet, um interaktive Prozesse gegenüber dauerhaft rechnenden Threads zu bevorzugen.

Innerhalb der Funktionen erfolgt der Zugriff auf Datenstrukturen der Scheduling-Klasse. Abhängig davon, an welchen Stellen diese Variablen verwendet werden, sind unterschiedliche Mechanismen zur Synchronisation erforderlich. Diese werden im folgenden Abschnitt vorgestellt.

Synchronisation verwendeter Datenstrukturen

Das Scheduling unter Linux arbeitet mit CPU-lokalen Ressourcen. Dies hat zwei wesentliche Gründe: Einerseits ist eine sehr gute Skalierbarkeit gegeben, da verschiedene Prozessoren bei Scheduling-Entscheidungen nicht auf die gleichen Datenstrukturen zugreifen müssen. Andererseits steigt dadurch gleichzeitig die Performance der Algorithmen, da die Ressourcen im lokalen CPU-Cache gehalten werden und somit ein schnellerer Zugriff möglich ist.

Die Datenstruktur, welche die Basis für das Scheduling bildet, ist die so genannte *Runqueue*. Innerhalb der Struktur sind weitere Strukturen definiert, die von den verschiedenen Scheduling-Klassen verwendet werden. Obwohl es sich um eine CPU-lokale Datenstruktur handelt, ist beim Zugriff eine Sperre notwendig, da es verschiedene Algorithmen gibt, die mit mehreren Runqueues arbeiten. Ein Beispiel hierfür ist die Migration von Threads zwischen CPUs innerhalb von CFS, um einen Lastausgleich herzustellen.

Der Aufruf der in der Struktur `sched_class` festgelegten Funktionen durch das Scheduling-Framework geschieht immer mit einer gehaltenen Sperre der Runqueue des jeweiligen Prozessors. Werden die in der Runqueue eingebetteten Datenstrukturen der Scheduling-Klasse ausschließlich innerhalb der Scheduling-Funktionen verwendet, dann muss sich der Programmierer nicht mit Sperren beschäftigen.

Beim ATLAS-Scheduler ist dies allerdings nicht der Fall, da der Ablaufplan eine CPU-lokale Datenstruktur ist, die auch außerhalb der Scheduling-Funktionen genutzt wird. Beispielsweise wird der Ablaufplan bei der Übertragung von neuen Jobs mittels `submit` bearbeitet – möglicherweise von einer anderen CPU. Auch der Start des nächsten Jobs innerhalb eines Threads beim Aufruf von `next` verändert den Ablaufplan, weil der alte Job entfernt werden muss.

Der einfachste Lösungsansatz ist das Sperren der beteiligten Runqueue. Im Fall der Übertragung eines Jobs und dessen Einplanung in den Ablaufplan ist dies aber problematisch, da die Operation wie in Abschnitt 3.4.3 beschrieben eine Komplexität von $O(n)$ hat.

Beim Sperren der Runqueue muss zwischen zwei verschiedenen Situationen unterschieden werden: Bei einer lokalen Sperre der Runqueue, also dem Sperren der Runqueue auf der CPU, auf welcher gerade die Abarbeitung stattfindet, werden lokale Interrupts deaktiviert und es erfolgt kein Aufruf des Schedulers. Das Sperren der Runqueue einer anderen CPU verhindert dort nicht den Start des Schedulers. Beim Aufruf des Schedulers muss dieser mittels Busy-Waiting bis zur Freigabe der Sperre warten. Es muss also die Zeitdauer, in der die Runqueue gesperrt bleibt, auf ein Minimum reduziert werden oder bestenfalls eine Sperrung der Datenstruktur komplett verhindert werden.

Anstatt die komplette Runqueue zu sperren, habe ich feingranularere Sperren bei einer Job-Übermittlung verwendet, welche die Datenstruktur zur Verwaltung des Ablaufplans schützen. Deshalb sind während der Übermittlung von Jobs auf der betreffenden CPU weiterhin Scheduling-Entscheidungen möglich, so lange nicht die Datenstruktur selbst innerhalb der Scheduling-Funktion benötigt wird.

4.3 Wahl der Zeitbasis

Der Linux-Kern stellt unterschiedliche Uhren mit jeweils verschiedenen Eigenschaften bereit. Die Uhr mit der Bezeichnung `CLOCK_REALTIME` beschreibt in Linux die systemweite, reale Zeit. Da der Administrator eines Systems die Uhrzeit jederzeit verstellen kann, verläuft die Uhr nicht monoton und Sprünge sind möglich. Durch das *Network Time Protokol* (NTP) kann der Linux-Kern über eine mögliche regelmäßige Abweichung (Drift) der zugrunde liegenden Hardware-Uhr informiert werden. Der Drift-Wert wird dann benutzt, um die Uhrzeit entsprechend anzupassen.

Eine weitere zur Verfügung stehende Uhr ist `CLOCK_BOOTTIME`. Diese Uhr misst die seit dem Start des Systems vergangene Zeit. Die Uhr enthält außerdem Zeitperioden, in denen sich ein Rechner im Standby-Betrieb befindet und enthält somit aus Sicht des Betriebssystems Sprünge.

Alternative Uhren dazu sind die monoton verlaufenden Uhren `CLOCK_MONOTONIC` beziehungsweise `CLOCK_MONOTONIC_RAW`. Die Implementierung stellt sicher, dass bei beiden Uhren keine Sprünge möglich sind. Der Unterschied liegt in der Behandlung des durch NTP bereitgestellten Drift-Werts: Dieser wird in `CLOCK_MONOTONIC` eingerechnet, während `CLOCK_MONOTONIC_RAW` die unangepassten, von der Hardware gelieferten Daten zur Berechnung der Zeit verwendet. POSIX [Soc08] spezifiziert den Startpunkt beider Uhren auf einen undefiniert Wert. In Linux sind beide Uhren beim Systemstart mit 0 initialisiert [Cor].

Bei der Erstellung des Ablaufplans ist die Nutzung einer Zeitbasis ohne Sprünge sinnvoll, wenn auch nicht zwingend notwendig. Sind Sprünge vorhanden, dann kann die Deadline einzelner oder aller Jobs im Ablaufplan bereits vor deren Ausführung in der Vergangenheit liegen. Der ATLAS-Scheduler würde die Jobs dann aus dem Ablaufplan entfernen und die dazugehörigen Threads an ATLAS-Recover weiterleiten. Aus semantischer Sicht sind Sprünge im Ablaufplan allerdings nicht sinnvoll: Die Einplanung von Jobs zur Dekodierung von Bildern eines Video-Players soll beispielsweise unabhängig von Änderungen der Systemzeit sein. Folglich muss zwischen den beiden monoton und ohne Sprüngen verlaufenden Uhren gewählt werden. Hierbei ist es sinnvoller, die Uhr `CLOCK_MONOTONIC` zu verwenden, da diese aus Sicht der Anwendung konsistent zur realen Zeit verläuft und theoretisch nicht für Drifts anfällig ist.

Im Kern verwende ich *High Resolution Timer (HR-Timer)* [GN06, Cor] zur Steuerung zukünftiger Ereignisse. Beispielsweise wird der ATLAS-Scheduler bei Überschreitung der Deadline eines Threads durch den Ablauf eines HR-Timers informiert. Zwar kann für den HR-Timer die Zeitbasis konfiguriert werden, jedoch wird die Uhr `CLOCK_MONOTONIC_RAW` derzeit nicht unterstützt. Deshalb verwende ich in der Implementierung ausschließlich die Uhr `CLOCK_MONOTONIC`.

Unterschiedlich schnell laufende Uhren im Linux-Kern

Beim Testen des ATLAS-Schedulers hat sich herausgestellt, dass die auf `CLOCK_MONOTONIC` basierende Uhr und die zur Abrechnung der Laufzeit eines Threads genutzte Uhr nicht mit der gleichen Frequenz arbeiten. Wird beispielsweise im Kern ein HR-Timer für eine Sekunde programmiert, dann steigt die Laufzeit des Threads, die mittels

`clock_gettime` und der Thread-eigenen Uhr `CLOCK_THREAD_CPUTIME_ID` ermittelt wird, nicht zwangsläufig um eine Sekunde. Die Werte variieren je nach System, so dass die Vermutung nahe liegt, dass als Basis für die Uhren unterschiedliche Hardware innerhalb des Systems genutzt wird. Die Beobachtung tritt insbesondere auch ein, wenn kein Drift im System eingestellt ist.

Praktisch handelt es sich hierbei um einen Fehler im Linux-Kern, denn wird ein Thread für eine Zeit t in Bezug auf die Uhr `CLOCK_MONOTONIC` ausgeführt, dann muss auch die lokale Ausführungszeit des Threads entsprechend um t steigen.

Der Unterschied beider Uhren ist abhängig vom System und bewegt sich in Größenordnungen von etwa einem Promille. Experimente haben gezeigt, dass sich die gemessenen Zeitabschnitte beider Uhren mittels eines konstanten Faktors ineinander umrechnen lassen.

Der Estimator rechnet mit der lokalen Ausführungszeit eines Threads, während die im Kern verwendeten HR-Timer mit `CLOCK_MONOTONIC` arbeiten. Deshalb muss die vom Estimator geschätzte Zeit bei der Job-Übermittlung um einen Faktor k angepasst werden. Nähere Details dazu werde ich im Abschnitt 5.3.1 bei der Evaluierung des ATLAS-Schedulers beschreiben.

4.4 Lost-Wakeup-Problem bei der Implementierung von `next`

Beim Aufruf von `next` blockiert ein Thread, wenn für diesen kein Job zur Verfügung steht. Sobald ein neuer Job durch einen anderen Thread übermittelt wurde, muss dieser den blockierten Thread aufwecken, um seine Ausführung fortzusetzen. Das Testen auf das Vorhandensein eines Jobs sowie die Übermittlung eines neuen Jobs können hierbei potentiell zur gleichen Zeit erfolgen, da beide Threads auf unterschiedlichen CPUs laufen können. Demzufolge muss die Implementierung neben dem Schutz von gemeinsam benutzten Datenstrukturen auch sicherstellen, dass Blockieren und Aufwecken entsprechend synchronisiert sind. Andernfalls sind Situationen möglich, in denen ein Thread trotz eines vorhandenen Jobs blockiert bleibt. Dieses Problem, welches auch als *Lost-Wakeup* bezeichnet wird, muss verhindert werden.

Abbildung 4.1 stellt einen Ablaufplan dar, in dem das Lost-Wakeup-Problem zum Blockieren von Thread 1 führt, obwohl für diesen von Thread 2 ein neuer Job übermittelt wurde. Das Problem tritt nicht nur auf Systemen mit mehreren Prozessoren, sondern auch auf Einprozessorsystemen auf, da auch die Ausführung im Kern unterbrochen und die CPU einem anderen Thread zugewiesen werden kann.

Die einfachste Lösungsmöglichkeit zur Verhinderung des Problems ist, das Testen auf das Vorhandensein eines Jobs sowie das Blockieren als atomare Operationen mittels Sperren umzusetzen. Unter Linux ist dies allerdings hier nicht möglich, da das Blockieren durch den Aufruf von `schedule` erreicht wird und dabei Sperren nicht wieder freigegeben werden können.

Die gängige Praxis zur Lösung des Problems ist die Nutzung einer Zustandsvariablen je Thread, die das Verhalten beim Aufruf von `schedule` beeinflusst. Beim direkten Aufruf des Schedulers wie im Abschnitt 2.4.3 beschrieben, blockiert der Thread, wenn dessen Zustand, gegeben durch die Zustandsvariable, ungleich `TASK_RUNNING` ist. Andernfalls

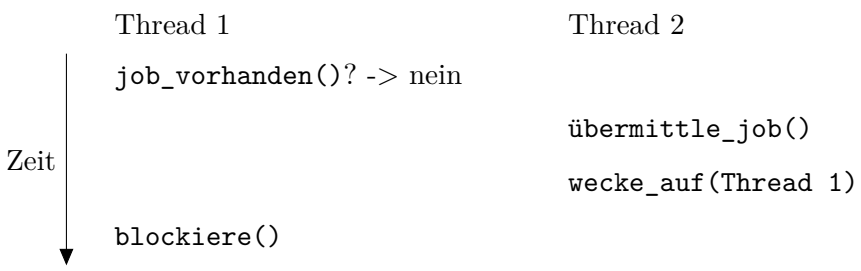


Abbildung 4.1: Lost-Wakeup-Problem

wird vom Scheduler zwar ein neuer Thread für die Ausführung bestimmt, aber der aufrufende Thread blockiert nicht, sondern bleibt im Zustand bereit.

Die `wake_up`-Funktion zum Aufwecken eines Threads ändert immer dessen Zustandsvariable zurück auf `TASK_RUNNING` und fügt den Thread, wenn dieser vorher blockiert war, wieder in die Menge der bereiten Threads ein.

Um ein Lost-Wakeup zu verhindern, muss, wie in Listing 4.1 dargestellt, der Zustand des Threads auf `TASK_INTERRUPTIBLE` geändert werden *bevor* die Existenz eines neuen Jobs geprüft wird. Wird zwischen beiden Instruktionen ein Job eingefügt und der Thread aufgeweckt, obwohl er noch gar nicht blockiert ist, dann stellt die Implementierung der `wake_up`-Funktion sicher, dass der Zustand wieder auf `TASK_RUNNING` gesetzt wird und `schedule` nicht blockiert.

```
1  for(;;) {
2          set_current_state(TASK_INTERRUPTIBLE);
3          if ((job = pop_job()))
4                  break;
5
6          schedule();
7  }
8  set_current_state(TASK_RUNNING);
```

Listing 4.1: Implementierung von next (vereinfacht)

Nach der Rückkehr von `schedule` muss geprüft werden, ob tatsächlich ein neuer Job verfügbar ist. Dies ist nicht immer der Fall, da beispielsweise auch bei der Zustellung von Signalen ein Thread aufgeweckt wird. Implementiert wird dies durch eine Schleife, die nur beim Vorhandensein eines neuen Jobs verlassen wird.

Wird einem Thread der Prozessor zwischen Zeile 2 und 3 preemptiv entzogen, dann kann fälschlicherweise angenommen werden, dass ein Thread trotz Vorhandenseins eines Jobs aufgrund seines Zustandes von `TASK_INTERRUPTIBLE` dauerhaft blockiert. Dies ist nicht der Fall, da die Implementierung von `schedule` diesen Fall erkennt. Erläuterungen zum Aufruf des Schedulers wurden im Abschnitt 2.4.3 beschriebenen. Nur der direkte Aufruf des Schedulers kann zum Blockieren eines Threads führen.

4.5 Probleme bei der Implementierung

Beim Aufruf der spezifischen Funktionen einer Scheduling-Klasse wird die Sperre der Runqueue bereits durch das Scheduling-Framework erwirkt. Dieser Zusammenhang wurde bereits im Abschnitt 4.2 diskutiert. Wird innerhalb dieser Funktionen versucht, die Sperre der Runqueue erneut zu erhalten, dann kommt es zu einer Deadlock-Situation: Die betreffende CPU wartet mittels Busy-Waiting auf die Freigabe der Sperre, die sie selbst hält. Dies muss unter allen Umständen verhindert werden, indem die Runqueue nicht erneut gesperrt wird.

Im ersten Moment wirkt es trivial, dieses einfache Prinzip zu beachten. Jedoch führt der Aufruf zahlreicher Funktionen zum Aufwecken blockierter Threads, die dann für die eigentliche Abarbeitung der Funktion verwendet werden. Die Implementierung des Aufweckens führt unweigerlich zu oben genannter Situation, da zum Einreihen eines bereiten Threads die `enqueue_task`-Funktion der entsprechenden Scheduling-Klasse aufgerufen wird. Davor wird versucht, die entsprechende Runqueue zu sperren. Zwei für diese Problematik anfällige Beispiele sind `printk`, die Ausgabe von Nachrichten, sowie die Verwendung von HR-Timern.

4.5.1 Verwendung von printk innerhalb des Schedulers

Eine Frage, die sich unweigerlich stellt, ist, warum innerhalb des Schedulers `printk` zur Ausgabe von Nachrichten aufgerufen werden soll. Für den praktischen Einsatz macht dies keinen Sinn, allerdings ist es vor allem in der Entwicklungsphase sehr hilfreich, wenn es eine Möglichkeit zur Nachrichtenausgabe gibt.

Das Problem, dass `printk` nicht zur Ausgabe innerhalb des Schedulers verwendet werden kann, ist bereits bekannt und wurde durch die Einführung der Funktion `printk_sched` ansatzweise gelöst. Die Funktion verzögert die Ausgabe bis zu einem sicheren Zeitpunkt, an dem die Runqueue nicht gesperrt ist. Der Nachteil der Implementierung der Funktion ist, dass nur genau eine Ausgabe möglich ist. Wird `printk_sched` mehrmals innerhalb des Schedulers aufgerufen, dann wird nur die letzte Nachricht ausgegeben. Für die Fehleranalyse während der Entwicklung ist diese Eigenschaft inakzeptabel. Aus diesem Grund habe ich die Funktion durch einen statisch allokierten Pool an Puffern erweitert, um die Anzahl der möglichen Nachrichtenausgaben zu erhöhen.

4.5.2 Verwendung von Timern innerhalb des Schedulers

Sowohl in ATLAS-LRT als auch in ATLAS-Recover werden HR-Timer verwendet, um den Ablauf entsprechend zu steuern. Die Programmierung der Timer kann sowohl auf absolute als auch auf relative Zeitangaben zurückgreifen. Besonders zuerst genannter Modus ist problematisch, wenn die absolute Zeitangabe sich bereits in der Vergangenheit befindet. In diesem Fall wird versucht, sofort die Ausführung der zugewiesenen Timer-Routine zu starten. Auch dabei wird erneut die Runqueue gesperrt und es kommt zum Deadlock, wenn die Programmierung des Timer aus dem Scheduler heraus durchgeführt wird.

Der verwendete Lösungsansatz beruht darauf, den Aufruf der Funktion, welche für die Einrichtung des Timers zuständig ist, anzupassen und das unmittelbare Aufwecken des Timers zu verhindern. Erst nach Verlassen des Schedulers wird durch einen *Softirq* die Abarbeitung der Timer-Routine gestartet.

5 Evaluierung

Die Evaluierung des ATLAS-Schedulers erfolgt in drei Stufen: Nachdem die grundlegende Funktionalität durch die Aufzeichnung und Auswertung von Events bestätigt wurde, wird die Einhaltung der in Jobs angeforderten Rechenzeiten in Mikrobenchmarks überprüft. Außerdem erfolgt eine Ermittlung der erforderlichen Kosten für die Einplanung von Jobs sowie für den Start eines neuen Jobs.

In einem komplexeren Experiment wird durch die angepasste Version eines Video-Players die Funktionalität des ATLAS-Schedulers überprüft. Dazu werden die Ergebnisse mit einer anderen Version des Video-Players verglichen, der nur von CFS eingeplant wird.

Vor der Evaluierung des ATLAS-Schedulers werden zunächst Details zur genutzten Testumgebung sowie zum benutzten Testsystem beschrieben.

5.1 Testumgebung und -system

Die Ausführung eines sich in Entwicklung befindlichen Schedulers auf echter Hardware ist nicht empfehlenswert, da auch kleinste Programmierfehler in der Regel zum Absturz des Systems führen. Dies macht sich in vielen Fällen durch das komplette „Einfrieren“ des Systems bemerkbar. Die Fehlersuche wird dann durch fehlende Debugging-Möglichkeiten am festgefahren System erschwert. Nach einem Absturz erhält man oft keinerlei Informationen über die Ursachen.

Aus diesem Grund habe ich für die frühe Entwicklung des Schedulers Qemu [Bel05] verwendet. In Qemu ist ein GDB-Server integriert, der es erlaubt, sich mit GDB über eine TCP-Verbindung mit Qemu zu verbinden. Mit dieser Technik kann die *Virtuelle Maschine* (VM), in welcher der Scheduler in einer kleinen Testumgebung läuft, in jedem beliebigen Zustand gestoppt und untersucht werden. Jeder Prozessor innerhalb der VM wird dabei als eigenständiger Thread abgebildet. Bei der Fehlfunktion kann so von außen der Zustand der virtuellen Maschine genau untersucht werden, indem man beispielsweise die Backtraces der einzelnen Prozessoren betrachtet. Dies führt in vielen Fällen schnell zum Finden offensichtlicher Fehler und hat sich besonders für Deadlock-Situationen bewährt.

Spätere Funktionstests und Evaluierungen mit einem stabilen Scheduler wurden auf einem Testsystem bestehend aus einem AMD-Phenom 9550 mit vier Kernen durchgeführt.

5.2 Überprüfung der Funktionalität

Vor einer Evaluierung des ATLAS-Schedulers ist es wichtig, zu überprüfen, ob der Scheduler entsprechend des Entwurfs einwandfrei funktioniert. Dies ist nicht trivial, da

die Überprüfung des Schedulers durch zusätzliche Anwendungen im Userland erfolgen muss, welche die Schnittstellen des Schedulers nutzen. Wird die Anwendung dabei durch einen Fehler innerhalb des ATLAS-Schedulers möglicherweise dauerhaft von CFS eingeplant oder erfolgt die Einplanung von Jobs beispielsweise zu früh, dann ist dies aus Sicht der Anwendung nicht immer feststellbar, insbesondere wenn die Anwendung ausreichend viel Rechenzeit erhält.

Zur Überprüfung der Funktionalität ist es deshalb notwendig, die Umstände, die zur Auswahl des nächsten Threads auf einer CPU geführt haben, nachzuvollziehen. Dazu sind Informationen über die Scheduling-Klasse, die den Thread ausgewählt hat, und die Deadline sowie verbleibende Ausführungszeit des mit dem Thread verknüpften Jobs notwendig. Außerdem muss der Zeitpunkt der Scheduling-Entscheidung gespeichert werden, um diesen mit den Parametern des Jobs in Verbindung bringen zu können. Mit diesen Informationen können Scheduling-Entscheidungen nachvollzogen und analysiert werden.

Im Abschnitt 5.2.1 werde ich eine Möglichkeit zur Erfassung der Informationen beschreiben und im Abschnitt 5.2.2 werde ich die Aufbereitung und Darstellung der Daten diskutieren.

5.2.1 Linux Trace Toolkit – next generation

Das Linux Trace Toolkit – next generation (LTTng) [lttb] ist ein Werkzeug, welches unter anderem die Erfassung von Events im Linux-Kern ermöglicht. Mit Hilfe eines im Userland laufenden Deamons werden die zu erfassenden Events gesteuert und es wird festgelegt, wo die erfassten Informationen gespeichert werden. Aus Performancegründen erfasst LTTng Daten nur in binärer Form im *Common Trace Format* [ctf]. Eine spätere Analyse benötigt deshalb zusätzliche Werkzeuge.

LTTng besteht aus einer Reihe von Kern-Modulen, die dynamisch zur Laufzeit des Systems eingebunden werden und die benötigte Funktionalität liefern. Der Linux-Kern ermöglicht zusammen mit LTTng in der Version 2.0 ohne weitere Anpassungen die Erfassung von Scheduling-Entscheidungen. Da das Event aber keine Informationen bezüglich der beteiligten Scheduling-Klasse enthält, sind weitergehende Anpassungen notwendig.

Ich habe deshalb die folgenden Events hinzugefügt und den Linux-Kern sowie LTTng entsprechend angepasst:

- `enqueue_task`
- `dequeue_task`
- `pick_next_task`
- `put_prev_task`

Die Erfassung der Events erfolgt in den gleichnamigen Funktionen des Scheduling-Frameworks.

5.2.2 Darstellung

Die von LTTng erzeugten Binärdaten müssen für die Funktionsüberprüfung weiterverarbeitet werden. Dabei bietet sich eine grafische Darstellung an, die es erlaubt, Scheduling-Entscheidungen besser nachzuvollziehen. Derzeit gibt es nur ein Tool zur grafischen Veranschaulichung als eine Erweiterung für *Eclipse* [ecl] mit der Bezeichnung *Eclipse LTTng plugin* [ltta]. Damit lassen sich die standardmäßig erfassbaren Events anzeigen, jedoch kann das Plugin nicht ohne weitere Anpassungen zur Darstellung eigener Events verwendet werden. Als Basis für die Visualisierung von Events dient ein Zeitstrahl, der in dieser Form keine leichte Nachvollziehbarkeit der Scheduling-Entscheidungen erlaubt.

Ein zusätzliches Tool mit der Bezeichnung *Babeltrace* [bab] wandelt die erzeugten Binärdaten in eine lesbare Form um und gibt diese aus.

Anstatt das Eclipse LTTng plugin zu erweitern, habe ich mich für die Implementierung eines einfachen Werkzeugs entschieden, welches die von Babeltrace generierte Ausgabe weiterverarbeitet. Das Werkzeug gibt die aufbereiteten Daten in Form einer Tabelle wieder aus und ermöglicht damit die Nachvollziehbarkeit von Scheduling-Entscheidungen.

t in ms	LRT	RECOVER	CFS	Infos			
				id	d	e	flags
17.221,53			heavy(118)	1	16.020	0	DE
18.017,95	heavy(118)			2	18.020	999	–
18.020,60		heavy(118)		2	18.020	996	D
19.017,54			kworker/0:1(23)				
19.017,55			heavy(118)	2	18.020	0	DE
19.020,87	heavy(119)			3	20.020	999	–
19.921,92			heavy(118)	2	18.020	0	DE
20.221,52			kworker/0:1(23)				
20.221,52			heavy(118)	2	18.020	0	DE
21.020,87	heavy(120)			5	22.020	999	–
21.921,94			kworker/0:1(23)				
21.921,95			heavy(118)	2	18.020	0	DE

Tabelle 5.1: Vereinfacht dargestellte Ausgabe des implementierten Werkzeugs zur Ausgabe der mit LTTng erstellten Traces

Tabelle 5.1 zeigt eine beispielhafte Ausgabe des entwickelten Auswerte-Werkzeugs. Die mittels Babeltrace aufbereiteten Daten von LTTng werden als Ereignisse in den Zeilen der Tabelle dargestellt. Die Spalten der Tabelle geben Informationen über das jeweilig eingetretene Ereignis: Die erste Spalte wird für die Ausgabe des erfassten Zeitstempels verwendet. Die Spalten zwei bis vier repräsentieren die Scheduling-Klassen ATLAS-LRT, ATLAS-Recover und CFS. Eine Angabe des Namens einer Anwendung in einer dieser Spalten bedeutet, dass die jeweilige Scheduling-Klasse den Thread mit der in Klammern angegebenen *Thread-ID* für die Abarbeitung ausgewählt hat. Auf die Darstellung der Scheduling-Klassen Stop, Real-Time und Idle wurde verzichtet, da diese für die Funktionsanalyse des ATLAS-Schedulers bedeutungslos sind. Die letzte Spalte enthält Informationen zum Job, welcher mit einem Thread verknüpft ist.

Das dargestellte Beispiel in Tabelle 5.1 zeigt ein System, in dem die Einplanung einer Anwendung mit dem Namen *heavy* bestehend aus drei Threads untersucht werden kann. Zur Vereinfachung wurde der Vorlauf von Threads in CFS im Beispiel deaktiviert. Für die verschiedenen Threads der Anwendung wurden Jobs mit einer Ausführungszeit von 1.000 ms übertragen. Im Gegensatz zu den Threads mit den IDs 119 und 120, deren Jobs die angegebene Rechenzeit niemals vollständig nutzen, überschreitet jeder Job von Thread 118 die angegebene Rechenzeit.

Zum Zeitpunkt 17.221,53 ms wurde Thread 118 von CFS ausgewählt und beendete zum Zeitpunkt 18.017,95 ms die Abarbeitung des Jobs mit der ID 1, um die Ausführung des nächsten Jobs zu starten. Dabei wurde der Scheduler aufgerufen, da nach dem Start der Abarbeitung eines neuen Jobs geprüft werden muss, ob der Thread weiterhin berechtigt ist, die CPU zu nutzen. Im Beispiel ist dies der Fall und die LRT Schicht des ATLAS-Schedulers wählt den gerade schon laufenden Thread erneut aus. Es erfolgt an dieser Stelle also eine sehr kurze Unterbrechung des Threads durch den Scheduler, jedoch wird kein Kontextwechsel zwischen zwei Anwendungen mit unterschiedlichen Adressräumen durchgeführt.

An den Informationen des neu zugewiesenen Jobs ist erkennbar, dass die Deadline bereits bei 18.020 ms liegt. Nach nur etwa 3 ms Ausführung ist die Deadline erreicht und der Thread wird an ATLAS-Recover abgegeben, da noch Rechenzeit zur Verfügung steht. Zusätzlich erfolgt der Aufruf des Schedulers. Die Auswertung des Ablaufplans ergibt, dass noch keine Ausführung eines anderen Threads in ATLAS-LRT notwendig ist, folglich erhält ATLAS-Recover Rechenzeit und der gerade schon rechnende Thread behält die CPU für die Dauer der restlichen Rechenzeit von 996 ms. Nach dieser Zeit wird der Thread an CFS abgegeben. Da keine weiteren Threads in der ATLAS-Recover-Schicht bereit sind, erhält CFS Rechenzeit. Nach kurzer Ausführung des *kworker*-Threads wird die Ausführung des heavy-Threads fortgesetzt.

Zum Zeitpunkt 19.020,87 ms wird, getriggert durch einen Timer-Interrupt, der Scheduler aufgerufen und ATLAS-LRT wählt den Thread mit der ID 119 für die Ausführung aus. Die weiteren dargestellten Scheduling-Entscheidungen in der Tabelle sind selbsterklärend. Die Spalte mit der Bezeichnung Flags markiert die Eigenschaften eines Jobs: *D* bedeutet, dass die zugewiesenen Deadline des Jobs bereits überschritten wurde, *E* steht für die Überschreitung der angegebenen Rechenzeit.

Eine unterstützende Suche nach Funktionsfehlern des ATLAS-Schedulers kann durch die automatisierte Auswertung der Ereignisse erfolgen. Zum Beispiel ist die Ausführung eines Jobs mit einer Deadline in der Vergangenheit durch ATLAS-LRT ein Fehler, der durch die Analyse der Ereignisse erkannt werden kann.

5.3 Mikrobenchmarks

In diesem Abschnitt werden eine Reihe von Mikrobenchmarks vorgestellt, die ausgewählte Anforderungen des ATLAS-Schedulers überprüfen. Um die Komplexität der durchgeführten Messungen gering zu halten, wurde zum Teil auf die Nutzung der ATLAS-Lib verzichtet und die implementierten Betriebssystemaufrufe direkt verwendet.

5.3.1 Vergleich von angeforderter und zugewiesener Rechenzeit

Anforderung 1 beschreibt die Durchsetzung weicher Echtzeitgarantien. Die Überprüfung der Anforderung ist mittels eines einfachen Experiments möglich: Die verfügbare Zeit für die Abarbeitung eines Jobs vor dem Eintreten der Deadline wird gemessen und mit der angeforderten Ausführungszeit des Jobs verglichen.

Aufbau der Messung

Für einen Thread werden Jobs mit der Periode p und Ausführungszeit e übermittelt. Die Abarbeitung eines Jobs besteht aus drei Schritten:

1. Auslesen der aktuellen Laufzeit e des Thread mittels des Betriebssystemaufrufs `clock_gettime` und der Uhr `CLOCK_THREAD_CPUTIME_ID`.
2. Warten auf das Eintreffen des Signals, welches die Überschreitung der Deadline anzeigt. Dabei darf der Thread nicht blockieren, um sicherzustellen, dass die Laufzeit des Threads erhöht wird.
3. Erneute Messung der aktuellen Laufzeit e' analog zum ersten Schritt.

Aus der Differenz der beiden gemessenen Laufzeiten $e' - e$ kann die Zeitdauer ermittelt werden, die für den Job vom ATLAS-Scheduler eingeplant wurde. Die gemessene Zeit setzt sich zusammen aus den Berechnungen des Threads sowie anderen Aktivitäten wie beispielsweise der Ausführung des Signal-Handlers. In der verwendeten Konfiguration des Linux-Kerns wird außerdem die Abarbeitung von Interrupts, Softirqs und Tasklets[Pal, Wil10] als Rechenzeit für den gerade laufenden Thread gezählt. Die Messung bestimmt deshalb die durch den ATLAS-Scheduler zugewiesene Rechenzeit, die sich allerdings von der *tatsächlich nutzbaren Zeit* für die Berechnungen des Threads unterscheidet. Letzere ist kleiner und wird im nächsten Experiment untersucht.

Bei einer Ausführung ohne Überlast muss die zugewiesene Ausführungszeit größer oder gleich der angegebenen Laufzeit sein, andernfalls wird Anforderung 1 verletzt.

Simulation von Hintergrundlast

Die Durchsetzung weicher Echtzeitgarantien des ATLAS-Schedulers ist in Abhängigkeit der Systemlast zu prüfen. Deshalb muss eine Hintergrundlast simuliert werden.

Dazu werden Threads erzeugt, die von CFS eingeplant werden und dauerhaft Rechenzeit für die Abarbeitung einer Endlosschleife nutzen. Durch eine Veränderung der Anzahl der Threads ist die Hintergrundlast skalierbar.

Durchführung der Messung

Die Messung wurde sowohl mit als auch ohne Vorlauf in CFS durchgeführt. Dabei wurde die Anzahl der Threads zur Simulation der Hintergrundlast von 0 bis 30 variiert. Bei jeder Messung wurden jeweils 20 Jobs mit einer Periode von 1.000 ms und einer Ausführungszeit von 500 ms abgearbeitet. Die beantragte Abarbeitungszeit wurde mit

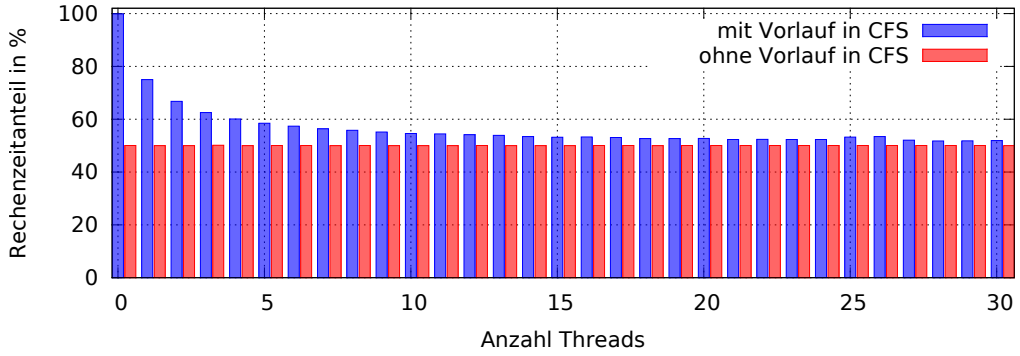


Abbildung 5.1: Anteil zugewiesener Rechenzeit bei 50% Reservierung

einem experimentell bestimmten Faktor von 1,0025 multipliziert, da innerhalb des Linux-Kerns die beteiligten Uhren eine unterschiedliche Frequenz verwenden. Details dazu wurden bereits im Abschnitt 4.3 beschrieben.

Auswertung

Abbildung 5.1 zeigt die Ergebnisse der durchgeführten Messungen. Dazu wurde auf der y-Achse die durchschnittlich gemessene Rechenzeit pro Job im Verhältnis zur Periode abgetragen. Die x-Achse spezifiziert die Anzahl der im Hintergrund laufenden Threads zur Generierung von Hintergrundlast.

Bei aktiviertem Vorlauf in CFS (blau) nutzt der Job bei fehlender Hintergrundlast wie erwartet 100% der verfügbaren Rechenzeit. Mit dem Anstieg der Hintergrundlast des Systems sinkt die verwendete Rechenzeit bis auf einen Wert von 51% bei 30 bereiten Hintergrund-Threads ab.

Wird dem Job kein Vorlauf in CFS gewährt (rot), dann beträgt die zugewiesene Rechenzeit durch den ATLAS-Scheduler konstant 50%, unabhängig von im Hintergrund laufenden Threads.

Jede durchgeführte Messung enthält auch die Kosten für den Aufruf und die Abarbeitung des Signal-Handlers beim Überschreiten der Deadline. Deshalb ist die gemessene Zeitdauer minimal größer als die vom ATLAS-Scheduler tatsächlich eingeplante Ausführungszeit des Jobs. Bei keiner der insgesamt 1.640 durchgeführten Messungen wurde die angeforderte Rechenzeit von 50% unterschritten. Für diese Messung konnte deshalb die Durchsetzung weicher Echtzeitgarantien gemäß Anforderung 1 bestätigt werden.

5.3.2 Vergleich von zugewiesener und tatsächlicher Rechenzeit

Im Experiment von Abschnitt 5.3.1 wurde die angeforderte und zugewiesene Rechenzeit eines Threads durch den ATLAS-Scheduler miteinander vergleichen. Zusätzlich muss auch die zugewiesene und tatsächlich zur Verfügung stehende Rechenzeit untersucht werden.

Die tatsächliche Rechenzeit ist geringer als die zugewiesene Rechenzeit, da letztere beispielsweise auch die Abarbeitung von Interrupts, Softirqs und Tasklets sowie den

Scheduling-Overhead enthält. Auch der System Management Mode [int11] beeinflusst die tatsächlich nutzbare Rechenzeit eines Threads. Hierbei handelt es sich um einen speziellen Ausführungsmodus der CPU, der die Abarbeitung von herstellerspezifischen Anweisungen zur Steuerung der Hardware ohne Kontrolle durch das Betriebssystem erlaubt.

Ich verwende die Bezeichnung *Störlast* für Einflussfaktoren, die die Rechenzeit eines Threads erhöhen, ohne dass dieser seine eigenen Berechnungen ausführt. Der Einfluss von Störlast ist nur schwer einplanbar, da er von äußeren Systemeinflüssen abhängig ist: Viele Interrupts und deren Abarbeitung werden beispielsweise durch Benutzeraktionen initialisiert und können nicht vorhergesagt werden.

Ziel des Experiments ist die Untersuchung des Zusammenhangs zwischen zugewiesener und tatsächlicher Rechenzeit.

Aufbau und Durchführung der Messung

Unter der Annahme, dass der Estimator der ATLAS-Lib die notwendige zuzuweisende Rechenzeit mit einer hinreichenden Genauigkeit richtig abschätzt, kann der Zusammenhang zwischen zugewiesener und tatsächlicher Rechenzeit untersucht werden. Dazu wird wird die vom Estimator geschätzte Zeit mit unterschiedlichen k-Faktoren multipliziert. Anschließend wird der Einfluss des k-Faktors auf die Anzahl verpasster Deadlines untersucht.

Der Estimator der ATLAS-Lib schätzt unter Nutzung einer Workloadmetric die notwendige Zeitdauer für die Ausführung eines Programmabschnitts. Um eine einfache Messung mit unterschiedlichen Rechenzeiten zu erhalten, wird der Estimator zur Schätzung der Ausführungszeit einer Schleife mit einer zufällig gewählten Anzahl an Iterationen i genutzt. Die Workloadmetric ergibt sich dabei ausschließlich aus i.

In einer Anlaufphase von zehn Messungen optimiert der Estimator seine Schätzungen. Danach werden nacheinander 10.000 Jobs an den ATLAS-Scheduler übertragen. Die Ausführungszeit eines Jobs ergibt sich dabei durch die vom Estimator geschätzte Zeit, welche mit dem Faktor k multipliziert wird. Zu keinem Zeitpunkt der Messung befinden sich mehr als zwei Jobs gleichzeitig im System.

Die Jobs werden von einem Thread abgearbeitet, der die Schleife mit der jeweiligen Anzahl an Iterationen i durchläuft. Kommt es dabei zu einer Überschreitung der Deadline, dann war die angeforderte Rechenzeit zu gering. Durch die Zählung der Deadline-Überschreitungen kann die relative Wahrscheinlichkeit einer Deadline-Überschreitung pro Job in Abhängigkeit des k-Faktors ermittelt werden.

Auswertung

Abbildung 5.2 zeigt die relative Anzahl verpasster Deadlines für verschiedene k-Faktoren zur Anpassung der angeforderten Rechenzeit. Je nach Anforderung der Anwendung kann durch die Wahl des k-Faktors beeinflusst werden, wie viele Jobs ihre Deadline im Durchschnitt verpassen. Je höher der Faktor gewählt wird, desto weniger Jobs verpassen ihre Deadline. Auf dem Testsystem sollte die vom Estimator ermittelte Schätzung mit einem k-Faktor von mindestens 1,025 multipliziert werden, wenn der Estimator

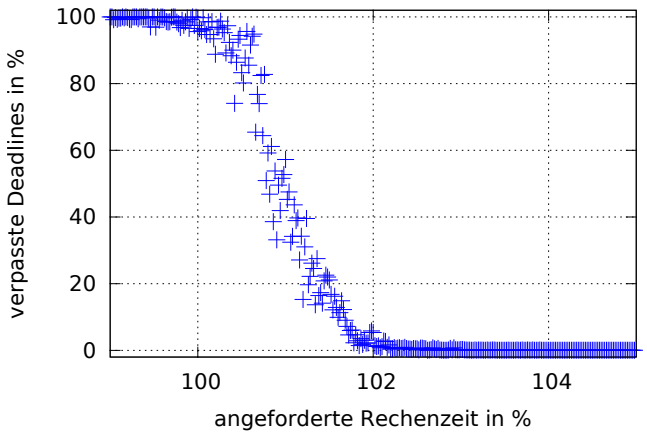


Abbildung 5.2: Verpasste Deadlines in Abhängigkeit des Faktors k zur Anpassung der angeforderten Rechenzeit

die Ausführungszeit einer einfachen Schleife schätzt. In diesem Fall ist die tatsächliche Ausführungszeit ausreichend groß, damit alle Jobs abgeschlossen werden können. Mit größeren k-Faktoren und somit längeren Rechenzeiten der Jobs erhöht sich die Wahrscheinlichkeit, dass es auf einer CPU zu einer Überlastsituation kommt, die entsprechend behandelt werden muss. Deshalb sollte der k-Faktor nicht zu groß gewählt werden.

Komplexere Schätzungen des Estimators, die auf einer Workloadmetric mit vielen Variablen basieren, bedürfen einer erneuten Untersuchung zur Bestimmung des notwendigen k-Faktors. Die Prüfung der Genauigkeit des Estimators sowie die Bestimmung der k-Faktoren sind nicht Gegenstand dieser Arbeit.

5.3.3 Kosten für die Einplanung eines Jobs

Bei der Übermittlung eines Jobs entstehen durch dessen Einplanung in den Ablaufplan Kosten. Zum einen muss die korrekte Position des neuen Jobs in der verwendeten Datenstruktur gefunden werden und zum anderen kann es im ungünstigsten Fall notwendig sein, alle bereits im Ablaufplan eingeplanten Jobs zu verschieben. Nähere Informationen dazu wurden bereits im Abschnitt 3.4.3 beschrieben.

Aufbau und Durchführung der Messung

Ausgehend von einem leeren Ablaufplan werden $n-1$ zufällig gewählte Jobs an den ATLAS-Scheduler übergeben und eingeplant. Bei der Übermittlung des n-ten Jobs wird die benötigte Zeitdauer für den Betriebssystemaufruf gemessen. Diese enthält damit die Kosten für den Wechsel in den Kern und wieder zurück ins Userland sowie die Kosten für die Einplanung des neuen Jobs innerhalb des Kerns. Im ungünstigsten Fall müssen bei der Einplanung alle $n-1$ sich bereits im Ablaufplan befindlichen Jobs verschoben werden. Innerhalb der Messungen werden in einem Zeitabschnitt von etwa 150 s insgesamt 15.000 Jobs eingeplant. Die Ausführungszeit eines Jobs wurde dabei mit gleichverteilter Wahrscheinlichkeit zufällig zwischen 5 ms und 15 ms gewählt.

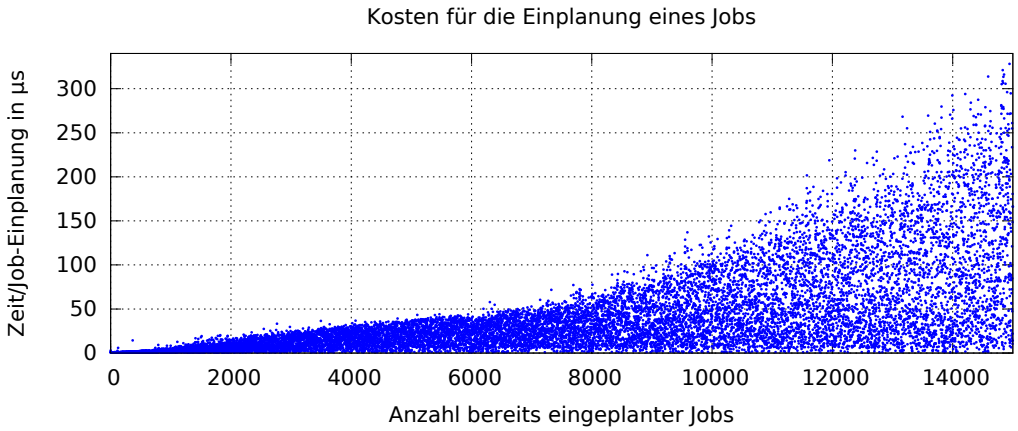


Abbildung 5.3: Kosten für die Einplanung eines neuen Jobs

Die Deadline der Jobs basiert auf einem mit gleicher Wahrscheinlichkeit zufällig ermittelten Wert x zwischen 0 s und 150 s. Ist t_{now} dabei der aktuelle Zeitpunkt zum Beginn der Ausführung, dann berechnet sich die Deadline d eines Jobs aus:

$$d = t_{now} + x + o$$

Dabei ist o ein Offset, der sicherstellt, dass es bei der Einplanung zu keinen Überlastsituationen kommt. Diese Überlastsituationen entstehen immer dann, wenn die Einplanung eines neuen Jobs zur Verschiebung bereits eingeplanter Jobs führt, so dass sich deren Deadline in der Vergangenheit befindet.

Auswertung

Abbildung 5.3 zeigt die Kosten für die Übermittlung und Einplanung eines Jobs in µs. Die x-Achse zeigt dabei die Anzahl bereits eingeplanten Jobs bei der Übermittlung. Die maximalen Kosten steigen mit der Anzahl bereits eingeplanter Jobs, da sich die Anzahl notwendiger Verschiebungen bei der Job-Übermittlung im ungünstigsten Fall erhöht. Im Diagramm ist erkennbar, dass die maximalen Kosten je Job-Übermittlung ab etwa 8.000 bereits eingeplanten Jobs stärker ansteigen.

Dieser Anstieg basiert auf einer größeren Anzahl von Last-Level-Cache-Misses (LLC-Misses) auf dem verwendeten System wie in Abbildung 5.4 dargestellt.

Gemessen wurde die Gesamtanzahl der LLC-Misses bei der Erzeugung, Übermittlung und Einplanung von den auf der x-Achse angegebenen Jobs. Dieser Wert wurde für eine bessere Vergleichbarkeit anschließend auf eine einzelne Job-Übermittlung normiert. Die Periode sowie die Parameter zur Erzeugung der Jobs sind identisch zur vorherigen Messung der Job-Übermittlungszeit.

Zu erkennen ist, dass die Anzahl der LLC-Misses bei mehr als 6.000 zu übertragenden Jobs deutlich ansteigt. Dies ist der Grund für den Anstieg der Kosten bei der Einplanung eines neuen Jobs: Da mehr Zugriffe zum Hauptspeicher notwendig sind, um die zu verschiebenden Jobs zu laden, steigt entsprechend die benötigte Gesamtzeit.

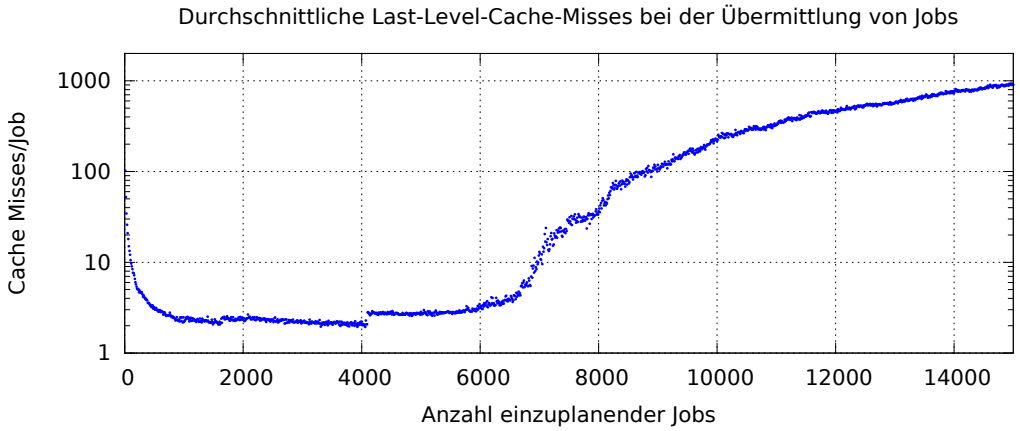


Abbildung 5.4: Cache-Misses bei der Übermittlung eines Jobs

Die erhöhte Anzahl an LLC-Misses speziell bei der Übermittlung bis zu 1.000 Jobs ist eine Ungenauigkeit bedingt durch die verwendete Messmethode: Die Ergebnisse durch die Initialisierung von Perf [dM] sowie der Start der Messanwendung verfälschen die Ergebnisse. Um hier genauere Ergebnisse zu erhalten müssen die Performance-Counter des Phenom-Prozessors [Dro08] direkt vor und nach der Job-Übermittlung ausgelesen und ausgewertet werden.

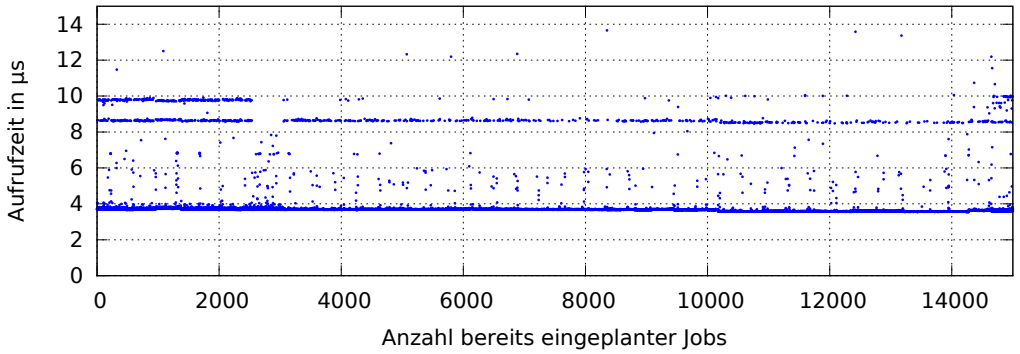
In einem weiteren Experiment wurde die Zeit für die Einplanung von bis zu 100.000 Jobs untersucht. Dabei kam es zu einem stärkeren Anstieg der maximalen Kosten bei mehr als 42.000 eingeplanten Jobs. Ab 60.000 eingeplanten Jobs stiegen die maximalen Kosten pro Job-Einplanung im durchgeführten Experiment linear an. Grund dafür ist, dass bei jeder Einplanung die zu verschiebenden Jobs neu aus dem Hauptspeicher geladen werden müssen und damit Einflüsse des Caches ausbleiben. Da eine so große Zahl eingeplanter Jobs untypisch ist und für die Praxis keine Bedeutung hat, wurde an dieser Stelle auf das entsprechende Diagramm verzichtet.

5.3.4 Kosten für den Start eines neuen Jobs

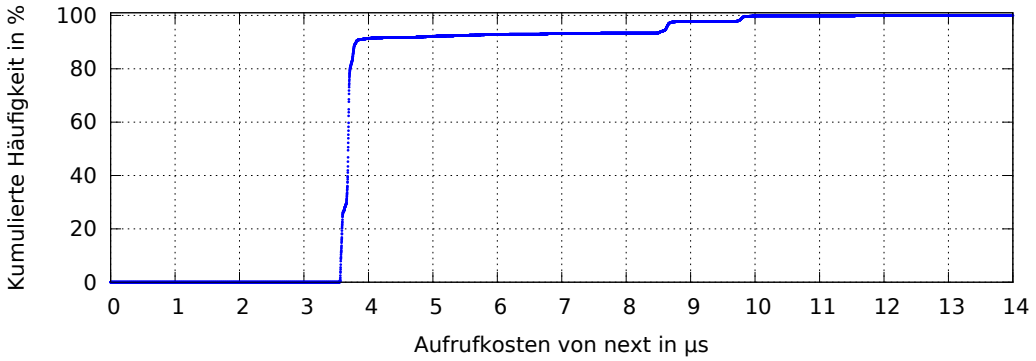
Die Kosten für den Betriebssystemaufruf `next`, der zur Implementierung des gleichnamigen Primitives verwendet wird, setzen sich zusammen aus der Ermittlung des nächsten Jobs für einen Thread sowie der Freigabe des zuletzt abgearbeiteten Jobs. Die Freigabe erfordert das Entfernen des Jobs aus dem Ablaufplan des ATLAS-Schedulers.

Aufbau und Durchführung der Messung

Für die Übermittlung der Jobs wird der gleiche Messaufbau wie im vorherigen Abschnitt 5.3.3 genutzt. Anstatt die Dauer einer Job-Übermittlung zu messen, wird die Dauer des Betriebssystemaufrufs `next` gemessen. Die Abarbeitung eines Jobs besteht dabei aus der Durchführung der Zeitmessung sowie der Ausgabe der gemessenen Zeit für eine Weiterverarbeitung.



(a) Absolute Kosten



(b) Kumulierte relative Häufigkeit

Abbildung 5.5: Kosten für den Aufruf von `next`

Auswertung

Abbildung 5.5(a) zeigt die absoluten Kosten für den Aufruf von `next`. Zur besseren Veranschaulichung der gemessenen Werte dient Abbildung 5.5(b): Rund 90% der Aufrufe benötigen eine Zeit von weniger als 3,8 µs. Bei 99,8% der gemessenen Werte liegen die Kosten für den Aufruf bei unter 10 µs.

Beim Abschluss eines Jobs wird dieser aus der Datenstruktur des Ablaufplans entfernt. Dafür wird ein Rot-Schwarz-Baum verwendet, der ausbalanciert ist. Je nach Situation erfordert das Löschen die Aktualisierung des Baumes zur Erhaltung der Struktureigenschaften und ist eine mögliche Erklärung für die unterschiedlichen Kosten. Zusammenfassend betrachtet sind die Kosten für den Aufruf von `next` aber vernachlässigbar.

5.4 FFplay

Mittels der im Abschnitt 5.3 beschriebenen Mikrobenchmarks wurden einzelne Aspekte des Schedulers evaluiert. In diesem Abschnitt wird der Scheduler als eine komplette Einheit getestet. Dabei soll unter anderem das Verhalten bei Überlastsituationen in

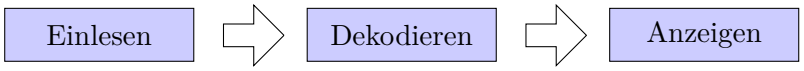


Abbildung 5.6: Abarbeitungsstufen bei der Anzeige eines Videos mit angepasstem FFplay

CFS verifiziert werden. Ebenso wird die korrekte Ausführung des Schedulers bei der Einplanung von Anwendungen untersucht, deren Threads sich nicht an die Spezifikation der übermittelten Jobs halten.

FFmpeg [ffm] ist ein platformübergreifendes Projekt verschiedener Entwickler zur Aufnahme, Wiedergabe, Konvertierung und zum Streamen von Audio- und Videoinhalten bestehend aus einer Reihe von Anwendungen und Bibliotheken. Vertrieben wird FFmpeg unter der *General Public License* [gpl] beziehungsweise *GNU Lesser General Public License* [lgp].

FFplay [ffp] ist ein einfacher Video-Player, der im Rahmen von FFmpeg erstellt wird. Neben den von FFmpeg bereitgestellten Funktionen nutzt FFplay eine weitere Bibliothek mit der Bezeichnung *Simple DirectMedia Layer* (SDL)[sdl]. Genau wie FFmpeg ist SDL eine platformübergreifende Bibliothek, die einen einfachen Zugriff auf die Ein- und Ausgabe des Systems ermöglicht.

Im Rahmen des ATLAS-Projekts wurde am Lehrstuhl eine angepasste Version von FFplay erstellt, welche die Funktionalität von ATLAS nutzt und sich somit zum Testen des ATLAS-Schedulers eignet.

5.4.1 Funktionsweise des angepassten FFplays

Beim Abspielen eines Videos mit der angepassten Version von FFplay werden drei Stufen durchlaufen, die in Abbildung 5.6 dargestellt sind.

In der ersten Stufe werden die notwendigen Daten gelesen, welche sich entweder schon im Speicher des Systems befinden oder noch von der Festplatte zu lesen sind. Die Dekodierung der gelesenen Daten zum fertigen Bild erfolgt in der zweiten Stufe. Die Anzeige des fertigen Bildes, welches auch als *Frame* bezeichnet wird, ist Aufgabe der letzten Stufe.

Für die Entkopplung der einzelnen Stufen werden zwischen diesen Puffer zur Speicherung der Daten verwendet. Vergleicht man die Zeitanforderungen der einzelnen Stufen der Abarbeitung, dann benötigt die Dekodierung die meiste Rechenzeit. Die Anzeige hingegen hat strikte Zeitvorgaben, die sich von den Eigenschaften des abzuspielenden Videos ableiten: Ein Video, das mit 25 Bildern pro Sekunde abgespielt wird, erfordert die Anzeige eines Bildes aller 40 ms. Die Anzeige in der letzten Stufe bei der Verarbeitung bestimmt somit den weichen Echtzeitcharakter der Anwendung. Die Zeitvorgaben des Videos müssen eingehalten werden, um die vom Benutzer erwartete Qualität zu gewährleisten.

Die praktische Umsetzung der verwendeten Version von FFplay nutzt vier Threads für das Abspielen des Videos. Je ein Thread wird dabei für das Einlesen der Daten und für die Dekodierung der Frames verwendet. Die Umsetzung der Anzeige durch zwei Threads hat technische Gründe und ist aufgrund der Nutzung der SDL-Bibliothek erforderlich. Genauere Details zur Umsetzung sind nicht Gegenstand der Arbeit.

Alle vier Threads nutzen die ATLAS-Lib zusammen mit dem Estimator zur Übermittlung von Jobs. Die geschätzten Ausführungszeiten des Estimators wurden in den Experimenten mit einem Faktor von 1,1 multipliziert, um fehlerhafte Schätzungen des Estimators auszugleichen. Die Vermutung liegt nahe, dass das Testsystem stark durch Störlast beeinflusst wird, weil in anderen Systemen bereits ein Faktor von 1,01 ausreichend ist, um genaue Schätzungen des Estimators zu erhalten.

5.4.2 Durchführung der Messung

Zur Untersuchung des ATLAS-Schedulers werden die Abstände der Anzeige von je zwei aufeinanderfolgenden Frames beim Abspielen eines Videos [BBC] bestimmt. Die Messungen werden dabei sowohl mit einem vom ATLAS-Scheduler eingeplanten FFplay als auch mit einem von CFS eingeplanten FFplay ausgeführt und anschließend verglichen.

Die Messung wird mit drei verschiedenen Systemzuständen durchgeführt, die sich durch die Zusammensetzung der Hintergrundlast unterscheiden:

Z1 *Nicht ausgelastetes System.* Die Messung in diesem Systemzustand wird mit einem Minimum laufender Hintergrundlast ausgeführt. Die zur Verfügung stehenden Rechenkapazitäten sind deshalb ausreichend und es kommt zu keinen Überlastsituationen bei der Ausführung von FFplay.

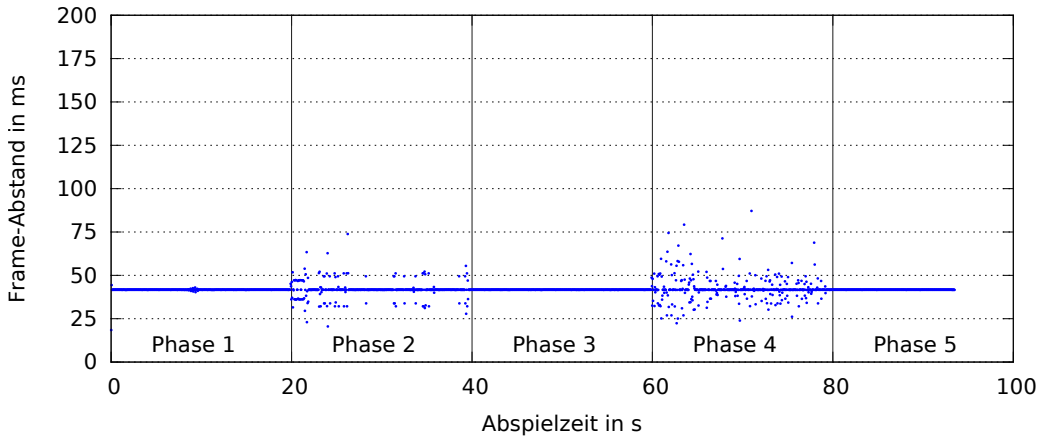
Z2 *Ausgelastetes System mit Threads eingeplant von CFS.* In diesem Zustand werden auf der CPU, die auch zur Ausführung von FFplay benutzt wird, zehn Threads in CFS gestartet, die eine Endlosschleife abarbeiten und somit eine Überlastsituation erzeugen. FFplay konkurriert mit den Threads um die zur Verfügung stehende Rechenzeit.

Z3 *Ausgelastetes System mit Threads eingeplant vom ATLAS-Scheduler.* Gestartet wird eine Anwendung, die Jobs für zehn Threads übermittelt. Die Jobs werden dabei mit einem Abstand von 40 ms und einer beantragten Rechenzeit von 12 ms übertragen. Die Parameter führen dazu, dass 30% des Ablaufplans mit Jobs der Anwendung ausgefüllt sind. Eine Überlastsituation im Ablaufplan bei der zusätzlichen Einplanung von FFplay ist damit unwahrscheinlich, da FFplay nicht mehr als 50% der Rechenkapazität einer CPU nutzt.

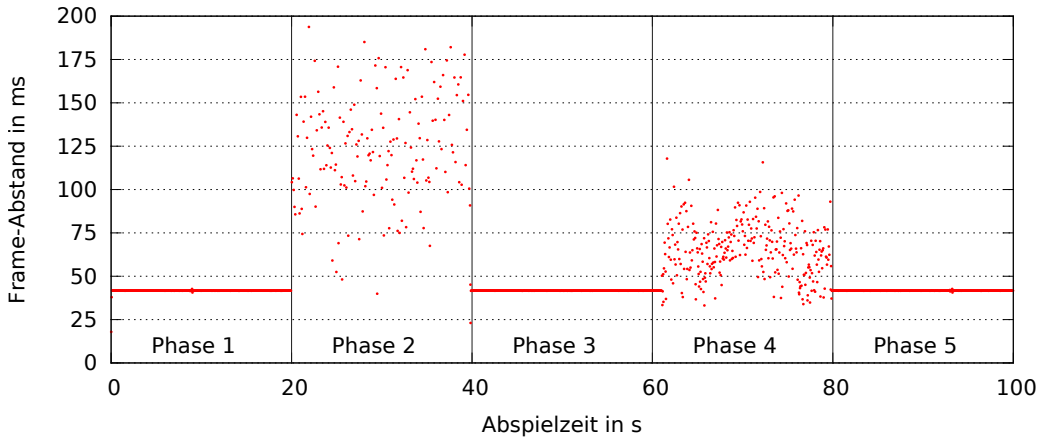
Die Hälfte der Threads beendet die Berechnung eines Jobs noch bevor die zugewiesene Rechenzeit vollständig aufgebraucht ist. Die andere Hälfte der Threads überschreitet die gemeldete Ausführungszeit sowie die Deadline. Die Konsequenz ist eine Verschiebung der Threads nach CFS und die Nutzung der zur Verfügung stehenden Rechenzeit. Der Systemzustand erlaubt die Evaluierung des ATLAS-Schedulers, wenn neben FFplay noch eine weitere Anwendung vom ATLAS-Scheduler eingeplant wird. Außerdem wird untersucht, in wie weit Threads, die sich nicht an die Spezifikation halten, die Ausführung von FFplay beeinträchtigen.

5.4.3 Auswertung

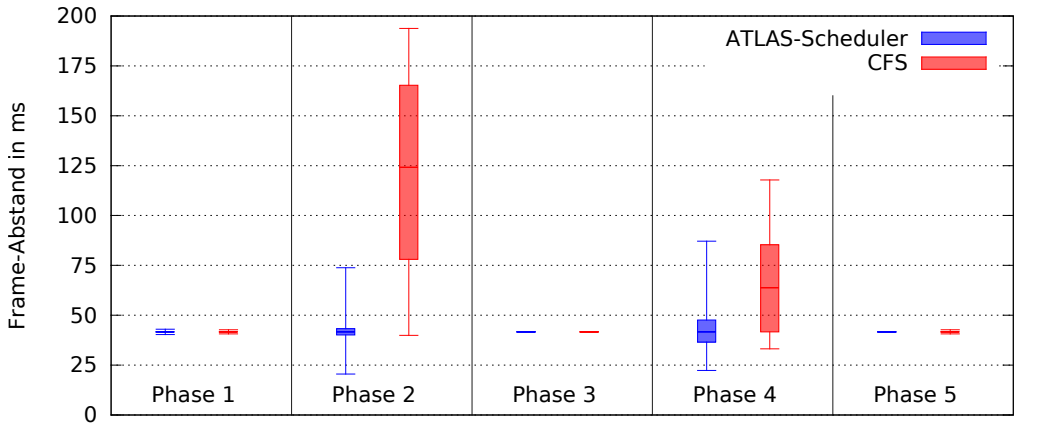
Abbildung 5.7 zeigt den Zeitabstand zweier aufeinanderfolgender Frames in Abhängigkeit der Abspielzeit des Videos mit FFplay. Die Wiedergabe wurde dabei in fünf Phasen mit



(a) Einplanung durch ATLAS-Scheduler



(b) Einplanung durch CFS



(c) Vergleich des ATLAS-Schedulers mit CFS mittels angepasster Boxplots

Abbildung 5.7: Zeit zwischen zwei aufeinanderfolgenden Frames beim Abspielen eines Videos mit FFplay

je 20 Sekunden Länge und wechselnden Systemzuständen eingeteilt. In Abbildung 5.7(a) erfolgt die Wiedergabe durch die Einplanung mit dem ATLAS-Scheduler, in Abbildung 5.7(b) dagegen bei Nutzung von CFS. In beiden Fällen wurde dasselbe Video mit einer Framerate von 24 Bildern je Sekunde abgespielt. Der optimale Abstand zwischen zwei aufeinanderfolgenden Frames liegt somit bei $41\frac{2}{3}$ms.

Für eine bessere Vergleichbarkeit wurden die durchgeführten Messungen in Abbildung 5.7(c) als angepasste Boxplots aufbereitet und zusammengefasst. Der blaue Boxplot in jeder Phase repräsentiert die Messungen mit dem ATLAS-Scheduler, der rote Boxplot mit CFS. Die untere Antenne einer Box zeigt das gemessene Minimum, gefolgt vom Perzentil P_{10} an der Unterkante der Box. Der Median jeder Messung wurde durch einen waagerechten Strich innerhalb der Box markiert. Analog befindet sich an der Oberkante jeder Box das Perzentil P_{90} und am Ende der oberen Antenne der gemessene maximale Abstand zweier aufeinanderfolgender Frames. Innerhalb einer Box befinden sich insgesamt 80% der gemessenen Werte. Originale Boxplots nutzen die Quartile als Grenzen für die Boxen, in denen sich nur 50% der Werte befinden. Die gewählte Form erlaubt eine genauere Analyse der Messungen.

Phase 1

In der ersten Phase zwischen 0 und 20 Sekunden wurde das Video in einem System im Zustand 1 betrieben. Dementsprechend ist in beiden Fällen die Zeit zweier aufeinanderfolgender Frames konstant und entspricht in etwa der optimalen Frame-Rate. Dies kann mit Abbildung 5.7(c) validiert werden: Die Abweichung zwischen dem minimalen und maximalen Abstand ist sowohl beim ATLAS-Scheduler als auch bei CFS vernachlässigbar. Entsprechend wird das Video in beiden Fällen mit einer hoher Qualität abgespielt.

Phase 2

Nach 20 Sekunden Laufzeit des Videos wurde in den Systemzustand 2 gewechselt. Bei einer Einplanung mit CFS steigt die durchschnittliche Zeit zwischen zwei aufeinanderfolgenden Frames um ein Vielfaches auf 124 ms: Der maximal gemessene Abstand beträgt 193,8 ms. 80% der Frame-Abstände verteilen sich auf ein Band von 78 ms bis 165 ms. Dementsprechend sinkt die Qualität beim Abspielen des Videos auf ein inakzeptables Niveau für den Betrachter: Das Video wird ruckelnd und zu langsam abgespielt.

Die in CFS eingeplanten Threads führen dazu, dass die für FFplay zur Verfügung stehende Rechenzeit nicht ausreichend ist. Die Anzeige der Frames erfolgt regelmäßig zu spät, weil die Rechenzeit für die Dekodierung zu gering ist. Je mehr Threads die CPU in CFS nutzen, desto geringer wird die nutzbare Rechenzeit aus Sicht von FFplay.

Beim Abspielen des Videos in Phase 2 mit der Einplanung durch den ATLAS-Scheduler vergrößert sich die durchschnittliche Zeit aufgrund einzelner Ausreißer zwischen zwei aufeinanderfolgenden Frames um vernachlässigbare 90 µs. Damit wird das Video im Gegensatz zur Einplanung mit CFS mit der richtigen Geschwindigkeit abgespielt. Abbildung 5.7 zeigt, dass mindestens 80% der Frames im richtigen Moment angezeigt werden.

Der Einfluss der Hintergrundlast bei der Einplanung durch den ATLAS-Scheduler ist gering. Abweichungen beim Abstand aufeinanderfolgender Frames basieren auf ungenauen

Schätzungen des ATLAS-Schedulers: Überschreitet ein Thread die Rechenzeit, dann wird er an CFS abgegeben. Eine Fertigstellung des Jobs dort ist schwierig, da die zugewiesene Rechenzeit zu gering ist. Erhöht man die Schätzungen der Ausführungszeit des Estimators um 50%, dann ist kein Einfluss der Hintergrundlast in CFS mehr feststellbar und die Anzahl von Ausreißern sinkt.

Die Betrachtung des Videos bei der Einplanung durch den ATLAS-Scheduler in ausgelasteten Systemen erfüllt die Qualitätserwartungen des durchschnittlichen Nutzers. Das Video läuft flüssig, nur in Ausnahmefällen kommt es zu kaum wahrnehmbaren Beeinträchtigungen durch eine ruckelnde Wiedergabe. Wichtig ist, dass bei der Einplanung mit dem ATLAS-Scheduler das Video weiterhin mit konstanter Geschwindigkeit abgespielt wird.

Phase 3

Nach 40 Sekunden Laufzeit des Videos wurde in Phase 3 das System wieder im Zustand 1 betrieben, um zum Beginn der nächsten Phase dieselben Voraussetzungen für beide Video-Player zu erhalten.

Phase 4

Ab 60 Sekunden Videolaufzeit erfolgt in Phase 4 die Ausführung von FFplay im Systemzustand 3. Die Einplanung mittels CFS ist vergleichbar zu Phase 2. Insgesamt ist die Streuung der Abstände geringer, weil sich nur fünf der zehn Threads, die ATLAS nutzen, nicht an die vorgegebene Spezifikation halten und von CFS eingeplant werden. Damit ist die Systemlast geringer im Vergleich zum Zustand 1. Die Wiedergabe des Videos erfolgt zu langsam und mit keiner zufriedenstellenden Qualität.

Die Einplanung von FFplay durch den ATLAS-Scheduler wird stärker im Vergleich zu Phase 2 beeinflusst: Der durchschnittliche Zeitabstand zwischen zwei aufeinanderfolgenden Frames steigt um 0.58 ms auf einen Wert von 42.25 ms. Erkennbar ist, dass es mehr Ausreißer gibt. Die Streuung der Werte ist wie in Abbildung 5.7(c) darstellt in Phase 4 erhöht.

Die größere Streuung der Werte basiert auf Verschiebungen der Jobs im Ablaufplan: Der Start der Anwendung zur Generierung von Hintergrundlast zum Zeitpunkt $t = 60\,\mathrm{s}$ plant für die nächsten 20 Sekunden Jobs ein. Deswegen werden die Jobs von FFplay bei der Einplanung entsprechend verschoben in den Ablaufplan eingefügt wie im Abschnitt 3.4.3 diskutiert. Bei der Abarbeitung des Ablaufplans werden die Jobs dann entsprechend eher gestartet oder, wenn Blockierungen zu kurzzeitigen Überlastsituationen führen, auch entsprechend später in ATLAS-Recover ausgeführt.

Bei der Betrachtung des Videos konnte ich keinen Unterschied zu Phase 2 feststellen. Die Qualität ist zufriedenstellend.

Weil sich fünf der zehn Hintergrund-Threads nicht an die Spezifikation ihrer Jobs halten, konnte in Phase 4 ebenfalls die korrekte Ausführung des ATLAS-Schedulers in dieser Situation nachvollzogen werden. Die entsprechenden Threads wurden bei der Überschreitung der angegebenen Rechenzeit von CFS eingeplant und haben die Ausführung der sich richtig verhaltenden Threads einschließlich FFplay nicht beeinflusst.

Phase 5

Phase 5 ab 80 Sekunden Videolaufzeit dient nur noch zum Abschluss des Videos und ist nicht relevant für weitere Betrachtungen.

Die Einplanung von FFplay mittels des ATLAS-Schedulers hat sich damit in allen getesteten Phasen bewährt.

6 Zusammenfassung

In der Arbeit habe ich den Entwurf, die Implementierung und die Auswertung des ATLAS-Schedulers beschrieben. An unterschiedlichen Stellen ergaben sich dabei verschiedene Lösungsansätze, die nicht alle innerhalb dieser Arbeit bis ins Detail betrachtet werden konnten. Die folgende Liste enthält deshalb einen Ausblick möglicher weiterführender Arbeiten, die die Funktionalität des ATLAS-Schedulers weiter optimieren können:

- *Mehrprozessorscheduling.* Im Abschnitt 3.7 wurde bereits ein Ansatz für das CPU-Scheduling auf mehreren Prozessoren vorgestellt. Damit werden Anwendungen, die ATLAS nutzen, in den Ablaufplänen verschiedener CPUs eingeplant und bei der Ausführung entsprechend migriert. In einer weiterführenden Arbeit kann dieser Ansatz als Basis für einen Entwurf sowie für die Implementierung genutzt werden.

- *Analyse und Umsetzung von Sicherheitsanforderungen.* Bei der Anforderungsanalyse des ATLAS-Schedulers habe ich bewusst auf Themen der Sicherheit verzichtet, da die Funktionalität im Vordergrund stande. Für einen praktischen Einsatz muss eine Sicherheitsanalyse erfolgen und der ATLAS-Scheduler entsprechend angepasst werden. Beispielsweise ist zu prüfen, ob es eine obere Grenze für die Rechenzeit von Jobs geben sollte, um eine zu lange Ausführung böswilliger Anwendungen durch den ATLAS-Scheduler zu verhindern. Entsprechend ist zu überprüfen, ob die maximale Anzahl von Job-Übermittlungen einzuschränken ist.

- *Umgang mit Überlast.* Im Abschnitt 3.6 wurde ein Ansatz für den Umgang mit Überlast beschrieben, der das Kürzungspotential von Jobs nutzt, um deren Rechenzeit im Falle einer Überlast zu kürzen. Der beschriebene Ansatz ist vielversprechend und muss genauer untersucht werden. Es gilt zu überprüfen, wie die Kommunikation zwischen ATLAS-Scheduler und Anwendung bei der Kürzung von Jobs erfolgen kann.

- *Repräsentation und Übermittlung relevanter Informationen durch das /proc-Dateisystem.* Im Abschnitt 4.1.2 wurde bereits die Verwendung des `/proc`-Dateisystems für die Umsetzung der ATLAS-Primitive diskutiert. Der Ansatz erlaubt die elegante Umsetzung des Primitives `cancel` und die Weitergabe von Informationen bereits übermittelter Jobs aus dem Kern wieder zurück ins Userland.

- *Test von ATLAS mittels weiterer Anwendungen.* Für die Evaluierung des ATLAS-Schedulers „als Ganzes“ wurde FFplay verwendet. Die Anpassung weiterer Anwendungen zur Nutzung von ATLAS erlaubt zusätzliche Untersuchungen der Eigenschaften des ATLAS-Schedulers.

Die Evaluierung in Kapitel 5 hat gezeigt, dass der ATLAS-Scheduler auch in ausgelasteten Systemen mit weichen Echtzeitanwendungen, wenn diese das ATLAS-Konzept nutzen, die benötigten Rechenressourcen zur Verfügung stellt. Für das getestete FFplay konnte somit eine Einplanung durchgesetzt werden, die für den Benutzer die Wiedergabe eines Videos mit hoher Qualität unabhängig von der Auslastung des Systems erlaubt. Selbst eine deutliche Überlastsituation in CFS führte nur zu einer minimalen Verschlechterung der Videoqualität, die auf ungenaue Schätzungen des Estimators zurückgeführt werden können.

In derzeitigen Systemen erfolgt die Einplanung von Anwendungen mit weichen Echtzeitanforderungen als Best-Effort-Anwendungen mit Hilfe von Prioritäten. Eine solche Einplanung erfordert eine globale Systemsicht für Aussagen bezüglich Rechenzeitzuweisungen. Die Nutzung des ATLAS-Konzepts erlaubt Anwendungen eine erfolgreiche Einplanung, ohne dass die Anwendung selbst ein globales Wissen über das System benötigt. Dies ist möglich, indem Rechenzeitanforderungen in Form von Jobs durch die Anwendung selbst an den ATLAS-Scheduler übertragen werden. Dieser Informationsfluss erlaubt anschließend eine bessere Einplanung.

Es liegt in der Natur interaktiver Systeme, dass aufgrund der Nutzereingaben eine Vielzahl von Anwendungen mit weichen Echtzeitanforderungen ausgeführt werden. Warum funktionieren diese Systeme aber, wenn Echtzeitanwendungen wie normale Anwendungen vom gleichen CPU-Scheduler eingeplant werden? Entweder gibt es keine Hintergrundlast oder aber die verfügbaren Rechenressourcen sind so dimensioniert, dass auch bei entsprechender Hintergrundlast ausreichend viel Rechenzeit zur Verfügung steht.

Nimmt der Benutzer eines Smartphones ein Bild auf und hört gleichzeitig Musik, ist nicht davon auszugehen, dass es keinerlei Hintergrundlast wie beispielsweise das Abspeichern des aufgenommenen Fotos gibt. Entsprechend müssen also genügend Rechenressourcen zur Verfügung stehen. Setzt man diese Gedankengänge fort und denkt über eine Einplanung der Anwendungen mittels ATLAS nach, dann können die gleichen Anwendungen auch mit weniger Rechenressourcen aber der gleichen Qualität umgesetzt werden. Zwar werden Hintergrundanwendungen erst später eingeplant, aber stattdessen sind weniger aktiv zur Verfügung stehende Rechenressourcen erforderlich, die zu einem geringeren Energieverbauch führen, was besonders bei mobilen Geräten mit Akku wichtig ist.

Die Umsetzung von ATLAS zeigt, dass es besser funktionierende Alternativen als das derzeitig verwendete prioritätsbasierte CPU-Scheduling für die Einplanung von Anwendungen mit weichen Echtzeitanforderungen gibt und dass es lohnenswert ist, auf diesem Gebiet weiterzuarbeiten.

Literaturverzeichnis

[ABSA98] Luca Abeni, Giorgio Buttazzo, Scuola Superiore, and S. Anna. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, 1998. 15

[ACL+05] Luca Abeni, Tommaso Cucinotta, Giuseppe Lipari, Luca Marzario, and Luigi Palopoli. QoS management through adaptive reservations. *REAL-TIME SYSTEMS*, 2005. 15

[AL02] Luca Abeni and Giuseppe Lipari. Implementing Resource Reservations in Linux. In *In Proceedings of the Fourth Real-Time Linux Workshop*, 2002. 8

[bab] Babeltrace. `http://lttng.org/babeltrace`. 51

[Bay72] Rudolf Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Inf.*, pages 290–306, 1972. 31

[BBC] BBC motion gallerie. `http://movies.apple.com/movies/us/hd_gallery/gl1800/bbc-r_m720p.mov`. 61

[BC06] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, 3. edition, 2006. 9

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, ATEC, 2005. 49

[CCAP10] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning schedulers for legacy real-time applications. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys, 2010. 16

[Cor] J. Corbet. The high-resolution timer API. `http://lwn.net/Articles/167897/`. 43

[CRM91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI, April 1991. 6

[ctf] Common Trace Format. `http://www.efficios.com/ctf`. 50

[DC99] Kenneth J. Duda and David R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP, 1999. 16

[Der74] Michael L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In *IFIP Congress*, pages 807–813, 1974. 10

[dM] Arnaldo Carvalho de Melo. The New Linux 'perf' Tools. `http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf`. 58

[Dro08] Paul J. Drongowski. *Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors*. AMD, September 2008. 58

[ecl] Eclipse. `http://www.eclipse.org/`. 51

[ffm] FFmpeg. `http://ffmpeg.org/`. 60

[ffp] FFplay. `http://ffmpeg.org/ffplay.html`. 60

[Gar09] Ankita Garg. Real-Time Linux Kernel Scheduler. *Linux Journal*, August 2009. `http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler?page=0,0`. 8

[GN06] Thomas Gleixner and Douglas Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Linux Symposium*, 2006. 43

[gpl] GNU General Public License. `http://www.gnu.org/licenses/gpl.html`. 60

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978. 31

[int11] *Intel®64 and IA-32 Architectures, Software Developer's Manual, Volume 1: Basic Architecture*, May 2011. 55

[Jon09] M. Tim Jones. Inside the Linux 2.6 Completely Fair Scheduler. Providing fair access to CPUs since 2.6.23. *IBM developer works*, December 2009. 7

[KSSG09] Charles Krasic, Mayukh Saubhasik, Anirban Sinha, and Ashvin Goel. Fair and timely scheduling via cooperative polling. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys, 2009. 16

[lgp] GNU Lesser General Public License. `http://www.gnu.org/licenses/lgpl.html`. 60

[lin] Linux Kernel Dokumentation (Nice-Level). `http://lxr.linux.no/#linux+v3.2.7/Documentation/scheduler/sched-nice-design.txt`. 16

[Liu00] Jane W. S. Liu. *Real-time systems*. Prentice Hall, Upper Saddle River, NJ, 2000. 4, 10

[ltta] Eclipse LTTng plugin. `http://lttng.org/eclipse`. 51

[lttb] Linux Trace Toolkit - next generation. `http://lttng.org/`. 50

[Mol07] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler (CFS), April 2007. `http://lkml.org/lkml/2007/4/13/180`. 7

[Pal] Venkatesh Pallipadi. Proper kernel IRQ time accounting. `http://lwn.net/Articles/405889/`. 53

[PCL08] Luigi Palopoli, Tommaso Cucinotta, and Giuseppe Lipari. AQuoSA - Adaptive Quality of Service Architecture, 2008. 15

[RWH13] Michael Roitzsch, Stefan Wächtler, and Hermann Härtig. ATLAS: Look-Ahead Scheduling Using Workload Metrics. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS. IEEE, April 2013. wird veröffentlicht. 1, 12, 13

[sdl] Simple DirectMedia Layer. `http://www.libsdl.org/`. 60

[Soc08] IEEE Computer Society. *Standard for Information Technology - Portable Operating System Interface (POSIX®), Base Specifications, Issue 7, IEEE Std 1003.1-2008*. IEEE, New York, NY, Dezember 2008. 8, 43

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, May 2008. 13

[Wil10] Matthew Wilcox. I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers, 2010. 53

[Wä12] Stefan Wächtler. Evaluation of migration costs in multi-core scheduling. Großer Beleg, TU Dresden, März 2012. 9

[YLB+08] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2008. 16

[YN] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. 6