Diplomarbeit

Rechenzeitabstraktion durch kontengesteuertes Scheduling

Stefan Lebelt
lebelt@os.inf.tu-dresden.de

22. September 2006

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dr. rer. nat. Claude-Joachim Hamann

Erklärung

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

Dresden, den 22. September 2006

Stefan Lebelt

Danke

Ich bedanke mich an dieser Stelle bei Prof. Dr. Hermann Härtig für die Möglichkeit, in der Betriebssystemgruppe arbeiten zu dürfen. Besonderer Dank gilt ebenfalls meinem Betreuer Dr. Claude Hamann für die sehr gute Unterstützung und Zusammenarbeit, des Weiteren Jean Wolter und Adam Lackorzynski für die große Geduld bei der Beantwortung vieler Fragen sowie Michael Roitsch und Andre Haferkorn. Ausserdem danke ich ganz besonders meinen lieben Eltern, die immer hinter mir stehen und mein Studium ermöglicht haben.

Inhaltsverzeichnis

1 Einleitung **9**

2 Grundlagen und Stand der Technik **11**
2.1 Grundsätzliches . . . 11
2.1.1 Mikrokernkonzept . . . 11
2.1.2 L4 . . . 11
2.1.3 Alienmodell . . . 12
2.1.4 Dealman-Konzept . . . 12
2.1.5 Semaphore . . . 13
2.2 Scheduling . . . 13
2.2.1 Round-Robin-Scheduling . . . 13
2.2.2 Fiasco-Scheduling . . . 14
2.2.3 Fair-Share-Scheduling . . . 14
2.2.4 Lottery-Scheduling . . . 14
2.2.5 CPU Inheritance Scheduling . . . 14
2.2.6 Scheduler-Activations . . . 15
2.2.7 Echtzeitscheduling . . . 15
2.3 Ökonomische Ansätze . . . 17
2.3.1 Auktionsbasierte Systeme . . . 17
2.3.2 GRIDs . . . 17
2.3.3 GRACE . . . 18

3 Entwurf **21**
3.1 Ziele . . . 21
3.2 Motivation . . . 22
3.2.1 Szenarien . . . 22
3.2.2 Zusammenfassung - allgemeine Ressourcenabstraktion . . . 25
3.3 Entscheidungen und deren Folgen . . . 26
3.3.1 Nutzung des Dealman-Konzeptes für die Ressource CPU . . . 26
3.3.2 Lottery-Scheduling . . . 26
3.3.3 Auktionen . . . 26
3.3.4 Userland-Scheduling . . . 26
3.3.5 Definition von Echtzeitthreads . . . 27
3.4 Schedulingmodell . . . 28
3.4.1 Das System . . . 28
3.4.2 Arbeitsweise des Schedulers . . . 30
3.4.3 Bedeutung der Priorität . . . 33
3.4.4 Zielabgleich . . . 35
3.4.5 Weitergehende Betrachtungen . . . 37

3.5 Budget-based-Userland-Scheduler . . . 39
3.5.1 Aufbau . . . 39
3.5.2 Scheduling . . . 39
3.5.3 Informationen über Threadzustände . . . 40
3.5.4 Service . . . 40

4 Implementierung **43**
4.1 Voraussetzungen und Aufbau . . . 43
4.2 Die wichtigsten Datenstrukturen . . . 43
4.2.1 Bereitliste . . . 43
4.2.2 Blockiertliste . . . 45
4.2.3 Senderwarteschlange, Liste der Echtzeitthreads, hpt-Liste . . . 45
4.2.4 Dealman-Hilfsstruktur t_index . . . 45
4.3 Alienguard . . . 45
4.3.1 Speichern des Timestampcounters beim Aktivwerden . . . 46
4.3.2 Allgemeiner Ablauf vor Beginn eines Syscalls . . . 46
4.3.3 Verwaltungsroutine vor Beginn des IPC-Syscalls . . . 46
4.3.4 Ablauf vor Beendigung des Syscall . . . 47
4.3.5 Verwaltungsroutine vor Beendigung des IPC-Syscalls . . . 47
4.3.6 Verbotene Syscalls . . . 47
4.3.7 Besonderheiten . . . 47
4.4 Watcher . . . 48
4.5 Scheduler . . . 48
4.5.1 Funktionsweise . . . 48
4.5.2 Besonderheiten bei Rückzahlungen nach lokalen Überweisungen . . . 50
4.6 Service . . . 50
4.6.1 Anmelden neuer Threads . . . 50
4.6.2 Ummelden als Echtzeitthread (Echtzeitadmission) . . . 51
4.6.3 Blockieren bis zum Beginn des nächsten Durchlaufes . . . 51
4.6.4 Lokale- und Dealmanüberweisungen . . . 51
4.7 Systemparameter und Optionen . . . 52
4.8 Start des Systems . . . 53
4.9 Einschränkungen des Prototypen . . . 53
4.9.1 Kein Paging . . . 53
4.9.2 Keine Interruptbehandlung . . . 53
4.9.3 Kein nachträgliches Starten von Threads/Tasks, kein Abmelden . . . 53
4.9.4 Kein Echtzeitscheduling mit dynamischen Prioritäten . . . 54
4.9.5 Keine Abgabe der Echtzeiteigenschaft . . . 54
4.9.6 Maximale Anzahl schedulbarer Threads . . . 54
4.9.7 Kein Lending . . . 54
4.10 Beispielszenarien . . . 54
4.10.1 IPC- und Timeoutszenario . . . 54
4.10.2 Echtzeitszenario . . . 55
4.10.3 Erzeuger-Verbraucher-Szenario . . . 55

5 Bewertung **57**
5.1 Mehraufwand (Overhead) . . . 57
5.1.1 Overhead des Modells - Durchlaufinitialisierung . . . 57
5.1.2 Overhead der Implementierung . . . 57
5.2 Optimierungspotential . . . 58
5.2.1 Optimierung des Modells - Durchlaufinitialisierung . . . 58
5.2.2 Optimierung der Implementierung . . . 59
5.3 Erzeuger-Verbraucher-Szenario . . . 59
5.4 Mehrwert . . . 61

6 Zusammenfassung **63**
6.1 Offene Probleme . . . 63
6.1.1 Implemenierung des Dealman! . . . 63
6.1.2 Beschränkungen . . . 63
6.2 Ausblick - Mehr-Knoten-Systeme . . . 63
6.3 Abschluss . . . 63

Abbildungsverzeichnis **64**

Tabellenverzeichnis **65**

Glossar **69**

Literaturverzeichnis **71**

1 Einleitung

Abstraktion (lat. Wegziehen) bezeichnet den „Vorgang des Abbildens von Objekten der realen Welt in solche einer Modellwelt“ [FH04]. Nahezu überall in der Informatik wird abstrahiert, um komplizierte Sachverhalte und Abläufe zu vereinfachen. Ein Beispiel dafür sind die vielen Programmiersprachen, die unterschiedlich stark von der Maschine abstrahieren. Ein Programmierer muss heute kaum noch wissen, wie der Computer, der sein Programm ausführt, funktioniert.
Obwohl eine solche Situation von einigen Experten als negativ empfunden wird, so scheint Vereinfachung eine wichtige Möglichkeit zu sein, in einer immer komplexer werdenden Welt, neue Probleme zu erkennen und zu lösen. Das neue Herausforderungen sich in einem einfachen und klar definierten System bedeutend leichter beschreiben und bearbeiten lassen liegt auf der Hand. Eine solche Einfachheit und Klarheit kann in vielen Fällen mittels Abstraktion erreicht werden.

In dieser Arbeit wird ein Schedulingmodell entworfen und prototypisch implementiert, das es ermöglicht von der Ressource CPU-Zeit zu abstrahieren. Damit soll die Vereinfachung von Problemen und Mechanismen wie zum Beispiel Donation oder Denial-of-Service-Sicherheit erreicht werden. Es wird zudem gezeigt, dass sich auf diese Weise ganz neue Lösungen bestimmter Aufgaben entwickeln lassen.

Gliederung

Die Arbeit ist in fünf Teile gegliedert. Das Folgende Kapitel beschäftigt sich mit dem derzeitigen Stand der Technik. Neben grundlegenden Konzepten werden insbesondere verschiedene Schedulingverfahren beschrieben. Darauf folgt der Entwurf, der unter anderem Ziele definiert und das Schedulingmodell vorgestellt. Im Kapitel Implementierung wird auf Details der prototypischen Umsetzung und verschiedene Beispielszenarien eingegangen. Zum Schluss folgen Bewertung und Zusammenfassung.

2 Grundlagen und Stand der Technik

2.1 Grundsätzliches

Das folgende Kapitel beschreibt grundlegende Techniken und Konzepte, die für diese Arbeit relevant sind. Die Informationen sind verschiedenen Quellen entnommen (Vlg. auch [Leb05]).

2.1.1 Mikrokernkonzept

Mikrokerne stellen im Gegensatz zu monolitischen Betriebssystemkernen nur grundlegende Dienste wie Adressräume, Aktivitäten und Kommunikation zur Verfügung. Treiber, Speicherverwaltung und Ähnliches können als Server im Userland implementiert werden.
Eine solche Architektur hat folgende Vorteile gegenüber monolithischen Systemen:

- Die Trusted Computing Base kann sehr viel kleiner sein.
- Es ist eine stärkere Trennung zwischen den verschiedenen Systemkomponenten möglich.
- Durch die Kompaktheit des Kerns ist es einfacher ihn zu verifizieren.

Dem Mikrokernansatz folgend, sollte bei Erweiterungen und der Entwicklung neuer Dienste versucht werden, diese im Userland und ohne Erweiterung des Kerns zu realisieren.

2.1.2 L4

L4 ist ein Mikrokern der zweiten Generation, das bedeutet, dass er nicht, wie z. B. Mach, durch den Umbau eines ehemals monolithischen Kerns, sondern von Grund auf als Mikrokern konstruiert wurde. L4 ist ursprünglich das Werk von Jochen Liedtke [Lie98].
Der Kern wird in Form von Michael Hohmuths Fiasco [Hoh] an der TU-Dresden eingesetzt und weiterentwickelt.

Tasks

Ein fundamentales Sicherheitskonzept sind Adressräume, also voneinander isolierte Speicherbereiche. Diese werden in L4 durch Tasks repräsentiert.

Threads

Threads sind die Aktivitäten. 128 Threads gehören zu einer Task und können mit dem Systemruf `lthread_ex_regs` aktiviert werden.

IPC

Während die Kommunikation zwischen Threads der selben Task über deren gemeinsam genutzten Speicher trivial ist, trifft das auf Inter-Prozess-Kommunikation (IPC) naturgemäß nicht zu. Taskübergreifende Kommunikation ist nur mit Hilfe des Mikrokerns möglich. IPC ist in L4 synchron. Das bedeutet, dass Sender und Empfänger zum Zeitpunkt der IPC dafür bereit sein müssen. Es werden im Kern keine Nachrichten gepuffert. Neben Short- und Long-IPC, bei denen jeweils tatsächlich Daten direkt oder indirekt von einem Adressraum in einen anderen kopiert werden, gibt es mit dem Flexpage-IPC auch die Möglichkeit, Teile des physischen Speichers anderen Tasks zu mappen und so gemeinsam genutzte Speicherbereiche zu erzeugen. Diese Technik bildet die Grundlage für Paging auf Nutzerebene.

2.1.3 Alienmodell

Für die Version zwei der Fiasco-L4-Kernels existiert eine experimentelle ABI-Erweiterung, die es erlaubt aus dem Userland heraus Syscalls bestimmter Threads zu kontrollieren. Dazu kann ein Thread in den Alienstatus versetzt werden. Vor und nach jedem, durch den Alien getätigten Syscall, wird eine Exceptionnachricht an dessen Pager gesendet. Erst nach der Bestätigung durch den Pager führt der Alien den Syscall durch, b. z. w. beendet ihn. Der Pager kann den Zustand des Alien zuvor verändern. Durch diesen Mechanismus ist es einem Userlandserver (z. B. einem Scheduler) möglich Informationen über den Zustand anderer Threads zu erhalten und bestimmte Syscalls für sie zu untersagen oder zu modifizieren.

2.1.4 Dealman-Konzept

Der Dealman-Bankserver ist ein Konzept, das ursprünglich zur „Vermeidung von Denial-of-Service-Angriffen in L4/DROPS“ [Leb05] eingeführt wurde. Nach Erstellung von [Leb05] hat sich herausgestellt, dass ein sehr ähnliches Kontosystem bereits 1986 in Amoeba eingeführt wurde [MT86, Kapitel 6], dessen Intension allerdings eine andere war.
Das Konzept beschreibt die Verwaltung von beschränkten Ressourcen durch die Einführung von virtuellem Geld, den Gums. Jeder Ressource wird ein Preis zugeordnet. Die Ressourcenprovider (Pager, Festplattentreiber, Scheduler, u. s. w.) teilen ihre Ressourcen nur nach Zahlung dieses Preises zu. Durch Überweisung von Gums von einem Thread zum anderen, ist die indirekte Übertragung von Anteilen an der Ressource möglich. Der Dealman-Bankserver bietet verschiedene in [Leb05, Kapitel 4.1.4, ab Seite 38] beschriebene Donation- und Lendingprotokolle an. Der allgemeine Fall des Lending wird im nächsten Abschnitt beschrieben.

Lending

Der Lendingprozess läuft wie folgt ab: Ein Thread t_1 sendet seinen Lendingauftrag an den Dealman-Bankserver. Dieser Auftrag beinhaltet folgende Daten:

- Empfängerthread t_2,
- Betrag,
- Lendingfrist.

Der Dealman startet daraufhin einen neuen Thread, den Lending-Observer. Dieser führt die Überweisung durch, indem er auf dem Konto des Empfängers t_2 ein Kontingent des Senders

t_1 anlegt und den entsprechenden Betrag vom Senderkonto auf dieses Kontingent überträgt. Danach blockiert er und wartet auf den Ablauf der Lendingfrist. Ist dieser Zeitpunkt erreicht, wird er aktiviert und invalidiert das betreffende Kontingent. Das invalidierte Kontingent steht t_2 nach wie vor zur Verfügung. Allerdings kann t_1 durch einen entsprechenden Aufruf an den Dealman dieses Kontingent auflösen und den Betrag zurücküberweisen lassen.

Authentifizierung

Der Dealman-Bankserver bietet zwei Möglichkeiten der Authentifizierung:

- die Absenderadresse des Auftraggebers der Donation und
- eine Capability, die beim Anlegen eines Kontos erzeugt und dem Besitzer übermittelt wird.

Das bedeutet, dass nur der Kontoinhaber selbst oder ein von ihm autorisierter Thread, der die Capability des Kontos besitzt, Geld von diesem Konto auf ein anderes überweisen kann.

2.1.5 Semaphore

Semaphoren sind Konstrukte zur Realisierung von wechselseitigem Auschluss und Synchronisation zwischen Threads. Ein Semaphore besteht aus den beiden Funktionen `P` (`up`) und `V` (`down`), einem Zähler und einer Wartschlange. Vor dem Betreten eines kritischen Abschnittes ruft ein Thread die Funktion `P` auf. Ist der kritische Abschnitt frei, so wird der Zähler dekrementiert und der Thread kann fortfahren. Ist der Abschnitt aber gesperrt (der Zähler hat den Wert 0), so wird der Thread in die Warteschlange eingereiht und blockiert. Bei Verlassen des kritischen Abschnittes ruft ein Thread die Funktion `V` auf. Der nächste wartende (blockierte) Thread wird nun aus der Warteschlange ausgekettet und aktiviert. Wartet kein Thread, so muss der Zähler inkrementiert werden.

2.2 Scheduling

„Unter Scheduling (englisch für Zeitplanerstellung), auch Zeitablaufsteuerung genannt, versteht man die Erstellung eines Ablaufplanes (schedule), der Prozessen zeitlich begrenzt Ressourcen zuweist“ [sch]. In dieser Arbeit bezeichnet Scheduling die Verwaltung und Vergabe der Ressource CPU. Im folgenden wird eine Vielzahl von Schedulingalgorithmen und -ansätzen vorgestellt.

2.2.1 Round-Robin-Scheduling

Laut [Tan02, Kapitel 2.5.3 - Scheduling in interaktiven Systemen, Seite 159] ist Round-Robin-Scheduling eines der meistbenutzten Schedulingverfahren. Dabei kann jeder Thread für ein kurzes Zeitquantum die Ressource nutzen. Nach dieser Zeit wird die Ressource entzogen und dem Nächsten zugeteilt.

2.2.2 Fiasco-Scheduling

Der Fiasco(kern)scheduler kennt 256 Prioritätsstufen. Es wird immer der jeweils höchstpriore bereite Thread aktiviert. Threads mit der gleichen Priorität werden nach dem Round-Robin-Verfahren geschedult. Blockiert ein Thread, so wird er aus der Bereitliste in die Blockiertliste umgekettet und nicht mehr aktiviert.

Time-Slice-Donation

Auf Fiasco ist es möglich Zeitscheiben zu donieren. Schaltet ein gerade laufender Thread z. B. mittels `l4_thread_switch` zu einem anderen ganz konkreten Thread hin, so übergibt er die verbleibende Zeit seiner Zeitscheibe an diesen Thread. Das Gleiche passiert bei IPC. Schickt Thread t_1 eine IPC-Nachricht an den wartenden Thread t_2, so wird zum Thread t_2 umgeschaltet. t_2 läuft dann auf der Zeitscheibe von t_1.

2.2.3 Fair-Share-Scheduling

Beim klassischen Round-Robin-Scheduling bekommt jeder Thread, unabhängig von seinem Besitzer, einen bestimmten zeitlichen Anteil an der Ressource. Das bedeutet aber, dass ein Benutzer, der mehrere Threads besitzt, mehr Anteil an dieser Ressource hat, als ein Benutzer mit beispielsweise nur einem Thread. Je nach Betrachtung kann das als unfair betrachtet werden (Vgl. [Tan02, Kapitel 2.5.3 - Scheduling in interaktiven Systemen, Seite 165]). Fair-Share-Scheduling beschreibt ein anderes Verhalten: Jedem Benutzer wird ein bestimmter Anteil der Ressource zugeteilt, unabhängig von der Anzahl seiner Threads.

2.2.4 Lottery-Scheduling

Ein Algorithmus, der Fair-Share-Scheduling unterstützt ist das so genannte Lottery-Scheduling [WW94]. Dabei besitzt jeder Thread eine Anzahl an Lotterielosen. Vor jeder Ressourcenzuteilung wird eine Verlosung durchgeführt, bei der zufällig eines der Lose ausgewählt wird. Der Gewinner erhält die Ressource für eine bestimmten Zeitraum. Danach wird erneut verlost. Je mehr Lose ein Thread besitzt, desto größer ist auf längere Sicht sein Anteil an der Ressource. Threads können Lose zwischeneinander übertragen. Auf diese Weise ist Donation und damit auch Fair-Share-Scheduling möglich, indem nur dem ersten Thread eines Benutzers eine bestimmte Menge an Losen zugeteilt wird. Da Lottery-Scheduling mit zufälligen Ereignissen (Lotterie) arbeitet, ergeben sich zwar auf längere Sicht die gewünschten Verteilungen der Anteile an der Ressource, kurzfristig gesehen kann dieses Verfahren aber auch zu unfairen Zuteilungen führen.

2.2.5 CPU Inheritance Scheduling

[FS96] beschreibt ein auf Donation basiertes Schedulingverfahren. Im Gegensatz zum klassischen Ansatz gibt es hier nicht einen zentralen Scheduler, sondern jeder Thread kann als Scheduler fungieren. Die Idee liegt darin, dass Threads von ihrem Scheduler Rechenzeit doniert bekommen. Diese Rechenzeit kann verbraucht werden, oder der Thread arbeitet selbst als Scheduler und doniert die Zeit entsprechend der Policy an seine Threads weiter. Auf diese Weise lassen sich unterschiedliche Schedulingalgorithmen und -policies in einem System betreiben.

2.2.6 Scheduler-Activations

Scheduler Activations [ABLL91, Cla05] beschreiben ein Konzept zur Unterstützung von User-Level-Threads durch den Kernel. Der Kern verwaltet dabei „virtuelle Prozessoren“ (Scheduler-Activations) und teilt sie Adressräumen (Tasks) zu. Jedes Threadsystem eines Adressraumes hat vollständige Kontrolle darüber, welcher Thread auf welchem „virtuellen Prozessor“ der Task läuft. Jedes schedulingrelevante Ereignis (Veränderung der Zuweisung von „virtuelle Prozessoren“, Blockieren von Threads, Erwachen von Threads, u. s. w.) teilt der Kern dem Threadsystem durch einen „upcall“ (Start der Ausführung an einem speziellen Eintrittspunkt) mit. Umgekehrt kann das Threadsystem durch entsprechende Syscalls den Kern über ungenutzte oder benötigte „virtuelle Prozessoren“ informieren.

2.2.7 Echtzeitscheduling

Echtzeit bedeutet Rechtzeitigkeit. Ein Echtzeitthread ist ein Thread, der seine Aufgabe zu einem ganz bestimmten Zeitpunkt erledigt haben muss, da sein Ergebnis sonst an Wert verliert oder sogar Schaden anrichtet. Erfolgt z. B. die Zündung einer Zündkerze im Ottomotor nur wenige Millisekunden zu spät, so hat der Kolben bereits seine Lage verändert und das Benzin ist nicht mehr maximal verdichtet.
Es gibt verschiedene Arten von Echtzeitthreads. In dieser Arbeit werden ausschließlich periodische Echtzeitthreads betrachtet, d. h. Threads, die ihre Aufgabe in periodischen Abständen erledigen müssen. Die Deadline eines periodischen Echtzeitthreads liegt genau am Beginn seiner nächsten Periode.
Die Algorithmen zum Scheduling von periodischen Echtzeitthreads lassen sich in zwei Klassen einteilen:

- Echtzeitschedulingalorithmen mit statischen Prioritäten und
- Echtzeitschedulingalrorithmen auf Basis dynamischer Prioritäten.

Im Folgenden wird jeweils ein Vertreter jeder Klasse vorgestellt.

Statische Prioritäten am Beispiel RMS

Statische Prioritäten bedeutet, dass bei der Zulassung (Admission) von Echtzeitthreads deren Prioritäten nach bestimmten Kriterien festgelegt werden und sich während der Laufzeit des Systemes nicht mehr ändern. Aktiviert wird der höchstpriore, bereite Thread. RMS steht für Ratenmonotones Scheduling [LL73, A Fixed Priority Scheduling Algorithm, Seite 49]. Der Name deutet bereits die grundlegende Vorgehensweise an. Die zugelassenen Threads werden nach steigenden Periodenlängen sortiert, d. h. je kürzer die Periode, desto höher die Priorität. Vor der Vergabe der Priorität erfolgt die Zulassungsprüfung (Admission). Ein neuer Thread kann nach RMS eingeplant werden, wenn folgende Bedingung erfüllt ist:

$$\sum_{i=1}^{n} \frac{e_i}{c_i} \leq n * (\sqrt[n]{2} - 1) \tag{2.1}$$

Dabei ist e_i die Ausführungszeit, c_i die Periodenlänge des Threads t_i und n die Anzahl der Echtzeitthreads. Dieses Kriterium ist hinreichend, aber nicht notwendig. Das folgende Beispiel, das einer Übungsaufgabe zur Vorlesung Betriebssysteme entnommen wurde, zeigt ein

Threadsystem, dass mit RMS eingeplant werden kann, obwohl das RMS-Zulassungskriterium nicht erfüllt ist.
Das Threadsystem besteht aus drei Threads, t_1, t_2 und t_3 mit folgenden Periodenlängen und Ausführungszeiten:

Thread	Ausführungszeit e_i	Periodenlänge c_i
t_1	$2s$	$5s$
t_2	$1s$	$3s$
t_3	$1s$	$5s$

Tabelle 2.1: RMS-Beispiel: Daten

Das Zulassungskriterium aus Formel 2.1 ist nicht erfüllt:

$$\sum_{i=1}^{n} \frac{e_i}{c_i} = \frac{2s}{5s} + \frac{1s}{3s} + \frac{1s}{5s} = \frac{14}{15} \approx 93,\bar{3}\% \tag{2.2}$$

$$n * (\sqrt[n]{2} - 1) = 3 * (\sqrt[3]{2} - 1) \approx 77,98\% \tag{2.3}$$

$$93,\bar{3}\% \not\leq 77,98\% \tag{2.4}$$

Abbildung 2.1 zeigt jedoch, dass das Threadsystem mit RMS eingeplant werden kann.

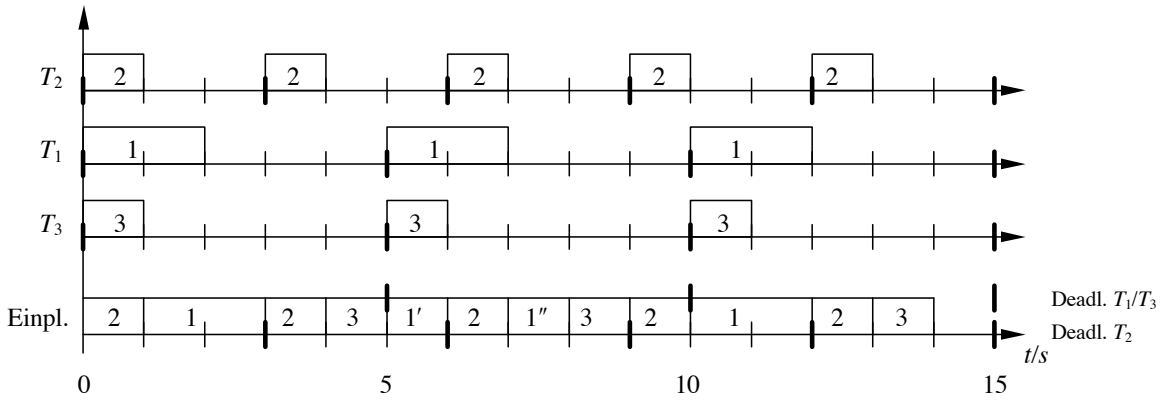

Abbildung 2.1: RMS-Beispiel: Einplanung

Das RMS-Verfahren ist optimal bezüglich Einplanbarkeit von Echtzeitthreads, mit statischen Prioritäten.

Dynamische Prioritäten am Beispiel EDF

Im Gegensatz zu RMS gibt es bei Verfahren mit dynamischen Prioritäten keine feste Prioritätszuordnung. Die Auswahl des zu aktivierenden Threads wird dabei nach anderen Kriterien bestimmt. Im Falle von EDF (Earliest Deadline First) [LL73, The Deadline Driven Scheduling Algorithm, Seite 55] prüft der Scheduler die nächsten Deadlines aller Threads und aktiviert den mit der zeitlich nächstgelegenen. Dem deutlich höheren Schedulingaufwand

steht bei diesem Verfahren ein weniger restriktives Zulassungskriterium gegenüber, das lediglich besagt, dass die Auslastung des Systems nicht größer sein darf als eins (Formel 2.5).

$$\sum_{i=1}^{n} \frac{e_i}{c_i} \leq 1 \tag{2.5}$$

Das EDF-Verfahren ist optimal bezüglich Einplanbarkeit von Echtzeitthreads.

2.3 Ökonomische Ansätze

Mark S. Miller und K. Eric Drexler stellen in ihre Agoric-Papers [MD88c, MD88b, MD88a] die These auf, dass sich Computersysteme im Hinblick auf Aktionen und Ressourcen als ökonomische Systeme begreifen und um Marktmechanismen erweitern lassen. Sie vergleichen dabei Speicher mit Land, Rechenzeit mit Treibstoff und Software mit Arbeitern.

2.3.1 Auktionsbasierte Systeme

In auktionsbasierten Systemen wird Rechenzeit versteigert. [MD88b, Kapitel 2] beschreibt verschiedene Arten von Auktionen. Diese lassen sich in zwei Kategorien einteilen:

1. in Auktionen, bei denen die Bieter (Threads) aktiv teilnehmen müssen (double auctions, English auctions, Dutch auctions) und

2. Auktionen, bei denen die Bieter nur ein einziges Gebot abgeben müssen und dann passiv bleiben können. (first-price- und second-price sealed-bid auctions)

Auktionen der zweiten Kategorie erscheinen als geeigneter, insbesondere die „second-price sealed-bid“ Aution, bei der das höchste Gebot gewinnt, der Bieter aber nur den zweithöchsten Preis zahlen muss. Die Passivität der Bieter hat aber auch einen entscheidenden Nachteil: Wenn der aktuelle Preis höher ist, als alle Gebote, „verhungert“ das System.

Escalator algorithm

Ein Algorithmus der zweiten Auktionskategorie, löst dieses Problem auf elegante Weise. Der „escalator algorithm“ (frei übersetzt Rolltreppenalgorithmus) beschreibt eine Form der „second-price sealed-bid“ Auktion, die so genannte „escalating-bid auction“. Jeder Thread gibt ein Startgebot ab und „legt“ es auf eine von mehreren Rolltreppen. Jede dieser Rolltreppen fährt mit einer anderen Geschwindigkeit. Je höher das Startgebot, desto weiter oben wird es auf die Treppe gelegt. Alle abgegebenen Gebote steigen nun mit der Geschwindigkeit ihrer Rolltreppe stetig an. Der Thread mit dem höchsten Gebot bekommt eine CPU-Zeitscheibe zum Preis des zweithöchsten Gebotes. Die Geschwindigkeiten der Rolltreppen können zusammen mit dem Startgebot als eine Art Priorität angesehen werden.

2.3.2 GRIDs

Eine besondere Form des verteilten Rechnens ist das Grid-Computing [gria]. Die Bezeichnung Grid ist vom engl. power grid (Stromnetz) abgeleitet. Die damit verbundene Vorstellung von „Rechenleistung aus der Steckdose“ [gric, grib] beschreibt die grundlegende Eigenschaft, dass es für den Nutzer eines Grids unerheblich ist, wo die genutzte Ressource sich befindet.

Lediglich die Tatsache, dass die gewünschte Anwendung irgendwo im Grid ausgeführt wird, ist von Interesse. Der Nutzer sendet einen Auftrag, eine Anwendung, an das Grid und wartet auf das entsprechende Ergebnis.
Grids bestehen in der Regel aus einem Multicomputersystem [Tan02, Kapitel 8.2 - Multicomputer] (oft verteilte Cluster, normaler PCs) und Komponenten zur Ressourcenverwaltung. Die Bedeutung solcher Systeme hat in den letzten Jahren stark zu genommen. Angefangen vom Projekten wie Seti [set] oder Einstein@Home [ein], bis hin zur Krebsforschung oder Wettervorhersage, gibt es unzählige denkbare Anwendungen. Nach Ansicht verschiedener Autoren (z. B. von [BSGA01, Buy02, BAG01, EHY02]) eignen sich ökonomische Ansätze sehr gut, b. z. w. sind sogar prädisteniert zur Ressourcenverwaltung in Grids.

2.3.3 GRACE

GRACE (**GR**id **A**rchitecture for **C**omputational **E**conomy) [Buy02] ist eine „Ökonomiebasierte Ressourcenmanagementarchitektur“ für Grids. Abbildung 2.2 zeigt den Aufbau der von

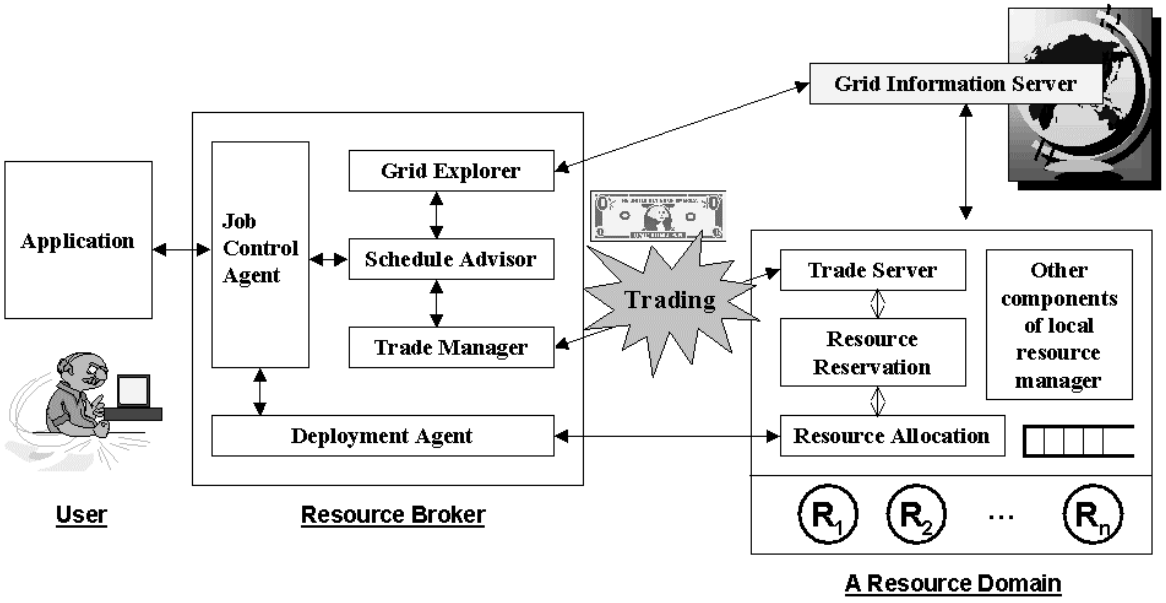

Abbildung 2.2: GRACE - Ressourcemanagement und Scheduling Systemarchitektur

den Autoren vorgeschlagenen Systemarchitektur. GRACE besteht aus folgenden Komponenten dieser Architektur:

- dem Trademanager,
- dem Tradeserver und
- den benötigten APIs und Verhandlungsprotokollen.

Ein Nutzer kann einen Auftrag, eine Anwendung, an den Ressource Broker senden. Der Ressource Broker ist dafür verantwortlich, die benötigten Ressourcen zu organisieren. Dazu

tritt er mittels des Trademanagers in Verhandlung mit den Tradeservern der Ressourceneigentümer. Die Tradeserver setzen dabei lokale Preis- und Vergabepolicies durch und arbeiten mit dem jeweiligen Accountingsystem ihres Ressourceneigentümers zusammen.

3 Entwurf

Das Kapitel Entwurf definiert zunächst die gestellten Ziele und zeigt, weshalb Rechenzeitabstraktion und kontengesteuertes Scheduling sinnvoll ist. Danach wird das entwickelte Schedulingmodell vorgestellt und mit den Zielen abgeglichen. Am Ende folgen einige weitergehende Betrachtungen und die Beschreibung des Prototypen.

3.1 Ziele

Das Ziel dieser Arbeit ist es, von der Größe Rechenzeit zu abstrahieren und so bekannte Probleme wie

- Rechenzeit-Donation/-Lending,
- Synchronisation,
- Denial-of-Service-Angriffe,
- Fairness
- Unterstützung von Echtzeit- und Nichtechtzeitprozessen,
- Rechenzeitbalancierung,

die bisher einzeln betrachtet und zum Teil gelöst wurden, zu vereinheitlichen. Zum besseren Verständnis werden im Kapitel 3.2.1 einige Beispielszenarien aufgeführt und deren bisherige Lösungen, möglichen neuen gegenübergestellt.

Dieses übergeordnete Ziel wird nun in Teilziele untergliedert. Das neue System soll folgende Möglichkeiten bieten:

- Abstraktion von Rechenzeit durch Nutzung des Dealmanansatzes,
- Feingranulare Verteilung der Rechenzeit auf alle bereiten Threads,
- Feingranulare Übertragung von Rechenzeit (Donation) zwischen verschiedenen Prozessen, unabhängig von der Art der Kommunikation (IPC/Shared Memory),
- Unterstützung von periodischen Echtzeit- und nichtperiodischen Nichtechtzeit-Prozessen,
- Frei implementierbarer Echtzeit-Schedulingalgorithmus auf Basis von statischen- oder dynamischen Prioritäten,
- Fairness (bereiten Threads mit Geld wird ein Minimum an Rechenzeit garantiert),
- Fair-Share-Scheduling (siehe Kapitel 2.2.3)

- Sicherheit (das Verfahren soll über die definierten Möglichkeiten hinaus nicht manipulierbar sein),
- Scheduler als Userlandtask

3.2 Motivation

Hinter der Idee der Rechenzeitabstraktion verbirgt sich der Wunsch, eine Menge von scheinbar ganz unterschiedlichen Problemen durch die Einführung eines neuen Schedulingansatzes zu vereinheitlichen und mit einem Schlag zu lösen, b. z. w. zu vereinfachen.

3.2.1 Szenarien

An dieser Stelle werden einige Szenarien beschreiben, die darlegen, warum kontengesteuertes Scheduling sinnvoll ist. Folgende, in Kapitel 3.4 zu beschreibende Symbole, finden dabei bereits Verwendung:

Symbol	Bedeutung
t_i	Thread Nummer i von n
Γ_{CPU}	Gesamtgeld im System für die Ressource CPU
Γ_{CPU_i}	Geld des Threads t_i für die Ressource CPU - repräsentiert dessen Rechenzeit

Tabelle 3.1: Symbole und deren Bedeutung

Rechenzeit-Donation

Nimmt ein Client die Dienstleistung eines Servers in Anspruch, so kann der Server, zum Beispiel zur Vermeidung von Denial-of-Service-Angriffen [Leb05, Kapitel 3.2.3], Rechenzeit-Donation vom Client verlangen. Auch aus Sicht des Clients kann ein solches Vorgehen sinnvoll sein, da der Server dessen Anfrage so schneller beantworten kann.

⇝**Bisherige Lösung:** Anfragen an Server erfolgen zumeist via IPC. Dabei findet automatisch Time-Slice-Donation statt, indem der Kern zum Server umschaltet, der Scheduling-Kontext des Clients aber aktiv bleibt. Damit läuft der Server zunächst auf der Zeitscheibe des Clients. Typischerweise blockiert der Client bei dem Aufruf und wartet auf eine Antwort des Servers. Das bedeutet, dass die Donation einmalig ist und der Server ggf. auf seiner eigenen Zeitscheibe weiterarbeiten muss. Dieses Problem lässt sich mit etwas Aufwand lösen, indem der Client vor dem Serveraufruf einen neuen Thread erzeugt, der bis zur Antwort fortlaufend mittels `l4_switch_to` zum Server umschaltet und so ebenfalls seine Zeitscheibe an diesen doniert. Ein anderer Ansatz ist der in [Ste04] beschriebenen „Donating Call“.
Im Falle von Shared-Memory-Kommunikation kann Time-Slice-Donation je nach Kommunikationsprotokoll ebenfalls mit Hilfe von `l4_switch_to` b. z. w. „Donating Call“ realisiert werden.

⇝**Lösung mittels Dealman-Donation:** Unabhängig von der Art der Kommunikation (IPC oder Shared Memory) doniert der Client vor dem Aufruf eine bestimmte Menge seines CPU-Budgets an den Server. Die Größe dieser Menge bestimmt, wieviel Rechenzeit der Client an

den Server abgeben will. Nach Abarbeitung der Anfrage, doniert der Server den entsprechenden Betrag an den Client zurück. Vertraut der Client dem Server nicht, kann statt Donation auch Lending eingesetzt werden.

Erzeuger-Verbraucher-Problem

[Tan02, Kapitel 2.3.4] beschreibt das Erzeuger-Verbraucher-Problem wie folgt. Die beiden Threads t_1 und t_2 kommunizieren über einen gemeinsamen n-elementigen Puffer. t_1 schreibt Daten in den Puffer und t_2 liesst daraus. Dabei gibt es zwei kritische Situationen:

1. Der Puffer ist leer.
2. Der Puffer ist voll.

t_1 und t_2 müssen derart synchronisiert werden, dass t_1 im Fall 2. nicht in den Puffer schreibt und t_2 im Fall 1. nicht aus dem Puffer liesst.

$\rightsquigarrow$**Klassische Lösung:** Klassisch wird dieses Problem mittels zweier Semaphoren [Tan02, Kapitel 2.3.4] gelöst.

$\rightsquigarrow$**Lösung mittels Dealman-Donation:** Völlig ohne zusätzliche Synchronisationsmechanismen (wie Semaphoren) kommt der folgende Lösungsansatz aus.
Zu Beginn ist der n-elementige Puffer leer. t_1 besitzt eine bestimmtes CPU-Budget Γ_{CPU_1}, während das entsprechende Konto von t_2 leer ist. Damit kann nur t_1 rechnen.
Nach dem Füllen eines Elements des Puffers doniert t_1, $\frac{\Gamma_{CPU_1}+\Gamma_{CPU_2}}{n}$ an t_2. Nach dem Entnehmen eines Elements des Puffers doniert t_2, $\frac{\Gamma_{CPU_1}+\Gamma_{CPU_2}}{n}$ an t_1.

Beispiel: t_1 und t_2 kommunizieren über einen 8-elementigen Ringpuffer. Zu Beginn ist das CPU-Budget von t_1 $\Gamma_{CPU_1} = 400\gamma$ und das von t_2 $\Gamma_{CPU_2} = 0\gamma$. Möglicher Ablauf:

- t_1 schreibt Daten in das erste Element des Puffers und doniert $\frac{400\gamma}{8} = 50\gamma$ an t_2
  (Anzahl der freien Pufferelemente: 7; $\Gamma_{CPU_1} = 350\gamma$; $\Gamma_{CPU_2} = 50\gamma$)
- t_1 schreibt Daten in das zweite Element des Puffers und doniert 50γ an t_2
  (Anzahl der freien Pufferelemente: 6; $\Gamma_{CPU_1} = 300\gamma$; $\Gamma_{CPU_2} = 100\gamma$)
- t_1 schreibt Daten in das dritte Element des Puffers und doniert 50γ an t_2
  (Anzahl der freien Pufferelemente: 5; $\Gamma_{CPU_1} = 250\gamma$; $\Gamma_{CPU_2} = 150\gamma$)
- t_2 entnimmt Daten aus dem dritten Element des Puffers und doniert 50γ an t_1
  (Anzahl der freien Pufferelemente: 6; $\Gamma_{CPU_1} = 300\gamma$; $\Gamma_{CPU_2} = 100\gamma$)
- u. s. w.

Im Beispiel ist zu erkennen, dass jeweils derjenige Thread mehr Rechenzeit bekommt, dessen Anteil am Puffer größer ist. So kann t_1 im ersten Punkt des Beispiels noch 7 Pufferelemente beschreiben und bekommt Rechenzeit im Wert von 350γ, während t_2 nur ein Pufferelement lesen kann und dafür lediglich Rechenzeit im Wert von 50γ erhält. Damit können auch große

Laufzeitunterschiede zwischen Erzeuger und Verbraucher ausgeglichen werden. Im Allgemeinen führt das Vorgehen dazu, dass der Puffer nach der Anlaufphase nie komplett gefüllt oder leer ist und so keiner der Threads warten muss.
Dieser Fakt grenzt diese Lösung scharf gegen den Ansatz mit Semaphoren ab. Voraussetzung dafür ist, dass der Puffer und die Geldgranularität hinreichend groß sind.
Des weiteren werden auch lokale Laufzeitschwankungen durch die, mit der Donation verbundenen, Erhöhung b. z. w. Absenkung der Rechenzeit ausgeglichen. Dadurch kann möglicherweise die Größe des Buffers reduziert werden.
Ein weiterer Vorteil gegenüber der Lösung mittels Semaphoren besteht darin, dass die Rechenzeit, die das Erzeuger-Verbraucher-Threadsystem zugeteilt bekommt immer konstant ist. Das kann in bestimmten Fällen zu einem schnelleren Fortschreiten des Systems führen. Das folgende Beispiel beschreibt einen solchen Fall.

Beispiel: Das Erzeuger-Verbraucher-System besteht aus den Threads t_1 und t_2. Beide kommunizieren über einen mehrelementigen Ringpuffer. Des weiteren existiert ein unbeteiligter Thread t_3. Der Erzeuger t_1 arbeite konstant schneller, als der Verbraucher t_2.
Im Falle der Lösung mittels Semaphoren und klassischem Round-Robin-Scheduling, passiert folgendes: Zu Beginn sind alle drei Threads rechenbereit und bekommen jeweils $\frac{1}{3}$ der verfügbaren Rechenzeit. t_1 und t_2, die das Erzeuger-Verbraucher-Threadsystem bilden, erhalten zusammen also $66,\bar{6}\%$. Nach einer gewissen Zeit ist der Puffer vollständig gefüllt und verhält sich von nun an wie ein einelementiger. Das bedeutet, dass von nun an t_1 oder t_2 und t_3 aktiv sind. t_1 oder t_2 und t_3 erhalten damit jeweils 50% der verfügbaren Rechenzeit. Das Erzeuger-Verbraucher-System verliert auf diese Weise $16,\bar{6}\%$ an t_3.
Nicht so im Falle der Lösung mittels Donation, bei dem Rechenzeit entsprechende des Geldes zugewiesen wird. Da sich das Vermögen des Threads t_3 nicht ändert, bekommt er immer exakt seinen Anteil ($\frac{1}{3}$) zugewiesen. Die verbleibenden $\frac{2}{3}$ erhalten anteilig t_2 und t_3. Das Erzeuger-Verbraucher-System kann deshalb in diesem Fall schneller fortschreiten.

Wechselseitiger Ausschluss, Synchronisation, Semaphore

In Mehrprozesssystemen werden Mechanismen zum Absichern von kritischen Abschnitten sowie zur Synchronisation benötigt. Ein klassisches Beispiel für einen derartigen Mechanismus ist der Semaphore (Kapitel 2.1.5).

⇝**Bisheriger Lösungsansatz:** Der Semaphoremechanismus wird durch einen Semaphoreserver implementiert, der den Zähler sowie die Warteschlange verwaltet und die beiden Funktionen `P` und `V` als Dienste anbietet.

⇝**Lösung mittels Dealman-Donation:** Der beschriebene Mechanismus kann mittels Dealman-Donation, ähnlich dem vorherigen Szenario, erweitert werden. Beim Aufruf der `P`-Operation durch den Thread t_1 wird auch dessen Dealman-Capability (siehe Kapitel 2.1.4) an den Semaphoreserver übermittelt. Dieser nutzt die Capability und doniert dem Thread t_2, der sich derzeit im kritischen Abschnitt befindet, das gesamte CPU-Geld von t_1. Je mehr Threads die `P`-Operation aufrufen und auf t_2 warten müssen, desto mehr Rechenzeit bekommt t_2 und desto schneller kann t_2 den kritischen Abschnitt verlassen.

Beim Verlassen des kritischen Abschnitts durch den Aufruf der V-Operation werden alle erhaltenen Kontingente an den Thread doniert, den der Semaphoreserver als nächstes für den kritischen Abschnitt auswählt.

Denial-of-Service-Angriff (Mißbrauch von Time-Slice-Donation)

Bisher ist folgender Angriff möglich ([Leb05, Seite 13]): Ein Angreifer startet alle möglichen 128 Threads. Während der erste Thread das eigentliche Programm abarbeitet, donieren die 127 anderen diesem ihre Rechenzeit. Das Ergebnis ist, dass der Angreifer effektiv 128 mal in der Bereitliste des Schedulers steht und 128 mal soviel Rechenzeit bekommt, wie alle anderen Prozesse, die den beschriebenen Angriff nicht durchführen (Vgl.: Fair-Share-Scheduling Kapitel 2.2.3).

⇝**Bisheriger Lösungsansatz:** In L4V2 ist diese Problem bisher nicht gelöst. Für L4.sec ist ein Mechanismus (Reservation Framework) geplant [Völ, Kapitel 2.4], durch den derartige Angriffe unmöglich werden. Dabei besitzt ein Thread eine Rechenzeitreservierung von der er einem neuen Thread einen Teil abgeben muss, damit dieser arbeiten kann. So ist es nicht mehr möglich „Rechenzeit zu schaffen“.

⇝**Lösung mittels Dealman-Donation und kontengesteuertem Scheduling:** Ähnlich der für L4.sec angedachten Idee kann das Problem auch mittels des in dieser Arbeit vorgestellten Systems gelöst werden. Ein neu erzeugter Thread hat zunächst kein Geld auf seinem CPU-Konto und wird damit vom Scheduler nicht aktiviert. Damit der Thread lauffähig wird, muss er eine Donation erhalten. Typischerweise bekommt er diese von seinem Erzeuger. Das bedeutet aber, dass die Gesamtrechenzeit eines Angreifers durch die Erzeugung neuer Threads nicht zunimmt, da er diesen Geld von seinem eigenen CPU-Konto abgeben muss. Dieses Vorgehen entspricht dem in Kapitel 2.2.3 beschriebenen Fair-Share-Scheduling.

Ergebnis der Szenarien

Die beschriebenen Beispielszenarien zeigen, dass sich verschiedene Probleme, denen bisher auf ganz unterschiedliche Weise begegnet wurde, durch Anwendung eines kontenbasierten Schedulingansatzes einheitlich lösen lassen.

3.2.2 Zusammenfassung - allgemeine Ressourcenabstraktion

Nahezu jede beschränkte Ressource lässt sich kontengesteuert verwalten (in [Leb05] wurde das konkret für die Ressource Arbeitsspeicher gezeigt). Neben den sicherheitsrelevanten Vorteilen führt ein solches Vorgehen zu einer starken Vereinfachung für Entwickler und Anwender, da diese anstatt vieler verschieden zu behandelnder Ressourcen eine einheitliche Schnittstelle, die Ressource Geld (Gums) vorfinden, deren Primitive (Donation/Lending) für alle Ressourcen gleich sind.

3.3 Entscheidungen und deren Folgen

3.3.1 Nutzung des Dealman-Konzeptes für die Ressource CPU

Der Dealman-Ansatz wurde ausgewählt, da er genau für die beschriebenen Probleme entworfen wurde und dessen Schwerpunkt auf sicherer Donation/Lending liegt.

„Striktes Lending“

Zur Verwaltung der Ressource CPU ist eine kleiner Erweiterung des Lending-Modells notwendig. Wie in Kapitel 2.1.4 beschrieben werden geborgte Kontingente nach Ablauf der Lendingfrist vom Lending-Observer invalidiert und können danach zurückgefordert werden. Im Falle von CPU-Konten ergibt sich folgendes Problem: Wenn ein Thread alle Gums seines CPU-Kontos einem anderen Thread borgt, so kann er nach Ablauf der Lendingfrist diese nicht zurückfordern, da er aufgrund fehlender Gums vom Scheduler nicht aktiviert wird. Daher wird das Lending-Modell um „Striktes Lending“ erweitert. Der einzige Unterschied zum „normalen“ Lending besteht darin, dass der Lending-Observer nach Ablauf der Lendingfrist das betreffende Kontingent nicht invalidiert, sondern direkt an den ursprünglichen Eigentümer zurücküberweist.

3.3.2 Lottery-Scheduling

Lottery-Scheduling (Kapitel 2.2.4) hat das Potential einige der gesetzten Ziele zu erreichen. Das liegt daran, dass dessen Lose ähnlich wie Gums (Geld) als Währung angesehen werden können. Aufgrund der Zufallskomponente ergeben sich aber zwei Probleme:

- die bereits im Kapitel 2.2.4 festgestellte Möglichkeit kurzfristiger Unfairness und
- die unmögliche garantierte Einhaltung harter Echtzeitgarantien durch den Indeterminismus dieses Verfahrens.

Es wurde daher entschieden, ein anderes Verfahren zu entwickeln, dass ähnliche Eigenschaften besitzt, aber deterministisch arbeitet und nicht auf dem Zufall basiert.

3.3.3 Auktionen

Der betrachtete auktionsbasiere Algorithmus (escalator algorithm in Kapitel 2.3.1) beschreibt lediglich den Prozess der Auswahl des zu aktivierenden Threads und geht von fixen Zeitscheibenlängen aus. Obwohl die Autoren von [MD88b] den Overhead des Algorithmus als „bescheiden“ bezeichnen erscheint er in Anbetracht dessen als hoch. Außerdem widerspricht diese Reduktion dem Entwurfsziel der feingranularen Rechenzeitverteilung (Kapitel 3.1).

3.3.4 Userland-Scheduling

Das zu entwerfende Schedulingmodell greift auf ein unabhängiges Kontensystem - den Dealman zurück. Der Dealman wird durch einen Userlandserver repräsentiert und zwingt jeden Ressourcenverwalter, der dessen Dienste in Anspruch nehmen will, ebenfalls im Userland zu existieren. Im Falle der in [Leb05] besprochenen Ressourcen Hauptspeicher, Hintergrundspeicher, Taskanzahl ist das unproblematisch, da die betreffenden Server bereits Userlandserver sind. Beim Scheduling sieht das anders aus. Die Verwaltung der Ressource CPU erfolgt bisher

im Kern. Jedoch gibt es bereits Ansätze, Teile des Scheduling z. B. durch User-Level-Threads [Die05] in das Userland zu holen.
Userland-Scheduling hat den großen Vorteil, dass damit die letzte Policy aus dem Kern verschwindet. Dort wird lediglich ein grundsätzlicher Mechanismus benötigt.
Aber es gibt auch eine andere Seite, den Userland-Scheduling ist mit zusätzlichen Kosten verbunden. So führt zum Beispiel jedes Umschalten zu einem anderen Thread durch den Userland-Scheduler zu mindestens einem weiteren Threadwechsel. Auf Architekturen wie x86, auf denen Threadwechsel teuer sind, ist das ein nicht zu vernachlässigender Fakt. Ausserdem werden Informationen über Threadzustände benötigt, die nur im Kern verfügbar sind. Insbesondere über Ereignisse, die bei IPC auftreten können wie:

- Blockieren von Threads,
- Erwachen von Threads,
- Einketten von Threads in Senderwarteschlangen,

muss der Userlandscheduler informiert werden.
Im Kapitel 2 wurden zwei Konzepte vorgestellt, die zur Lösung dieses Problems herangezogen werden können:

- das Alienmodell (2.1.3) und
- die Scheduler-Activations (2.2.6).

Da das Alienmodell Bestandteil von L4V2 ist und bereits Erfahrungen über dessen Nutzung vorliegen fällt die Entscheidung auf diesen Ansatz.

Ein Userlandscheduler baut auf den Schedulingmechanismen des Betriebssystemkernes auf. Um Verwechselungen zu vermeiden, wird im Folgenden explizit erwähnt, wenn es um Ereignisse und Begriffe im Kontext des Kernes geht.

3.3.5 Definition von Echtzeitthreads

Wie bereits in Kapitel 2.2.7 erwähnt, behandelt diese Arbeit ausschließlich streng periodische Echtzeitthreads. Das bedeutet, dass jeder Echtzeitthread eine feste Periode und Ausführungszeit besitzt. Die Ausführungszeit beschreibt dabei die Zeit, die innerhalb einer Periode benötigt wird. Die Deadlines solcher Threads entsprechen jeweils dem Beginn ihrer nächsten Periode.

3.4 Schedulingmodell

Das folgende Kapitel widmet sich dem entwickelten Schedulingmodell. Es begint mit der mathematischen und verbalen Beschreibung der zu Grunde gelegten Infrastruktur. Eine existierende Kontenverwaltung nach dem Dealmankonzept (siehe Kapitel 2.1.4) mit der Möglichkeit von Donation wird vorausgesetzt. Daran schließt sich die Vorstellung der Algorithmen an. Zum Schluss folgt der Abgleich mit den gestellten Zielen aus Kapitel 3.1.

3.4.1 Das System

Es existieren $0 \leq i \leq n$ Threads $t_i \in T$ im System. Das Gesamtvermögen an CPU-Gums im System beträgt Γ_{CPU}.

Sicht eines Threads

Jeder Thread t_i legt für sich eine Reihe von Schedulingparametern fest. t_i besitzt eine bestimmte Menge an CPU-Gums (Γ_{CPU_i}) auf seinem Schedulingkonto. Diese Menge repräsentiert den jeweiligen Anteil an der Gesamtrechenzeit. Ein gewisser Betrag des Budgets kann auf das Prioritätenkonto des Threads überwiesen werden (Γ_{PRIO_i}) und erhöht ggf. dessen Priorität, bei gleichzeitiger Reduktion seiner Rechenzeit. Neben diesen beiden Konten kann ein weiteres, ein Sparkonto (Γ_{STORE_i}), existieren, das vom Scheduler nicht berücksichtigt wird. Ausserdem existiert für jeden Thread t_i die minimale Periode C_i. Deren Wert entscheidet, auf welche Weise die absolute Rechenzeit s_i bestimmt wird, die t_i erhält, und welcher Schedulingalgorithmus für diesen Thread Anwendung findet.

- $C_i \leq 0$: Der Thread t_i ist ein Nichtechtzeitthread.
  Die absolute Zeitscheibenlänge s_i wird derart berechnet, dass gilt:

  $$\forall i < n : \frac{\Gamma_{CPU_i}}{\Gamma_{CPU_{(i+1)}}} = \frac{s_i}{s_{(i+1)}} \tag{3.1}$$

  Das bedeutet, dass die Verteilung des Geldes (Gums) auf die Konten aller Nichtechtzeitthreads zu einer äquivalenten Verteilung der Rechenzeit führt.

- $C_i > 0$: Der Thread t_i ist ein Echtzeitthread.
  Die absolute Zeitscheibenlänge s_i wird derart berechnet, dass gilt:

  $$\frac{\Gamma_{CPU_i}}{\Gamma_{CPU}} = \frac{s_i}{C_i} \tag{3.2}$$

  Das bedeutet, dass der prozentuale Anteil an der Gesamtrechenzeit, durch einen Echtzeitthread mit der entsprechend anteiligen Menge der CPU-Gums des Systems, bezahlt werden muss.

Sicht des Schedulers

Der Scheduler berechnet aus verschiedenen Systemparametern und den von den Threads vorgegebenen Schedulingparametern, die jeweiligen absoluten Rechenzeiten s_i (Gesamtzeitscheiben) und Prioritätsstufen P_i. Um die maximale Antwortzeit des Systems zu begrenzen

wird die Zeitscheibe s_i jedes Threads t_i in $1 \leq j \leq m$ Realzeitscheiben s_{ij} aufgeteilt.

Systemparameter:

- s_{MIN} ... minimale Realzeitscheibenlänge
  Dieser Wert gibt an, wie lang eine Realzeitscheibe mindestens sein muss. Dadurch wird der Schedulingoverhead begrenzt.

- s_{MAX} ... maximale Realzeitscheibenlänge
  Dieser Wert gibt an, wie lang eine Realzeitscheibe höchstens sein darf. Die Gesamtzeitscheibe s_i wird ggf. in mehrere Realzeitscheiben s_{ij} geteilt (siehe Formel 3.12). Auf diese Weise wird die maximale Antwortzeit des Systems begrenzt.

- s_Q ... Länge eines Zeitquantum
  Das Zeitquantum ist das kleinstmögliche Zeitmaß. Aus der Verteilung der Budgets Γ_{CPU_i} wird dessen Preis E_{s_Q} errechnet und bestimmt, wieviele Zeitquanten jeder Nichtechtzeitthread bekommt (siehe Formeln 3.4 und 3.5).

- $E_P(P)$ ... Prioritätsstufen-Kostenfunktion (z. B.: $E_P(P) = 100\gamma * \lfloor 1,5^{(P-1)} \rfloor + 50\gamma$)
  Mit dieser Funktion werden die Kosten der einzelnen Prioritätsstufen berechnet. Da zudem auch die Umkehrung $E_P^{-1}(\Gamma_{PRIO_i})$ zur Berechnung der Prioritätsstufe aus einem gegebenen Prioritätsbudget benötigt wird, sollte diese Funktion invertierbar sein.

- Γ_{CPU} ... Anzahl der CPU-Gums im System

Schedulingparameter (vorgegeben von den Threads t_i, siehe vorheriger Abschnitt):

- Γ_{CPU_i} ... Anzahl der CPU-Gums (CPU-Budget) des Threads t_i
- Γ_{PRIO_i} ... Anzahl der Prioritäts-Gums (Prioritäts-Budget) des Threads t_i
- C_i ... Periode des Threads t_i

Aus den System- und Schedulingparametern sind folgende Werte zu berechnen:

- Für Nichtechtzeitthreads ($C_i = 0$):

$$E_{s_{MIN}} = \min(\Gamma_{CPU_i}) \text{... Kosten einer Minimalzeitscheibe} \tag{3.3}$$

  Der Thread mit dem geringsten CPU-Budget bestimmt den Preis für eine minimale Realzeitscheibe der Länge s_{MIN}. Die Forderung nach Fairness ist damit erfüllt (Kapitel 3.1). Es wird so gewährleistet, dass jeder bereite Thread, der Geld besitzt, auf jeden Fall einen Anteil der Rechenzeit erhält. Dieser Anteil entspricht s_{MIN}.

$$E_{s_Q} = \frac{E_{s_{MIN}} * s_Q}{s_{MIN}} \text{ ... Kosten eines Zeitquantum} \tag{3.4}$$

$$k_i = \left\lfloor \frac{\Gamma_{CPU_i}}{E_{s_Q}} \right\rfloor \text{ ... Anzahl der Zeitquanten des Threads } t_i \tag{3.5}$$

$$s_i = k_i * s_Q \text{ ... Gesamtlänge der Realzeitscheiben des Threads } t_i \tag{3.6}$$

$$P_i = E_P^{-1}(\Gamma_{PRIO_i}) \text{ ... Prioritätsstufe des Threads } t_i \tag{3.7}$$

- Für Echtzeitthreads ($C_i < 0$):

$$U_i = \frac{\Gamma_{CPU_i}}{\Gamma_{CPU}} \text{ ... Anteil an der Gesamtrechenzeit des Threads } t_i \tag{3.8}$$

$$s_i = U_i * C_i \text{ ... Gesamtlänge der Realzeitscheiben des Threads } t_i \tag{3.9}$$

Im Gegensatz zu Nichtechtzeitthreads wird die Prioritätsstufe P_i von t_i im Echtzeitfall nicht in Abhängigkeit von den Prioritätspreisen, sondern in Abhängigkeit vom eingesetzten Echtzeitschedulingverfahren berechnet. Das kann im Falle von statischen Prioritäten einmalig zum Startzeitpunkt des Threads erfolgen, oder aber, im Falle von dynamischen Prioritäten bei jedem Bereitwerden eines Echtzeitthreads. Für die Prioritäten von Echtzeitthreads gilt:

$$P_i > E_P^{-1}(\Gamma_{CPU}) \tag{3.10}$$

Auf diese Weise ist sichergestellt, dass Echtzeitthreads immer höhere Prioritäten besitzen als Nichtechtzeitthreads.

- Allgemein:

$$m_i = \left\lceil \frac{s_i}{s_{MAX}} \right\rceil \text{ ... Anzahl der Realzeitscheiben des Threads } t_i \tag{3.11}$$

$$s_{ij} = \begin{cases} s_{MAX} & \text{für } j < m_i \\ s_i \pmod{s_{MAX}} & \text{für } j = m_i \end{cases} \text{ ... Länge der j-ten Realzeitscheibe von } t_i \tag{3.12}$$

3.4.2 Arbeitsweise des Schedulers

Die Bereitliste

Die Bereitliste besteht aus zwei Bestandteilen:

- der Prioritätenliste und
- den Threadlisten.

Die Prioritätenliste ist eine sortierte Liste. Jeder Prioritätsknoten verweist auf eine Threadliste, in der alle Threads dieser Priorität, in der Reihenfolge ihres Bereitwerdens, aufgelistet sind. Threadlisten Listen von Threadknoten.
Jeder Threadknoten enthält folgende Werte:

- die Größe seines verbleibenden CPU-Budgets $\Gamma_{CPU_{i_{local}}}$ des aktuellen Durchlaufes und
- ggf. den Verweis auf einen „Rücksprungthreadknoten“.

Donation

Die grundlegenden Protokolle für Donation und Lending sind in [Leb05, Kapitel 4.1.4] beschrieben. Beim Scheduling müssen einige zusätzliche Dinge beachtet werden:

- Donations werden erst zu Beginn des nächsten Schedulingdurchlaufs berücksichtigt[1]. Das passiert, indem der Scheduler am Anfang jedes Durchlaufs die Größe der CPU-Budgets $\Gamma_{CPU_{i_{local}}}$ aller Threadknoten mit den der zugehörigen (Dealman-)Konten Γ_{CPU_i} abgleicht[2].
- Echtzeitthreads können weder Donations empfangen, noch anweisen, da sonst eine erneute Echtzeitadmission notwendig wäre.
- Ausserdem muss nach jeder Donation überprüft werden, ob sich durch die Überweisung der Preis eines Zeitquantums E_{s_Q} (siehe Formel 3.3) verändert hat. Ist das der Fall, ist dieser Wert zu aktualisieren.

Local Donation

Da „normale“ Donations erst zu Beginn eines neuen Schedulingdurchlaufes aktiv werden, wird ein weiterer Mechanismus benötigt, der es erlaubt innerhalb einer Periode die verbleibende Rechenzeit an einen Anderen zu donieren. Dieser Mechanismus wird „Local Donation“ genannt. Im Unterschied zur „normalen“ Donation, die vom Dealman durchgeführt wird, ist für local Donation der Scheduler verantwortlich, er überweist dabei Geld zwischen $\Gamma_{CPU_{i_{local}}}$ der beteiligten Threads.

Start eines neuen Threads

Ein neu erzeugter Thread ist zunächst nicht lauffähig, da sich weder auf seinem CPU- noch dem Prioritätenkonto Geld (Gums) befindet. Daher muss der Erzeuger die entsprechende Menge an Gums an den neuen Thread donieren.
Der Scheduler berechnet daraus die Priorität und ordnet den Thread in die Threadliste der entsprechenden Prioritätsstufe ein.

Start eines neuen Echtzeitthreads

Beim Start eines Echtzeitthreads findet zusätzlich die sogenannte „Admission“ (Zulassungsprüfung) statt. Je nach angewendetem Echtzeitschedulingverfahren wird zunächst überprüft, ob der Thread gegenwärtig eingeplant werden kann. Ist di e Admission erfolgreich wird die notwendige Prioritätsstufe berechnet. Diese ist gemäß Formel 3.10 in jedem Fall höher als alle Stufen, die Nichtechtzeitthreads kaufen können.

Einfacher Schedulingablauf

Zu Beginn eines jeden Durchlaufes werden alle $\Gamma_{CPU_{i_{local}}}$ mit den Γ_{CPU_i} des Dealman abgeglichen (aufgeladen) und bei Änderungen von Prioritätskontoständen Γ_{PRIO_i}, die betreffenden Threads in die entsprechenden Threadlisten umgekettet. Der Scheduler aktiviert nun die

[1] Die Gründe dafür werden in der Sicherheitsbetrachtung im Kapitel 3.4.4 erläutert.
[2] Durch verschiedene Optimierungen (Kapitel 5.2) kann der so entstehende Overhead minimiert werden.

höchstpriore Threadliste. Danach erfolgt die Abarbeitung der Liste gemäß der Realzeitscheiben s_{ij} der jeweiligen Threads nach dem Round-Robin-Verfahren.

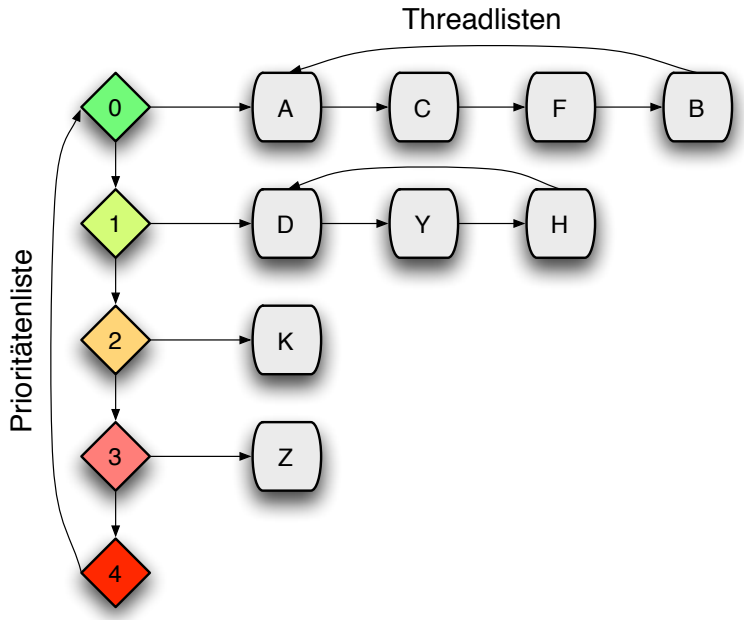

Abbildung 3.1: Bereitliste mit einfachem Schedulingablauf

Vor Aktivierung eines Threads wird dessen $\Gamma_{CPU_{i_{local}}}$ um den Wert der zugeteilten Realzeitscheibe s_{ij} gekürzt. Haben alle Threads ihre Rechenzeit aufgebraucht (alle $\Gamma_{CPU_{i_{local}}}$ dieser Prioritätsstufe sind 0), wird die nächstpriore Threadliste aktiviert. Nachdem die Threadlisten aller Prioritätsstufen durchlaufen wurden, gleicht der Scheduler erneut alle $\Gamma_{CPU_{i_{local}}}$ mit den Γ_{CPU_i} des Dealman ab, passt ggf. die Prioritäten an und beginnt mit dem nächsten Durchlauf.

Bereitwerden eines höherprioren Threads

Der soeben beschriebene Ablauf kann unterbrochen werden wenn ein Thread bereit wird, der eine höhere Priorität besitzt als der gerade laufende. In diesem Fall wird der niederpriore Thread unterbrochen, er bekommt ggf. eine Rückzahlung für die nicht verbrauchte Rechenzeit auf sein $\Gamma_{CPU_{i_{local}}}$-Konto, und sein Threadknoten wird als „Rücksprungthreadknoten“ im Threadknoten des Höherprioren gespeichert. Dieser höherpriore Threadknoten wird in die entsprechende Threadliste eingeordnet und der Thread danach aktiviert. Nach Ablauf dessen Rechenzeit s_i wird zum vorherigen, niederprioren Thread zurückgesprungen. Da in diesem Szenario kein anderer Thread auf der höheren Prioritätsstufe rechnen kann (alle Threads dieser Stufe haben ihre Zeitscheiben in diesem Durchlauf bereits aufgebraucht), kann der neue Thread seine Gesamtzeitscheibe ohne Unterbrechung aufbrauchen, solange in dieser Zeit kein weiterer Höherpriorer bereit wird.

Bereitwerden eines Echtzeitthreads

Da Echtzeitthreads prinzipiell eine höhere Priorität haben als alle Nichtechtzeitthreads, wird ein Bereitwerdender sofort aktiviert, sofern kein höherpriorer Echtzeitthread läuft. Bei Anwendung eines Schedulingverfahrens mit dynamischen Prioritäten (z. B. EDF), müssen zuvor ggf. die Prioritäten aller Echtzeitthreads angepasst werden.

Der Scheduler achtet darauf, dass kein Echtzeitthread innerhalb seiner Periode mehr Rechenzeit bekommt als ihm gemäß Formel 3.2 zusteht und sorgt im Gegenzug für dessen rechtzeitige periodische Aktivierung.

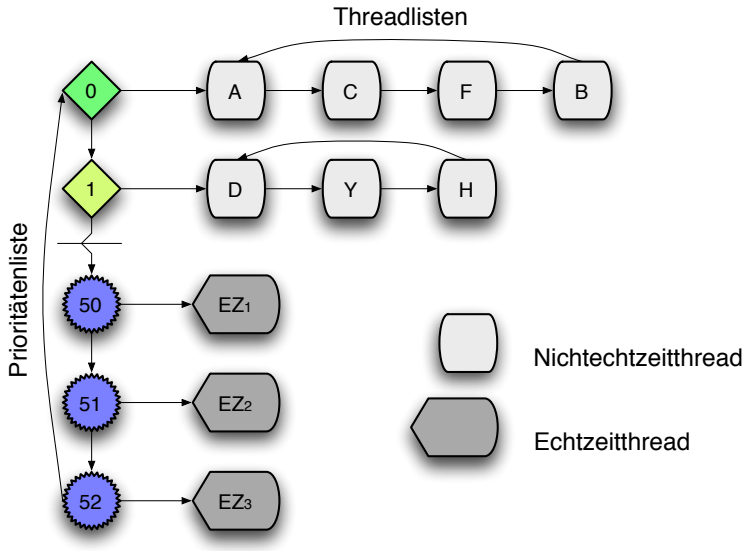

Abbildung 3.2: Bereitliste mit Echtzeitthreads

Bereitwerden eines niederprioren Threads

Ein bereitgewordener Thread mit einer niedrigeren Priorität als der aktuell Laufende, wird lediglich an das Ende der entsprechenden Threadliste angehangen und erst aktiviert, wenn diese Liste aktiv ist.

Blockieren eines Threads

Blockiert ein Thread, so wird dessen Threadknoten aus der Bereitliste ausgekettet und in eine Blockiertliste eingehangen. Das ist insbesondere dann der Fall, wenn zu Beginn eines neuen Schedulingdurchlaufes festgestellt wird, dass dessen $\Gamma_{CPU_i} = 0$ ist, da er im vorherigen Durchlauf eine entsprechend hohe Überweisung getätigt hat.

3.4.3 Bedeutung der Priorität

Die Höhe der Priorität bestimmt, wie lange es dauert, bis ein Thread nach dem Bereitwerden aktiviert wird. Wird ein Thread t_{23} bereit, wird er entsprechend seiner Priorität an das Ende

der passende Threadliste angehangen.
Nun gibt es genau drei Möglichkeiten: Der gerade laufende Thread t_5 hat ...

1. eine höhere Priorität als t_{23}.
2. eine niedrigere Priorität als t_{23}.
3. die gleiche Priorität wie t_{23}.

Im ersten Fall, muss t_{23} warten, bis seine Threadliste aktiviert wird. Im zweiten Fall wird t_{23} sofort aktiviert, und im dritten Fall nach dem Round-Robin-Verfahren entsprechend der Realzeitscheiben der Threads seiner Threadliste geschedult.

Die Prioritätsstufen-Kostenfunktion

Aus folgenden Gründen ist es nicht sinnvoll, die Preise der Prioritätsstufen für Nichtechtzeitthreads dynamisch mit dem Preis des Zeitquantums E_{s_Q} anzupassen:

1. Formel 3.1 ist unabhängig von E_{s_Q}.
2. Es gilt:
$$\Gamma_{CPU_i} + \Gamma_{PRIO_i} = const. \tag{3.13}$$
Damit gibt es bereits einen Zusammenhang zwischen der maximal bezahlbaren Priorität eines Threads und den Kosten für Rechenzeit E_{s_Q}.
3. Ändernde Prioritäten bringen diverse Probleme mit sich, w. z. B. die potentielle Notwendigkeit des Umketten von Threadknoten bei jeder Änderung E_{s_Q}.

Geringe Antwortzeiten lassen sich nur dann erreichen, wenn sich möglichst viele Threads in den Threadlisten der jeweilige Priorität befinden, b. z. w. die Gesamtzeitscheiben s_i von Threads in dünn besetzten Threadlisten möglichst klein sind. Durch den Zusammenhang zwischen CPU- und Prioritätenbudget (siehe Formel 3.13) können diese beiden Anforderungen mit einer entsprechende Preisgestaltung für die Prioritäten erreicht werden.
Die Prioritätsstufen-Kostenfunktion $E_P(P)$, b. z. w. deren Umkehrung, dient der Berechnung der Priorität eines Threads in Abhängigkeit von dessen Prioritätsbudget Γ_{PRIO_i}. Intuitiv erscheint es sinnvoll, den Preis mit steigender Priorität überprotortional wachsen zu lassen. Die Funktion $E_P(P) = 100\gamma * \lfloor 2^{(P-1)} \rfloor$ bewirkt beispielsweise, dass sich der Preis mit jeder Stufe verdoppelt.

$\boldsymbol{P}$	$\boldsymbol{E_P(P)}$ **in** $\boldsymbol{\gamma}$
0	0
1	100
2	200
3	400
4	800
5	1600

Tabelle 3.2: Prioritätsstufen-Kostenfunktion $E_P(P) = 100\gamma * \lfloor 2^{(P-1)} \rfloor$)

Im Ergebnis wird die Besetzung der Threadlisten mit steigender Priorität stark abnehmen. Im gleichem Maße sollte auch der Rechenzeitanteil wie gewünscht sinken, da mit steigender Priorität weniger Gums auf den CPU-Budgets verbleiben. Im konkreten Fall hängt das Ergebnis aber stark vom „Vermögen“ der einzelnen Threads ab.

3.4.4 Zielabgleich

Nun soll betrachtet werden, in wie weit das vorgestellte Modell die Zielanforderungen aus Kapitel 3.1 erfüllt.

Rechenzeitabstraktion und Fair-Share-Scheduling

Rechenzeitabstraktion ist das oberste Ziel dieser Arbeit. Sie wird durch die Zuordnung eines Preises und die Nutzung eines Kontensystems erreicht. Der Anteil an der Gesamtrechenzeit wird ausschließlich durch das „Vermögen“ jedes Threads festgelegt. Das benötigte Geld kann nur von anderen Threads doniert werden. In diesem und auch dem nächsten Punkt ähnelt das Modell dem Lottery-Scheduling (Kapitel 2.2.4).

Fairness

Fairness ist die Forderung, dass jeder rechenbereite Thread, der ein Minimum an Gums ($\Gamma_{CPU_i} < 0$) besitzt, auch ein Minimum an Rechenzeit (s_{min}) zugeteilt bekommt. Durch die Berechnung des Preises einer Minimalzeitscheibe (Formel 3.3) wird diese Forderung erfüllt.

Feingranulare Verteilung der Rechenzeit auf alle bereiten Threads

Die Verteilung des Geldes wird durch den Scheduler eins-zu-eins auf die Verteilung der Rechenzeit umgesetzt. Erreicht wird das durch ein Reservierungszeit-ähnliches Modell. Die Menge des Geldes, die ein Thread besitzt bestimmt die Länge seiner Zeitscheibe.
Die Granularität wird durch zwei Faktoren beeinflußt:

- den Preis eines Zeitquantum E_{s_Q} und
- dessen Größe s_q.

Da E_{s_Q} von $E_{s_{min}}$ und s_{min} abhängt (Formel 3.4), gibt es eine untere Grenze der Granularität. Diese Grenze wird erreicht, wenn $E_{s_{min}} = 1$ und damit $E_{s_q} = \frac{s_q}{s_{min}}$ ist.
Aufgrund der möglichen Schwankung muss gesagt werden, dass dieses Ziel nur annähernd erreicht werden konnte.

Kommunikationsunabhängige Donation

Bisher erfolgte „Time-Slice-Donation“ durch das Umschalten zu einem anderen Thread bei gleichzeitiger Beibehaltung des Scheduling-Kontextes. Da das Scheduling im beschriebenen Modell durch die Geldverteilung des Systems wie gewünscht manipuliert werden kann, ist mittels Dealman-Donation und -Lending eine abstraktere und feingranularere Rechenzeitdonation möglich.

Unterstützung von Nichtechtzeit und Echtzeit

Das beschriebene Modell unterstützt Nichtechtzeit- und streng periodische Echtzeitthreads. Zudem ist es möglich jeden Echtzeitschedulingalgorithmus auf Basis von Prioritäten (statische und dynamische) intergieren.

Sicherheit

Sicherheit ist eines der Probleme, weswegen das Dealman-Konzept in [Leb05] eingeführt wurde. Durch Nutzung dieses Ansatzes ist ein „Missbrauch von Time-Slice-Donation“ (Kapitel 3.2 und [Leb05, Seite 13]) nicht mehr möglich, da ein neu gestarteter Thread zunächst keine Gums besitzt und daher keine Rechenzeit erhält.
Des weiteren muss betrachtet werden, ob die Beträge der Schedulingkonten und die vergebene Rechenzeit in jedem Fall äquivalent sind. Beim „Einfachen Schedulingablauf“ ist das per Definition (Formeln 3.5 und 3.6) gegeben.

Zu untersuchen ist folgender Fall:

- t_{11} startet t_{37}
- t_{11} doniert sein gesamtes Budget ($\Gamma_{CPU_i} + \Gamma_{PRIO_i}$) an t_{37}
- t_{37} bekommt eine höhere Priorität und wird damit sofort aktiviert
- t_{37} startet und doniert kurz vor Ablauf seiner Zeitscheibe das gesamte Geld zurück an t_{11}.
- gemäß „Rücksprungadressknoten“ wird nun wieder t_{11} aktiviert

Da die Gesamt-/Restrechenzeit jedes Threads vor dessen Aktivierung aus $\Gamma_{CPU_{i_{local}}}$ und E_{s_Q} berechnet wird, erhielte t_{11} Rechenzeit, die t_{37} bereits aufgebraucht hat. Da Donations aber grundsätzlich erst zu Beginn eines neuen Schedulingdurchlaufs aktiviert werden, tritt dieses Problem nicht auf.
Anders verhält es sich bei local Donations. Diese werden sofort wirksam, gelten aber nur bis zum Ende des aktuellen Durchlaufes, da zu Beginn des nächsten alle $\Gamma_{CPU_{i_{local}}} = \Gamma_{CPU_i}$ gesetzt werden. $\Gamma_{CPU_{i_{local}}}$ wird vor Aktivierung des Threads belastet, so kann maximal der verbleibende, noch nicht verbrauchte Teil, doniert werden.

Die letzte Frage, die beim Thema Sicherheit gestellt werden muss ist, ob es als Angriff betrachtet werden muss, wenn ein Nichtechtzeitthread sich dem System gegenüber als Echtzeitthread ausgibt. Um diese Frage zu beantworten, ist zu betrachten, welche Konsequenzen ein solches Verhalten für den Thread und das System hat.
Um Echtzeitgarantien zu erhalten, muss eine minimale Periode C_i angegeben werden, deren Einhaltung der Scheduler durchsetzt. Ausserdem zahlt ein Echtzeitthread immer den maximalen möglichen Preis, exakt den Anteil an Γ_{CPU}, der seiner Auslastung U_i entspricht. Nichtechtzeitthreads dagegen erhalten im Normalfall deutlich mehr Rechenzeit für die gleiche Anzahl an Gums. Dieser Sachverhalt wird durch das folgende Beispiel veranschaulicht.

Beispiel: In einem System mit $\Gamma_{CPU} = 10000\gamma$ existieren vier Threads: t_1, t_2, t_3, t_4.
t_1 ist ein Echtzeitthread, mit einer Auslastung von $U_1 = 0,5 : \Gamma_{CPU_1} = 5000\gamma$ (Formel 3.8).

Die verbleibenden 50% Rechenzeit werden entsprechend der Verteilung der restlichen 5000γ auf die Nichtechtzeitthreads aufgeteilt.

Fall 1:	$\Gamma_{CPU_2} = 5000\gamma$	:	$U_2 = 0,5$
	$\Gamma_{CPU_3} = 0\gamma$		$U_3 = 0$
	$\Gamma_{CPU_4} = 0\gamma$		$U_4 = 0$
Fall 2:	$\Gamma_{CPU_2} = 4000\gamma$	:	$U_2 = 0,4$
	$\Gamma_{CPU_3} = 500\gamma$		$U_3 = 0,05$
	$\Gamma_{CPU_4} = 500\gamma$		$U_4 = 0,05$
Fall 3:	$\boldsymbol{\Gamma_{CPU_2} = 2000\gamma}$	:	$\boldsymbol{U_2 = 0,375}$
	$\boldsymbol{\Gamma_{CPU_3} = 500\gamma}$		$\boldsymbol{U_3 = 0,125}$
	$\Gamma_{CPU_4} = 2500\gamma$		$U_4 = 0$
	$\boldsymbol{t_4}$ **ist nicht rechenbereit!**		

Die beiden ersten Fälle zeigen den worst case (alle Threads, die Gums besitzen sind rechenbereit), bei dem die Rechenzeit am teuersten ist. Dabei gibt es preislich keinen Unterschied zwischen Echtzeit und Nichtechtzeit.
Fall 3 hingegen zeigt den average case (einige Threads, die Gums besitzen sind nicht rechenbereit). Dabei erhalten Nichtechtzeitthreads für das gleiche Geld mehr Rechenzeit als Echtzeitthreads. Als Konsequenz dieser Betrachtung muss gelten, dass Echtzeitthreads keine Donations durchführen dürfen.

Zusammenfassend kann gesagt werden, dass ein Thread mit Echtzeitgarantie weniger Rechenzeit bekommt als ohne. Damit wird die letzte Frage zum Thema Sicherheit mit Nein beantwortet.

3.4.5 Weitergehende Betrachtungen

Das Modell als auktionsbasiertes System

Das entwickelte Schedulingmodell kann als spezielle Form eines auktionsbasierten Systems verstanden werden:

- Threads können innerhalb eines Schedulingdurchlaufes durch Donations (ggf. auch zwischen den eigenen CPU-, Prio- und Sparkonten) ihre Kontostände verändern und damit die Gebote für den nächsten Durchlauf festlegen.

- Der Thread mit dem höchsten Prioritätsgebot und einem CPU-Gebot, das größer als Null ist, wird aktiviert und bekommt Rechenzeit entsprechend seines CPU-Gebotes.

- Durch Konsumieren der Rechenzeit sinkt das CPU-Gebot auf Null.

- Dieser Vorgang wird solange fortgesetzt, bis kein Thread mehr ein aktives CPU-Gebot besitzt. Danach beginnt ein neuer Durchlauf

Andere Schedulingpolicies

Der Scheduler kann durch Einflussnahme auf die Geldverteilung des Systems manipuliert werden. Das trifft sowohl auf die Rechenzeit, als auch die Prioritäten zu. Ein übergeordneter Scheduler kann durch gezieltes Umverteilen unterschiedliche Schedulingpolicies (z. B. prioritätenbasiertes Round-Robin) erzwingen. Durch den Schwerpunkt auf Donation, bietet sich insbesondere eine Umsetzung von CPU-Inheritance-Scheduling (Kapitel 2.2.5) an. Damit ist das Modell wahrscheinlich mächtiger als die meisten anderen Modelle.

Integration in ökonomibasierte GRIDsysteme

Durch die Nutzung eines unabhängigen Kontensystems - dem Dealman, besteht potentiell die Möglichkeit der Integration in ökonomiebasierte GRIDsysteme. Es ist beispielsweise denkbar, dass ein Tradeserver des GRACE-Systems, wie im Kapitel 2.3.3 beschrieben, mit dem Dealman (als lokales Accountingsystem) zusammenarbeitet.

3.5 Budget-based-Userland-Scheduler

Im folgenden wird dargelegt, wie das im Kapitel 3.4 beschriebene Schedulingmodell auf dem L4 Mikrokern (L4V2 - Fiasco) realisiert werden soll.

3.5.1 Aufbau

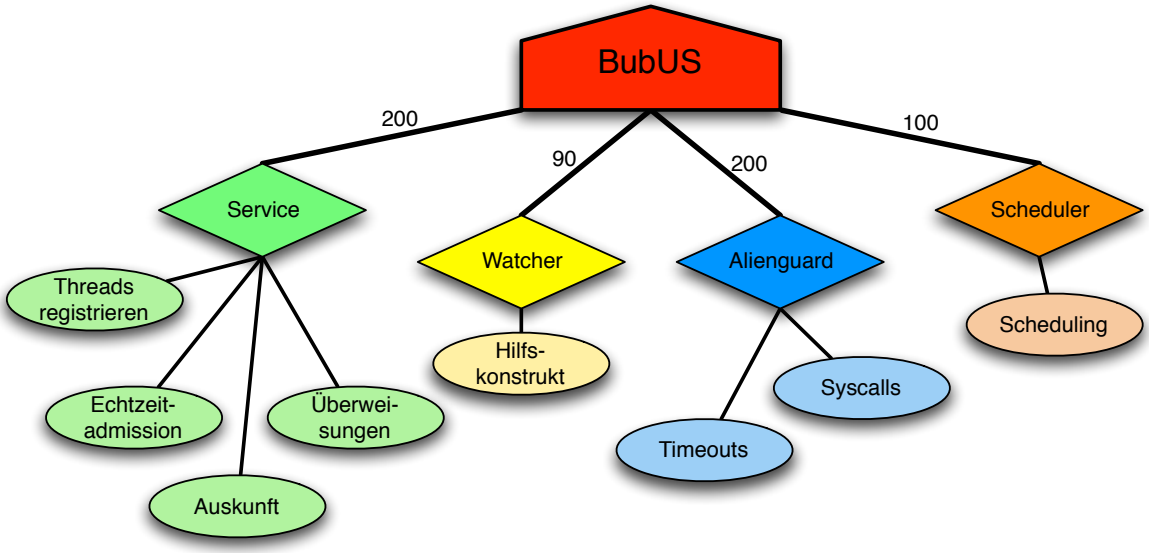

Abbildung 3.3: BubUS - grundsätzlicher Aufbau

Abbildung 3.3 zeigt schematisch den Aufbau des Budget-based-Userland-Schedulers, der aus vier Komponenten und Threads besteht. An jedem Pfad stehen die Priorität der jeweiligen Komponente. Scheduler, Alienguard und Service repräsentieren dabei die drei grundlegenden Aufgaben des Systems

- Scheduling
- Sammeln von Informationen über Threadzustände
- Anbieten verschiedener Dienste wie zum Beispiel Donation.

Der Watcher ist ein Hilfskonstrukt.

3.5.2 Scheduling

Allgemein verläuft das Scheduling genau wie in Kapitel 3.4.2 dargelegt. Jeder Durchlauf beginnt mit der Durchlaufinitialisierung. Dabei werden die lokalen Konten aufgeladen, Threads ohne Geld blockiert, aus den Budgets aller bereiten Threads der Preis einer Minimalzeitscheibe bestimmt u. s. w. Danach werden die Threads gemäß ihrer Prioritäten und Budgets aktiviert und wieder unterbrochen. Der Scheduler durchläuft dabei die Prioritätenliste und die zu jeder Stufe gehörenden Threadlisten, bis alle Budgets aufgebraucht sind - dann endet der Durchlauf.

Aktivieren und Unterbrechen

Alle nicht aktivierten Threads laufen mit der Kernpriorität 1. Der Scheduler aktiviert einen Thread, indem er seine Priorität auf einen Wert zwischen der Priorität des Watchers und des Schedulers anhebt. Nun schläft der Scheduler bis zum Ende der zugeteilten Rechenzeit und unterbricht durch sein Aufwachen nach Ablauf dieser Zeit den aktivierten Thread.

Vorzeitige CPU-Aufgabe

Gibt der aktive Thread vorzeitig die CPU ab, so wird der Watcher im Kontext des Kernschedulers zum höchstprioren Thread und wird aktiviert. Die einzige Aufgabe des Watchers besteht darin, den Scheduler aufzuwecken. Auf diese Weise kann der Scheduler bei vorzeitiger CPU-Aufgabe seine Arbeit sofort fortsetzen und es gibt keine Idelingphasen.

3.5.3 Informationen über Threadzustände

Der Scheduler benötigt Informationen über Änderungen von Threadzuständen. Blockiert ein Thread im Kontext des Kernscheduling weil er auf IPC wartet, so muss er auch im BubUS-kontext blockiert werden. Wird ein derart blockierter Thread wieder erweckt, weil z. B. ein Timeout abgelaufen ist, so sollte BubUS ihm zum entsprechenden Zeitpunkt auch die CPU wieder zuteilen.
Diese Informationen sammelt der Alienguard. Er fungiert als Alienpager gemäß des im Kapitel 2.1.3 beschriebenen Alienmodells. Alle zu schedulenden Threads sind Alienthreads, die diesen Pager nutzen. Damit wird der Alienguard immer dann aktiv, wenn ein Thread versucht einen Syscall durchzuführen oder ihn zu beenden.

Funktionsweise

Anhand der Exceptionnachricht erkennt der Alienguard, um welchen Syscall, mit welchen Parametern es sich handelt und kann so vorhersehen, welche schedulingrelevanten Aktionen der Kern daraufhin ausführen wird. Entsprechend dieser Vorhersage führt er nun selbst alle notwendigen Schritte bezüglich des Userlandscheduling, z. B. das Blockieren des betreffenden Threads, aus.

Datenkonsistenz

Da Datenstrukturen wie die Prioritätenliste und auch die Threadlisten ggf. durch den Alienguard modifiziert werden, besitzt dieser eine höhere Kernpriorität als der Scheduler. Auf diese Weise kann er in jedem Fall ohne Unterbrechung alle notwendigen Modifikationen durchführen. Danach wird immer der Scheduler aktiviert, auch wenn der aktive Thread noch Rechenzeit zur Verfügung hat. So kann der aktive Thread keinen weiteren Syscall auslösen und es besteht immer Datenkonsistenz zwischen Alienguard und Scheduler. Die ggf. zu wenig erhaltene Rechenzeit wird dem Thread in Form einer Rückzahlung erstattet.

3.5.4 Service

BubUS bietet seinen Threads verschiedene Dienste an. Diese werden von der Servicekomponente umgesetzt.
Folgende Dienste können die von BubUS geschedulten Threads in Anspruch nehmen:

- Anmelden neuer Threads (Einketten des Threads in die Blockiertliste),
- Ummelden als Echtzeitthreads (Echtzeitadmission),
- Blockieren bis zum Beginn des nächsten Durchlaufes (Abgeben des für diesen Durchlauf verbleibenden Geldes),
- lokale Überweisungen (local donation),
- Abfragen des Preises eines Zeitquantum.

Abweichend vom Modell ist es nicht möglich einen Thread von Beginn an als Echtzeitthread anzumelden. Neue Threads sind zunächst immer Nichtechtzeitthreads und können durch Ummelden zu Echtzeitthreads werden. Dieses Vorgehen hat den Vorteil, dass die Echtzeiteigenschaft beim Start eines Threads noch nicht feststehen muss, sondern auch später erworben werden kann.

Datenkonsistenz

Genau wie der Alienguard, modifiziert auch der Service ggf. schedulingrelevante Datenstrukturen. Deshalb besitzt auch diese Komponente eine erhöhte Kernpriorität. Da das in Anspruch nehmen von Servicefunktionen einen IPC-Syscall bedingt, wird dabei immer auch der Alienguard aktiv. Deshalb wird auch nach jeder Modifikation durch den Service immer der Scheduler aktiviert.

4 Implementierung

Das folgende Kapitel beschreibt die prototypische Implementierung des BubUS. Dabei werden zum Teil Abläufe, die im Entwurf bereits allgemein dargelegt wurden, nochmals konkret besprochen.

4.1 Voraussetzungen und Aufbau

Grundlage der Implementierung bildet der L4 Mikrokern in Version 2 (Fiasco) (siehe Kapitel 2.1.1 und 2.1.2) mit der Alienthread-ABI-Extension (siehe Kapitel 2.1.3).
Das im Kapitel 3.4 beschriebene Schedulingmodell setzt einen Dealman-Bankserver voraus. Da dieser derzeit noch nicht existiert werden die benötigten Funktionen rudimentär vom Scheduler implementiert.

Der Aufbau aus Kapitel 3.5.1 wird eins-zu-eins übernommen, da die Funktionen der einzelnen Komponenten voneinander losgelöst betrachtet werden können und das System auf diese Art übersichtlicher und leichter verständlich erscheint. Der BubUS-Server besteht deshalb aus vier Threads:

- dem Scheduler,
- dem Watcher,
- dem Alienguard und
- dem Service.

4.2 Die wichtigsten Datenstrukturen

BubUS benötigt für die verschiedenen Funktionen eine Reihe von Datenstrukturen. Diese Datenstrukturen werden an dieser Stelle erläutert.

4.2.1 Bereitliste

Der allgemeine Aufbau der Bereitliste wurde im Kapitel 3.4.2 bereits dargelegt. Sie besteht aus der Prioritätsliste und den Threadlisten.

Prioritätsliste

Die Prioritätsliste ist eine doppelt verkettete Liste, die aufsteigend nach Prioritäten sortiert ist. Jedes Element dieser Liste (*Variable:* `prionode`) besteht aus folgenden vier Elementen:

- der Priorität (*Variable:* `int prio`),

- einem Verweis auf den Anfang der zugehörigen Threadliste (*Variable:* `threadnode *threadlist_head`),
- einem Verweis auf das Ende der zugehörigen Threadliste (*Variable:* `threadnode *threadlist_end`),
- einem Verweis auf das nächste Listenelement (*Variable:* `prionode *next`) und
- einem Verweis auf das vorherige Listenelement (*Variable:* `prionode *prev`).

Threadlisten

Zu jeder Prioritätsstufe gehört eine Threadliste. Jedes Threadlistenelement (*Variable:* `struct threadnode`) beinhaltet die folgenden Daten:

- einen Verweis auf die Thread-ID des Threads (*Variable:* `l4_threadid_t *t_id`),
- das lokale Budget (*Variable:* `int l_budget`),
- Echtzeitinformationen (falls der Thread ein Echtzeitthread ist):
  - die Ausführungszeit (Zeitscheibe) (*Variable:* `int rt_s`),
  - die Länge der Periode (*Variable:* `int rt_c`),
  - die Priorität (*Variable:* `int rt_prio`),
  - die Information, ob der Thread bereits gestartet wurde (*Variable:* `char rt_started`),
  - den nächsten Aktivierungszeitpunkt (*Variable:* `l4_uint64_t rt_next_activation`),
- IPC-Informationen:
  - die Information, ob seit dem Blockierzeitpunkt bereits ein neuer Schedulingdurchlauf gestartet wurde (*Variable:* `char new_period`), anhand derer beim Aktivieren des Threads entschieden wird, ob sein lokales Budget aufgeladen werden darf oder nicht.
  - der IPC-Status (*Variable:* `char ipc_state`),
  - die Syscallinformationen vor dem IPC-Syscall (*Variable:* `l4_utcb_t *utcb`)
  - ein Verweis auf den Threadknoten des IPC-Partners (*Variable:* `threadnode *ipc_partner`)
  - die Senderwarteliste (*Variable:* `sndnode *ipc_snd_partner`),
- der Rücksprungthreadknoten (*Variable:* `threadnode *jumps_back`),
- ein Zeiger auf den zugehörigen Prioritätsknoten (*Variable:* `prionode *pnode`),
- ein Verweis auf den nächsten Threadknoten der Liste (*Variable:* `threadnode *next`) und
- ein Verweis auf den vorherigen Threadknoten der Liste (*Variable:* `treadnode *prev`).

4.2.2 Blockiertliste

Die Liste der blockierten Threads entspricht im Wesentlichen der Bereitliste, mit dem Unterschied, dass nur ein einziger Prioritätsknoten mit der Priorität -1 existiert.

4.2.3 Senderwarteschlange, Liste der Echtzeitthreads, hpt-Liste

Senderwarteschlange (*Variable:* `sndnode`), Echtzeitthreads (*Variable:* `realtimenode`) und hpt (*Variable:* `hptnode`) sind ebenfalls doppelt verkettete Listen. Jedes Element enthält lediglich einen Verweis auf den betreffenden Threadknoten und die benötigten Verkettungsinformationen.

Die Liste der **H**öher**p**rioren **T**hreads wird genutzt um dem Scheduler mitzuteilen, wenn Threads mit höherer Priorität, als die der aktuell aktiven Threadliste, bereit werden. Das passiert, wenn Timeouts ablaufen oder Echtzeitthreads aktiviert werden müssen.
Die Echtzeitthreadliste enthält Verweise auf alle zugelassenen Echtzeitthreads des Systems und wird zur Vereinfachung der Admission als Zwischenspeicher genutzt.
Die Senderwarteschlange ist Bestandteil jedes Threadknotens. Sie enthält Verweise auf die Threadknoten aller sendewilligen Threads.

4.2.4 Dealman-Hilfsstruktur t_index

Das Feld *Variable:* `t_index` ist ein Hilfskonstrukt, dass zum einen dem schnelleren Auffinden von Threadknoten dient und zum anderen Informationen beherbergt, die normalerweise von einem Dealman-Bankserver verwaltet würden. Das Feld besteht aus 128 Zeilen und Spalten. Die Zeilennummer entspricht einer Tasknummer, die Spaltennummer einer Threadnummer. `t_index` hält zu jedem Thread folgende Informationen:

- einen Verweis auf den zugehörigen Threadknoten und
- folgende Dealmankonten:
  - Budget (*Variable:* `budget`): Dieses Konto repräsentiert Γ_{CPU_i}.
  - Blockiertes Budget (*Variable:* `blocked_budget`): Auf dieses Konto wird der Bestand vom Budgetkonto transferiert, wenn ein Thread blockiert (z. B. während IPC). Beim Reaktivieren des Threads erfolgt die Rückübertragung.
  - Prioritätsbudget (*Variable:* `prio_budget`): Dieses Konto repräsentiert Γ_{PRIO_i}.
  - Budgetspeicher (*Variable:* `store_budget`): Dieses Konto repräsentiert Γ_{STORE_i}.

4.3 Alienguard

Der Alienguardthread implementiertiert die in 3.5.3 dargelegte Funktionalität. An dieser Stelle wird auf einige Besonderheiten eingegangen und der konkrete Ablauf vor und nach einem Syscall beschrieben.

4.3.1 Speichern des Timestampcounters beim Aktivwerden

Sobald der Alienguard durch eine Exception-IPC aktiv wird, speichert er den aktuellen Timestampcounter. Der Wert wird später vom Scheduler benötigt um ggf. Gums für nicht verbrauchte Rechenzeit zurückzuerstatten (siehe Kapitel 4.5).

4.3.2 Allgemeiner Ablauf vor Beginn eines Syscalls

Zunächst prüft der Alienguard, um welchen Syscall es sich handelt und führt im Falle von IPC eine Verwaltungsroutine aus. Als nächstes speichert er die Syscallinformationen im Threadknoten (Kapitel 4.2.1) des betreffenden Threads (*Variable:* `l4_utcb_t *utcb`) und vermerkt damit, dass dieser nun auf die Erlaubnis für diesen Syscall wartet. Danach blockiert er (im Kontext des Kernscheduling) durch den Aufruf von `ipc_wait` und wartet auf die nächste Exception- oder andere IPC-Nachricht. Jetzt bekommt der Scheduler wieder die Kontrolle. Vor der nächsten Aktivierung des betreffenden Threads, bekommt der Alienguard eine IPC-Nachricht vom Scheduler und erteilt mit Hilfe der gespeicherten Syscallinformationen die Erlaubnis für den Syscall.

Dieser Ablauf ist notwendig um folgende Situation zu vermeiden:

- Ein Thread macht fortlaufend Syscalls.
- Der Alienguard erlaubt den ersten Syscall sofort und sendet die entsprechende Nachricht an den Thread.
- Der Thread läuft jetzt kurzzeitig auf der Zeitscheibe (Kernscheduling) des Alienguard und beendet den Syscall.
- Der Alienguard erlaubt das Ende des Syscalls.
- Der Thread läuft erneut auf der Zeitscheibe (Kernscheduling) des Alienguard und startet den zweiten Syscall.
- ...
- Im Schlimmsten Fall kann nie wieder ein anderer Thread bereit werden, da der Alienguard aus Datenkonsistenzgründen die höchste Priorität im Kontext des Kernscheduling besitzt.

4.3.3 Verwaltungsroutine vor Beginn des IPC-Syscalls

Im Falle des IPC-Syscall wird als erstes untersucht, um welche Art von IPC es sich handelt (`call`, `wait`, `send_and_reply`, u. s. w.). Aus dieser Information kann nun geschlossen werden, in welchem Zustand sich der Thread nach dem Syscall befindet. Diese Information wird ebenfalls im Threadknoten abgelegt (*Variable:* `threadnode.ipc_state`).
Abhängig von der Art des IPC-Calls wird nun entsprechend weiterverfahren:

- Im Falle einer Sendeoperation wird ermittelt, ob der IPC-Partner empfangsbereit und damit blockiert ist (*Variable:* `threadnode.ipc_state` des IPC-Partners). Befindet sich dieser im Zustand `closed wait` wird zudem noch überprüft, ob Sender und erwarteter Sender (*Variable:* `threadnode.ipc_partner` des IPC-Partners) übereinstimmen.

Ist dem so oder befindet der Empfänger sich im Zustand `open wait`, dann wird dieser Empfänger aktiviert und die Verwaltungsroutine ist beendet. Wenn nicht, so wird der Sender in die Senderwarteschlange im IPC-Partner-Threadknoten eingereiht (*Variable:* `threadnode.ipc_snd_partner` des IPC-Partners) und blockiert.

- Im Falle einer Empfangsoperation wird zunächst geprüft ob es sich um offenes- oder geschlossenen Empfang (`open wait` / `closed wait`) handelt. Ist ein berechtigter Sender sendebereit und damit blockiert, so wird er aktiviert und die Verwaltungsroutine ist beendet. Wenn nicht, wird der Thread blockiert.

4.3.4 Ablauf vor Beendigung des Syscall

Im Gegensatz zum Beginn des Syscalls wird das Ende nach dem Ausführen einer weiteren Verwaltungsroutine (IPC) sofort erlaubt.

4.3.5 Verwaltungsroutine vor Beendigung des IPC-Syscalls

Vor Beendigung des IPC-Syscalls wird der betreffende Thread ggf. aus der Senderwarteschlange seines IPC-Partners (*Variable:* `threadnode.ipc_snd_partner` des Partners) gestrichen, sein Zustand (*Variable:* `threadnode.ipc_state` des betreffenden Threads) auf den definierten Wert `IPC_STATE_NULL` und der IPC_Partner (*Variable:* `threadnode.ipc_partner` des betreffenden Threads) auf `NULL` gesetzt

4.3.6 Verbotene Syscalls

Aufgrund der Funktionsweise des Schedulers (beschrieben im Kapitel 4.5) darf der Syscall `l4_thread_schedule` von keinem zu schedulenden Thread ausgeführt werden und wird daher vom Alienguard nicht erlaubt.
Im BubUS-Prototypen sind zudem auch `task_new`, `lthread_ex_regs` sowie `thread_switch` untersagt, da keine Taskverwaltung implementiert, b. z. w. eventuelle negative Auswirkungen nicht untersucht wurden.

4.3.7 Besonderheiten

Zwei spezielle IPC werden gesondert behandelt:

- Serveraufruf an den BubUS-Servicethread zum Umwandeln in einen Echtzeitthread

- Serceraufruf an den BubUS-Servicethread zum Warten bis zum nächsten Schedulingdurchlauf

In beiden Fällen soll der betreffende Thread nach dem Call bis zum nächsten Schedulingdurchlauf nicht mehr aktiviert werden. Aus diesem Grund wird der IPC-Status (*Variable:* `ipc_state`) in diesen speziellen Situationen auf den Wert `IPC_STATE_WAITING_FOR_NEW_ROUND` gesetzt und der Thread bleibt blockiert (bis zur Initialisierung des nächsten Schedulingdurchlaufes).

4.4 Watcher

Der Watcher-Hilfsthread, dessen Priorität (im Kontext des Kernscheduling) zwischen der des gerade laufenden Threads und der aller nicht laufenden Threads liegt schaltet wie in 3.5.2 erläutert, immer, wenn er aktiviert wird, mittels `l4_thread_switch` sofort zum Scheduler.

4.5 Scheduler

Im Folgenden wird im Detail besprochen, wie das Kernstück des BubUS arbeitet.

4.5.1 Funktionsweise

Allgemein läuft ein Schedulingdurchlauf folgendermassen ab:

1. Zu Beginn eines Durchlaufes erfolgt dessen Initialisierung:
   - Die lokalen Budgets ($\Gamma_{CPU_{i_{local}}}$) aller Nichtechtzeitthreads werden mit den globalen Konten (Γ_{CPU_i}) abgeglichen und so wieder aufgeladen.
   - Bereite Threads, deren $\Gamma_{CPU_i} = 0$ ist, werden blockiert.
   - Blockierte Threads, deren $\Gamma_{CPU_i} > 0$ ist, werden erweckt.
   - Blockierte Threads, die bis zum nächsten Schedulingdurchlauf warten (*Variable:* `threadnode.ipc_state` = `IPC_STATE_WAITING_FOR_NEW_ROUND`) werden erweckt.
   - Der Rechenzeitpreises ($E_{s_{min}}$ und E_{s_q}) wird gemäß der Formeln 3.3 und 3.4 neu berechnet
   - Threads mit verändertem Prioritätskontostand (Γ_{PRIO_i}) werden ggf. umgekettet. Dabei kann es vorkommen, dass ein bestehender Prioritätsknoten aufgelöst (wenn der umzukettende Thread der letzte dieser Stufe war) oder ein neuer Knoten angelegt (wenn der umzukettende Thread der erste dieser neuen Stufe ist) werden muss.
   - Die hpt-Liste[1] wird geleert.
   - Die höchstprioren Threadliste wird aktiviert.
   - Der zu aktivierenden Threads wird aus der höchstprioren Threadliste ausgewählt.
2. Die hpt-Liste[1] wird geprüft. Ist ein höherpriorer Thread bereit so verdrängt dieser den ursprünglich zu aktivierenden:
   - Der höherpriore Threads wird als zu aktivierender Thread ausgewählt.
   - Der ursprünglich zu aktivierende Thread wird im Rücksprungthreadknoten des höherprioren gespeichert
   - Der höherpriore Thread wird aus der hpt-Liste[1] ausgekettet.
3. Eine Realzeitscheibe (Länge gemäß Formel 3.12) wird an den zu aktivierenden Thread „verkauft“. Dabei wird sein lokales Budget um den entsprechenden Gumsbetrag gekürzt.

[1] Liste der höher prioren Threads

4. Es wird geprüft, ob innerhalb der Realzeitscheibe (oder bereits davor) ein Echtzeitthread aktiviert werden muss. Ist das der Fall:
   - Die eben „verkaufte“ Realzeitscheibe wird entsprechend gekürzt.
   - Der entsprechenden Gumsbetrag wird auf das lokale Konto des zu aktivierenden Threads rückerstattet.
   - Der demnächst zu aktivierende Echtzeitthread wird in die hpt-Liste[1] eingekettet.
5. Die Priorität (Kontext des Kernscheduling) des zu aktivierenden Threads (Aufruf von `l4_thread_schedule`) wird auf einen Wert gesetzt, der höher als die Priorität des Watchers und kleiner als die Priorität des Schedulers ist.
6. Es wird geprüft, ob der zu aktivierende Thread auf eine Syscall-Erlaubnis durch den Alienguard wartet. Wenn ja, bekommt der Alienguard die Anweisung zum Senden der Erlaubnis.
7. Der Timestampcounters wird gelesen und gespeichert.
8. Der ausgewählte Thread wird durch Blockieren des Schedulers (`open wait`) mit einem Timeout, der der Realzeitscheibe des gewählten Threads entspricht, aktiviert. Nach Ablauf dieser Zeit oder vorzeitiger CPU-Abgabe, wird der Scheduler wieder aktiv.
9. Es wird geprüft, ob der aktivierte Thread innerhalb seiner Zeit blockiert ist. Wenn ja, wird seine Priorität auf die höchstmögliche Priorität (Kontext des Kernscheduling) unterhalb der Priorität des Schedulers angehoben. Auf diese Weise können Timeouts wirksam werden, sobald ein anderer, durch den Scheduler geschedulter Thread läuft. Wenn nicht, wird seine Priorität auf einen Wert unterhalb der Priorität des Watchers abgesenkt. Auf diese Weise wird der Thread vom Kern nicht aktiviert, da es immer einen bereiten höherprioren Thread gibt.
10. Wurde innerhalb der Rechenzeit des aktivierten Threads der Alienguard aktiv, weil der Thread einen Syscall auslösen wollte oder der Timeout eines anderen Threads wirksam geworden ist, so konnte ggf. nicht die gesamte Zeitscheibe aufgebraucht werden. In diesem Fall wird jetzt dem aktivierten Thread der Differenzgeldbetrag für die nicht genutzte Rechenzeit auf das lokales Konto rückerstattet.
11. Ist innerhalb der Rechenzeit ein höherpriorer Thread bereit geworden, so wird dieser in die hpt-Liste[1] eingekettet.
12. Der letzte Arbeitsschritt besteht in der Auswahl des nächsten zu aktivierenden Threads:
    - Existiert ein Rücksprungthreadknoten, ein bereitgewordener höherpriorer Thread hatte den ursprünglich zu aktivierenden verdrängt, so wird dieser verdrängte Thread ausgewählt, sofern das Geld des höherprioren aufgebraucht ist.
    - Ist der zuletzt Aktivierte blockiert worden und war er der letzte in seiner Threadliste, so wird die nächstpriore Threadliste aktiviert und deren erster Thread ausgewählt.
    - Haben alle Threads der aktiven Threadliste ihre Gums aufgebraucht, so wird ebenfalls die nächstpriore Threadliste aktiviert und deren erster Thread ausgewählt.
    - In jedem anderen Fall wird der nächste Thread der aktuellen Threadliste gewählt.

4.5.2 Besonderheiten bei Rückzahlungen nach lokalen Überweisungen

Local Donations (Kapitel 3.4.2) sind Bestandteil normaler Überweisungen, für die der Service (Kapitel 4.6) verantwortlich ist. Dabei werden Gums von einem lokalen Konto zum anderen überwiesen. Als besonderer Fall ist dabei folgende beispielhafte Situation zu betrachten:

- Thread t_1 hat noch 1000γ auf seinem lokalen Konto.
- Vor der Aktivierung des Threads t_1 bekommt er Rechenzeit für 150γ. Auf seinem lokalen Konto verbleiben also 850γ
- Nach seiner Aktivierung versucht t_1 925γ an den Thread t_2 zu überweisen.
- Da t_1 nur noch 850γ besitzt werden nur diese 850γ an t_2 überwiesen.
- Da eine Überweisung einen IPC-Syscall auslöst, wird der Alienguard bereit und übergibt, nachdem seine Aufgabe erfüllt ist, die Kontrolle zurück an den Scheduler.
- Der Scheduler zahlt nun einen Betrag von 100γ zurück, da t_1 durch das Aktivwerden des Alienguard nicht die volle bezahlte Zeit nutzen konnte.

Im Ergebnis hat t_1 noch genügend Geld auf seinem lokalen Konto, konnte aber den gewünschten Betrag nicht überweisen.
Aus diesem Grund prüft der Scheduler vor einer Rückzahlung nach der Aktivierung eines Threads, ob dieser eine lokale Überweisung durchgeführt hat, die nur unvollständig erfolgt ist. Ist das der Fall, so wird der Rücküberweisungsbetrag derart aufgeteilt, dass zunächst der Fehlbetrag der lokalen Donation an den Donationpartner übertragen und der Rest an den zuletzt aktiven Thread gezahlt wird. Im obigen Beispiel bedeutet das:

- t_2 bekommt die fehlenden 75γ auf sein lokales Konto gutgeschrieben.
- t_1 bekommt die noch verbleibenden 25γ zurücküberwiesen.

4.6 Service

Neben denen im Kapitel 3.5.4 aufgelisteten Funktionen implementiert der Servicethread zusätzlich zwei Dealmandienste:

- das Abfragen des Gesamtgeldes Γ_{CPU} und
- Überweisungen (Donations) zwischen den nichtlokalen Konten der Threads.

Alle Funktionen werden im Folgenden näher betrachtet.

4.6.1 Anmelden neuer Threads

Mit dem Aufruf `bubus_new_thread(l4_threadid_t *t_id)` können neue Threads beim Scheduler angemeldet werden. Aus dem Prioritätsbudgets eines neuen Threads, wird mit Hilfe der Prioritätsstufen-Kostenfunktion (Kapitel 3.4.1) errechnet, in welche Threadliste er eingekettet und ob ggf. ein neuer Prioritätsknoten angelegt werden muss.
Diese Funktion kann im Prototypen lediglich von einer weiteren vertrauenswürdigen Komponente (siehe 4.8) sinnvoll genutzt werden, da „normale“ von BubUS geschedulte Threads derzeit keine Threads starten können.

4.6.2 Ummelden als Echtzeitthread (Echtzeitadmission)

Mittels `bubus_mk_rt(l4_threadid_t *t_id, int period_length)` kann ein Thread zum Echtzeitthread werden. Für den BubUS-Prototypen wurde Echtzeitscheduling nach RMS implementiert. Die Zulassung erfolgt daher wie folgt:

1. Gemäß der Formel 3.2 wird aus der angegebenen Periode, dem Geld des Threads und dem Gesamtgeld im System, die Gesamtzeitscheibe des neuen Echtzeitthreads berechnet.

2. Aus Periode und Gesamtzeitscheibe des neuen Echtzeitthreads und den Perioden und Gesamtzeitscheiben der anderen, bereits zugelassenen Echtzeitthreads, wird nun das RMS-Admissionkriterium 2.2.7 geprüft. Ist es nicht erfüllt, wird der Thread nicht zugelassen, dann endet die Zulassung an dieser Stelle und der Thread verbleibt an der bisherigen Stelle in seiner Threadliste.

3. Ist das Kriterium erfüllt, wird der Thread zugelassen, wird der Thread auf in die entsprechend hochpriore Threadliste umgekettet. Ggf. kommt es dabei zu einer Veränderung der Prioritäten anderer Echtzeitthreads. Diese Änderungen sind nicht mit weiteren Umkettoperationen verbunden.

4. Nach dem Umketten wird der IPC-Status des Threads (*Variable:* `threadnode.ipc_state`) auf den Wert `IPC_STATE_WAITING_FOR_NEW_ROUND` gesetzt. Auf diese Weise wird erreicht, dass der Thread zu einem definierten Zeitpunkt startet.

Ein Echtzeitthread kann keine Donations (weder lokale noch nicht-lokale) durchführen. Der Grund dafür lieg darin, dass sein Geld vollständig für die gegebene Echtzeitgarantie vergeben ist (vgl. Kapitel 3.4.4 Punkt Sicherheit). Des weiteren kann ein Echtzeitthread nicht wieder zu einem „normalen“ Thread werden.

4.6.3 Blockieren bis zum Beginn des nächsten Durchlaufes

Ein Thread kann die verbleibende Rechenzeit eines Schedulingdurchlaufes, durch Aufruf von `bubus_wait_for_next_round()`, aufgeben. Im Gegensatz zu einer lokalen Überweisung, bei der ein Teil dieser Rechenzeit an einem bestimmten anderen Thread übergeben wird, kommt die aufgegebene Zeit dabei dem gesamten Threadsystem zugute.

4.6.4 Lokale- und Dealmanüberweisungen

Aufgrund des Fehlen eines Dealman-Bankservers implementiert BubUS die zwingend benötigten Dienste dieses Servers selbst. Zu diesem Zweck gibt es das in Kapitel 4.2 beschriebene Array *Variable:* `t_index`, das für jeden Thread die vier Konten

- Budget (*Variable:* `budget`): Dieses Konto repräsentiert Γ_{CPU_i}.
- Blockiertes Budget (*Variable:* `blocked_budget`): Auf dieses Konto wird der Bestand vom Budgetkonto transferiert, wenn ein Thread blockiert (z. B. während IPC). Beim Reaktivieren des Threads erfolgt die Rückübertragung.
- Prioritätsbudget (*Variable:* `prio_budget`): : Dieses Konto repräsentiert Γ_{PRIO_i}.

- Budgetspeicher (*Variable:* `store_budget`): Gums auf diesem Konto werden beim Scheduling nicht berücksichtigt.

sowie einen Verweis auf den entsprechenden Threadknoten *Variable:* `*t_nd` enthält.

Durch Aufrufen der Servicefunktion `bubus_remid(int tasknr, int threadnr, char from, char to, int budget, int l_budget)` können lokale- und auch Dealmanüberweisungen angewiesen werden. Die einzelnen Parameter haben dabei folgende Bedeutungen:

- `int tasknr, int threadnr` definiert den Empfänger der Überweisung
- `char from` gibt an, **von** welchem Konto (Budget, Prioritätsbudget, Budgetspeicher) des Senders belastet werden soll.
- `char to` gibt an, **auf** welches Konto (Budget, Prioritätsbudget, Budgetspeicher) des Empfängers der Betrag überwiesen werden soll.
- `int budget` definiert den zu überweisenden Betrag.
- `int l_budget` definiert den lokal zu überweisenden Betrag.

Auf diese Weise können mit einem Aufruf zwei Überweisungen durchgeführt werden, eine sofort wirksame lokale und eine, zu Beginn nächsten Durchlaufes aktiv werdende Dealmandonation. Bei beiden Arten wird jeweils der maximal mögliche Betrag überwiesen. Der Aufruf

```
int gamma_cpu = bubus_info_gamma_cpu();
bubus_remid(9,5,DEALMAN_BUDGET,DEALMAN_STORE,gamma_cpu,gamma_cpu);
```

bewirkt somit eine Dealman-Donation des gesamten Geldes auf dem Budgetkonto des aufrufenden Threads auf das Budgetspeicherkonto des Threads 9.05. Diese Überweisung wird zu Beginn des nächsten Schedulingdurchlaufes, bei dessen Initialisierung (siehe Punkt 1 der Funktionsweise des Schedulers im Kapitel 4.5) aktiv. Erhält der aufrufende Thread bis dahin selbst keine Donation so wird er zu diesem Zeitpunkt blockiert.
Neben der Dealman-Donation wird zudem auch eine lokale Überweisung des gesamten lokalen Budgets vom aufrufenden Thread zum Empfänger 9.05 durchgeführt. Diese Donation zeigt sofort Wirkung, indem der aufrufende Thread bis zum nächsten Durchlauf nicht mehr aktiviert wird, da sein Geld für den aktuellen Durchlauf aufgebraucht ist. Im Gegenzug erhält der Empfänger 9.05 entsprechend mehr Rechenzeit.

4.7 Systemparameter und Optionen

Die im Kapitel 3.4.1 beschriebenen Systemparameter sind entsprechend Tabelle 4.1 festgelegt.

s_{MIN}	s_{MAX}	s_Q	$E_P(P)$	Γ_{CPU}
$100ms$	$200ms$	$10ms$	$100\gamma * \lfloor 2^{(P-1)} \rfloor$	10000γ

Tabelle 4.1: Systemparameter

Der Scheduler verfügt über die Startoption `-v`. Wird sie angegeben startet BubUS im Verbosemodus und protokolliert jede Aktion auf die Console.

4.8 Start des Systems

Da das BubUS-System auf dem Alienmodell basiert und nur Alienthreads korrekt schedulen kann, wird eine vertrauenswürdige Komponente zum Starten der ersten Threads benötigt. Dieser erste Thread bekommt, sobald er sich beim Scheduler durch Aufruf von `bubus_new_thread` registriert hat, das gesamte Schedulinggeld des Systems Γ_{CPU} überwiesen. Nun kann er Alienthreads erzeugen, diese ebenfalls bei BubUS anmelden und die erhaltenen Gums unter ihnen aufteilen. Am Ende dieses Vorgangs blockiert der erste Thread für immer. Nun gibt es nur noch Alienthreads und das System arbeitet korrekt.

4.9 Einschränkungen des Prototypen

Der Prototyp des BubUS-Systems dient in erster Line dazu, das beschriebene Schedulingmodell und seine Vorzüge zu demonstrieren, er weist deshalb noch einige Einschränkungen auf. Keine dieser Einschränkungen ist modellbedingt!

4.9.1 Kein Paging

Beim Alienmodell (siehe Kapitel 2.1.3) werden die entsprechenden Exceptions derzeit nur an den Pager des Alienthreads zugestellt. Um diese Exceptions nutzen zu können gibt es den Alienguard (siehe Kapitel 4.3), der Pager jedes zu schedulenden Threads sein muss. Damit diese Threads dynamisch Speicher allokieren können müsste entweder der Alienguard Pagerfunktionalität implementieren oder das Alienmodell derart abgeändert werden, dass die entsprechenden Exceptions auch an einen anderen Thread, einen reinen Exceptionhandler, übermittelt werden können. Da die erste Variante eine Vermischung von Scheduler- und Speicherverwaltungsfunktionalität zur Folge hätte ist Variante zwei zu bevorzugen. Die Beispielszenarien, die im Kapitel 4.10 beschrieben sind, benötigen keine dynamische Speicherallokation.

4.9.2 Keine Interruptbehandlung

Aus zeitlichen Gründen und der Tatsache heraus, dass die Behandlung von Interrupts keine zusätzlichen Erkenntnisse über das beschriebene Schedulingmodell bringt, wurde auf deren Implementierung verzichtet. Grundsätzlich ist die Interruptbehandlung nahezu identisch mit der implementierten Behandlung von IPC und Timeouts.

4.9.3 Kein nachträgliches Starten von Threads/Tasks, kein Abmelden

Das Starten neuer Threads ist eine sicherheitskritische Funktion, da nicht bei BubUS registrierte Threads den definierten Schedulingablauf stören können. Für dieses Problem gibt es zwei Lösungsmöglichkeiten:

- die Kontrolle und ggf. Modifikation der entsprechenden Syscalls (`lthread_ex_regs` und `thread_switch`) durch den Alienguard
- oder die Einführung eines vertrauenswürdigen Task-/Threadservers.

Bei der Entscheidung für eine der beiden Varianten muss eine weitere Frage berücksichtigt werden: Wer bekommt das Geld eines beendenden Threads?

Diese Frage wurde bereits in [Leb05, Kapitel 5.2] beantwortet. Dort wird unter anderem eine hierarchische Threadstruktur beschrieben. Beendet ein Thread, so bekomm sein Erzeugerthread dessen Geld. Dafür wird ein Task-/Threadserver benötigt. Dieses Vorgehen ergibt auch bezüglich des Entwurfsziels Fair-Share-Scheduing (Kapitel 2.2.3) Sinn. Deshalb ist die zweite Lösungsvariante zu bevorzugen. Durch den erheblichen zeitlichen Mehraufwand und die fehlende Notwendigkeit dieser Funktion für die Beispielszenarien, wurde auf die Implementierung verzichtet.

4.9.4 Kein Echtzeitscheduling mit dynamischen Prioritäten

Das beschriebene Schedulingmodell unterstützt Echtzeitschedulingverfahren mit statischen und dynamischen Prioritäten. BubUS implementiert zu Demonstrationszwecken lediglich ein Verfahren mit statischen Prioritäten (RMS).

4.9.5 Keine Abgabe der Echtzeiteigenschaft

Wurde ein Thread einmal zu einem Echtzeitthread umgemeldet, so kann er nie wieder zu einem Nichtechtzeitthread werden.

4.9.6 Maximale Anzahl schedulbarer Threads

Durch die größe des Felds *Variable:* `t_index` ist die maximale Anzahl der Threads derzeit auf $128 * 128 = 16384$ beschränkt.

4.9.7 Kein Lending

Donation und Lending sind Dealmandienste, die normalerweise von einem Dealmen-Bankserver angeboten werden sollen. Da ein solcher Server derzeit noch nicht existiert, Donation zu Demonstrationszwecken aber zwingend benötigt wird, implementiert der im Servicethread des Prototypen (Kapitel 4.6) diesen Dienst. Auf den allgemeineren Fall, das Lending, wurde dabei verzichtet, weil diese Funktionalität für die Beispielszenarien nicht benötigt wird.

4.10 Beispielszenarien

Im diesem Kapitel werden drei Szenarien vorgestellt, die die Fähigkeiten des BubUS demonstrieren.

4.10.1 IPC- und Timeoutszenario

Das erste Szenario demonstriert die Ünterstützung von IPC und Timeouts, sowie den allgemeinen Schedulingablauf. Es besteht aus vier Threads mit jeweils verschiedenen Aufgaben.

- Thread t_1 wartet mittels des Calls `l4_ipc_wait` ständig auf eine Nachricht des Threads t_2. Dabei wird ein Empfangstimeout von $2000ms$ verwendet.
- Thread t_2 sendet mittels `l4_ipc_call` zunächst eine Nachricht an Thread t_1 und wartet $5000ms$ auf eine IPC-Nachricht, danach geht er in eine Endlosschleife über.

- Thread t_3 und t_4 geben mittels `printf` ständig „`hello\` aus.

Das Geld wird wie in Tabelle 4.2 dargestellt verteilt. Gemäß dem Modell erhält damit t_1 dreimal soviel Rechenzeit wie t_2, t_2 doppelt soviel Rechenzeit wie t_4 und t_3 wird nicht aktiviert, da er kein Geld besitzt. t_1 und t_2 bekommen laut Prioritätsstufen-Kostenfunktion (Tabelle 3.2) die Priorität 1, t_3 und t_4 erhalten die Priorität 0.

Thread	Γ_{CPU}	Γ_{PRIO}	Γ_{STORE}
t_1	3000γ	500γ	0γ
t_2	1000γ	500γ	0γ
t_3	0γ	0γ	4500γ
t_4	500γ	0γ	0γ

Tabelle 4.2: IPC- und Timeoutszenario: Gumsverteilung

BubUS wird in diesem Szenario mit der Option `-v` gestartet, damit sind können alle Aktionen, die er ausführt, beobachtet werden.

4.10.2 Echtzeitszenario

Das zweite Szenario demonstriert die Echtzeitunterstützung von BubUS. Dafür werden drei Threads, t_1, t_2 und t_3, gestartet. Die im System verfügbaren Gums Γ_{CPU} werden dabei so verteilt, dass die Gesamtzeitscheiben für die einzelnen Threads, gemäß Rechenzeitverteilung bei Echtzeitthreads (Formel 3.9), denen in Tabelle Tabelle 4.3 angegebenen Ausführungszeiten entsprechen. Nach Anmeldung und Start der Threads, versuchen alle drei sich durch Aufruf von `bubus_mk_rt` als Echtzeitthreads zu registrieren. Dabei geben sie die in Tabelle 4.3 angebenenen Periodenlängen an. Alle drei Threads laufen danach in Endlosschleifen und konsumieren lediglich die zugeteilte Rechenzeit.

Thread	**Ausführungszeit**	**Periodenlänge**
t_1	$2000ms$	$5000ms$
t_2	$1000ms$	$3000ms$
t_3	$1000ms$	$5000ms$

Tabelle 4.3: Echtzeitszenario

Hier wird ein Grenzfall der RMS-Admission beschrieben: Das Admission-Kriterium (siehe 2.2.7) ist nicht erfüllt, die drei Threads ließen sich aber dennoch einplanen. Bei der RMS-Admission des BubUS-Prototypen wird dieser Fall nicht geprüft. Deshalb wird Thread t_3 nicht als Echtzeitthread zugelassen und als Nichtechtzeitthread geschedult.
Auch hier wird BubUS im Verbosemodus gestartet. Das Szenario zeigt, wie die beiden Echtzeitthreads t_1 und t_2 periodisch, teilweise mehrfach innerhalb eines Schedulingdurchlaufes, aktiviert werden und immer Vorrang vor dem Nichtechtzeitthread t_3 haben.

4.10.3 Erzeuger-Verbraucher-Szenario

Szenario Drei beschreibt eines der Motivationsszenarien aus Kapitel 3.2.1 - eine rechenzeitbalancierende Lösung des Erzeuger-Verbraucher-Problems. Das Szenarion besteht aus drei

Threads:

- dem Erzeuger,
- dem Verbraucher und
- dem Reporter, der ständig den Status des Puffers graphisch darstellt.

Erzeuger und Verbraucher schreiben, b. z. w. lesen aus einem 23-elementigen Puffer. Da der Puffer zu Beginn leer ist, bekommt der Verbraucher zunächst kein Geld und wird deshalb vom Scheduler nicht aktiviert. Der Erzeuger erhält 6000γ, die verbleibenden 3000γ ($\Gamma_{CPU} = 10000\gamma$) bekommt der Reporter. Der Erzeuger erzeugt in etwa 10-mal so schnell, wie der Verbraucher konsumiert.
Genau wie im Motivationsszenario beschrieben donieren sich beide gegenseitig jeweils $\frac{1}{n} * (\Gamma_{CPU_{Erzeuger}} + \Gamma_{CPU_{Verbraucher}})$, konkret also $\frac{1}{23} * 6000$ nach dem Schreiben, b. z. w. Entnehmen eines Elementes aus dem Puffer.
Die Ausgabe des Reporters zeigt das Erwartete: der Puffer ist niemals ganz leer und niemals ganz voll. Durch die Donations erzeugen und verbrauchen Erzeuger und Verbraucher nach einer kurzen Angleichphase etwa mit gleicher Geschwindigkeit, obwohl der Erzeuger um den Faktor 10 schneller ist als der Verbraucher.

Zum Vergleich wurde die gleiche Simulation mittels klassischer zählender Semaphoren implementiert (Auszüge siehe Tabelle 4.4). Im Gegensatz zur Lösung mittels Rechenzeitabstraktion und Donation, ist der Puffer hier nach kurzer Zeit komplett gefüllt und verhält sich von da an wie ein einelementiger Puffer.

Konkrete Ergebnisse folgen zusammen mit einer entsprechenden Auswertung im Kapitel 5.2

Klassische Lösung mit Semaphoren

```
int consumer_code(){
 int i;
 for(;;){
   l4semaphore_down(&sem_full);
   buffer[start] = EMPTY;
   start = (start + 1) % N;
   /*consume some time*/
   for(i=0;i<(1000000*100);i++){
     l++;l--;l++;l--;l++;l--;
   }
   l4semaphore_up(&sem_empty);




 }
return 0;
}
```

Rechenzeitbalancierende Lösung

```
int consumer_code(){
 int i;
 for(;;){

   buffer[start] = EMPTY;
   start = (start + 1) % N;
   /*consume some time*/
   for(i=0;i<(1000000*100);i++){
     l++;l--;l++;l--;l++;l--;
   }
 bubus_remid(producer_id.id.task,
      producer_id.id.lthread,
      DEALMAN_BUDGET,
      DEALMAN_BUDGET,
      gums2remid, gums2remid);
 }
return 0;
}
```

Tabelle 4.4: Erzeuger-Verbraucher-Problem: Gegenüberstellung von klassischer- und rechenzeitbalancierender Lösung

5 Bewertung

5.1 Mehraufwand (Overhead)

Zur Bewertung des Overheads müssen zwei verschiedene Quellen betrachtet werden:

- der Overhead des Modells und
- der Overhead der Implementierung.

5.1.1 Overhead des Modells - Durchlaufinitialisierung

Der größte Overhead des Modells tritt durch die Initialisierung der einzelnen Schedulingdurchläufe auf. Dabei werden beide Listen, die Bereitliste und die Blockiertliste, einmal komplett traversiert. Der Mehraufwand steigt daher linear mit der Anzahl der zu schedulenden Threads. Ausserdem müssen die lokalen CPU-Konten aufgeladen, ggf. Threads umgekettet, blockiert oder erweckt, der aktuelle Rechenzeitpreis berrechnet und die hpt-Liste geleert werden. Es wurden an dieser Stelle keine konkreten Messungen vorgenommen, da der Arbeitsaufwand der Durchlaufinitialisierung stark von der jeweiligen Situation abhängt.
In den Szenarien des Prototypen ist dieser Overhead durch ein schubweises Fortschreiten des jeweiligen Threadsystems zu bemerken.

5.1.2 Overhead der Implementierung

Die Art der Implementierung bringt weiteren Mehraufwand mit sich. Der größte Teil entsteht durch die Entscheidung für Userland-Scheduling (Vlg. 3.3.4).

Zusätzliche Threadwechsel - grundsätzlich

Aufgrund des Userland-Scheduling kommt es grundsätzlich bei jedem Schedulingereignis zu mindestens einem zusätzlichen Threadwechsel, der Aktivierung des Schedulerthreads.

Zusätzliche Threadwechsel - bei Syscalls

Da jeder Syscall eines Alienthreads zwei Exceptionnachrichten (vor- und nach dem Call) an den Alienguard zur Folge hat, müssen insgesamt sechs zusätzliche Threadwechsel in Kauf genommen werden:

1. Aktivieren des Alienguard durch die Exceptionnachricht vor dem Call - Dieser merkt sich die Anfrage und schalte um zum Scheduler.
2. Aktivieren des Schedulers - Der Scheduler schaltet vor der nächsten Aktivierung des betreffenden Threads zum Alienguard.

3. Aktivieren des Alienguard - Der Alienguard schickt die Exceptionbestätigung an den betreffenden Thread.

4. Aktivieren des betreffenden Threads. Dieser führt nun den Syscall durch.

5. Aktivieren des Alienguard durch die Exceptionnachricht nach dem Call - Dieser antwortet sofort mit der Exceptionbestätigung.

6. Aktivieren des betreffenden Threads. Dieser beendet den Call.

Häufige Änderung von Kernprioitäten

Pro Aktivierung eines Threads, wird zweimal dessen Kernpriorität verändert (vor dem Aktivieren und nach dem Unterbrechen) Dabei muss der Thread vom Kernscheduler umgekettet werden.

5.2 Optimierungspotential

Bei der Entwicklung des BubUS-Prototypen wurde kein Wert auf Performance gelegt. Es gibt aus diesem Grund noch eine Menge an Optimierungspotential. Drei Ansätze sollen an dieser Stelle angesprochen werden.

5.2.1 Optimierung des Modells - Durchlaufinitialisierung

Der Vorgang der Durchlaufinitialisierung kostet viel Zeit. Ein Teil dieser Zeit kann unter Bestimmten Umständen eingespart werden.

- Es ist nicht erforderlich, alle CPU-Konten exakt zu Beginn eines neuen Durchlaufes aufzuladen. Ferner kann das vor der ersten Aktivierung jedes Threads innerhalb des Durchlaufes passieren.

- Tätigt ein Thread eine Überweisung, bei der er sein CPU-Konto leert, dann wird der Thread sofort blockiert.

- Bekommt ein Thread, der auf Grund von Geldmangel blockiert ist, eine Überweisung, dann wird er sofort erweckt. Das lokales Konto wird erst bei seiner ersten Aktivierung des nächsten Durchlaufes aufgeladen.

- Tätigt der Thread mit den wenigsten Gums auf seinem CPU-Konto eine Überweisung von seinem CPU-Konto, dieser Thread bestimmt den Rechenzeitpreis, so wird der neue Rechenzeitpreis gespeichert und vor dem nächsten Durchlauf ggf. aktualisiert.

Lediglich das Umketten von Threads auf andere Prioritätsstufen, bei Veränderungen von Prioritätskontoständen und das leeren der hpt-Liste, muss zwingend zu Beginn eines neuen Durchlaufes durchgeführt werden.

Unter der Annahme, dass Threads selten ihre Priorität ändern, wird die Durchlaufinitialisierung auf diese Weise stark verkürzt.

5.2.2 Optimierung der Implementierung

Scheduler als Alienguard

Es ist zu überprüfen, ob der Schedulerthread die Aufgaben des Alienguard übernehmen kann. Damit könnten die Threadwechsel zwischen den beiden Threads, beim Auftreten eines Syscalls, entfallen.
Folgende zwei Dinge sprechen für diese Möglichkeit:

- Scheduler und Alienguard können niemals parallel bereit sein, da der Alienguard eine höhere Kernpriorität besitzt als der Scheduler, aber nur durch Alien-Exceptions bereit werden kann. Alien-Exceptions werden wiederum nur von Threads ausgelöst, die eine niedrigere Kernriorität besitzen als der Scheduler.

- Der Scheduler befindet sich immer im Wartezustand (warten auf IPC), wenn eine Exceptionnachricht zugestellt werden kann.

Alienguard als Service

Eine weitere Möglichkeit Threadwechsel einzusparen besteht darin, Service- und Alienguardfunktionalität in einen einzigen Thread einzubauen. Ein IPC-Call an den Service könnte direkt vom Alienguard abgefangen und bearbeitet werden. Es würden dabei, je nach Umsetzung, mindestens zwei Threadwechsel entfallen.
Bei dieser Optimierung muss zwischen Funktionskapselung und Performance abgewogen werden.

5.3 Erzeuger-Verbraucher-Szenario

Das wichtigste und aussagekräftigste Szenario, das Erzeuger-Verbraucher-Szenario, wird nun bewertet. Im Kapitel 3.2.1 als motivierendes Szenario eingeführt und in 4.10.3 konkret beschrieben zeigt es, wie kontenbasiertes Scheduling zusammen mit Donation sinnvoll genutzt werden kann. In 3.2.1 wurden zwei Behauptungen aufgestellt:

- zum einen sollen mittels Donation, permanente Laufzeitunterschiede zwischen Erzeuger und Verbraucher ausgeglichen werden (Rechenzeitbalancierung) und

- zum anderen soll ein solches System unter bestimmten Umständen schneller fortschreiten können, als in einer klassischen, semaphorbasierten Umsetzung.

Da im Rahmen dieser Arbeit beide Ansätze, donation- und semaphorbasierter, implementiert wurden, ist ein direkter Vergleich möglich. Dazu wurde jeweils die Zeit gemessen, die zum kompletten Füllen des Puffers vergeht, also die Zeit, die der Erzeuger benötigt, um jedes Pufferelement einmal zu beschreiben. Es werden dabei zwei Fälle unterschieden:

1. das erste Füllen, dabei kann der Erzeuger noch nicht durch den langsameren Verbraucher ausgebremst werden, und

2. jedes weitere Füllen, bei dem der Einfluss des langsameren Verbrauchers auf den Erzeuger maximal ist.

Im konkreten Fall trifft Punkt zwei ab dem dritten Füllen des Puffers zu. Das liegt daran, dass der Verbraucher es während des ersten Füllens bereits schafft, einige Elemente zu leeren. Im semaphorbasierten Fall ist das daran zu erkennen, dass der Puffer nach dem ersten Füllen bereits wieder leere Elemente enthält. Die Tabellen 5.1 und 5.2 zeigen die Messergebnisse.

Messung	**erstes Füllen**	**Folgefüllen**
1	$5006,0ms$	$46471,0ms$
2	$5576,0ms$	$46208,0ms$
3	$5145,0ms$	$46341,0ms$
4	$4862,0ms$	$46579,0ms$
5	$5331,0ms$	$46588,0ms$
6	$4959,0ms$	$47034,0ms$
7	$5021,0ms$	$46573,0ms$
8	$4932,0ms$	$46595,0ms$
9	$5047,0ms$	$46565,0ms$
10	$4873,0ms$	$46091,0ms$
∅	$5075,2ms$	$46504,5ms$

Tabelle 5.1: Zeiten der semaphorbasierten Lösung

Messung	**erstes Füllen**	**Folgefüllen**
1	$4157,0ms$	$20537,0ms$
2	$4097,0ms$	$20115,0ms$
3	$3592,0ms$	$21242,0ms$
4	$3583,0ms$	$20197,0ms$
5	$3612,0ms$	$22754,0ms$
6	$3549,0ms$	$20155,0ms$
7	$3568,0ms$	$21340,0ms$
8	$4085,0ms$	$21116,0ms$
9	$4105,0ms$	$21007,0ms$
10	$4129,0ms$	$20926,0ms$
∅	$3847,7ms$	$20938,9ms$

Tabelle 5.2: Zeiten der donationbasierten Lösung

Die Zeit zum ersten Füllen verhält sich zur Zeit des Füllens bei maximalem Einfluss des langsameren Verbrauchers wie folgt:

- semaphorebasierter Ansatz: $1 : 9,2$
- donationbasierter Ansatz: $1 : 5,4$

In beiden Fällen entspricht das Ergebnis der Erwartung. Da der Erzeuger etwa um den Faktor 10 schneller ist als der Verbraucher, wird der Erzeuger in der semaphorbasierten Lösung ca. auf ein Zehntel seiner möglichen Geschwindigkeit abgebremst. Im rechenzeitbalancierenden Fall werden die Geschwindigkeiten beider Threads aneinander angeglichen. Durch den zusätzlichen Overhead wird das theoretische Ergebnis von $1 : 5$ nicht erreicht.

5.4 Mehrwert

An dieser Stelle sollen nochmals alle Funktionen aufgelistet werden, durch die sich das in dieser Arbeit beschriebenen Systems der Rechenzeitabstraktion durch kontengesteuertes Scheduling gegenüber klassischen Schedulingverfahren abgrenzt. Erwartungsgemäß stimmt diese Liste in den meisten Punkten mit den gesetzten Zielen aus Kapitel 3.1 überein.

- Feingranulare Rechenzeit-Donation/-Lending ist auf Userebene möglich.
- Die Verteilung der verfügbaren Rechenzeit auf alle bereiten Threads wird durch die Verteilung des Geldes parametrisiert.
- Fair-Share-Scheduling wird unterstützt. Damit verbunden ist der Schutz vor Denial-of-Service Angriffen auf die Ressource Rechenzeit
- Es werden periodische Echtzeit- und nichtperiodische Nichtechtzeitthreads unterstützt.
- Jedem bereiten Threads mit Geld wird ein Minimum an Rechenzeit garantiert (Fairness).

Der Mehrwert des Systems ist dabei in der Gesamtzeit aller Punkte zu sehen.

6 Zusammenfassung

6.1 Offene Probleme

6.1.1 Implemenierung des Dealman!

An verschiedenen Stellen dieser Arbeit wurde darauf hingewiesen, dass das beschriebene Schedulingmodell einen Dealman-Bankservice voraussetzt. Da ein solcher Service bisher nicht existiert, müßen die benötigten Dienste zur Zeit vom Scheduler selbst angeboten und umgesetzt werden. Bei einer Implementierung des Dealman ist darauf zu achten, dass es die Möglichkeit zum schnellen Kontenabgleich zwischen Dealmanserver und einem entsprechend autorisierten Thread, wie dem Scheduler, gibt. Eine entsprechende Möglichkeit ist in [Leb05, Kapitel 4.2.4] beschrieben.

6.1.2 Beschränkungen

Im Kapitel 4.9 wurden eine Reihe von Beschränkungen des Prototypen dargelegt. Insbesondere das Fehlen von Paging und der Möglichkeit nachträglich neue Tasks und Threads zu starten verringert die Zahl der Anwendungsmöglichkeiten derzeit stark. Jedoch ist eine Implementierung der fehlenden Funktionen ohne Beschränkung des Schedulingmodells möglich. Die jeweils notwendigen Schritte wurden zum Teil an den jeweiligen Stellen (Kapitel 4.9.1, 4.9.2, 4.9.3) bereits angesprochen.

6.2 Ausblick - Mehr-Knoten-Systeme

Abgesehen von der im Kapitel 3.4.5 angedeutete Möglichkeit der Integration in bestehende Gridlösungen, erscheint auch eine Dealman-basierte Ressourcenverwaltung in Mehr-Rechner-Systemen als denkbar.
Dabei existieren aber noch zu lösende Probleme wie zum Beispiel die Frage, wie die unterschiedlichen Wertigkeiten des Geldes der verschiedenen Rechnerknoten miteinander vereinbart werden können oder wie knotenübergreifende Donations umzusetzen und zu bewerten sind, b. z. w. welche Folgen Geldmangel für einen Knoten hat.

6.3 Abschluss

Abschliessend ist zu sagen, dass die gesetzten Ziele (Kapitel 3.1 und 3.4.4) im wesentlichen erreicht wurden und die Vorteile des beschriebenen Systems durch den BubUS-Prototypen und verschiedene Szenarien nachgewiesen werden konnten. Durch die Beschränkungen des Prototypen (Kapitel 4.9) und das Fehlen eines Dealman-Bankservers ist das vorgestellte System zum jetzigen Zeitpunkt aber noch nicht praxistauglich.

Abbildungsverzeichnis

2.1 RMS-Beispiel: Einplanung 16
2.2 GRACE - Ressourcemanagement und Scheduling Systemarchitektur 18

3.1 Bereitliste mit einfachem Schedulingablauf 32
3.2 Bereitliste mit Echtzeitthreads 33
3.3 BubUS - grundsätzlicher Aufbau 39

Tabellenverzeichnis

2.1 RMS-Beispiel: Daten 16

3.1 Symbole und deren Bedeutung 22
3.2 Prioritätsstufen-Kostenfunktion $E_P(P) = 100\gamma * \lfloor 2^{(P-1)} \rfloor$) 34

4.1 Systemparameter 52
4.2 IPC- und Timeoutszenario: Gumsverteilung 55
4.3 Echtzeitszenario 55
4.4 Erzeuger-Verbraucher-Problem: Gegenüberstellung von klassischer- und rechenzeitbalancierender Lösung 56

5.1 Zeiten der semaphorbasierten Lösung 60
5.2 Zeiten der donationbasierten Lösung 60

Glossar

ABI Application Binary Interface

Aktivieren Als Aktivieren wird die Zuteilung der *Ressource* CPU bezeichnet

Average case normaler (durchschnittlicher) Fall

Blockieren Blockieren bedeutet, dass ein *Thread* aus der Bereitliste in die Blockiertliste umgekettet und damit nicht mehr aktiviert wird. Blockieren ist das Gegenstück zum *Erwecken*.

Capability Capabilities sind Rechte für bestimmte Aktionen auf Objekten. Gegenüber anderen Schutzsystemen haben Capabilities den Vorteil, dass sie einfach weitergegeben werden können. Eine Beispiel für Capabilities sind Passwörter. Jeder, der das Passwort einer verschlüsselten Datei kennt, darf/kann diese entschlüsseln.

Dealman Dealman steht für Deal-Manager. Es ist die Bezeichnung für einen allgemeinen Accounting-Manager, der vergleichbar ist mit einer Bank im Sinne des Finanzwesens. Der Dealman verwaltet γ (*Gums*) auf verschiedenen Konten und Unterkonten (*Kontingenten*).

Donation Das Übergeben einer Ressource an eine andere Task b. z. w. einen anderen Thread. Im Sinne des Dealmankonzeptes ist Donation die Überweisung von Gums vom Konto eines Threads auf das Konto eines anderen.

DoS Denial-of-Service

Echtzeit Rechtzeitigkeit

Erwecken Erwecken bedeutet, dass ein *Thread* aus der Blockiertliste in die Bereitliste umgekettet und damit zum entsprechenden Zeitpunkt wieder aktiviert wird. Erwecken ist das Gegenstück zum *Blockieren* .

Gums Gums sind die Währung im Kontext des *Dealman-Konzeptes*. Gums ist ein Kunstwort, das nur im Plural existiert. Es heisst also sowohl 2 Gums wie auch 1 Gums. Das Symbol für Gums ist der griechische Buchstabe γ. Ja, ich hatte bei der Einführung dieser Währung Gummibären vor Augen :-).

IPC Inter-Prozess-Kommunikation

Kontingent Kontingente sind Unterkonten des *Dealman*.

Kritischer Abschnitt Kritische Abschnitte sind Stellen eines Programmes, in denen beliebige Unterbrechungen durch einen Scheduler zu Problemen, wie Dateninkonsistenz, führen können.

L4 L4 ist der Name eines *Mikrokerns* der zweiten Generation. L4 wurde ursprünglich von Jochen Liedtke entwickelt und wird in Form von Michael Hohmuths Fiasco an der TU-Dresden eingesetzt und weiterentwickelt.

L4V2 L4V2 steht für die zweite Version des *L4-Kerns*. Die Spezifikation bildet die Grundlage dieser Arbeit.

Lending Das zeitweise Übergeben einer Ressource an eine andere *Task* b. z. w. einen anderen *Thread*. Im Sinne des Dealmankonzeptes ist Lending eine Überweisung von *Gums* vom Konto eines *Threads* auf das Konto eines andere, die nach einer festgelegten Zeit ungültig wird.

Mikrokern Mikrokerne sind kleine Betriebssystemkerne, die im Gegensatz zu monolithischen Kernen nur grundlegende Dienste wie Adressräume, Aktivitäten und Kommunikation zur Verfügung stellen.

Overhead Mehraufwand

Pager Thread, der für die Behandlung von Pagefaults zustandig ist. Pager implementieren in *L4* virtuellen Speicher auf *Userlandebene* .

Policy Regelwerk

Ressource Betriebsmittel

Ressourcen-Provider Server, die grundlegende *Ressourcen*, wie Speicher, Netzbandbreite, CPU-Zeit, ..., verwalten, b. z. w. zur Verfügung stellen.

Scheduling Planung und Zuteilung von *Ressourcen*. In dieser Arbeit wird der Begriff Scheduling synonym zu CPU-Scheduling verwendet.

Task Tasks sind im Kontext von *L4* Adressräume. Zu jeder Tasks gehören in *L4V2* 128 *Threads*.

Thread Aktivitätsträger

Userland Eingeschränkte ‘Umgebung’, in der Nutzerprogramme laufen

Worst case schlimmstmöglicher Fall

Literaturverzeichnis

[ABLL91] Anderson, Thomas E., Brian N. Bershad, Edward D Lazowska und Henry M Levy: *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* Technischer Bericht, University of Washington, 1991.

[BAG01] Buyya, Rajkumar, David Abramson und Jonathan Giddy: *An Economy Grid Architecture for Service-Oriented Grid Computing.* Technischer Bericht, 2001.

[BSGA01] Buyya, Rajkumar, Heinz Stockinger, Jonathan Giddy und David Abrams: *Economic Models for Resource Management and Scheduling in Grid Computing.* Technischer Bericht, 2001.

[Buy02] Buyya, Rajkumar: *Economic-based Distributed Ressource Management and Scheduling for Grid Computing.* Doktorarbeit, Monash University, 2002.

[Cla05] Clauss, Dietrich: *Investigation of Mechanisms to Support User-Level Thread Packages on Top of the L4-Fiasco Microkernel.* Diplomarbeit, Technische Universität Dresden, 2005.

[Die05] Dietrich, Claus: *Investigation of Mechanisms to Support User-Level Thread Packages on Top of the L4-Fiasco Microkernel.* Diplomarbeit, Technische Universität Dresden, 2005.

[EHY02] Ernemann, Carsten, Volker Hamscher und Ramin Yahyapour: *Economic Scheduling in Grid Computing.* Technischer Bericht, 2002.

[ein] *Das Einstein@home Projekt - http://www.boinc.de/einstein.htm.*

[FH04] Fischer, Prof. Peter und Peter Hofer: *Lexikon der Informatik.* Smart Books, 2004.

[FS96] Ford, B. und S. Susarla: *CPU Inheritance Scheduling.* In: *In Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, 1996.

[gria] *Grid-Computing - http://de.wikipedia.org/wiki/Grid-Computing.*

[grib] *Rechenleistung aus der Steckdose? - http://www.uni-hannover.de/de/forschung/tdf/programm06/rechenleistung-aus-der-steckdose/.*

[gric] *Tag der Forschung - http://www.rrzn.uni-hannover.de/grid_tdf.html.*

[Hoh] Hohmuth, Michael: *The Fiasco kernel: System Architecture.*

[Leb05] Lebelt, Stefan: *Konzepte zur Vermeidung von Denial-of-Service-Angriffen in L4/DROPS.* Technischer Bericht, Technische Universität Dresden, 2005.

[Lie98] Liedtke, Jochen: *Lava Nucleus (LN) Reference Manual - 486 Pentium PentiumPro - Version 2.2*, 1998.

[LL73] Liu, C. L. und James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* J. ACM, 20(1):46–61, 1973.

[MD88a] Miller, Mark S. und K. Eric Drexler: *Comparative Ecology: A Computational Perspective.* The Ecology of Computation, 1988.

[MD88b] Miller, Mark S. und K. Eric Drexler: *Incentive Engineering: for Computational Resource Management.* The Ecology of Computation, 1988.

[MD88c] Miller, Mark S. und K. Eric Drexler: *Markets and Computation: Agoric Open Systems.* The Ecology of Computation, 1988.

[MT86] Mullender, S. J. und A. S. Tanenbaum: *The Design of a Capability-Based Distributed Operating System.* The Computer Journal, 29(4):289–299, 1986.

[sch] *Scheduling - http://de.wikipedia.org/wiki/Scheduling.*

[set] *SETI.Germany - http://www.setigermany.de/.*

[Ste04] Steinberg, Udo: *Quality-Assuring Scheduling in the Fiasco Microkernel.* Diplomarbeit, Technische Universität Dresden, 2004.

[Tan02] Tanenbaum, Andrew S.: *Moderne Betriebssysteme.* Pearson Studium, 2002.

[Völ] Völp, Markus: *L4.Sec Preliminary Microkernel Reference Manual.*

[WW94] Waldspurger, Carl A. und William E. Weil: *Lottery-Scheduling: Flexible Proportional-Share Ressource Management.* Technischer Bericht, MIT Laboratory for Computer Science, 1994.