Diplomarbeit

Effiziente Unterstützung von Multiprozessorsystemen im Fiasco-Mikrokern unter Beachtung des zeitlichen Ausführungsverhaltens

Matthias Lange

26. September 2007

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl. Inf. Michael Peter

Erklärung

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

Dresden, den 26. September 2007
Matthias Lange

Danksagung

Hiermit möchte ich meinem Betreuer Michael Peter für seine Unterstützung und Geduld bei der Anfertigung dieser Arbeit danken. Seine Hilfe bei den zahlreichen Debugging-Sessions und die anregenden Diskussionen sowie seine Kritik haben entscheidend zum Gelingen dieser Arbeit beigetragen.

Danken möchte ich hiermit auch Fabrice Bellard, der mit der Entwicklung von Qemu ein für mich unverzichtbares Testwerkzeug geschaffen hat. An dieser Stelle gebührt Michael Peter zusätzlicher Dank, denn durch dessen Erweiterung von Qemu ist es praktisch zum „Schweizer Taschenmesser“ beim Debugging geworden.

Ebenfalls danken möchte ich Neal Walfield, dessen hartnäckige Fragen geholfen haben, manchen Sachverhalt klarer und deutlicher zu formulieren. Seine Hinweise haben dazu beigetragen, diese Arbeit in den richtigen Kontext zu rücken.

Meinem Vater möchte ich für das Korrekturlesen danken. Seine Anmerkungen haben geholfen, zahlreiche Abschnitte dieser Arbeit verständlicher und klarer zu formulieren.

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren sind Mehrkernprozessor- und Multiprozessorsysteme ein allgemeiner Trend in der Halbleiterindustrie geworden. Es wird erwartet, dass in den nächsten Jahren Prozessoren auf den Markt kommen, die mit Dutzenden oder sogar Hunderten Kernen ausgestattet sind (siehe [ea05]). Der Grund ist, dass Taktratensteigerungen von Prozessoren allein nicht mehr zu den gewünschten Leistungssteigerungen führen. Zusätzlich führen hohe Taktraten zu hohen Verlustleistungen. Multiprozessorsysteme werden in der Regel niedriger getaktet und versuchen Taskparallelität besser auszunutzen. Um die Leistungsfähigkeit dieser Architekturen vollständig zu erschließen, müssen Betriebssysteme und Anwendungsprogramme daran angepasst werden.

Ein Mikrokern ist ein minimaler Betriebssystemkern. Er stellt grundlegende Primitive für die Konstruktion von Betriebssystemen zur Verfügung. Gegenüber monolithischen Systemen bieten Mikrokerne eine zuverlässige Basis für den Aufbau komplexer Systeme, da Komponenten, für die kein Wohlverhalten garantiert werden kann, voneinander isoliert sind. Mikrokerne der ersten Generation litten unter Geschwindigkeitsproblemen, die bei der Entwicklung von Mikrokernen der zweiten Generation gezielt gelöst wurden. L4 ist ein Mikrokern der zweiten Generation und bezeichnet sowohl eine Schnittstelle als auch eine Implementierung. Der an der TU Dresden entwickelte Mikrokern Fiasco ist eine echtzeifähige Implementierung der L4-Schnittstelle. Seine Architektur erlaubt es neben Nicht-Echtzeitaufgaben auch Echtzeitaufgaben zu bearbeiten. Trotzdem können für Echtzeitaufgaben Garantien für die Einhaltung von Bearbeitungsfristen gegeben werden. Echtzeitaufgaben, aber auch allgemeine Arbeitslasten wie z.B. Compiler, können von SMP-Systemen profitieren. Deshalb ist eine Unterstützung von Multiprozessorsystemen im Fiasco-Mikrokern wünschenswert.

Die bisher existierende Unterstützung von Fiasco für Mehrprozessorsysteme stellt hohe Ressourcenanforderungen und leidet unter Leistungsdefiziten bei Anwendungen, die auf mehr als einem Prozessor laufen. Ziel der FiascoMP-Implementierung war es, mit Hilfe eines einfachen Kern-Modells Multiprozessorhardware für Fiasco nutzbar zu machen und die Echtzeiteigenschaften der Uniprozessorvariante zu erhalten. Die aktuelle FiascoMP-Variante hat einige Limitierungen. *Threads* einer Task können nur auf einem Prozessor laufen und Speicher-Mappings nicht über Prozessorgrenzen hinweg vergeben werden. Das Modell hat bei der Implementierung von gemeinsam genutztem Speicher einen hohen Ressourcenbedarf. In dieser Diplomarbeit werde ich Möglichkeiten untersuchen, wie FiascoMP weiterentwickelt werden kann, um derartige Beschränkungen zu beseitigen. Die vorgeschlagenen Lösungen werden bewertet und anhand einer prototypischen Implementierung wird die Eignung zum Erreichen der Ziele untersucht.

Um Ineffizienzen der aktuellen Implementierung zu beseitigen, wird auf Kern-Ebene ein gemeinsam genutzter Adressraum implementiert. *Threads* einer Task können auf un-

terschiedlichen Prozessoren laufen. Der Zugriff auf globale Datenstrukturen muss mit Hilfe eines echtzeitkompatiblen Synchronisationsmechanismus synchronisiert werden. Der *Ex-Regs*-Systemruf muss erweitert werden, damit die Erzeugung von *Threads* auf anderen Prozessoren möglich ist. Seitenfehlernachrichten über Prozessorgrenzen hinweg werden erlaubt. Ein Konsistenzprotokoll stellt die Konsistenz der *Translation Lookaside Buffer* (TLBs) der einzelnen Prozessoren mit den Seitentabellen der Tasks sicher.

Die Mapping-Datenbank ist eine gemeinsam genutzte Ressource und deshalb muss der Zugriff darauf synchronisiert werden. Die Synchronisation hat Auswirkungen auf das Ausführungsverhalten des Gesamtsystems. Die Synchronisation mit Hilfe eines nichtunterbrechbaren Spin-Locks umzusetzen, brachte Probleme mit dem Zeitverhalten für Echtzeitaufgaben. Ein unterbrechbares Spin-Lock würde zwar das Zeitverhalten verbessern, allerdings gibt es mit einem solchen Mechanismus Probleme mit der Fairness, denn *Threads* können verhungern. Deshalb wird ein unterbrechbarer Lock-Mechanismus benötigt, der das Zeitverhalten nicht negativ beeinflusst und der keine Fairness-Probleme hat.

Im bisherigen Fiasco-Kern werden zum Schutz von einigen kritischen Abschnitten *Helping*-Locks (siehe [Hoh02]) verwendet. Bei *Helping*-Locks ist der Lock-Halter unterbrechbar, und unbegrenzte Prioritätsinversion wird vermieden. *Helping*-Locks funktionieren im Cross-Prozessor-Fall nicht mehr, da das direkte Umschalten zum Lock-Halter im Multiprozessorfall nicht mehr möglich ist. Ein potentieller Helfer kann nicht direkt zum Lock-Halter umschalten, da im Multiprozessorfall nicht sichergestellt ist, dass der Lock-Halter nicht gerade selbst aktiv ist. In dieser Arbeit stelle ich einen *Helping*-Mechanismus vor, der auch im Cross-Prozessor-Fall anwendbar ist. Der Lock-Mechanismus ermöglicht die Unterbrechbarkeit des Lock-Halters und vermeidet das *Threads*, beim Versuch auf das Lock zuzugreifen, verhungern können. Unterbrechbarkeit ist notwendig, damit Echtzeitaufgaben erfüllt werden können. Mit dem MP-fähigen *Helping*-Mechanismus lässt sich eine systemglobale Mapping-Datenbank, die Grundlage für einen gemeinsamen Adressraum über Prozessorgrenzen hinweg, implementieren. Anhand einer Referenzimplementierung für den Fiasco-Mikrokern untersuche ich Eigenschaften dieses Locks und vergleiche sie mit denen anderer Lock-Varianten.

1.1 Über diese Arbeit

Diese Diplomarbeit beschäftigt sich mit der Weiterentwicklung der Multiprozessorvariante des Fiasco-Mikrokerns. Ziel ist es, Limitierungen der aktuellen Implementierung zu beseitigen und damit eine effizientere L^4Linux-MP-Implementierung auf dem Multiprozessorkern zu ermöglichen.

Im folgenden Kapitel werde ich zunächst das Konzept von Mikrokernbetriebssystemen vorstellen und anschließend die L4-Schnittstelle erläutern. Die folgenden Abschnitte beschäftigen sich mit verwandten Arbeiten auf dem Gebiet der SMP-Betriebssysteme. Darin wird auch untersucht, inwiefern Echtzeiteigenschaften eine Rolle beim Design dieser Systeme gespielt haben. Am Ende des Kapitels stelle ich die aktuelle FiascoMP-Implementierung vor und ich erläutere für diese Diplomarbeit relevante Eigenschaften

und Limitierungen dieser Implementierung.

Im Analyse-Kapitel stelle ich drei Modelle vor, nach denen FiascoMP weiterentwickelt werden kann. Die Ziele dabei sind, den Ressourcen-Bedarf durch

- Vermeidung der Duplikation der Seitentabellen im Kern und
- eine dedizierte Mapping-Datenbank für jeden Prozessor

zu senken. Die in den drei im Analyse-Kapitel vorgestellten Modellen auftretenden Probleme bei der Erhaltung der Echtzeiteigenschaften von Fiasco und die Verzögerung von prozessorlokalen Operationen sollen vermieden werden. Das in diesem Kapitel vorgestellte Modell 2 ist ein symmetrisches Modell, d.h. alle Operationen sind auf allen Prozessoren möglich. *Thread*-Migration wird nicht durch den Kern unterstützt. Modell 2 implementiert auf Kern-Ebene einen gemeinsam genutzten Adressraum, indem bislang prozessorlokal definierte Datenstrukturen global definiert werden. Die im aktuellen FiascoMP-Modell bislang dediziert für jeden Prozessor vorhandenen Mapping-Datenbanken werden durch eine systemglobale ersetzt.

Im Design-Kapitel beschäftige ich mich mit der Entwicklung eines echtzeitkompatiblen MP-Locking-Mechanismus. Dieser Mechanismus wird benötigt, um Zugriffe auf die systemglobale Mapping-Datenbank zu synchronisieren und dabei vorhersagbares Verhalten von Echtzeit-*Threads* zu ermöglichen.

Das Implementierungs-Kapitel beschreibt einige Aspekte der Implementierung des im Analyse-Kapitel vorgestellten Modells 2. Zunächst erläutere ich, welche Änderungen an der bestehenden FiascoMP-Implementierung notwendig waren. Den Abschluss bildet ein Abschnitt über die Umsetzung des MP-Preemption-Locks.

Im vorletzten Kapitel untersuche ich, wie gut die im Design-Kapitel genannten Ziele erreicht wurden. Es wird gezeigt, dass der Ressourcenverbrauch gegenüber der ursprünglichen Implementierung gesenkt werden konnte. Gleichzeitig wurde die Laufzeit lokaler Operationen erhalten. Weiterhin werde ich anhand unterschiedlicher Messungen und Kriterien zeigen, dass das MP-Preemption-Lock die Echtzeiteigenschaften des Fiasco-Mikrokerns erhält.

Den Abschluss dieser Arbeit bildet eine kurze Zusammenfassung. Anschließend gebe ich einen Ausblick darauf, welche zukünftigen Weiterentwicklung möglich sind.

1.2 Bezeichnungen

Die Informatik ist stark durch englische Begriffe und Bezeichnungen geprägt. Ich habe mich bemüht, soweit wie möglich passende deutsche Entsprechungen dafür zu finden. Für manche Begriffe lassen sich jedoch keine vernünftigen Entsprechungen finden. Um in dieser Diplomarbeit einen schwer verständlichen „Denglisch-Stil“ zu vermeiden, werden solche Begriffe von mir in kursiver Schrift hervorgehoben, zum Beispiel *Idle-Thread*. Quelltexte werden in Schreibmaschinenschrift geschrieben, zum Beispiel `__main`.

2 Grundlagen und verwandte Arbeiten

Im ersten Abschnitt dieses Kapitels stelle ich das Konzept von Mikrokernbetriebssystemen vor und erläutere die Eigenschaften der Fiasco-Schnittstelle. Anschließend stelle ich Arbeiten vor, die sich mit SMP-Betriebssystemen beschäftigen. Im zweiten Teil beschreibe ich für meine Arbeit wichtige Aspekte des aktuellen Fiasco-Multiprozessorkerns. Im letzten Teil stelle ich Mechanismen vor, die für die Konstruktion von SMP-Betriebssystemen wichtig sind.

2.1 L4-Mikrokerne

Ein Mikrokern ist ein minimaler Betriebssystemkern. Er stellt grundlegende Primitive wie z.B. Adressräume, *Threads* und Inter-Prozesskommunikation (im folgenden IPC genannt) zur Verfügung. Mit Hilfe dieser Primitive werden Betriebssystemdienste, zum Beispiel Gerätetreiber und Dateisysteme, außerhalb des Kerns implementiert (siehe [Lie95]).

Im Vergleich zu monolithischen Betriebssystemkernen ergibt sich eine höhere Stabilität gegenüber Fehlfunktionen einzelner Komponenten, da diese bei Mikrokernsystemen voneinander isoliert sind. Systeme mit hohen Sicherheitsanforderungen profitieren von der kleineren Codebasis von Mikrokernen, da diese leichter zu überprüfen ist.

In unterschiedlichen Varianten wurde in der Vergangenheit bereits versucht, die Idee der Mikrokernsysteme in die Praxis umzusetzten. Das bekannstete Projekt dürfte hier das Mach-Projekt der Carnegie Mellon University [BBB+90] sein. Solche Projekte führten zunächst zu keinen Erfolgen, da die Ausführungsgeschwindigkeit dieser Systeme im Vergleich zu klassischen monolithischen Systemen zu gering war [Raw97]. Dafür verantwortlich waren unterschiedliche Designschwächen, hauptsächlich mangelnde IPC-Geschwindigkeit und *Double-Paging*. Derartige Mikrokerne werden als Mikrokerne der ersten Generation bezeichnet.

Jochen Liedtke entwickelte mit L4 einen Mikrokern der zweiten Generation, wobei L4 sowohl die erste Implementierung als auch eine Schnittstellendefinition bezeichnet. Mit den von ihm in [Lie96] beschriebenen Mechanismen (Adressräume, *Threads* und IPC) wurde versucht, die Geschwindigkeitsprobleme zu lösen. Die IPC-Kommunikation wurde grundlegend vereinfacht. Solche Kerne bieten einen sehr effizienten Weg für IPC. Sie verfügen weiterhin nur über drei Abstraktionen (*Threads*, Adressräume und IPC) und versuchen, sämtliche Strategien, zum Beispiel Seitenersetzungsalgorithmus, aus dem Kern zu entfernen. Der Vorteil dieses Designs ist, dass auf Benutzerebene implementierte Strategien von zusätzlichem Wissen über die Ausführungsumgebung profitieren können. Dieses Wissen ist nur schwer in einem Betriebssystemkern zu implementieren.

Fiasco ist ein an der TU Dresden entwickelter Mikrokern, der eine der L4-Schnittstellen implementiert. Bei der Entwicklung von Fiasco wurde viel Wert auf Echtzeitfähigkeiten gelegt, die sich in einem Kern-Design mit nur sehr kurzen, nicht unterbrechbaren Abschnitten und kurzen Interrupt-Latenzen wiederfinden. Der Fiasco-Mikrokern wurde in einer vereinfachten Variante der Programmiersprache C++ geschrieben. Es gibt Varianten für IA-32, AMD64 und ARM.

Weitere L4-Implementierungen sind:

L4ka::Pistachio: Pistachio wird an der Universität Karlsruhe entwickelt und implementiert das L4v4-API für IA-32, IA-64, Alpha, AMD64, ARM, MIPS 64-bit, PowerPC 32-bit und PowerPC 64-bit. Er wird in eingebetteten Systemen, Standardsystemen und Multiprozessorsystemen eingesetzt. Ein wichtiges Ziel bei der Entwicklung von Pistachio ist Portabilität [Pis].

L4/MIPS: Wird an der University of New South Wales entwickelt und läuft auf MIPS-Prozessoren (für Forschung und Lehre, wurde durch L4/Pistachio im Rahmen des L4/NICTA-Projekts abgelöst) (siehe [Hei01]). Die 64-bit-Variante diente als Basis für das SASOS-Projekt (Mungi)

OKL4: Open Kernel Labs (für eingebette Systeme) (siehe [okl]), ist aus L4ka::Pistachio hervorgegangen

Sysgo P4 Sysgo wird für eingebette Systeme in der Automobilindustrie verwendet (siehe [sys]).

L4.sec TU-Dresden, Erweiterung des L4-APIs um Mechanismen für Kommunikationskontrolle und Kern-Ressourcen-Management (siehe [Kau05])

2.1.1 Fiasco

In den folgenden Abschnitten erläutere ich die L4-Schnittstelle.

Adressräume

Adressräume sind Schutzdomänen mit einer eindeutigen Bezeichnung (Adressraum-ID). Sie sind gegenseitig voneinander isoliert. Zugriffsrechte auf Ressourcen werden an Adressräume vergeben. Adressräume können eine bestimmte, feste Anzahl (bei Fiasco maximal 128) an *Threads* beinhalten.

Adressräume werden in L4-Systemen rekursiv aufgebaut. Ausgehend vom Wurzel-Adressraum (σ_0) wird Speicher durch Map-Operationen (*map* und *grant*) weitergegeben bzw. durch die Unmap-Operation (*unmap*) wieder entzogen. Der σ_0-Server bekommt beim Systemstart vom Kern den physischen Speicher zugewiesen. Der Kern kann bestimmte Speicherregionen reservieren, die dann vom σ_0-Server nicht weitergegeben werden. Der physische Speicher wird bei σ_0 gleich 1:1 (d.h. physische Adresse = virtuelle Adresse) eingeblendet.

Threads

Threads sind eine Prozessorabstraktion. Sie haben zwei Funktionen: sie bilden Endpunkte für die Kommunikation und bekommen vom *Scheduler* Rechenzeit für einen Prozessor zugewiesen.

Das Scheduling in Fiasco basiert auf statischen Prioritäten. *Threads* mit höherer Priorität verdrängen *Threads* mit niedrigeren Prioritäten. *Threads* mit gleichen Prioritäten werden nach dem *Round-Robin*-Verfahren eingeplant.

IPC

Zwischen *Threads* können über IPC Nachrichten ausgetauscht werden. Die Kommunikation ist bei L4 synchron und erfolgt über ein sog. Rendezvous von Sender und Empfänger. Nachrichten, die durch IPC übertragen werden, können einzelne Maschinenworte (Short-IPC), Speicher-Mappings und Puffer (Long-IPC) enthalten.

Seitenfehler werden auf IPC abgebildet. Der auslösende *Thread* wird blockiert und der Kern schickt im Namen dieses *Threads* eine Nachricht mit der Seitenfehleradresse an den Pager-Thread. Der *Pager-Thread* antwortet mit der Rücksendung eines Speicher-Mappings. Erst nach Empfang der Antwort kann der *Thread*, der den Seitenfehler ausgelöst hat, weiter laufen. Für den Empfang eines Speicher-Mappings muss der Empfänger ein sog. *Receive-Window* angeben. Das ist ein Bereich des Adressraums, in den das Speicher-Mapping eingefügt werden soll. Damit kann ein Empfänger vermeiden, dass Teile seines Adressraums durch manipulierte Speicher-Mappings verändert werden.

Für einen *Thread* gehört der *Pager-Thread* zur *Trusted Computing Base*, denn bei einem Seitenfehler wird der gesamte Adressraum als *Receive-Window* angegeben. Deshalb muss ein *Thread* seinem *Pager-Thread* vertrauen. Ein böswilliger *Pager-Thread* kann durch manipulierte Speicher-Mappings den Empfänger korrumpieren.

Durch Hardware oder Software ausgelöste Ausnahmen (im folgenden Interrupts) werden ebenfalls auf IPC-Nachrichten abgebildet. Dazu registriert sich ein *Thread* auf Benutzerebene für einen bestimmten Interrupt. Wird dieser Interrupt ausgelöst, so stellt der Kern eine entsprechende Nachricht an den registrierten *Thread* zu.

2.2 Verwandte Arbeiten

In diesem Abschnitt stelle ich andere Arbeiten und Implementierungen zu symmetrischen Multiprozessorsystemen vor. Bei der Analyse dieser Arbeiten zeigt sich, dass verbreitete SMP-Betriebssysteme häufig einen gemeinsam genutzten Adressraum anbieten.

2.2.1 Linux-MP

Linux unterstützt SMP-Systeme auf einer Vielzahl von Plattformen. Eine zentrale Linux-Abstraktion ist der Prozess. Ein Prozess ist ein Programm in Ausführung. Einem Prozess werden durch das Betriebssystem Ressourcen (Betriebsmittel und Rechenzeit) zugewiesen. *Threads* sind in Linux leichtgewichtige Prozesse, die sich Ressourcen mit dem Vater-

Prozess teilen können. Der Kern implementiert für Prozesse einen auf allen Prozessoren gemeinsam nutzbaren Adressraum. Prozesse benötigen kein Wissen über die darunterliegende Architektur, um z.B. von der parallelen Ausführung unterschiedlicher *Threads* zu profitieren. Der Kern organisiert selbstständig eine Lastverteilung. In [BH03] beschreibt Ray Bryant, wie u.a. durch Integration eines O(1)-Schedulers und die Ersetzung des *Big-Kernel*-Locks durch eine feingranularere Variante die Skalierbarkeit des Linux-Kerns auf bis zu 64 Prozessoren verbessert werden konnte. Auf einem Intel-Itanium-System mit 64 Prozessoren konnte damit ein fast linearer Geschwindigkeitszuwachs erzielt werden.

In der Veröffentlichung von Bryant wird gezeigt, dass die globale *Runqueue* im Linux-Kern 2.4 (und früher) einen Flaschenhals darstellt. Deshalb wird in Linux ein *Scheduler* verwendet, der mit prozessorlokalen Warteschlangen arbeitet. Die Lastverteilung wird durch einen periodisch laufenden Prozess realisiert, der lauffähige *Threads* anhand bestimmter Kriterien auf weniger ausgelastete Prozessoren migriert. Diese Kriterien sind nicht trivial und können große Auswirkungen auf die Gesamtgeschwindigkeit haben. So werden z.B. *Threads* mit warmen Caches nicht zur Migration ausgewählt.

Linux-MP ist ein SMP-Betriebssystemkern für eine allgemeine Arbeitslast des Systems. Echtzeitfähigkeiten spielen bei der Entwicklung nur eine untergeordnete Rolle.

2.2.2 RTLinux-MP

RTLinux [YB] eine Erweiterung des Linux-Kerns, die mit dem POSIX 1003.13 Standard kompatibel ist. RTLinux ist ein echtzeitfähiger Betriebssystemkern. Auf diesem Betriebssystemkern wird parallel zu Echtzeitanwendungen ein Standard-Linux-System ausgeführt. RTLinux unterstützt SMP auf PowerPC-Prozessoren. Echtzeit-*Threads* lassen sich bestimmten Prozessoren zuordnen. Ein „Prozessor-Reservierungs-Schema“ erlaubt es, bestimmte Prozessoren eines SMP-Systems für Echtzeitaufgaben zu reservieren. Damit werden Störungen durch die Ausführung von Nicht-Echtzeitanwendungen vermieden. Das Scheduling erfolgt prozessorlokal.

Die Zukunft von RTLinux ist unklar, da momentan keine Weiterentwicklung stattfindet. Dennoch ist das Design des Systems interessant, denn für FiascoMP wird eine ähnliche Arbeitslast, bestehend aus Echtzeitaufgaben und parallel dazu ein L^{4}Linux-MP, angenommen. RTLinux bietet allerdings keinen Adressraumschutz für Echtzeit-Tasks.

2.2.3 K42

K42 [ea06] ist ein Betriebssystem von IBM für Multiprozessorsysteme. Es ist ein Forschungsprojekt mit dem Ziel, sowohl kleine als auch große Multiprozessorsysteme zu unterstützen. Es basiert auf einem Mikrokern und zielt auf Geschwindigkeit und Skalierbarkeit auf bis zu mehreren hundert Prozessoren ab. K42 unterstützt das Linux-API und -ABI, so dass Linux-Anwendungen ohne Anpassung auf diesem Betriebssystem ausgeführt werden können. Echtzeitfähigkeiten waren ursprünglich nicht Bestandteil des Systemdesigns.

2.3 MP-Erweiterungen für die L4-Schnittstelle

In [Völ02] schlägt Völp ein Multiprozessormodel für das L4v2-API vor. Die Kernschnittstelle bleibt dabei mit der Uniprozessorvariante kompatibel, so dass unmodifizierte Anwendungen weiter ausgeführt werden können. Weitere Ziele waren Flexibilität, Geschwindigkeit, Transparenz für unmodifizierte und Nicht-Transparenz für an SMP angepasste Anwendungen.

Der Scheduler arbeitet prozessorlokal und deshalb haben Prioritäten auch nur eine lokale Bedeutung.

Dieser Ansatz geht von einer allgemeinen Arbeitslast des Systems aus.

2.3.1 Skalierbare Mikrokernsysteme

In [Uhl05] beschreibt Uhlig einen komplexen Synchronisationsmechanismus für Mikrokerne, der Geschwindigkeit und Skalierbarkeit des Systems erhält. Dazu schlägt er eine dynamische Anpassung der Synchronisationstrategie und -granularität vor.

Er beschreibt einen Mechanismus zur Anpassung der Synchronisationsmethode zur Laufzeit, der *adaptive locks* genannt wird. Locks können prozessorlokal oder prozessorübergreifend sein. Anwendungen müssen dem Kern durch Hinweise mitteilen, welches Schema benutzt werden soll. Das erfordert, dass die Anwendungen die darunter liegende Architektur kennen müssen, um sie nutzen zu können.

Die Arbeit beschreibt ein allgemeines Modell für eine allgemeine Arbeitslast des Systems. Uhligs Ziel war es, dass alle Anwendungen, wenn auch mit möglichen Geschwindigkeitsverlusten, lauffähig bleiben. Echtzeitfähigkeiten waren nicht Ziel der Arbeit.

2.3.2 Fiasco-SMP

Michael Peter beschreibt in [Pet01] einen ersten Ansatz, den Fiasco-Mikrokern für SMP-Systeme anzupassen. Da der Kern bereits voll unterbrechbar war, wurde es als ausreichend erachtet, die Lock-Infrastruktur für Multiprozessorsysteme anzupassen. Locks haben im SMP-Fall unterschiedliche Aufgaben, die sie im Uniprozessorfall nicht haben. Eine einfache Verallgemeinerung der Lock-Semantik ist daher nicht möglich. Das von Peter entwickelte Design bedingte einen hohen Mehraufwand für lokale Operationen und wurde bei Weiterentwicklungen von Fiasco in den folgenden Jahren nicht weiter angepasst.

2.3.3 Aktuelle FiascoMP-Variante

In [Sch06] stellt Sven Schneider einen alternativen Ansatz vor. Ziel dieser Implementierung war es, ein Modell zu realisieren, welches:

- die Echtzeiteigenschaften des Fiasco-Mikrokerns erhält,
- das genug Funktionalität bietet, um darauf L^{4}Linux auf mehreren Prozessoren auszuführen und

- möglichst wenig Änderungen am Kern verlangt.

Es wurde ein System realisiert, welches auf jedem Prozessor eine Instanz von Fiasco ausführt. Nachrichten können mit Hilfe eines Message-Passing-Systems sehr effizient zwischen einzelnen Instanzen ausgetauscht werden. Andere Interaktionen sind nicht möglich.

Kommunikation

Für die Cross-Prozessor-Kommunikation wird ein *Message-Box-System* verwendet. Es wurde im Hinblick darauf entwickelt, dass eine Vielzahl an Operationen weiterhin prozessorlokal und damit schnell erfolgen können. Daher ist es nicht wünschenswert, alle Operationen, die parallel von Prozessoren durchgeführt werden können, mit einem teuren Synchronisationsmechanismus zu schützen.

Jeder Prozessor darf zu jedem Zeitpunkt nur eine ausstehende Nachricht haben. Während er auf die Bearbeitung seiner verschickten Nachricht wartet, muss er evtl. einkommende Nachrichten bearbeiten. Damit werden Blockierungen vermieden.

Nachrichten haben einen bestimmten Nachrichtentyp, anhand dessen die richtige Funktion für die Bearbeitung der Nachricht ausgewählt wird. *Threads* kommunizieren bei Fiasco mittels Inter-Prozess-Kommunikation (IPC). In der aktuellen Implementierung steht *Threads* für die Kommunikation über Prozessorgrenzen hinweg nur *Short*-IPC zur Verfügung. Der *Long*-IPC-Mechanismus ist prinzipiell möglich, wurde aber bislang nicht implementiert. Speicher-Mappings können nicht über Prozessorgrenzen hinweg verschickt werden.

Mapping-Datenbank

Fiasco unterstützt die Möglichkeit, Adressräume rekursiv aufzubauen (siehe [Lie96]). Die hierarchischen Abhängigkeiten von eingeblendeten Seiten werden in der Mapping-Datenbank gespeichert. Das ist nötig, damit eine Seite später aus jenen Adressräumen, denen die selbe Seite eingeblendet ist, wieder entfernt werden kann.

Jeder Prozessor verwaltet eine eigene Mapping-Datenbank für die bei ihm laufenden Tasks. Per Konstruktion sind keine Cross-Prozessor-Mappings erlaubt. Ein gemeinsam genutzter Adressraum muss daher auf Benutzerebene durch kooperierende Tasks konstruiert werden.

Limitierungen

Das bislang implementierte Modell hat einige Limitierungen:

Tasks: *Threads* einer Task können nur auf einem Prozessor ausgeführt werden. Dieser wird bei der Task-Erzeugung festgelegt und kann später nicht geändert werden.

Mappings: Das Modell erlaubte keine Mappings über Prozessorgrenzen hinweg.

Aus diesen Limitierungen ergibt sich bei der Konstruktion von gemeinsam genutztem Speicher ein hoher Ressourcenbedarf. Auf jedem Prozessor muss eine eigenständige Instanz des Sigma0-Servers, die Zugriff auf den gesamten physischen Speicher hat, laufen. Um gemeinsam genutzten Speicher zu implementieren, wird pro Prozessor eine Task benötigt, welche prozessorlokal Zugriff auf den gemeinsam genutzten Speicher erhält. Die Tasks einer Anwendung müssen daher miteinander kooperieren, um gemeinsam genutzten Speicher zu realisieren.

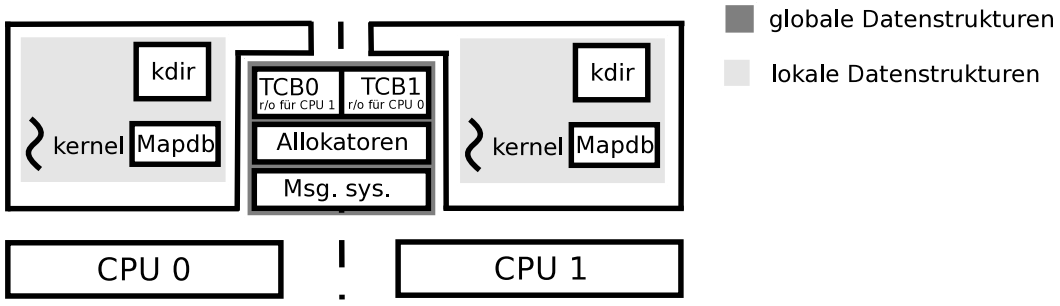

Abbildung 2.1: Schematische Darstellung des Kern-Modells der aktuellen Implementierung

Der Kern bietet keine Unterstützung, um Seiten, die von mehreren Tasks genutzt werden, all diesen Tasks zu entziehen. Eine Implementierung würde zu einer vergleichsweise hohen Komplexität führen, da die Seitentabellen der Tasks, die gemeinsam genutzten Speicher implementieren, als Cache einer übergeordneten Datenstruktur funktionieren. Deshalb müssen beim Entzug einer Seite auf jedem Prozessor die entsprechenden Einträge ungültig gemacht werden. Dafür sind zusätzliche Übergänge von der Kern- zur Benutzerebene nötig.

Das Fehlen von durch den Kern unterstützter Thread-Migration verhindert eine transparente Lastverteilung für Anwendungen.

3 Analyse

Durch die Limitierungen (siehe Abschnitt 2.3.3) der aktuellen Implementierung wurden bislang die folgenden Probleme umgangen:

Mapping-Datenbank: Es gibt keine von mehreren Prozessoren gemeinsam genutzte Mapping-Datenbank. Jeder Prozessor verwaltet seine eigene Mapping-Datenbank. Konkurrierende Zugriffe auf die Mapping-Datenbank sind somit nicht möglich, da per Konstruktion keine Operationen erlaubt sind, die Modifikationen an mehr als einer Mapping-Datenbank erfordern.

TLB-Konsistenz: Der Kern muss nach Seitentabellenänderungen keine Konsistenz der TLBs über Prozessorgrenzen hinweg sicherstellen, da TLBs niemals Einträge von Seitentabellen von Tasks anderer Prozessoren enthalten können.

Migration: Transparente Thread-Migration ist mit dem aktuellen Modell nicht möglich. Es lassen sich auf Benutzerebene nur komplette Tasks transferieren, indem erst der Zustand der Task extrahiert, eine neue Task erzeugt und ihr der Zustand der zu transferierenden Task übertragen wird. Bei diesem Vorgang bleibt die Identität der Task nicht erhalten. Das muss von Anwendungen berücksichtigt werden.

Im folgenden beschreibe ich Weiterentwicklungen, mit denen die oben genannten Beschränkungen beseitigt werden sollen. Ich werde die Vor- und Nachteile diskutieren und sie hinsichtlich ihrer Eignung zum Erreichen der nachfolgend genannten Ziele untersuchen.

Für die Definition der Ziele, die mit der Weiterentwicklung von FiascoMP erreicht werden sollen, gehe ich bei der Arbeitsbelastung des Systems von den folgenden zwei Annahmen aus:

- Anwendungen sind für den lokalen Fall optimiert und
- verwenden gemeinsam genutzten Speicher.

In [Völ02] beschreibt Völp einen Mechanismus, mit dem das Gesamtsystem partitioniert wird (Zuweisung von *Threads* auf Prozessoren). Es gibt einen Rückmeldemechanismus, der angibt, ob bestimmte Operationen eines *Threads* besonders häufig über Prozessorgrenzen hinweg erfolgen. In diesem Fall kann das Partitionierungsschema angepasst werden. Es wird zwischen dynamischer und statischer Partitionierung unterschieden. Die dynamische Partitionierung ermöglicht eine bessere Anpassung an sich ändernde Lastsituationen des Systems und ist für allgemeine Arbeitslasten daher besser geeignet.

Auf Lokalität optimierte Anwendungen sollen nicht durch einen langsameren Kernmechanismus für lokale Operationen verlangsamt werden. Mit der Weiterentwicklung sollen außerdem die in Abschnitt 2.3.3 genannten Limitierungen der bisherigen FiascoMP-Implementierung beseitigt werden.

Ziele für die Weiterentwicklung sind daher:

- Verringerung des Ressourcen-Bedarfs durch:
  - Vermeidung der Duplikation der Seitentabellen im Kern,
  - Beseitigung der dedizierten Mapping-Datenbank für jeden Prozessor und
  - Verringerung des Mehraufwands an Laufzeit
- Verzögern prozessorlokaler Operationen durch Cross-Prozessor-Synchronisation vermeiden und
- Erhaltung der Echtzeiteigenschaften von Fiasco.

Als Anwendungsszenario für das System nehme ich ein Multiserver-Betriebssystem und eine hypothetische Mehrprozessorvariante von L^{4}Linux an.

In den folgenden Abschnitten werde ich einige Begriffe verwenden, die hier zur besseren Verständlichkeit zunächst definiert werden.

Heimat-Prozessor: Dieser Begriff bezieht sich auf eine Task und benennt den Prozessor, auf dem ihre Seitentabelle verwaltet wird. Üblicherweise ist das der Prozessor, auf dem die Task erzeugt worden ist.

Heimat-Thread: Ein Heimat-*Thread* ist ein Thread, der auf dem Heimat-Prozessor seiner Task ausgeführt wird.

Fern-Thread: Ein Fern-*Thread* wird auf einem anderen Prozessor als dem Heimat-Prozessor seiner Task ausgeführt.

In den folgenden Abschnitten werde ich drei Modelle vorstellen, nach denen der FiascoMP-Kern weiterentwickelt werden kann. Das erste Modell, Modell 1, ermöglicht es, einzelne *Threads* einer Task auf unterschiedlichen Prozessoren laufen zu lassen. Im Modell 2 implementiert der Kern einen gemeinsam genutzten Adressraum. Das Modell 3 bietet durch den Kern unterstützte *Thread*-Migration. Um die Unterscheidung der einzelnen Modelle zu erleichtern, werde ich im folgenden das bisherige Modell *Modell A* nennen. Danach folgt die Bewertung aller Modelle und die Implementierung eines Modells.

3.1 Modell 1 - Cross-Prozessor-Tasks

Im bisherigen FiascoMP-Modell sind Tasks und deren *Threads* auf einen Prozessor beschränkt. Dadurch kann eine Anwendung den Geschwindigkeitsgewinn durch die parallele Ausführung von unabhängigen *Threads* nicht für sich ausnutzen, ohne den Mehraufwand für die Erzeugung mehrerer Tasks aufzubringen.

Das Modell 1, welches ich in den folgenden Abschnitten vorstelle, ermöglicht es, einzelne *Threads* einer Task auch auf anderen Prozessoren auszuführen. Damit können Adressräume über Prozessorgrenzen hinweg genutzt werden. Es wird erwartet, dass die Anzahl der für eine bestimmte Last benötigten Tasks reduziert werden kann.

3.1.1 Scheduling

Wie von Marcus Völp in [Völ02] beschrieben, sind die Laufzeitkosten für die Synchronisation einer globalen Ready-Liste sehr hoch. Der aktuelle Fiasco-MP-Kern verwendet prozessorlokales Scheduling. Prioritäten haben deshalb nur prozessorlokal eine Bedeutung, da *Threads* unterschiedlicher Prioritäten auf unterschiedlichen Prozessoren parallel laufen können. Die Strategie, Scheduling-Entscheidungen prozessorlokal zu treffen, wird, wie aus den im Kapitel 2 vorgestellten Arbeiten hervorgeht, häufig verwendet.

Um diese Laufzeitkosten zu vermeiden, wird das prozessorlokale Scheduling weiterhin bevorzugt. Fern-*Threads* werden den Scheduling-Entscheidungen des Prozessors, auf dem sie laufen, unterworfen.

3.1.2 Speicherverwaltung

Der *Translation Lookaside Buffer* (TLB) ist ein Cache der übergeordneten Seitentabellen und speichert Adressübersetzungen. Mit der Nutzung von Adressräumen auf mehreren Prozessoren müssen TLBs unterschiedlicher Prozessoren mit der Seitentabelle der Task konsistent gehalten werden. Wird ein Eintrag aus der Seitentabelle gelöscht oder wird einer Task ein Mapping entzogen, dann müssen zugehörige TLB-Einträge in den TLBs aller betroffenen Prozessoren ungültig gemacht werden.

Die Uni-Prozessor-Implementierung der Unmap-Operation beinhaltet die TLB-Inva lidierung. Im Remote-Fall funktioniert dieser Mechanismus nicht mehr, da die Unmap-Operation nur auf einem Prozessor durchgeführt wird. Deshalb wird ein Konsistenzprotokoll für die TLBs aller Prozessoren benötigt. Die folgenden Abschnitte stellen zwei unterschiedliche Strategien vor, nach denen die TLBs aller Prozessoren konsistent gehalten werden können.

Eifrige TLB-Löschung

Bei diesem Ansatz werden die TLBs aller Prozessoren bei jeder Unmap-Operation gelöscht. Dazu wird eine Nachricht an alle Prozessoren verschickt, die daraufhin ihre TLBs löschen. Einen weiteren Ansatz, der die Nutzung der TLBs mit berücksichtigt, stelle ich im folgenden Abschnitt vor.

Dieser Ansatz hat große Auswirkungen auf die Gesamtgeschwindigkeit, da potentiell auch TLBs von Prozessoren gelöscht werden, auf denen keine Fern-*Threads* laufen. Der Einfluss auf die Gesamtgeschwindigkeit muss experimentell überprüft werden.

Selektive TLB-Löschung

Das Wissen darüber, welche Tasks von Änderungen eines Mappings betroffen sind, steckt in den Mapping-Bäumen selbst. Mit Hilfe dieses Wissens lässt sich der TLB gezielter löschen.

Zunächst wird an jeden Prozessor wiederum eine Nachricht zum Löschen des TLBs geschickt. Jeder Prozessor überprüft vor dem Löschen, ob gerade ein *Thread* einer betroffenen Task läuft und löscht ggf. den TLB.

Mit Hilfe einer globalen Kern-Datenstruktur könnte ermittelt werden, auf welchem Prozessor welche *Threads* laufen. Damit kann entschieden werden, welche Prozessoren von einer Unmap-Operation betroffen sind. Die Nachrichten brauchen nur noch an diese Prozessoren geschickt zu werden. So werden die Auswirkungen auf die betroffenen Prozessoren begrenzt.

In [Uhl05] wird mit der *Processor Cluster Mask* ein Verfahren beschrieben, mit dem sich die Zuordnung von Ressourcen zu Prozessoren effizient verwalten lässt. Dieses Verfahren kann auch hier verwendet werden.

3.1.3 Prozessor-ID

Die *Kernel-Info-Page* (KIP) ist in L4 eine Datenstruktur, die Informationen über den Speicher, den Prozessor und die Kern-Schnittstelle enthält. Im bisherigen Modell konnte ein *Thread* sich die KIP einblenden lassen und daraus die Prozessor-ID auslesen. Fern-*Threads* haben in diesem Fall jedoch die KIP des Heimat-Prozessors eingeblendet und können daher nicht die korrekte Prozessor-ID ermitteln.

Als mögliche Lösung schlage ich einen neuen Systemruf vor, der die Prozessor-ID aus dem *Thread Control Block* (TCB) des jeweiligen *Threads* ausliest. Ebenfalls möglich ist die Erweiterung des `l4_myself`-Systemrufs mit dem z.B. die *Thread*-ID ermittelt werden kann. Unterstützt der Kern *Userlevel Thread Control Blocks* (UTCBs), so kann man die Prozessor-ID in einem Feld des UTCBs speichern und der *Thread* kann sie dort auslesen.

3.1.4 Kommunikation

Für Fern-*Threads* werden nur Short- und Long-IPC erlaubt. Fern-*Threads* können keine Speicher-Mappings annehmen oder verschicken, da jeder Prozessor weiterhin seine eigene Mapping-Datenbank für seine Tasks verwaltet. Die Mapping-Datenbanken unterschiedlicher Prozessoren werden nicht synchronisiert.

3.2 Modell 2 - Gemeinsam genutzte Mapping-Datenbank

In dem hier vorgestellten Modell (im Folgenden Modell 2) werden die bisher für jeden Prozessor dedizierten Mapping-Datenbanken durch eine einzige globale Mapping-Datenbank ersetzt. Dafür muss festgelegt werden, auf welchen Prozessoren Mappings

entgegengenommen werden dürfen bzw. wie der Zugriff auf die Mapping-Datenbank synchronisiert wird.

Es entfallen die begrenzenden Einschränkungen:

- Ressourcen-Vervielfältigung (Seitentabellen und Mapping-Datenbank) und
- keine Mappings über Prozessorgrenzen hinweg.

Für die Benutzerebene wird im Modell 2 durch den Kern ein gemeinsamer Adressraum auf allen Prozessoren zur Verfügung gestellt. Damit wird der Aufwand (Speicher und Laufzeit) für Anwendungen mit Cross-Prozessor-Adressräumen reduziert.

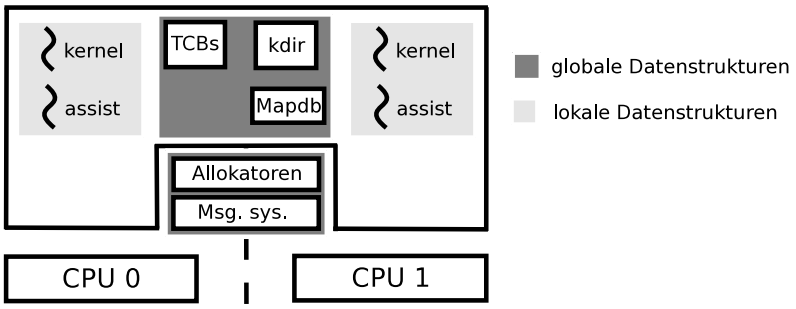

Abbildung 3.1: Schematische Darstellung des Kern-Modells von Modell 2

3.2.1 Speicherverwaltung

Zugriffe auf gemeinsam genutzte Daten des Kerns, z.B. die Mapping-Datenbank, müssen synchronisiert werden, da der Zugriff parallel auf unterschiedlichen Prozessoren erfolgen kann.

Seitentabellen

Änderungen an Seitentabellen müssen sychronisiert werden. Dazu kann entweder die gesamte Seitentabelle durch ein Lock oder durch eine feiner granulare Variante geschützt werden. Bei der feingranularen Lösung wird zunächst die erste Ebene (Seitentabellenverzeichnis) durch ein Lock geschützt. Ist die entsprechende Seitentabelle (zweite Ebene) gefunden, wird diese ebenfalls gelockt. Anschließend kann das Lock des Seitentabellenverzeichnisses wieder freigegeben werden. Dieses Locking-Schema erhöht die Skalierbarkeit gegenüber der grobgranularen Lösung. Der Nachteil ist, dass diese Lösung aufwändiger zu implementieren ist.

Allokatoren

Tabelle 3.1: Allokatoren in Fiasco und deren Verwendung

Allokator	**Verwendung**
slab_cache_anon	einfacher Slab-Allokator
Kmem_slab_simple	leitet von slab_cache_anon ab, Allokation von Speicher in Seitengröße
Kmem_slab	leitet von Kmem_slab_simple ab, Allokation von Speicher größer als eine Seite, ein Allokator für jede Mapping-Baum-Größe, Taskerzeugung, Erzeugung von Scheduling-Kontexten, Initialisierung des Region-Managers, Allokation für FPU-Zustand
Mapped_allocator	Kmem_slab::block_alloc und Kmem_slab_simple::block_alloc für Speicher kleiner als eine Seite, Physframe::alloc, Vmem_alloc::init (für Zero-Page und `page_alloc`)
Kmem_alloc	leitet von Mapped_allocator ab, Allokator für Kern-Speicher, Seitentabellenallokation in Space::v_insert, JDB
List_alloc	Kmem_alloc::alloc
Vmem_alloc	Initialisierung von UTCBs und der IDT, Kmem_slab::block_alloc für Speicher größer als eine Seite, Initialisierung einer neuen Task, Initialisierung des Kern-Threads

Die Allokatoren reservieren Speicher aus einem globalen Pool. Aus diesem Grund müssen Allokatoren, die parallel auf unterschiedlichen Prozessoren verwendet werden können, durch Locks vor wechselseitigem Zugriff geschützt werden. Im Fiasco-Kern werden die in Tabelle 3.1 beschriebenen Allokatoren verwendet. Aus dieser Tabelle geht hervor, dass alle Allokatoren parallel verwendet werden können, da sie z.B. für die Task-Erzeugung und Allokation von Speicher für Kern-Objekte benutzt werden. Deshalb müssen sie synchronisiert werden. In Abbildung 3.2 sind die Abhängigkeiten der in Fiasco verwendeten Allokatoren dargestellt.

Mapping-Datenbank

Mit der Implementierung eines gemeinsam genutzten Adressraums durch den Kern wird nur noch eine systemglobale Mapping-Datenbank benötigt. Da Mappings parallel auf unterschiedlichen Prozessoren etabliert werden können, muss die Mapping-Datenbank vor wechselseitigem Zugriff geschützt werden. Mapping-Hierarchien werden in der Mapping-Datenbank in Mapping-Bäumen gespeichert. In der aktuellen Implementierung werden zum Schutz der Mapping-Bäume Helping-Locks verwendet. Diese Helping-Locks können nicht mehr verwendet werden, da die implizite Annahme, dass bei Lock-*Contention* zum Lock-Halter umgeschaltet werden kann, nicht mehr für MP-Systeme gilt. Da Map- und Unmap-Operationen relativ lange Laufzeiten haben können, wird ein neuer, unterbrechbarer Locking-Mechanismus benötigt, um den Einfluss auf die Latenz von Echtzeitaufgaben zu minimieren.

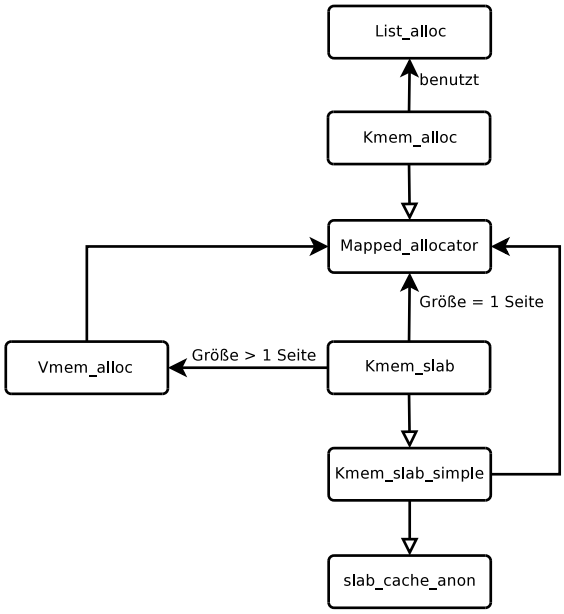

Abbildung 3.2: Abhängigkeiten der in FIASCO verwendeten Allokatoren

Für die Allokation von Speicher für die Mapping-Bäume wird für jede Baumgröße ein spezifischer Slab-Allokator verwendet. Diese Allokatoren sind in einem Feld gespeichert.

3.3 Modell 3 - Thread-Migration

Im Modell A gibt es keine Möglichkeit, einzelne *Threads* für eine bessere Lastverteilung auf andere Prozessoren zu migrieren. Darin lassen sich nur komplette Tasks transferieren, wobei aber die Identität dieser Task nicht erhalten bleibt. Es gibt keine Unterstützung durch einen Kern-Mechanismus. Mit dem im Abschnitt 3.2 vorgestellten Modell 2 ist Thread-Migration von der Nutzerebene aus möglich. Allerdings ist dieser Vorgang aufgrund der zwischenzeitlichen Nicht-Existenz dieses *Threads* für die Außenwelt sichtbar, denn der Migrationsvorgang ist nicht atomar und kann daher unterbrochen werden.

Die Task-Migration im Modell A setzt sich aus insgesamt fünf Schritten zusammen und wird vollständig auf der Benutzerebene ausgeführt:

Einfrieren: Alle *Threads* der zu migrierenden Task werden angehalten und der Zustand wird, damit er sich nicht mehr ändert, in einem Schnappschuss festgehalten.

Extrahieren: Der die Migration durchführende *Thread* extrahiert den Zustand jedes *Threads* der zu migrierenden Task.

Transferieren: Auf dem Ziel-Prozessor wird eine neue Task mit der entsprechend benötigten Anzahl an *Threads* angelegt. Die *Threads* sind noch nicht lauffähig, da ihnen noch nicht der Zustand der zu transferierenden *Threads* übertragen wurde.

Injizieren: Der extrahierte Zustand jedes *Threads* wird auf einen der neu erzeugten *Threads* übertragen.

Auftauen: Die *Threads* der migrierten Task werden auf dem Ziel-Prozessor aufgeweckt und können dann dort ausgeführt werden. Anschließend müssen mögliche Kommunikationspartner benachrichtigt werden, da sich die Identität der *Threads* geändert hat.

Die *Thread*-Migration im Modell 2 läuft ähnlich ab.

Dieser komplexe Vorgang eignet sich nicht für dynamische Lastverteilung zwischen einzelnen Prozessoren. Diese Art der Migration ist nicht identitätserhaltend und damit für Kommunikationspartner sichtbar. Außerdem werden kooperierende Tasks auf unterschiedlichen Prozessoren benötigt, die die Migration durchführen.

Um diese Mängel zu beseitigen, stelle ich im nächsten Abschnitt ein weiteres Modell vor.

3.3.1 Migration

Für *Thread*-Migration wird ein Kern-Mechanismus benötigt, der transparent bezüglich der Interaktion (IPC) zwischen *Threads* ist. Es ergeben sich Probleme mit: zwischenzeitlicher Migration, nicht abgeschlossenen Nachrichten und dem Locking von *Threads*.

Um Verklemmungssituationen zu vermeiden, muss verhindert werden, dass sich zwei *Threads* gegenseitig migrieren wollen. Deshalb dürfen *Threads* nur von einem dedizierten *Thread* migriert werden. Der Migrations-*Thread* übernimmt in diesem System die Lastverteilung. Der Migrations-*Thread* kann ein zusätzlicher Kern-*Thread* oder ein Server-*Thread* auf Benutzerebene sein.

3.3.2 Kommunikation

Threads müssen um eine Warteschlange für ausstehende Operationen und Nachrichten erweitert werden. Das ist nötig, da während der Migration Nachrichten an den zu migrierenden *Thread* eintreffen können. Damit keine Nachrichten verloren gehen, muss der Kern solche Nachrichten in einem Puffer zwischenspeichern und nach der Migration zustellen. Wird ein *Thread* gerade migriert, so muss sichergestellt werden, dass die Bearbeitung jeder Nachricht nicht beliebig verzögert wird.

3.4 Diskussion

In diesem Abschnitt fasse ich zunächst noch einmal die Eigenschaften der drei von mir vorgestellten Modelle zusammen. Anschließend diskutiere ich Vor- und Nachteile der jeweiligen Modelle.

3.4.1 Modelleigenschaften

Das im Abschnitt 3.1 vorgestellte Modell 1 ermöglicht die Ausführung von *Threads* einer Task auf unterschiedlichen Prozessoren. Die sog. Fern-*Threads* dürfen Seitenfehler auslösen und *Short*- und *Long*-IPC-Nachrichten verschicken. Das Verschicken von Speicher-Mappings von und an Fern-*Threads* ist nicht erlaubt.

Im Modell 2 werden die bislang dediziert für jeden Prozessor vorhandenen Mapping-Datenbanken durch eine systemglobale ersetzt. Die Duplikation der Seitentabellen entfällt, da vom Kern ein gemeinsamer Adressraum auf allen Prozessoren zur Verfügung gestellt wird.

Im Modell 3 wird ein für Anwendungen teilweise transparentes Multiprozessorsystem geschaffen. Multi-Thread-Anwendungen ohne Wissen über die Multiprozessorhardware können aufgrund der Unterstützung durch den Kern trotzdem den Vorteil aus paralleler Ausführung einzelner *Threads* ziehen, da der Kern oder ein Server auf Benutzerebene *Threads* dynamisch auf unterschiedliche Prozessoren verteilen kann. Die Management-Komponenten müssen jedoch die Zuordnung von *Threads* zu Prozessoren kennen. Deshalb kann man hier nur von einem teilweise transparenten Modell sprechen.

3.4.2 Vor- und Nachteile

Aufgrund der Beschränkung von Fern-*Threads* auf *Short*-IPC im Modell 1 sind keine Speicher-*Mappings* über Prozessorgrenzen hinweg möglich. Die Anordnung von *Threads* auf den Prozessoren kann sich kritisch auf die Gesamtgeschwindigkeit des Systems auswirken, da bei ungünstiger Anordnung zahlreiche Nachrichten an den Heimat-Prozessor notwendig werden können. Der Heimat-Prozessor stellt einen potentiellen Flaschenhals dar, da er den gesamten Verwaltungsaufwand für die Seitentabellen und die Mapping-Datenbank seiner Tasks zu tragen hat.

Die im Abschnitt 3 genannten Ziele lassen sich mit dem Modell 1 nicht erreichen. Jedoch werden wichtige Grundlagen (z.B. TLB-Invalidierungen und Cross-Prozessor-Tasks) gelegt, die für das Modell 2 benötigt werden.

Im Modell 2 wird der Ressourcen-Verbrauch (Speicher) des Kerns gesenkt. Es wird erwartet, dass die für eine bestimmte Last benötigte Anzahl von Tasks reduziert wird, da z.B. der Kern bereits einen gemeinsam genutzten Adressraum implementiert. Außerdem entfällt der Laufzeitmehraufwand für die Erzeugung mehrerer Tasks, um gemeinsam genutzten Speicher auf Nutzerebene zu konstruieren.

Die Vorteile (geringerer Ressourcenverbrauch und weniger Laufzeitaufwand) werden durch ein komplexeres Kern-Modell erzielt. Um Fortschritt von *Threads* bei langen kritischen Abschnitten auch über Prozessorgrenzen hinweg erzielen zu können, wird ein komplexer Synchronisationsmechanismus für die Mapping-Datenbank benötigt. Ein einfaches Spin-Lock zum Schutz der Mapping-Datenbank erfüllt zwar die Fairness-Eigenschaft, da der kritische Abschnitt aber nicht unterbrechbar ist, hat es negative Auswirkungen auf die Echtzeiteigenschaften. Das lässt sich mit einem unterbrechbaren Spin-Lock verhindern. Ein unterbrechbares Spin-Lock kann jedoch zu unbegrenzter Verzögerung von Operationen einzelner *Threads* führen. Deshalb wird ein unterbrechbarer und fairer

Lock-Mechanismus benötigt, der das Verhungern von *Threads* verhindert.

Das Kern-Modell von Modell 3 ist das komplexeste der drei vorgestellten Modelle. Die Auswirkungen davon auf das Echtzeitverhalten von Fiasco sind unklar. Es ist am besten für eine allgemeine Arbeitslast eines MP-Systems geeignet, aber für das von mir angenommene Anwendungsszenario (L^{4}Linux) nicht nötig, da Linux selbst eine Lastverteilung implementiert.

3.4.3 Schlussfolgerung

Aus der Diskussion in Abschnitt 3.4 geht hervor, dass das in Abschnitt 3.1 vorgestellte Modell 1 mit Cross-Prozessor-Tasks bereits die für eine bestimmte Arbeitslast benötigte Taskmenge reduzieren kann. Dieses System ist dafür geeignet, mehrere L^{4}Linux-Uniprozessorinstanzen parallel auf unterschiedlichen Prozessoren auszuführen. Für die Ausführung einer hypothetischen L^{4}Linux-MP-Variante ist es weniger geeignet, da nach wie vor ein hoher Ressourcenverbrauch durch die duplizierten Kern-Datenstrukturen benötigt wird.

Der hohe Ressourcenverbrauch wird im zweiten Modell beseitigt. Durch die Implementierung eines gemeinsam genutzten Adressraums im Kern entfällt die Duplikation von Kern-Datenstrukturen. Das Modell 2 vereinfacht die Konstruktion eines L^{4}Linux-MP-Systems, da Linux-SMP selbst auch einen gemeinsam genutzten Adressraum für Anwendungsprogramme implementiert.

Das Modell 3 mit durch den Kern unterstützter *Thread*-Migration bietet für die Umsetzung von L^{4}Linux-MP keine zusätzlichen Vorteile. Linux implementiert selbst weitere SMP-Mechanismen wie z.B. Lastverteilung. Daher wird die Funktionalität von Modell 3 für L^{4}Linux-MP nicht benötigt.

Ich werde deshalb im Rahmen dieser Arbeit das Modell 2 mit einem gemeinsam genutzten Adressraum implementieren. Es bietet die Eigenschaften, die zum Erreichen der im Abschnitt 3 genannten Ziele

- Verringerung des Ressourcen-Bedarfs,
- Verhinderung von langsamen prozessorlokalen Operationen durch Cross-Prozessor-Synchronisation und
- Erhaltung der Echtzeiteigenschaften von Fiasco

nötig sind. Es ist für die angenommene Arbeitslast geeignet.

4 Design

In diesem Kapitel stelle ich den Entwurf für die Weiterentwicklungen von FiascoMP vor. Im vorangegangenen Kapitel habe ich mich für die Implementierung des Modells 2 entschieden. Der zentrale Punkt dieses Modells ist die system-globale Mapping-Datenbank. Um bei konkurrierenden Zugriffen von unterschiedlichen Prozessoren auf die Mapping-Datenbank die Echtzeitfähigkeiten zu erhalten, wird ein unterbrechbarer Locking-Mechanismus über Prozessorgrenzen hinweg benötigt. Deshalb bildet der Algorithmus für das Cross-Prozessor-Helping den Schwerpunkt dieses Kapitels.

4.1 Thread-Interaktion

4.1.1 IPC

Der Cross-Prozessor-IPC-Pfad ist mit dem lokalen IPC-Pfad integriert. Damit ist sichergestellt, dass der atomare Übergang zwischen dem Sende- und dem Empfangsteil einer Nachricht auch im Cross-Prozessorfall gewährleistet ist.

Um Adressräume über Prozessorgrenzen hinweg zu nutzen, wird *Pagefault*-IPC über Prozessorgrenzen erlaubt.

Um die Komplexität des Kerns zu begrenzen, ist es ein akzeptabler Kompromiss, dass der Cross-Prozessor-IPC-Pfad nur Register-Short-IPC implementiert. Der Einfluss von Cross-Prozessor-IPC auf die lokale IPC soll so klein wie möglich sein.

4.1.2 Remote-Ex-Regs

Über den Ex-Regs-Systemruf kann durch *Threads* der Zustand von anderen *Threads* verändert werden. Dazu wird das *Thread*-Lock des betroffenen *Threads* gegriffen und anschließend wird der *Thread* als nicht lauffähig markiert. Im Multiprozessorfall kann es zu einem Deadlock kommen, wenn zwei *Threads* auf unterschiedlichen Prozessoren sich gegenseitig mit dem Ex-Regs-Systemruf manipulieren.

Mit Hilfe des Message-Box-Systems und dessen Ausführung im Kontext eines zusätzlichen Kern-*Threads* wird eine Blockierung vermieden. Die Bearbeitung des Ex-Regs-Aufrufs erfolgt im Kontext dieses Kern-*Threads*.

4.1.3 Nachrichtenrate

Die Rate, mit der ein Prozessor Nachrichten an andere Prozessoren schickt, muss begrenzt werden. Damit wird sichergestellt, dass auf dem Empfänger-Prozessor in den Sendepausen *Threads* laufen können.

Die Rate, mit der Nachrichten verschickt werden, ist ein anpassbares Kriterium und hängt stark von der Arbeitslast des Systems ab. Es ist sinnvoll, diese empirisch durch Messung zu ermitteln.

4.2 Synchronisation der Mapping-Datenbank

In diesem Abschnitt beschreibe ich, wie im Modell 2 Zugriffe auf die Mapping-Datenbank synchronisiert werden, damit die Auswirkungen auf das Zeitverhalten begrenzt werden können. In vielen Modellen werden Ressourcen an Prozessoren gebunden [Liu00]. Zusätzlich wird die Annahme gemacht, dass die Ressourcennutzungszeit begrenzt ist. Beide Annahmen sind schlecht mit den Charakteristiken der Mapping-Datenbank vereinbar. Mapping-Operationen können Echtzeit-Operationen zwar nicht behindern, können aber nicht Bestandteil von Echtzeitaufgaben sein, da die maximale Ausführungszeit nicht begrenzt ist (abhängig von der Mapping-Baum-Größe). Die Mapping-Datenbank an einen Prozessor zu binden, würde diesen Prozessor potentiell zu einem Flaschenhals machen.

Jeder einzelne Prozessor kann zu jedem Zeitpunkt nur einen Ausführungspfad bearbeiten. Durch Zeitmultiplex mehrerer Codepfade auf einem Prozessor wird innerhalb eines bestimmten Zeitrahmens das Weiterlaufen des Gesamtsystems erreicht. Bestimmte Codepfade dürfen nicht parallel laufen (kritische Abschnitte), da sie auf gemeinsame Daten zugreifen. Solche Abschnitte müssen mit Mechanismen geschützt werden, die eine exklusive Abarbeitung der kritischen Abschnitte sicherstellen. Eine Möglichkeit dafür sind Locks.

Für die Nutzung der durch ein Lock geschützten Ressource muss zunächst das Lock gesetzt werden. Ist das Lock bereits belegt, blockieren *Threads* beim Zugriff darauf. Durch diese Konstruktion kann es im System zum Problem der Prioritätsumkehr kommen, was den Fortschritt von lauffähigen *Threads* verhindert. Fiasco vermeidet Prioritätsumkehr durch Helping-Locks.

Ein an einem besetzten Lock blockierter *Thread* kann, um seine eigene Blockierzeit zu minimieren, zu dem Lock-Halter umschalten und so die Ausführung des kritischen Abschnitts fortsetzen. Damit wird erreicht, dass der Lock-Halter das Lock möglichst schnell wieder freigibt. Auf Einprozessorsystemen ist das problemlos möglich, da die unterschiedlichen Codepfade durch Zeitmultiplex serialisiert werden. Bei MP-Systemen treten dabei folgende Probleme auf:

- Ausführung des kritischen Abschnitts zu jedem Zeitpunkt nur auf einem Prozessor sicherstellen und
- mehrere gleichzeitig zum Helfen bereite Helfer.

Diese Probleme werden durch Synchronisation zwischen Besitzern und Helfern sowie zwischen Helfern untereinander gelöst.

Für die Synchronisation sind unterschiedliche Implementierungen möglich. Das angestrebte Ziel ist es, die bestehenden Codepfade möglichst wenig zu verändern. Die

Synchronisation zwischen Besitzer und dem ersten Helfer erfolgt durch die Unterbrechung des Besitzers durch den ersten Helfer mit Hilfe eines *Disable-Requests*. Solange es lauffähige Helfer gibt, führt der Besitzer den kritischen Abschnitt nicht auf seiner eigenen Zeitscheibe aus: der Helfer führt den kritischen Abschnitt auf seiner Zeitscheibe aus. Die Synchronisation zwischen einzelnen Helfern erfolgt über eine lockspezifische Helferliste, in die sich jeder Helfer einträgt. Mit Hilfe dieser Liste wird das Helfen serialisiert. Threads, die sich in dieser Liste befinden, werden aus der *Ready*-Liste ausgetragen. Damit können sie nicht vom *Scheduler* auf ihrem Prozessor ausgewählt werden. Helfer warten auf die Benachrichtigung, mit dem Helfen beginnen zu können. Nach dem Eintreffen einer Benachrichtigung, mit dem Helfen zu beginnen, wird die Ausführung fortgesetzt.

Eine andere Möglichkeit der Synchronisation wäre, dass potentielle Helfer permanent den Zustand des Besitzers überprüfen (Polling) und bei Unterbrechung desselben mit dem Helfen beginnen. Während des ständigen Abfragens verstreicht die Prozessorzeit ungenutzt. Aufgrund der zusätzlichen Instruktionen und der aufwändigen atomaren Operationen wird der *switch_to*-Pfad langsam.

4.2.1 Algorithmus

Bei der Lockanforderung wird überprüft, ob das Lock bereits besetzt ist. Ist das Lock bereits besetzt, wird der *Thread* in die Anwärterliste und die Helferliste eingetragen. Ist der Helfer der erste Helfer, d.h. es gibt keine weiteren lauffähigen Helfer und der Besitzer führt den kritischen Abschnitt auf seiner eigenen Zeitscheibe aus, unterbricht der Helfer den Besitzer. Anschließend führt der Helfer den Besitzer auf seiner Zeitscheibe aus.

Wird ein Helfer bei der Ausführung des kritischen Abschnitts unterbrochen, wird überprüft, ob weitere Helfer in der Helferliste eingetragen sind. Ist dies der Fall, müssen zwei Fälle unterschieden werden.

1. Der nächste Helfer führt auf demselben Prozessor wie der unterbrochene Helfer aus. Der unterbrochene Helfer wird in die lokale Ready-Liste eingetragen. Ein direktes Umschalten ist nicht möglich, da so das lokale Scheduling ignoriert werden würde. Alle nachfolgenden Helfer, die lokal sind, werden ebenfalls in die Ready-Liste eingefügt, da nicht bekannt ist, ob sie lokal die höchste Priorität haben und damit lauffähig sind.

2. Der nächste Helfer führt auf einem anderen Prozessor als der unterbrochene Helfer aus. Der neue Helfer wird benachrichtigt, dass er mit dem Helfen beginnen kann.

Erhält ein potentieller Helfer die Nachricht, mit dem Helfen beginnen zu können, muss überprüft werden, ob der Helfer überhaupt noch lauffähig ist. Während seiner Wartezeit kann seine Lauffähigkeit z.B. durch einen höher priorisierten *Thread* verändert worden sein. Im lokalen Fall wird der Helfer in die Ready-Liste eingetragen und ein Schedule durchgeführt. Im *Remote*-Fall wird die Priorität des Helfers mit der Priorität des aktuell auf dem Prozessor laufenden *Threads* verglichen. Deswegen ist das Scheduling einfacher, als eine Erweiterung der Scheduling-Schnittstelle, um die höchste Priorität ermitteln zu

können. Hat der Helfer die höchste Priorität, wird durch `switch_to` auf dem Prozessor des Helfers zu ihm umgeschaltet. Andernfalls wird sofort der nächste Helfer aus der Helferliste benachrichtigt.

Mit dem vorgestellten Mechanismus wird Fortschritt garantiert und die Ausführung des kritischen Abschnitts auf jedem Prozessor ermöglicht. Der kritische Abschnitt wird jedoch zu jedem Zeitpunkt auf höchstens einem Prozessor ausgeführt.

Der Helping-Vorgang wird durch eine implizite Migration des Besitzer-Kontexts für maximal die Dauer des kritischen Abschnitts durchgeführt. Der Helfer führt den Besitzer-Kontext auf seiner Zeitscheibe lokal auf seinem Prozessor aus. Die Migration erfolgt höchstens für die Dauer der Ressourcennutzung und beinhaltet keine Migration auf Benutzerebene.

Während des Helping-Vorgangs nehmen der Besitzer und die Helfer unterschiedliche Zustände an. Diese sind in dem folgenden Zustandsübergangsdiagramm (Abbildung 4.1) dargestellt. In Tabelle 4.1 sind die einzelnen Zustände beschrieben.

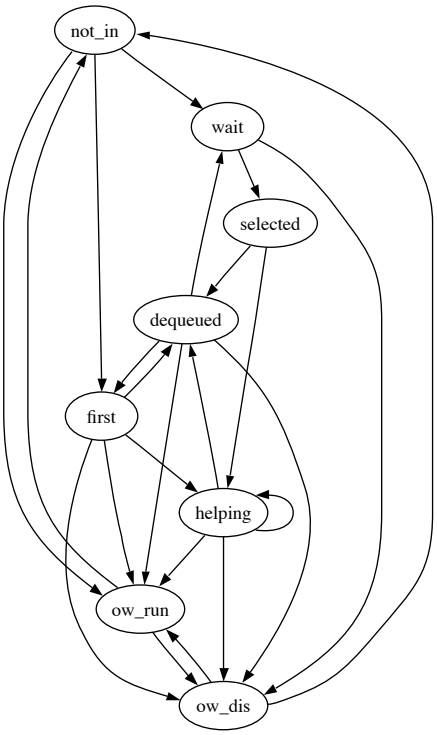

Abbildung 4.1: Zustandsübergangsdiagramm für Zustände, die Besitzer und Helfer einnehmen können

Tabelle 4.1: Beschreibung der Zustände des Zustandsgraphen

Name	**Beschreibung**
not_in	Zustand *not involved*. Alle Threads, die weder das Recht zur exklusiven Nutzung der Ressource besitzen, noch die Ressource anfordern.
ow_run	Zustand *owner running*. Der *Thread* besitzt das Recht zur exklusiven Nutzung der Ressource. Der Besitzer ist in der lokalen *Ready*-Liste und arbeitet den kritischen Abschnitt selbst ab oder er wurde durch einen höher priorisierten *Thread* unterbrochen.
ow_dis	Zustand *owner disabled*. Der *Thread* besitzt das Recht zur exklusiven Nutzung der Ressource. Der Besitzer wurde durch einen Helfer deaktiviert. Der Helfer führt den kritischen Abschnitt aus (aktives Helfen).
wait	Zustand *waiting*. Die Ressource wurde angefordert, aber das Recht noch nicht zugeteilt. Ein anderer Helfer hilft momentan. Der Helfer befindet sich in der Helferliste und wartet auf eine Benachrichtigung, mit dem Helfen zu beginnen. Er befindet sich nicht mehr in der Ready-Liste und kann damit vom Scheduler nicht ausgewählt werden.
first	Zustand *first*. Die Ressource wurde angefordert, aber das Recht noch nicht zugeteilt. Der Helfer ist an erster Stelle in der Helferliste und wartet auf den erfolgreichen Abschluss der Deaktivierung des Besitzers. Zum Zeitpunkt des Eintreffen des Helfers gab es noch keine weiteren Helfer.
selected	Zustand *selected*. Die Ressource wurde angefordert, aber das Recht noch nicht zugeteilt. Dieser Zustand ist ein Zwischenzustand. Der Helfer ist an erster Stelle in der Helferliste und eine Benachrichtigung, mit dem Helfen zu beginnen, ist unterwegs. Der Besitzer wurde bereits von einem anderen Helfer deaktiviert.
helping	Zustand *helping*. Die Ressource wurde angefordert, aber das Recht noch nicht zugeteilt. Der Helfer ist erster in der Helferliste. Der Besitzer führt den kritischen Abschnitt auf dem Prozessor des Helfers und dessen Zeitscheibe aus.
dequeued	Zustand *dequeued*. Die Ressource wurde angefordert, aber das Recht noch nicht zugeteilt. Der Helfer wurde von einem lokalen *Thread* verdrängt. Er wurde aus der Helferliste entfernt und in die Ready-Liste eingefügt. Sobald er vom Scheduler erneut ausgewählt wird, reiht er sich wieder in die Helferliste ein.

Parallele Aktivitäten

Beim MP-fähigen Helping-Mechanismus gibt es einige Aktivitäten, deren strikte Serialisierung, z.B. durch ein Spin-Lock im switch_to-Pfad, zu Leistungseinbußen führen würde. Die Serialisierung würde die Skalierbarkeit verschlechtern. Deshalb wird an bestimmten Punkten ein gewisser Grad an Unbestimmtheit erlaubt und im Bedarfsfall wird mit Hilfe

eines serialisierenden Ereignisses ein bestimmter Zustand herbeigeführt.

- Übergang des Besitzers von `ow_run` nach `not_in` und parallel Übergang eines Helfers von `first` nach `helping` (Serialisierung durch Nachricht).
- Übergang mehrerer *Threads* von `not_in` nach `wait` (Serialisierung durch Lock).
- Übergang mehrerer *Threads* von `dequeued` nach `wait` (Serialisierung durch Lock).

Invarianten

Bei der Konstruktion des Algorithmus habe ich mehrere Invarianten festgelegt, deren Verletzung das Nichterreichen der gesetzten Ziele zur Folge hätte:

1. Höchstens ein *Thread* führt zu jedem Zeitpunkt den kritischen Abschnitt aus (Korrektheit).
2. Wenn es mindestens einen Helfer gibt, existiert auch ein Besitzer.
3. Wenn sowohl Besitzer als auch Helfer ausführungsbereit sind, haben Helfer Vorrang und führen den kritischen Abschnitt aus. Damit werden Änderungen am switch_to-Pfad vermieden, da potentielle Helfer nicht den Zustand des Besitzers abfragen und entsprechend handeln müssen.
4. Der kritische Abschnitt wird ausgeführt, wenn mindestens ein Anforderer des Locks lauffähig ist.
5. Der erste *Thread* in der Helferliste ist im Zustand `first`, `selected` oder `helping`. Alle weiteren *Threads* in der Helferliste befinden sich im Zustand `wait`. Wenn ein *Thread* im Zustand `wait` ist, dann gibt es auch einen *Thread* im Zustand `first`, `selected` oder `helping`.
6. Der Besitzer befindet sich nicht in der Helferliste. Damit wird die Weitergabe des Helfens vereinfacht. Die Ausführung des kritischen Abschnitts kann so an den nächsten *Thread* in der Helferliste weitergegeben werden kann.

Gekoppelte Übergänge

Der Zustandsübergang eines *Threads* im MP-Helping-Mechanismus kann möglicherweise zur Verletzung einer Invariante führen. Zur Aufrechterhaltung der Invarianten kann es daher nötig sein, den Zustand anderer *Threads* zu verändern. Die Tabellen 4.2 bis 4.7 listen alle relevanten Zustandsübergänge auf und zählen die Bedingungen, die für ihren Eintritt nötig sind, auf.

4.2.2 Ablaufszenarien

Beim Ablauf des Multiprozessor-Helpings gibt es unterschiedliche Szenarien. Im Folgenden sind typische Abläufe dargestellt.

1. Das Lock wird freigegeben. Während dessen greift ein neuer *Thread* auf die Ressource zu. Der *Disable-Request* schlägt fehl (Meldung: neuer Besitzer, Freigeben des Locks und Einreihen in die Helferliste sind synchronisiert). Der neue *Thread* wird neuer Besitzer.

2. Ein Helfer beendet den kritischen Abschnitt. Dabei können die folgenden zwei Fälle auftreten:
   a) Ein anfordernder *Thread*, der auch Helfer sein kann, wird neuer Besitzer. Der neue Besitzer geht in den Zustand ow_dis über und der aktuelle Helfer setzt das Helfen für den neuen Besitzer fort.
   b) Der Helfer wird selbst neuer Besitzer.
      i. Es gibt keine weiteren Helfer. Der neue Besitzer setzt die Ausführung unmittelbar fort (Zustand ow_run).
      ii. Es gibt weitere Helfer in der Helferliste. Der neue Besitzer deaktiviert sich und benachrichtigt anschließend den nächsten Helfer.

3. Der letzte Helfer wird verdrängt. Die Helferliste ist leer. Der Besitzer wird aktiviert und in die *Ready*-Liste seines Prozessors eingereiht. Der Scheduler kann den Besitzer zur Ausführung auswählen.

Tabelle 4.2: Zustandsübergänge ausgehend vom Zustand ***not_in***

Ereignis	**Bedingung**	**Folgezustand**	**Beschreibung**
Zugriff auf Ressource	Ressource nicht belegt	ow_run	*Thread* wird Besitzer
Zugriff auf Ressource	Ressource belegt, andere *Threads* versuchen bereits zu helfen	wait	*Thread* in Helferliste einreihen
Zugriff auf Ressource	Ressource belegt, Helferliste leer	first	*Thread* ist erster Helfer, Zustand des Besitzers nicht bekannt, daher Deaktivierung des Besitzers

Der Zustandsübergang $not\_in \mapsto first$ hat bei einem aktuell ausführenden Besitzer den Übergang $ow\_run \mapsto ow\_dis$ zur Folge. Der Helfer geht dann von `first` nach `helping` über.

Tabelle 4.3: Zustandsübergänge ausgehend vom Zustand ***wait***

Ereignis	**Bedingung**	**Folgezustand**	**Beschreibung**
Lock-Freigabe des vorherigen Besitzers		ow_dis	Helfer ist erster in der Warte-Liste, weitere bereite Helfer existieren

Tabelle 4.9: Zustandsübergänge ausgehend vom Zustand ***first***

Ereignis	**Bedingung**	**Folgezustand**	**Beschreibung**
disable-Request erfolgreich abgeschlossen		helping	Besitzer deaktiviert
disable-Request schlägt fehl, Lock-Freigabe durch Besitzer vor Deaktivierung	Helfer ist Erster in der Warte-Liste und Helferliste ist leer	ow_run	Helfer ist neuer Besitzer
disable-Request schlägt fehl, Lock-Freigabe durch Besitzer vor Deaktivierung	Helfer ist Erster in der Warte-Liste, Helferliste nicht leer	ow_dis	Helfer ist neuer Besitzer
disable-Request schlägt fehl, Lock-Freigabe durch Besitzer vor Deaktivierung	Helfer ist nicht Erster in der Warte-Liste	helping	der neue Besitzer wird deaktiviert
disable-Request schlägt fehl, Lock-Freigabe durch Besitzer vor Deaktivierung	Helfer ist Erster in der Warte-Liste, Helferliste ist leer, Helfer hat lokal nicht die höchste Priorität	dequeued	während des Abschlusses der Unterbrechungsnachricht, wählt der lokale Scheduler einen höher priorisierten *Thread* aus

Der Zustandsübergang $first \mapsto helping$ führt beim neuen Besitzer zum Übergang $wait \mapsto ow\_dis$ oder $dequeued \mapsto ow\_dis$.

Tabelle 4.4: Zustandsübergänge ausgehend vom Zustand ***helping***

Ereignis	**Bedingung**	**Folgezustand**	**Beschreibung**
Lock-Freigabe durch Helfer	Erster in Warte-Liste, Helferliste leer	ow_run	Helfer ist neuer Besitzer
Lock-Freigabe durch Helfer	Erster in Warte-Liste, Helferliste nicht leer	ow_dis	Helfer ist neuer Besitzer
Helfer wird unterbrochen		dequeued	der nächste Helfer wird benachrichtigt oder der Besitzer wird aktiviert
Lock-Freigabe durch Helfer	Helfer ist nicht Erster in der Warte-Liste	helping	Helfer setzt Helfen für neuen Besitzer fort

Der Zustandsübergang *helping* $\mapsto$ *ow_dis* hat bei der Existenz weiterer bereiter Helfer den Übergang *wait* $\mapsto$ *helping* des nächsten Helfers zur Folge. Ein Übergang von `wait` über `first` nach `helping` ist nicht nötig, da der Besitzer bereits deaktiviert ist.
Der Übergang *helping* $\mapsto$ *dequeued* löst bei Vorhandensein weiterer Helfer den Übergang *wait* $\mapsto$ *selected* des nächsten Helfers aus. Gibt es keine weiteren Helfer, geht der Besitzer von *ow_dis* nach *ow_run* über.

Tabelle 4.5: Zustandsübergänge ausgehend vom Zustand ***ow_run***

Ereignis	**Folgezustand**
Lockfreigabe	not_int
disable-Request	ow_dis

Der Zustandsübergang *ow_run* $\mapsto$ *not_in* eines Besitzers löst, falls parallel Helfer eingetroffen sind, den Übergang $first \mapsto ow\_run$ (nur ein Helfer), den Übergang *wait* $\mapsto$ *ow_dis* (mehrere Helfer) oder den Übergang $first \mapsto ow\_dis$ (weitere Helfer vorhanden) des Helfers bzw. neuen Besitzers aus.
Der Übergang *ow_run* $\mapsto$ *ow_dis* führt zum Übergang $first \mapsto helping$ des Helfers, der den disable-Request initiiert hat.

Tabelle 4.6: Zustandsübergänge ausgehend vom Zustand ***ow_dis***

Ereignis	**Bedingung**	**Folgezustand**	**Beschreibung**
Helfer wird unterbrochen	es gibt keine Helfer	ow_run	Helfer hat lokal nicht mehr die höchste Priorität
Lock-Freigabe durch Helfer		not_in	Helfer beendet kritischen Abschnitt

Der Zustandsübergang $ow\_dis \mapsto not\_in$ hat für den neuen Besitzer entweder den Übergang $wait \mapsto ow\_dis$ (weitere Helfer vorhanden), $helping \mapsto ow\_run$ (aktiver Helfer wird neuer Besitzer, keine weiteren Helfer vorhanden) oder $helping \mapsto ow\_dis$ (aktiver Helfer wird neuer Besitzer, weitere Helfer vorhanden) zur Folge. Wird der Helfer unterbrochen und existieren weitere Helfer, so verbleibt der Besitzer im Zustand `ow_dis`

Tabelle 4.7: Zustandsübergänge ausgehend vom Zustand ***selected***

Ereignis	**Bedingung**	**Folgezustand**	**Beschreibung**
einkommende Benachrichtigung	Helfer hat lokal die höchste Priorität	helping	
einkommende Benachrichtigung	Helfer hat lokal nicht die höchste Priorität	dequeued	Benachrichtigung weiterer Helfer

Tabelle 4.8: Zustandsübergänge ausgehend vom Zustand ***dequeued***

Ereignis	**Bedingung**	**Folgezustand**	**Beschreibung**
Helfer vom lokalen Scheduler ausgewählt	Helferliste ist nicht leer	wait	Helfer reiht sich erneut in die Helferliste ein und wartet auf Benachrichtigung
Helfer vom lokalen Scheduler ausgewählt	Helferliste leer	first	der Besitzer muss erst deaktiviert werden, dann Beginn des Helfens, Erfüllung der Invariante 3
Helfer wird neuer Besitzer	Helferliste ist nicht leer	ow_dis	der Helfer wird nach der Unterbrechung neuer Besitzer
Helfer wird neuer Besitzer	Helferliste leer	ow_run	der Helfer wird neuer Besitzer

5 Implementierung

In diesem Kapitel beschreibe ich einige Aspekte der Implementierung des Modells 2. Im Abschnitt 5.8 gehe ich näher auf die Umsetzung des MP-Preemption-Locks ein.

5.1 Prozessorlokale Daten

Für die Implementierung der prozessorlokalen Daten verwende ich die Template-Technologie von C++. Prozessorlokale Daten werden mit Hilfe dieses Templates definiert. Mittels des Templates kann über einen Index auf die entsprechenden lokalen Daten zugegriffen werden.

5.1.1 Task-State-Segment (TSS)

Bei einem Kerneintritt eines *Threads* werden ein neuer Instruktionspointer und ein neuer Stackpointer benötigt. Der Instruktionspointer wird aus dem Interrupt- oder Trapgate der Interrupt-*Descriptor-Table* (IDT) ermittelt. Der Stackpointer wird aus dem TSS geladen. Das TSS ist auf allen Prozessoren an der gleichen virtuellen Adresse eingeblendet. Ein TSS kann nicht von mehreren Prozessoren genutzt werden.Treten ein Thread auf dem Heimat-Prozessor und ein Fern-Thread parallel in den Kern ein, muss der Stackpointer aus unterschiedlichen TSS geladen werden. Deshalb muss das TSS jedes Prozessors an einer anderen virtuellen Adresse eingeblendet werden, wenn die gleiche Seitentabelle benutzt wird.

Das TSS muss auf einer speziellen Seite zusammen mit der IO-Bitmap liegen. Es wird ein statisches Feld angelegt, welches für eine bestimmte Anzahl (momentan durch die Konfiguration auf vier beschränkt) von Prozessoren das jeweilige TSS speichert. Beim Zugriff auf das TSS wird dann anhand der Prozessor-ID der richtige Eintrag aus diesem Feld ausgewählt.

5.1.2 IPC-Fenster

Das IPC-Fenster wird für Long-IPC-Nachrichten verwendet und ist an festen virtuellen Adressen im Kern-Adressraum eingeblendet. Beim gleichzeitigen Zugriff von Heimat- und Fern-Threads auf das IPC-Fenster kann es zu Daten-Korrumpierung kommen.

5.2 Startvorgang

In der bisherigen Implementierung wird beim Bootvorgang der weiteren Prozessoren der Stack des Bootstrap-Prozessors kopiert. Da der Kern auf dem neuen Prozessor die gleichen virtuellen Adressen verwendet, kann er den kopierten Call-Frame auf dem Stack verwenden und bei der Rückkehr aus der Startfunktion einfach anhand der Prozessor-ID entscheiden, was zu tun ist.

Bei einem gemeinsam genutzten Adressraum funktioniert diese Technik nicht mehr. Deshalb wird jedem Prozessor beim Starten ein dedizierter Bootstack zur Verfügung gestellt. Auf diesem Bootstack ruft jeder Prozessor die Initialisierungsfunktion `init_ap` auf. Darin werden der APIC, die Timer und weitere allgemeine prozessorlokale Daten (UTCB und FPU) initialisiert. Anschließend wird die `main`-Funktion aufgerufen. Darin wird der Kern-Thread gestartet, der wiederum die Benutzerthreads für Sigma0 und die Roottask startet.

Das bisherige Clonen der Seitentabelle entfällt, da alle Prozessoren die gleiche Kern-Seitentabelle verwenden. Alle weiteren Prozessoren werden in der `__main`-Funktion (vom Bootstrap-Prozessor ausgeführt) gestartet.

5.3 TLB-Konsistenz

Im Abschnitt 3.1.2 habe ich unterschiedliche Strategien vorgestellt, mit denen Konsistenz zwischen den TLBs gewährleistet werden kann. Die eifrige Variante löscht immer alle TLBs aller Prozessoren. Bei der selektiven Löschung werden nur TLBs betroffener Prozessoren gelöscht.

Da TLBs eine Quelle von Unvorhersagbarkeit sind, müssen die Kosten für Adressumsetzungen ohne TLB in der Berechnung der *Worst Case Execution Time* (WCET) berücksichtigt werden. Zum Erreichen der Ziele „Reduzierung des Ressourcenverbrauchs“ und „Erhalten der Laufzeit von lokalen Operationen“, war es ausreichend, die eifrige TLB-Löschung zu implementieren. Bei einer Unmap-Operation, bei der die Löschung des TLBs erforderlich ist, wird eine *TLB-Shootdown*-Nachricht an alle Prozessoren im System verschickt. Bei Erhalt einer solchen Nachricht wird dann der TLB gelöscht.

5.4 Assistenten-Thread

Der Assistenten-Thread wird von der Klasse `Assist_thread` implementiert. Er bekommt die niedrigste Priorität des Systems, damit er niemals vom Scheduler ausgewählt wird. Dazu wird die Priorität des *Idle*-Threads um eins angehoben. Somit ist sichergestellt, dass der Assistenten-*Thread* nur durch ein explizites Umschalten zu seinem Kontext läuft. Ein Umschalten zum Assistenten-*Thread* darf nur bei gesetztem CPU-Lock erfolgen. Das bedeutet, dass der Assistenten-*Thread* scheinbar mit der höchsten Systempriorität läuft, da er niemals unterbrochen werden kann. Damit wird sichergestellt, dass der Assistenten-*Thread* nicht durch externe Ereignisse verdrängt werden kann.

5.5 Allokatoren

Im Abschnitt 3.2.1 habe ich die in FIASCO verwendeten Allokatoren aufgezählt. Der Zugriff auf diese Allokatoren muss synchronisiert werden. Dafür verwende ich ein einfaches Spin-Lock.

5.6 Kern-Thread-IDs

Mit dem Wegfall der Partitionierung des Systems entfällt auch die Partitionierung des TCB-Bereichs. Alle Prozessoren erzeugen ihre Kern-Threads (Idle- und Assist-Thread) daher nach folgendem Schema:

Prozessor 0: Kern-Thread-ID 00.00, Assist-Thread-ID 00.01

Prozessor 1: Kern-Thread-ID 00.02, Assist-Thread-ID 00.03 usw.

5.7 Taskerzeugung

Der `ex_regs`-Systemruf dient dazu, den Zustand von *Threads* zu verändern. Damit der Kern einen *Thread* auf einem anderen Prozessor erzeugen kann, benötigt er Informationen über die Prozessornummer, auf dem der *Thread* gestartet werden soll. Dazu wird der `Ex-Regs`-Systemruf mit zwei Parametern aufgerufen. Zum einen wird die Erzeugung eines Fern-*Threads* signalisiert und zum anderen wird der Prozessor für diesen *Thread* angeben.

Der dem Kern mit diesem Systemruf übergebene 32-Bit-Wert (L4v2-API) hat die in Abbildung 5.1 dargestellte Struktur.

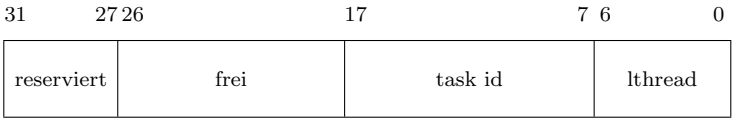

Abbildung 5.1: Struktur des mit `l4_thread_ex_regs` übergebenen Parameters

Daraus geht hervor, dass die Bits 26 bis 18 für weitere Parameter zur Verfügung stehen. Bit 26 wird verwendet, um anzuzeigen, ob ein *Thread* auf einem anderen Prozessor erzeugt werden soll. Dafür steht die neue Konstante `L4_THREAD_EX_REGS_REMOTE_THREAD` zur Verfügung. In den drei folgenden Bits wird die Prozessor-ID übertragen. Damit können *Threads* nur auf den ersten 8 Prozessoren erzeugt werden. Der 32-Bit-Wert bei Erzeugung eines Fern-Threads sieht wie in Abbildung 5.2 aus.

Der Kern wertet bei diesem Systemruf erst das `L4_THREAD_EX_REGS_REMOTE_THREAD`-Feld aus. Soll ein *Thread* auf einem anderen Prozessor erzeugt werden, wird eine Nachricht mit den benötigten Parametern an den entsprechenden Prozessor verschickt. Dafür ist ein neuer Nachrichten-Typ notwendig. Um diese Parameter bequem ermitteln zu können, habe ich die Klasse `Sys_ex_regs_frame` um zwei Funktionen erweitert.

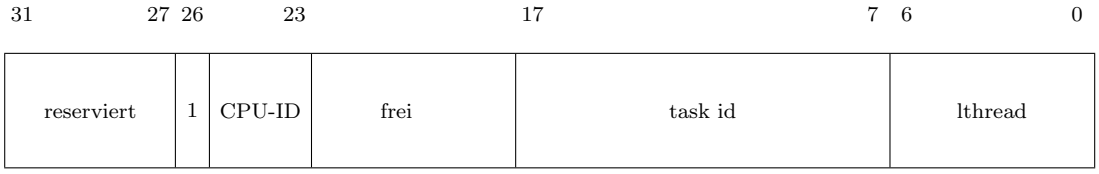

Abbildung 5.2: 32-Bit-Wert für die Erzeugung eines *Threads* auf einem anderen Prozessor

5.7.1 Tasks und Threads

Im Modell A wird davon ausgegangen, dass alle *Threads* einer Task entweder in der *Present-* (globale Liste) oder *Ready*-Liste nur eines Prozessors eingetragen sind. Das trifft für Fern-*Threads* nicht mehr zu.

Um alle *Threads* bei der Taskzerstörung zu erfassen, wird deshalb durch die *Present*-Liste iteriert und alle *Threads* werden zerstört.

5.8 MP-Preemption-Lock

Das MP-Preemption-Lock wird von der Klasse `Mp_preemption_lock` implementiert. Die Klassen `Cpu` und `Context` mussten erweitert werden, damit sie die für das Lock benötigte Funktionalität bereitstellen. Dazu gehören die *Thread*-Zustände und die Möglichkeit zur Unterbrechung des Lock-Halters durch einen Helfer.

Jeder Kontext verfügt über ein Zustandsfeld, in dem der in der Tabelle 4.1 beschriebene Lock-Zustand gespeichert wird.

Um die Lock-Funktionalität zu testen, habe ich mit Hilfe des Kern-Testframeworks Tests implementiert, die reproduzierbar ausgewählte Situationen testen. In Tabelle 5.1 findet sich ein Überblick über die Tests und welche Fälle damit getestet werden. Die Testläufe sehen so aus, dass in einer Schleife das Lock gegriffen und wieder freigegeben wird. Die Lockhaltedauer wird über eine Schleife simuliert. Die Länge der Schleife wird dynamisch modifiziert, um unterschiedliche Lock-Haltezeiten zu simulieren.

Tabelle 5.1: Tests mit denen das MP-Preemption-Lock getestet wurde

Testname	**Testsituation**
test_mppl_single_1	Test auf einem Prozessor. Ein Thread greift exklusiv auf das Lock zu, mit und ohne Unterbrechung im kritischen Abschnitt

Tabelle 5.1: Tests mit denen das MP-Preemption-Lock getestet wurde

Testname	**Testsituation**
test_mppl_single_2	Test auf einem Prozessor. Zwei Threads konkurrieren um das Lock. Der Besitzer wird unterbrochen und • der Helfer beendet den kritischen Abschnitt oder • wird selbst auch unterbrochen und der Besitzer beendet anschließend selbst den kritischen Abschnitt
test_mppl_single_3	Test auf einem Prozessor. Drei Threads konkurrieren um das Lock. Weitergabe der Ausführung des kritischen Abschnitts zwischen Helfern
test_mppl_multi_2	Zwei Threads auf unterschiedlichen Prozessoren konkurrieren um das Lock. Der Besitzer wird deaktiviert und der Helfer beendet den kritischen Abschnitt, Unterbrechung des Helfers (durch *Timeout* und anschließend Aktivierung des Besitzers möglich
test_mp_preemptioni_lock	Test auf zwei Prozessoren mit drei Threads. Die *Threads* konkurrieren jeweils um das Lock. Übergabe zwischen Helfern, Aktivierung des Besitzers, lokales Helfen

5.9 Benutzerebene

Das im Modell A benötigte Server-Stub-Paradigma für den Sigma0- und Roottask-Server wird nicht mehr benötigt. Speicher-Mappings können auf allen Prozessoren entgegengenommen werden und auch über Prozessorgrenzen hinweg vergeben werden. Deshalb lassen sich für beide Server die Uniprozessorvarianten verwenden.

6 Evaluierung

In diesem Kapitel werde ich anhand unterschiedlicher Kriterien untersuchen, ob das von mir prototypisch implementierte Modell die gewünschten Ziele:

- Verringerung des Ressourcen-Verbrauchs,
- Erhalt der Echtzeiteigenschaften und
- ausreichende Funktionalität für eine hypothetische L^{4}Linux-Implementierung

erreicht. Die Messungen habe ich, soweit nicht anders angegeben, auf einem Pentium 3 Multiprozessorsystem mit 2 Prozessoren und 256MB RAM durchgeführt. Die Prozessoren sind mit 450MHz getaktet.

6.1 Funktionalität

In diesem Abschnitt vergleiche ich die Funktionalität meiner Implementierung mit der Funktionalität der Uniprozessorvariante und der alten FiascoMP-Implementierung von Sven Schneider.

Das Modell 2 implementiert gemeinsam genutzte Adressräume über Prozessorgrenzen hinweg. Damit lässt sich die Menge der für eine bestimmte Arbeitslast benötigten Tasks reduzieren. Zusätzlich entfällt der höhere Speicherverbrauch durch die Duplikation der Kern-Seitentabellen und der Mapping-Datenbanken (vgl. Abschnitt 2.3.3).

Um das zu zeigen, habe ich das folgende Messszenario verwendet. Als Arbeitslast habe ich eine multithreadfähige Berechnung der Mandelbrotmenge gewählt. Das Beispiel existiert in drei Varianten:

1. für die Implementierung des Modells A
2. für die Implementierung des Modells 2
3. als Uniprozessorvariante aufbauend auf Modell 2

Den nachfolgenden Abbildungen 6.1 und 6.2 kann der prinzipielle Aufbau des Beispielprogrammes in den unterschiedlichen Varianten entnommen werden. Es gibt zwei *Threads* (W_0 und W_1), die jeweils einen Teilbereich (Kachel) der Mandelbrotmenge berechnen. Den Teilbereich bekommen sie durch den Koordinatorthread C zugewiesen. R ist der Regionmanager, der Seitenfehler der Arbeiterthreads an einen Pager weiterleitet.

Bei der Implementierung dieses Beispiels für das Modell A werden auf der Remote-Seite zusätzliche Proxy-Threads benötigt. A_0 nimmt Speicher-Mappings, die vom Proxy-Pager P_1 verschickt werden, entgegen. R_1 ist der Proxy des Region-Managers.

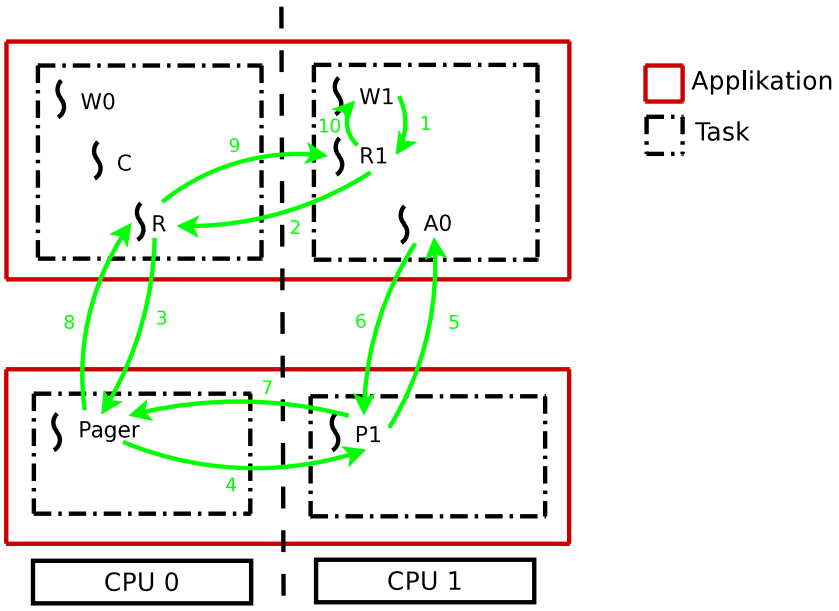

Abbildung 6.1: Schematische Darstellung des Testprogramms für Modell A

Ein möglicher Ablauf des Beispielprogramms im Modell A (Abbildung 6.1) sieht wie folgt aus:

1. W_1 löst einen Seitenfehler aus Der Kern generiert eine IPC (1), die an R_1 geschickt wird (direktes Senden an R nicht möglich).

2. R_1 schickt die Nachricht (2) weiter an den Region-Manager R. Der Region-Manager ist für Speicheranforderungen und Benachrichtigungen zuständig.

3. Der Region-Manager fordert die Seite zuerst selbst durch einen Seitenfehler an. Dabei wird IPC (3) an den Pager geschickt.

4. Der Pager muss dafür sorgen, dass die Seite in alle Tasks der Anwendung eingeblendet wird. Das lässt sich am einfachsten mit einer Greedy-Strategie realisieren. Dazu schickt der Pager die Nachricht (4) an seinen Proxy-Pager P_1, der ein Speicher-Mapping an A_0 verschicken.

5. Zwischen Pager und Akzeptor-*Thread* gibt es ein Vertrauensverhältnis. Der Akzeptor-*Thread* schränkt seinen Empfangsbereich für Speicher-Mappings nicht ein, so dass der Pager beliebige Seiten einblenden kann. P_1 verschickt an A_0 ein Speicher-Mapping (5) und wartet auf die Antwort (6), anschließend bestätigt die Vergabe des Mappings mit Nachricht (7) an den Pager

6. Der Pager schickt ebenfalls ein Speicher-Mapping (8) zurück an den Region-Manager.

7. Der Region-Manager informiert seinen Proxy R_1, dass der Seitenfehler aufgelöst worden ist (9).

8. R_1 benachrichtigt W_1 (10).

Im Modell 2 ist dieses Ablaufszenario (siehe Abbildung 6.2) einfacher, da keine Proxy-Threads benötigt werden. Die Seitenfehlernachricht kann direkt an den Region-Manager geschickt werden. Der Pager schickt das Speicher-Mapping zurück an den Region-Manager. Damit wird es auch für W_1 sichtbar. Der Umweg über die Proxies P_1 und A_0 entfällt.

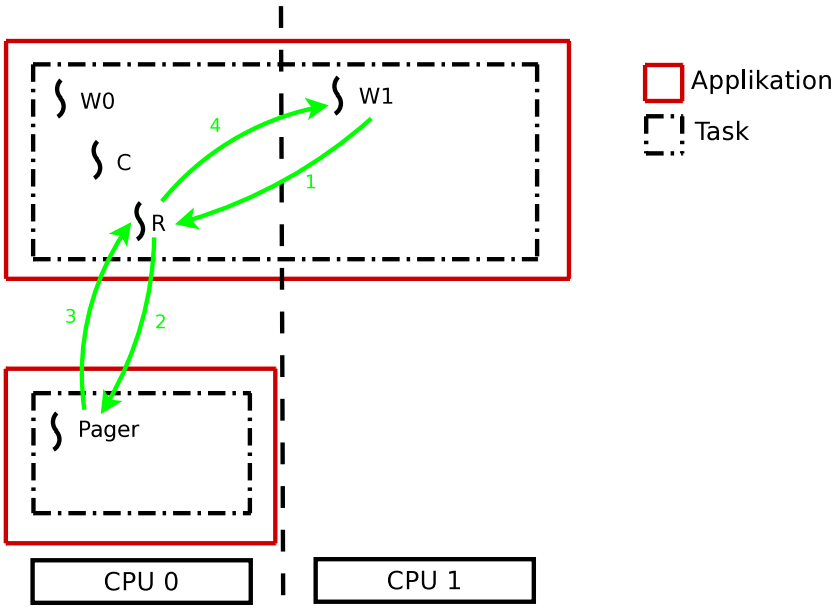

Abbildung 6.2: Schematische Darstellung des Testprogramms für Modell 2

6.1.1 Parallele Nutzung mehrerer Prozessoren

Durch die parallele Nutzung mehrerer Prozessoren soll ein Geschwindigkeitsgewinn gegenüber einer Uniprozessorlösung bei der Berechnung erzielt werden. Im Idealfall steigt die Geschwindigkeit linear mit der Anzahl der Prozessoren in einem System, so dass man bei einem 2-Prozessorsystem im Vergleich zu einem Uniprozessorsystem die doppelte Geschwindigkeit erreicht. Dieser theoretische Wert lässt sich in der Praxis kaum erreichen, da bei steigender Anzahl von Prozessoren der Aufwand für die Verteilung von Arbeitsaufgaben auf die Prozessoren zunimmt. Dadurch wird der Geschwindigkeitsgewinn verringert.

In Tabelle 6.1 ist zu erkennen, dass der Geschwindigkeitsgewinn durch die Berechnung auf zwei Prozessoren im Model A bei 2,00 liegt. Die Schwankungen sind dadurch zu erklären, dass einzelne Abschnitte der Berechnung unterschiedlich komplex sind und deshalb keine absolut gleichmäßige Verteilung der Berechnung auf zwei Prozessoren möglich ist. Der von den beiden Threads gemeinsam genutzte Speicher für die Speicherung des Ergebnisses wird zum Anfang der Messung eingeblendet und danach nicht wieder entzogen. Deshalb ist die Laufzeit bei der Zoomstufe 1 höher als in der darauf folgenden

Stufe. Die Ergebnisse dieser Messung im Modell 2 sind vergleichbar. Hier liegt der durch-

Tabelle 6.1: Geschwindigkeitsgewinn bei der Berechnung der Mandelbrotmenge durch zwei Threads auf einem und zwei Prozessoren im Modell A in Milliarden Takten (Pentium 3 Multiprozessor mit 450MHz und 256MB RAM

Zoomstufe	**Ein Prozessor**	**Zwei Prozessoren**	**Speedup**
1	3,01	1,51	1,98
2	3,05	1,52	2,00
3	3,11	1,55	2,00
4	3,17	1,57	2,00
5	3,23	1,61	2,00
6	3,29	1,64	2,00
7	3,35	1,67	2,00
8	3,42	1,70	2,00
9	3,48	1,74	2,00
10	3,55	1,77	2,00
		Durchschnitt	2,00

schnittliche Geschwindigkeitsgewinn bei 1,99. Eine detailierte Tabelle ist im Anhang zu finden (Tabelle 8.1).

Um die Skalierbarkeit meiner Implementierung zu messen, habe ich die Messungen des Geschwindigkeitsgewinns auch auf vier und acht Prozessoren gemessen. Die Messungen habe ich auf einem Intel-Xeon-System (Pentium D) mit 2x2 Prozessorkernen und Hyperthreading durchgeführt. Die Prozessoren sind mit 2,66GHz getaktet und den für Fiasco verfügbaren Hauptspeicher habe ich auf 256MB begrenzt. Der durchschnittliche Geschwindigkeitsgewinn kann Tabelle 6.2 entnommen werden. Es ist zu erkennen, dass

Tabelle 6.2: Durchschnittlicher Geschwindigkeitsgewinn bei bis zu 8 Prozessoren eines Intel-Xeon-Systems mit 2,66GHz

Prozessoren	**Stride**	**Zuwachs**
4	0	3,83
4	1	3,90
8	0	7,51

der Geschwindigkeitszuwachs bei acht Prozessoren nicht mehr linear ist, sondern langsam abflacht. Die Gründe dafür sind der steigende Mehraufwand bei der Verteilung der Berechnungspakete an die Arbeiter-*Threads*. Außerdem ist die Berechnung der einzelnen Pakete nicht völlig symmetrisch, so dass keine optimale Verteilung der Rechenlast möglich ist.

In der Tabelle 6.2 zeigt sich auch, dass Hyperthreading kein vollwertiger Ersatz für einen vollständigen Prozessor ist. Nutzt man die virtuellen Prozessorkerne (Stride = 1) nicht, so ist der Geschwindigkeitszuwachs bei vier Prozessoren höher, als wenn man zwei physische und zwei virtuelle Prozessorkerne (Stride = 0) für die Berechnung nutzt. Der Grund ist, dass zwei Hyperthreads um knappe Ressourcen des Prozessors (ALU, Dekoder) konkurrieren und damit gegenseitige Wartezeiten erzeugen.

Einfluss der Arbeitslast

Während der Messungen zeigte sich, dass die Art der Arbeitslast großen Einfluss auf den tatsächlich erzielbaren Geschwindigkeitszuwachs durch die Nutzung mehrerer Prozessoren hat. In Abbildung 6.3 ist zu erkennen, dass bei der Zerlegung des 800x600 Pixel großen Bildes in 1024 Kacheln der Geschwindigkeitsgewinn am höchsten ist. Bei weniger Kacheln werden die Prozessoren nicht mehr optimal ausgelastet, so dass Wartezeiten entstehen. Außerdem nimmt der Kommunikationsaufwand zu.

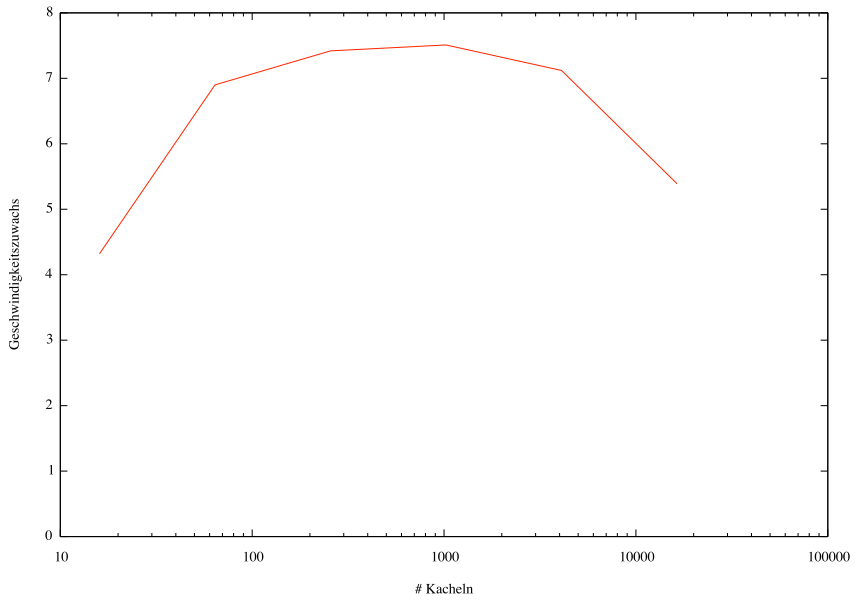

Abbildung 6.3: Geschwindigkeitszuwachs in Abhängigkeit von der Anzahl der Bildkacheln

6.1.2 Dynamische Arbeitslast

Das im vorangegangenen Abschnitt beschriebene Mandelbrot-Beispiel verwende ich, um Dynamik einer Arbeitslast zu simulieren. Dazu wird Speicher dynamisch zugeteilt und wieder entzogen. Dabei wird das Verhältnis zwischen Berechnungsdauer und Speicherreservierung und -freigabe verändert. Bei langen Berechnungen wird erwartet, dass der

Einfluss durch die Speicherverwaltung geringer ist, als bei kurzen Berechnungen. Insgesamt sollte ein deutlicher Mehraufwand für die Speicherverwaltung im Modell A sichtbar sein, da die Mapping-Datenbanken mehrerer Prozessoren aufwändig auf Benutzerebene konsistent gehalten werden müssen. Die Synchronisation erfordert zusätzliche Operationen mit sich daraus ergebenden Blockierzeiten.

Die folgenden Abbildungen geben jeweils die Berechnungszeiten für insgesamt 10 Bilder an. Der Speicher wurde dabei einmal nach der Berechnung jedes Bildes bzw. nach der Berechnung jeder einzelnen Kachel eines Bildes entzogen. Ein Bild bestand aus insgesamt neun Kacheln. Abbildung 6.4 zeigt deutlich den Mehraufwand, der durch die

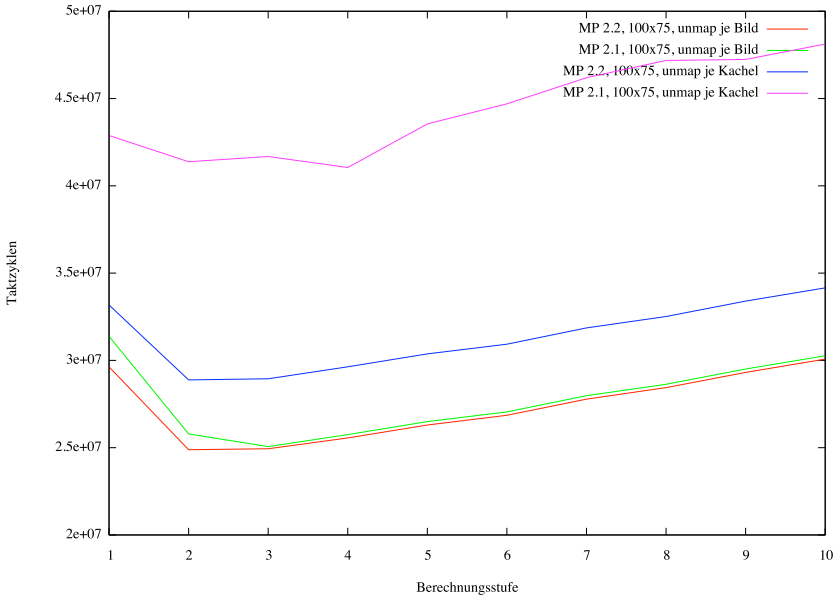

Abbildung 6.4: Berechnungszeiten für Apfelmännchen, Bildgröße 100x75 Pixel

häufige Unmap-Operation entsteht. Wird der Speicher häufig entzogen, verschlechtert sich bei beiden Modellen die Ausführungszeit. Bei Modell A zeigt sich deutlich der Mehraufwand, der bei einer Unmap-Operation entsteht. Der Mehraufwand liegt bei bis zu 30 Prozent gegenüber Modell 2. Wächst die Berechnungszeit im Verhältnis zur Zeit, die für die Unmap-Operation benötigt wird, so gleichen sich die Werte zwischen Modell 2 und Modell A an. In Abbildung 6.5 ist zu erkennen, dass beim Modell A der Mehraufwand bei häufigeren Unmap-Operationen stärker steigt, als bei Modell 2.

Das Modell 2 bietet gegenüber dem Modell A bei dynamischen Arbeitslasten also einen deutlichen Effizienzgewinn. Die Geschwindigkeitsverbesserungen liegen bei bis zu 30 Prozent gegenüber dem Modell A. Aus den Abbildungen 6.1 und 6.2 geht zudem hervor, dass das Applikationsdesign beim Modell 2 vereinfacht wurde, da z.B. die umständliche Konstruktion mit Proxy-Tasks entfällt.

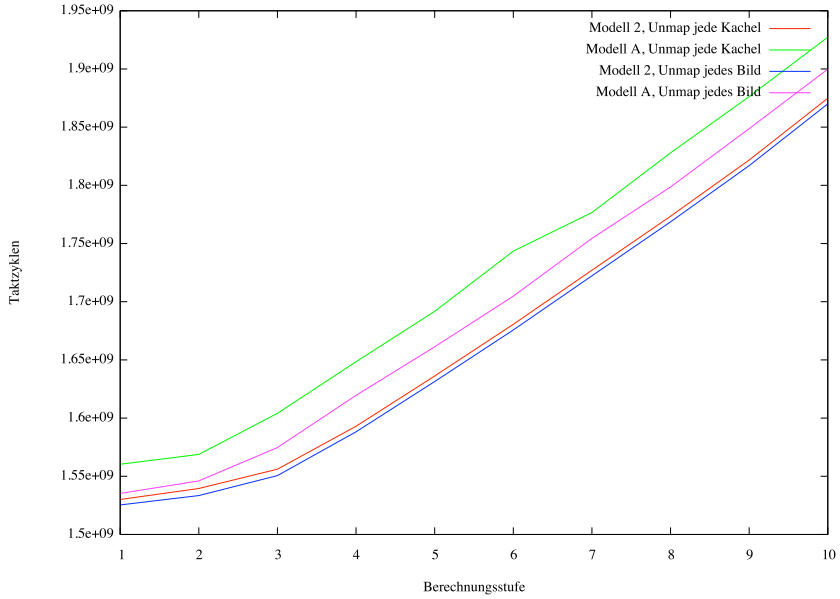

Abbildung 6.5: Berechnungszeiten für Apfelmännchen, Bildgröße 800x600 Pixel

6.1.3 Lokale-IPC-Geschwindigkeit

Ein Ziel der Weiterentwicklung von FiascoMP ist der Erhalt der Geschwindigkeit der lokalen Operationen. Die folgende Tabelle vergleicht die IPC-Geschwindigkeit des FiascoMP-Kerns mit der des Uniprozessorkerns. Dazu habe ich den Pingpong-Benchmark verwendet.

Wie Tabelle 6.3 zeigt, ist der IPC-Pfad im Multiprozessorkern in etwa gleich schnell wie beim Uniprozessorkern.

Tabelle 6.3: Lokale IPC-Geschwindigkeit (ohne ASM-Shortcut) in Taktzyklen gemessen auf einem Pentium III 2-Prozessor-System mit 450MHz

Testname	**Uniprozessor-Fiasco**	**FiascoMP**
int30/warm	1344	1393
sysenter/warm	1165	1165

Um die Unabhängigkeit der lokalen IPC-Operation von der Auslastungen der anderen Prozessoren zu zeigen, habe ich auf mehreren Prozessoren ein einfaches IPC-Pingpong laufen lassen. Zwei *Threads* auf einem Prozessor schicken sich lokal eine *Short*-IPC-Nachricht. Es wird die Zeit zwischen Abschicken einer Nachricht und Erhalt der Antwort vom Kommunikationspartner gemessen. Die Werte in Tabelle 6.4 zeigen, dass eine Arbeitslast auf einem Prozessor nicht die IPC-Operation auf anderen Prozessoren beeinflusst. Diese Messung habe ich sowohl auf einem Zwei-Prozessor (Pentium 3), als

auch auf einem Acht-Prozessorsystem (Xeon) durchgeführt Für die Streuung der Wer-

Tabelle 6.4: Lokale IPC-Geschwindigkeit in Abhängigkeit einer Arbeitslast auf anderen Prozessoren, Pentium 3 Multiprozessor mit 450MHz und 256MB RAM, Werte in Taktzyklen über 100.000 Runden gemittelt

Short-IPC (CPU 0)	**Short-IPC (lokal) parallel auf CPU 0 und CPU 1**
787	785

Tabelle 6.5: Lokale IPC-Geschwindigkeit in Abhängigkeit einer Arbeitslast (IPC) auf anderen Prozessoren, Intel-Xeon-System mit 2,66GHz, Werte in Taktzyklen über 100.000 Runden gemittelt

Prozessornummer	**IPC-Geschwindigkeit**
1	2370
2	2103
3	2528
4	2030
5	2402
6	2001
7	2209
8	2047

te auf dem Intel-Xeon-System sind vermutlich Cache- und Pipeline-Effekte sowie das Hyperthreading der Prozessoren verantwortlich. Den Einfluss des Hyperthreadings erkennt man daran, dass Prozessoren mit einer ungeraden Prozessornummer einen höheren Mittelwert aufweisen.

6.2 Echzeiteigenschaften

Für ein Lock können unterschiedliche Größen gemessen werden.

Blockierzeit: Die Blockierzeit ist die Zeit, die ein *Thread* nach der Anforderung eines Locks bis zur Gewährung des Locks warten muss. Die Blockierzeit wird durch Lock-Contention beeinflusst. Lock-Contention ist ein Maß für die Anzahl von Versuchen, bei denen das Lock bereits besetzt war. Lock-Contention hängt in hohem Maße von der Arbeitslast des Systems ab. Bei nichtunterbrechbaren Locks ist die Blockierzeit höchstens die Zeit, für die das Lock längstens gehalten wird plus die Blockierzeiten aller Threads, die vor diesem das Lock greifen.

Lock-Haltezeit: Die Lock-Haltezeit ist eine Eigenschaft (statisch) des Designs. Sie wird zum einen von der Aufgabe des kritischen Abschnitts und zum anderen von der Lock-Granularität bestimmt. Feingranulare Locks ermöglichen eine kurze Lock-Haltezeit und damit kurze Blockierzeiten von *Threads*.

Neben den Metriken können Locks auch hinsichtlich bestimmter Eigenschaften bewertet werden.

Fairness: Ist ein Lock fair, dann erhält ein Thread nach einer endlichen Wartezeit Zugriff auf das Lock. Anforderungen von *Threads* können nicht unbegrenzt verzögert werden.

Echtzeiteigenschaften: Wird durch ein Lock die Blockierzeit von Echtzeitthreads beeinflusst, obwohl keine Ressourcenabhängigkeit vorliegt? Können Threads, die ein Lock halten unterbrochen werden, kann damit die Dauer von Blockierzeiten vermindert werden.

6.2.1 Mess-Szenario

Um die Eigenschaften unterschiedlicher Lock-Varianten zu messen, habe ich ein einfaches Beispielprogramm entwickelt. Zwei *Threads* auf unterschiedlichen Prozessoren konkurrieren um ein Lock. Nachdem einer der beiden *Threads* das Lock gegriffen hat, wird er von einem höher priorisierten *Thread* auf seinem Prozessor verdrängt.

Ist das Lock ein nicht-unterbrechbares Spin-Lock, können Echtzeitthreads unter Umständen nicht ihre Deadlines einhalten, da sie den Lock-Halter, um selbst zu laufen, nicht unterbrechen können. Das ist bei Mapping-Datenbank-Operationen problematisch, da deren Bearbeitungszeit nicht begrenzt ist.

Tabelle 6.6: Ausführungszeiten für einen Zyklus bestehend aus: Lock greifen, kritischen Abschnitt bearbeiten, Lock freigeben, Zeiten in Taktzyklen angegeben

Zyklus	Lockhaltedauer	Gesamtzeit
1	100.000	85.748.441
2	100.000	45.102.602
3	100.000	45.102.629
1	300.000	266.158.207
2	300.000	135.307.620
3	300.000	135.307.644
1	600.000	536.767.247
2	600.000	270.615.903
3	600.000	270.615.918

Ein unterbrechbares Spin-Lock löst das Problem der fehlenden Unterbrechbarkeit. Allerdings tritt hier das Problem des Verhungerns von *Threads* auf. Das zu Anfang dieses

Abschnitts geschilderte Beispiel führt dazu, das ein Lock-Halter *Threads* auf anderen Prozessoren beliebig verzögern kann.

Das MP-Preemption-Lock löst beide der genannten Probleme. *Threads* können, während sie ein Lock halten, unterbrochen werden. Ein unterbrochener Lock-Halter kann *Threads* auf anderen Prozessoren nicht verzögern, da Lock-Anwärter den Lock-Halter auf ihrer Zeitscheibe bis zur Freigabe des Locks ausführen können.

In Tabelle 6.6 sind die Taktzyklen für einen kompletten Durchlauf durch einen kritischen Abschnitt (Lock greifen, kritischen Abschnitt bearbeiten, Lock freigeben) für einen *Thread* auf Prozessor 1 dargestellt. Der Lock-Halter auf Prozessor 0 wurde durch einen höher priorisierten Thread verdrängt. Es ist zu sehen, dass der erste Zyklus etwa doppelt so lang dauert, wie die nachfolgenden Zyklen. Das liegt daran, dass der *Thread* auf Prozessor 1 zunächst dem unterbrochenen Lock-Halter hilft und anschließend seinen eigenen kritischen Abschnitt bearbeitet.

In Abbildung 6.6 habe ich die Ausführungszeiten der Unmap-Operation bei steigender Größe des Speicherbereichs dargestellt. Die Achsen des Diagramms haben eine logarithmische Einteilung. Es ist gut zu sehen, dass der Anstieg der Dauer der

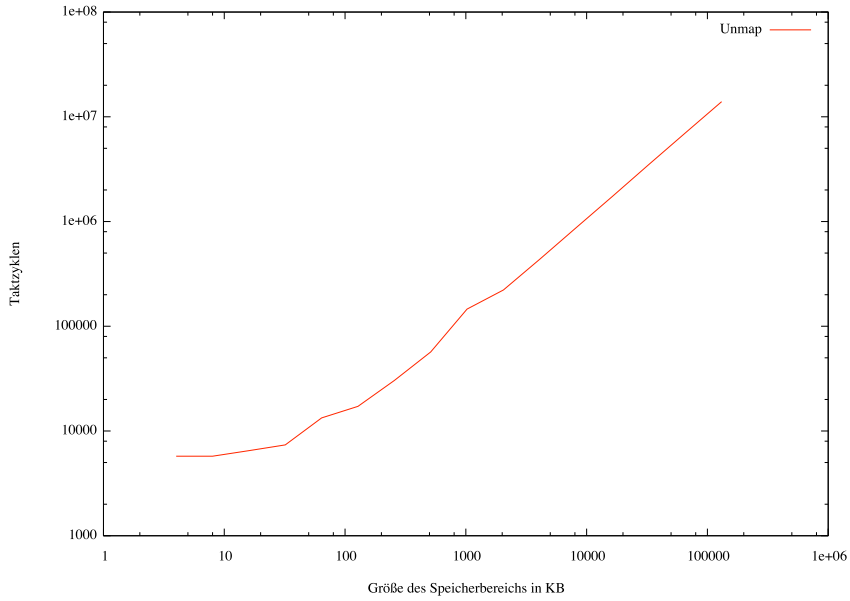

Abbildung 6.6: Laufzeit der Unmap-Operation bei steigender Größe des Speicherbereichs

Unmap-Operation mit steigender Größe des Speicherbereichs monoton anwächst. Wird die Mapping-Datenbank mit einem nicht unterbrechbaren Lock-Mechanismus synchronisiert, kann diese Eigenschaft negative Auswirkungen auf Echtzeitthreads haben, da diese dadurch beliebig verzögert werden können. Deshalb wird an dieser Stelle ein unterbrechbarer Lock-Mechanismus benötigt, weil die Konstruktion großer Mapping-Bäume im derzeitigen L4-Modell nicht verhindert werden kann. Ein möglicher Angreifer kann einen solchen Mapping-Baum aufbauen und dann eine Unmap-Operation starten. Die

Garantie, dass Echzeitthreads ihre Zeitschranken einhalten, kann nur mittels Unterbrechbarkeit erreicht werden.

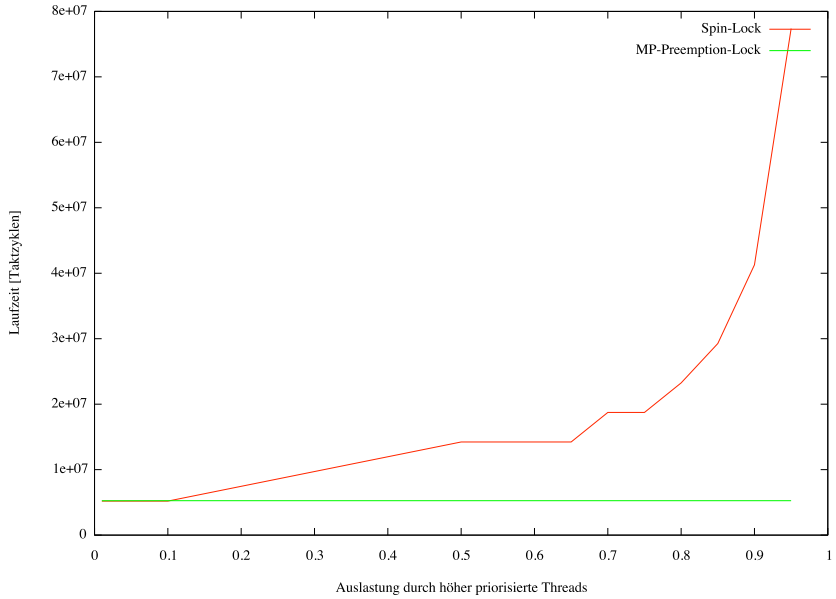

Abbildung 6.7: Verzögerungszeiten beim Zugriff auf ein Lock (Maximalwerte)

In einem synthetischen Messszenario habe ich die maximale Verzögerung und die maximale Dauer für die Bearbeitung eines kritischen Abschnitts gemessen. Zwei *Threads* auf unterschiedlichen Prozessoren konkurrieren um ein Lock. Auf einem der Prozessoren läuft ein unbeteiligter *Thread* (HP-*Thread*) mit einer höheren Priorität, als der Lock-Anwärter auf diesem Prozessor. Gemessen wird die Verzögerung und Gesamtlaufzeit des *Threads*, der allein auf einem Prozessor läuft. Die Messwerte habe ich bei steigender Auslastung des Prozessors durch den HP-*Thread* ermittelt und in Abbildung 6.7 dargestellt.

Es ist deutlich zu erkennen, dass die maximale Verzögerung bei der Benutzung eines unterbrechbaren Spin-Locks stark ansteigt. Die maximale Verzögerung bei Nutzungs des MP-Preemption-Locks bleibt konstant. Der gleiche Effekt ist bei der Gesamtausführungszeit zu sehen. Diese ist in Abbildung 6.8 dargestellt. Nicht nur bei der maximalen Ausführungszeit, sondern auch bei der durchschnittlichen Ausführungszeit bietet das MP-Preemption-Lock Vorteile gegenüber einem Spin-Lock. Beispielsweise sind bei Anwendungen mit graphischen Oberflächen, die keine periodischen Echtzeitaufgaben mit harten Deadlines haben, kurze Antwortzeiten bei der Bedienung wichtig. In Abbildung 6.9 ist zu erkennen, dass mit Hilfe des MP-Preemption-Locks die durchschnittliche Ausführungszeit bei der Messung unter einer gewissen Schranke bleibt. Die Beobachtungen lassen daher auf qualitative Eigenschaften diesbezüglich schließen. Die Peaks in der dargestellten Kurve sind durch Resonanzeffekte bei der Messung zu erklären.

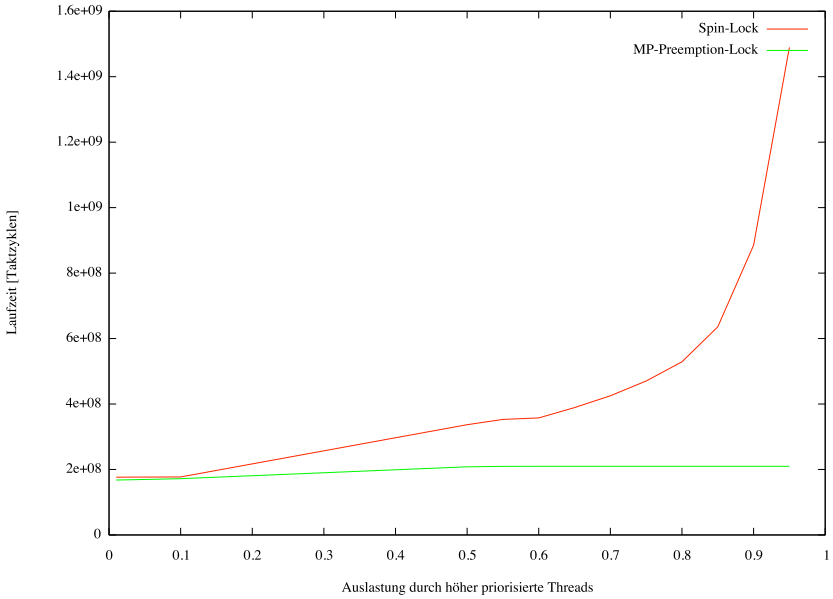

Abbildung 6.8: Gesamtausführungszeiten bei der Benutzung eines Spin- und MP-Preemption-Locks im Worst-Case-Fall

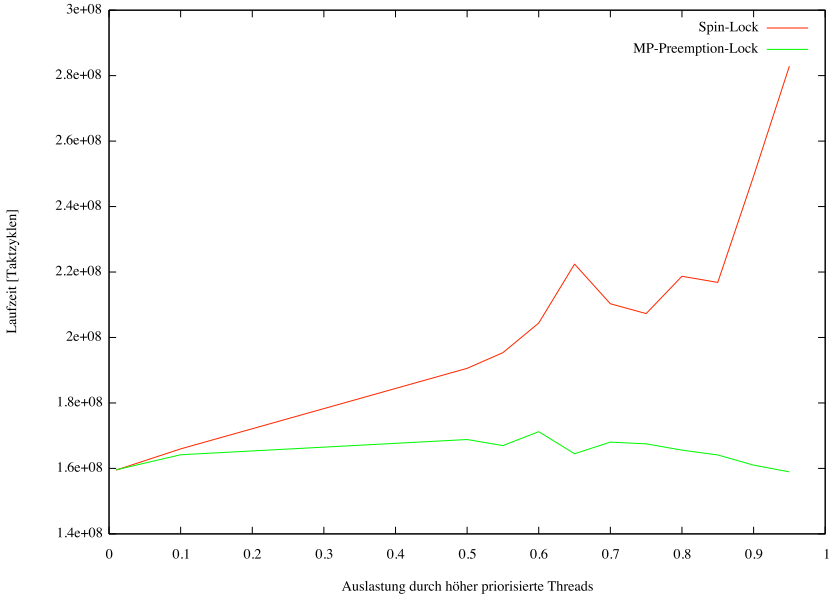

Abbildung 6.9: Gesamtausführungszeiten bei der Benutzung eines Spin- und MP-Preemption-Locks (durchschnittlich)

7 Zusammenfassung und Ausblick

In den folgenden Abschnitten möchte ich zunächst die Arbeit und die Ergebnisse kurz zusammenfassen. Anschließend gebe ich einen Ausblick auf mögliche zukünftige Arbeiten.

7.1 Zusammenfassung

Ziel dieser Arbeit war die Weiterentwicklung von FiascoMP, um die im Abschnitt 2.3.3 genannten Limitierungen, keine Cross-Prozessor-Tasks und keine Speicher-Mappings über Prozessorgrenzen hinweg, zu beseitigen. Bei der Weiterentwicklung des Kern-Modells wurde als Arbeitslast eine L^4Linux-MP-Implementierung angenommen.

Im Kapitel 3 habe ich drei Modelle vorgestellt, nach denen FiascoMP weiterentwickelt werden kann. Die Modelle bauen aufeinander auf. Modell 1 ermöglicht die Erzeugung von Cross-Prozessor-Tasks. Modell 2 beseitigt den hohen Ressourcenverbrauch, weil die Duplikation von Kern-Datenstrukturen entfällt. Modell 3 bietet ein für Anwendungen halb-transparentes SMP-System, in dem *Thread*-Migration durch einen Kern-Mechanismus unterstützt wird. In der abschließenden Diskussion im Abschnitt 3.4 habe ich mich entschieden, das Modell, welches einen gemeinsamen Adressraum implementiert, umzusetzen. Die Funktionalität des Modells 2 ist aus meiner Sicht ausreichend, um die gewünschten Ziele

- Verringerung des Ressourcenbedarfs,
- Vermeiden von Verlangsamung von lokalen Operationen durch Cross-Prozessor-Synchronisation und
- Erhaltung der Echtzeiteigenschaften von Fiasco

zu erreichen.

Mit Hilfe von Cross-Prozessor-Tasks kann die Effizienz gesteigert werden. Globale Kern-Datenstrukturen, z.B. die Mapping-Datenbank, müssen synchronisiert werden. Um die Echtzeitfähigkeiten zu erhalten, wurde ein Synchronisationsmechanismus entwickelt, der echtzeitkompatibel ist und Helping über Prozessorgrenzen hinweg ermöglicht. Dieser Algorithmus ermöglicht Fortschritt von Threads beim Zugriff auf eine gemeinsam genutzte Ressource und vermeidet unbegrenzte Prioritätsinversion. Dieser Mechanismus wird für die Synchronisation der systemglobalen Mapping-Datenbank benötigt, damit die Echtzeiteigenschaften erhalten werden können.

Im Kapitel 5 habe ich einige interessante Aspekte der Implementierung herausgegriffen. Um die Restriktionen der ursprünglichen FiascoMP-Implementierung zu beseitigen, wurde ein gemeinsam genutzter Adressraum implementiert. Dazu wurden Datenstrukturen (Kern-Seitentabelle und Mapping-Datenbank) global für alle Prozessoren sichtbar gemacht. Für die Implementierung eines echtzeitkompatiblen Synchronisationsmechanismus waren Anpassungen und Erweiterungen an bestehenden Fiasco-Subsystemen nötig.

Im Kapitel 6 habe ich gezeigt, dass die im Abschnitt 3 genannten Ziele erreicht wurden. FiascoMP ermöglicht die parallele Nutzung mehrerer Prozessoren und erschließt damit mögliche Geschwindigkeitssteigerungen durch die Ausnutzung von Taskparallelität. Auf einem 8-Prozessorsystem konnte für eine Arbeitslast eine Geschwindigkeitssteigerung von 7,51 gemessen werden. Gegenüber dem vorher implementierten Modell A sind beim Modell 2 bei dynamischen Arbeitslasten zusätzliche Geschwindigkeitssteigerungen von bis zu 30 Prozent möglich, da die aufwändige Synchronisation der Mapping-Datenbanken unterschiedlicher Prozessoren bei der Konstruktion von gemeinsam genutztem Speicher auf Benutzerebene entfällt. Durch die Implementierung von gemeinsam genutztem Speicher auf Kern-Ebene wurde zudem das Applikationsdesign vereinfacht. Das aktuelle Kern-Modell ist für eine Implementierung von L^{4}Linux-MP geeignet. Das neue Kern-Modell bietet ausreichend Funktionalität, um darauf L^{4}Linux-MP zu implementieren. Diese Aufgabe war aber nicht Bestandteil meiner Arbeit, weshalb ich hier nur kurz eine mögliche Struktur von L^{4}LinuxMP skizzieren möchte. Das Fehlen von kernunterstützter *Thread*-Migration stellt kein Problem dar, da Linux selbst über einen Lastverteilungsmechanismus verfügt.

Im L^{4}Linux-Server laufen unterschiedliche Aktivitäten in einem *Thread*-Kontext, von mir Linux-*Threads* genannt. Zu diesen Aktivitäten zählen der Service-*Thread*, der Interrupt-*Thread* und der Paging-*Thread*. Die einzelnen Linux-*Threads* werden für die Ausführung auf einen L4-*Thread* gemultiplext. Für die Konstruktion von L^{4}Linux-MP müssen die Linux-*Threads* für alle weiteren Prozessoren dupliziert werden und auf jedem Prozessor muss im L^{4}Linux-Server ein L4-*Thread* vorhanden sein.

Für jede Linux-Task gibt es im L^{4}Linux-Server einen L4-*Thread*, auf den die einzelnen *Threads* der Linux-Task gemultiplext werden. Deshalb muss bei der Erzeugung einer Linux-Task im L^{4}Linux-Server auf jedem Prozessor ein entsprechender L4-*Thread* angelegt werden. Das ist nötig, damit L^{4}Linux selbstständig eine Lastverteilung vornehmen kann.

7.2 Ausblick

In der aktuellen Implementierung des MP-Preemption-Locks gibt es eine globale Warteliste, in der sich Lock-Anwärter eintragen. Das Problem dabei ist, dass keine lokale Aussage über das Ausführungsverhalten gemacht werden kann, da sich beliebig viele *Threads* auf anderen Prozessoren in die Warteliste eingereiht haben können. Eine prozessorlokale Warteliste könnte dieses Problem lösen. Der neue Besitzer wird dann nach dem Round-Robin-Verfahren aus diesen Listen ausgewählt. Damit ließe sich eine ge-

rechtere Vergabe des Locks erreichen, denn häufige Anforderungen von *Threads* durch den gleichen Prozessor hätten weniger Auswirkungen auf Echtzeitaufgaben auf anderen Prozessoren. Außerdem ist eine Erweiterung des MP-Preemption-Locks um einen globalen Ressourcenzustand sinnvoll. Damit lassen sich z.B. Helping-Vorgänge bei der Lock-Übergabe an einen neuen Besitzer vermeiden.

Ein interessantes Aufgabengebiet ist die nicht-transparente Thread-Migration auf Benutzerebene zu implementieren. Das Problem des vorgeschlagenen Migrationsmechanismuses ist die Sichtbarkeit des Migrationsprozesses auf der Benutzerebene. Die Sichtbarkeit muss bei der Konstruktion von Anwendungen beachtet werden und führt zu einer höheren Komplexität im Vergleich zu einem durch den Kern unterstützten Migrationsmechanismus. Es muss untersucht werden, ob es evtl. sinnvoll ist, die Schedulingparameter nach einer Migration anzupassen.

Das Messagebox-System stellt im aktuellen Modell ein Problem für die Echtzeitfähigkeit des Kerns dar. Nachrichten werden synchron verschickt. Das bedeutet, dass bis zur Verarbeitung einer Nachricht die Aktivität auf dem verschickenden Prozessor blockiert ist. Ein asynchroner Mechanismus böte den Vorteil, dass nach dem Verschicken der Nachricht die Ausführung der ursprünglichen Aktivität fortgesetzt werden kann. Es muss untersucht werden, welche Änderungen am IPC-Pfad, insbesondere für Remote-IPC, nötig wären.

Um bei bestimmten Arbeitslasten, die eine große Menge an Cross-Prozessor-Nachrichten erzeugen, Fortschritt des Gesamtsystems zu ermöglichen, kann es evtl. nötig sein, die Rate, mit der IPIs ausgelöst werden, zu begrenzen. Ein bösartiger *Thread* kann z.B. durch häufiges Verschicken von Nachrichten über Prozessorgrenzen hinweg die Gesamtgeschwindigkeit des Systems beeinflussen. Das kann sogar dazu führen, dass eine Reservierung (Zeit) für einen Echtzeitthread nicht erfüllt werden kann. Deshalb muss untersucht werden, ob z.B. eine dynamische oder statische Begrenzung vorteilhafter ist.

8 Anhang

A Algorithmus in Pseudocode

Listing 8.1: Kritischer Abschnitt

```
void Resource::sync_work() {
  lock();
  do_work(); /* ressourcenspezifisch */
  unlock();
}
```

Listing 8.2: Ergreifen des Locks

```
/* globale Synchronisation des Aufrufers */
bool Resource::try_lock(thread) {
  /* Zustand von thread: not_in */
  if(_owner || _helper) {
    enqueue_in_waiterlist(thread);
    enqueue_helper(thread);
    /* Zustand: first, wait (selected) */
    return false;
  } else {
    thread->set_state(OW_RUN);
    _owner = thread;
    return true;
  }
}
```

Listing 8.3: Einreihen in die Helferliste

```
/* Ausführung wird synchronisiert */
void Resource::enqueue_in_helperlist(thread) {
  if(thread->get_state() == DEQUEUED)
    enqueue_helper(thread);
}
```

Listing 8.4: Lock-Belegung

```
void Resource::lock() {
  /* Unterbrechbarkeit aus */
  /* Zustand: not_in */
  bool flag = true;
  self = current_thread();
  /* globale Serialisierung */
  if(!try_lock(self)) {
    /* Zustand: wait, first o. selected */
    while(flag) {
      switch(self->get_state()) {
        /* Uebergang asynchron, daher keine
         * Unterscheidung, sonst Race */
        case WAIT: /* nur bei erster Iteration */
        case SELECTED:
          /* auf Benachrichtigung warten */
          ready_dequeue(self);
          schedule();
          /* Zustand: helping, dequeued o. ow_run/ow_dis */
          break;
        case HELPING:
          switch_to(_owner);
          break;
        case DEQUEUED:
          enqueue_in_helperlist(self)
          /* Zustand: dequeued, ow_run o. ow_dis */
          break;
        case FIRST:
          if(disable_owner())
            switch_to(_owner);
          break;
        case OW_RUN:
        case OW_DIS:
          flag = false;
          break;
      }
    }
  }
  /* Unterbrechbarkeit ein */
}
```

Listing 8.5: Festlegung des neuen Besitzers

```
/* synchronisiert mit try_lock und enqueue_helper */
/* liefert false wenn es keinen neuen Besitzer
 * gibt oder next_helper im Zustand first ist */
bool Resource::set_new_owner() {
  _owner->set_state(NOT_IN);
  _helper = peek_helperlist();
  _owner = peek_waiterlist();
  if(!_owner)
    return false;
  if(_helper) {
    switch(_helper->get_state()) {
      case FIRST:
        /* ausstehender disable-Request
         * neuer Besitzer wird in
         * disable_owner gesetzt */
        _owner = NULL;
        return false;
      case WAIT:
        _helper->set_state(SELECTED);
        break;
      case HELPING:
        /* Besitzer wird in schedule aus
         * der Ready-Liste ausgetragen */
        break;
    }
    /* Invariante 2 */
    dequeue_from_waiterlist(_owner);
    _owner->set_state(OW_DIS);
    return true;
  } else {
    /* Besitzer gibt selbst Ressource frei */
    dequeue_from_waiterlist(_owner);
    _owner->set_state(OW_RUN);
    if(is_local(_owner))
      /* setzt running flag */
      ready_enqueue(_owner);
    else
      remote_ready_enqueue(_owner);
    return true;
  }
}
```

Listing 8.6: Helfer in die Helferliste einfügen

```
void Resource::enqueue_helper(thread) {
  if(thread == _owner)
    return;
  if(!has_first_helper())
    thread->set_state(FIRST);
  else
    thread->set_state(WAIT);
  helperlist_add(thread);
}
```

Listing 8.7: Deaktivierung des Besitzers

```
void Cpu::on_disable_request() {
  /* Zustaende: not_in,ow_run,ow_preempt */
  while(has_pending_requests()) {
    request = head_of_requestlist();
    resource = request->resource();
    resource->inc_dis_count();
    owner = request->owner();
    preemptee = resource->owner();
    /* Test ob Besitzer inzwischen geaendert */
    if(owner == preemptee) {
      owner->set_state(OW_DIS);
      if(owner == current_thread()) {
        Disable_Op dis_op;
        dis_op.target = resource->helper();
        dis_op.ret = DIS_SUCC;
        do_assist(&dis_op);
      } else
        send_reply(DIS_SUCC, resource->helper());
      ready_dequeue(owner);
    } else {
      /* neuer Besitzer */
      send_reply(DIS_FAIL, resource->helper());
    }
  }
}
```

Listing 8.8: Benachrichtigung eines Helfers

```
void Resource::notify_helper() {
  helper = peek_helperlist();
  /* helper im Zustand selected */
  while(is_local(helper) {
    /* Zustand von helper: wait */
    remove_from_helperlist(helper);
    helper->set_state(DEQUEUED);
    ready_enqueue(helper);
    helper = peek_helperlist();
    helper->set_state(SELECTED);
  }
  if(helper)
    send_notification(helper);
  else {
    _owner->set_state(OW_RUN);
    if(is_local(_owner)) {
      ready_enqueue(_owner);
      if(highest_prio(_owner))
        switch_to(_owner);
      else
        schedule();
    }
    else
      remote_ready_enqueue(_owner);
  }
}
```

Listing 8.9: Deaktivierung des Besitzers

```
/* Synchronisation wie set_new_owner */
bool Resource::disable_owner() {
  /* Zustand: first */
  self = current_thread();
  Request request(owner(), this);
  switch(send_disable_request(&request)) {
    case DIS_SUCC:
      self->set_state(HELPING);
      return true;
    case DIS_FAIL:
      dequeue_from_helperlist(_owner);
      if(_owner == self) {
        if(helper_avail()) {
          self->set_state(OW_DIS);
          _helper->set_state(SELECTED);
          send_notification(_helper);
        } else {
          self->set_state(OW_RUN);
          ready_enqueue(self);
        }
        return false;
      } else {
        _owner->set_state(OW_DIS);
        self->set_state(HELPING);
        return true;
      }
  }
}
```

Listing 8.10: Verschiedenes

```
bool do_assisted_notify() {
  notify_helper();
  return true;
}

void Cpu::on_kernel_entry() { /* timer, irq */
  /* Zustaende: ow_dis, ow_run */
  self = current_thread();
  Preempt_Op preempt_op(self);
  do_assist(&preempt_op);
}

void Resource::activate_owner() {
  _owner->set_state(OW_RUN);
  if(is_local(_owner)
    ready_enqueue(_owner);
  else
    remote_ready_enqueue(_owner);
}
```

Listing 8.11: Assistenzfunktionen

```
/* Deallokation des Stacks beachten */
bool Disable_Op::do_op() {
  return send_reply(this.return, this.target);
}

bool Notify_Op::do_op() {
  return do_assisted_notify();
}

bool Enqueue_op::do_op() {
  /* Argument auf lokalen Stack kopieren weil nach
  remote_ready_enqueue evtl. Deallokation */
  res = this->resource;
  prev_helper = this->helper;
  dis_count = res.dis_count;
  if(is_local(this->old_owner))
    ready_enqueue(this->old_owner);
  else {
    /* moegliche Disable-Requests an
         dieser Stelle */
    remote_ready_enqueue(this->old_owner);
  }
  if(dis_count == res.dis_count) {
    /* Zugriff auf owner und helper moeglich weil nur durch
      set_new_owner modifiziert */
    if( (res->owner == prev_helper) {
      if(res->owner->get_state() == OW_RUN) {
        switch_to(res->owner);
        return false;
      } else {
        notify_helper();
        return true;
      }
    }
    /* res->helper == prev_helper */
    switch_to(prev_helper);
    return false;
  }
  return true;
}

bool Preempt_Op::do_op() {
  return preempt_function(this.owner, this.helper);
}
```

Listing 8.12: Lock-Freigabe

```
void Resource::unlock() {
  /* Unterbrechbarkeit aus */
  /* Zustaende: ow_run, helping */
  old_owner = _owner;
  old_helper = _helper;
  if(set_new_owner()) {
    Enqueue_op op;
    op.old_owner = old_owner;
    op.helper = old_helper;
    op.resource = this;
    do_assist(&op);
    }
  } else {
    /* alter Besitzer ist in Ready-Liste, da
     * er selbst ausgeführt hat */
  }
  /* Unterbrechbarkeit ein */
}
```

Listing 8.13: Unterbrechung eines Helfers

```
bool Cpu::preempt_function(thread) {
  if( thread->has_ownership()
    && thread->get_state() == OW_DIS) {
    select_next_helper(thread, resourcelist);
  }
  /* liefert Thread mit hoechster Prioritaet */
  highest_prio_thread = prio_schedule();
  highest_prio = highest_prio_thread->prio();
  local_helper = null;
  gather_notifications(resourcelist);
  do {
    activity = 0;
    foreach(r in resourcelist) {
      while(r->helper_avail()) {
        next = r->peek_helperlist();
        if(is_local(next)) {
          if(highest_prio < next->prio()) {
            activity = 1;
            highest_prio = next->prio();
            local_helper = next;
            break;
          } else {
            next->set_state(DEQUEUED);
            r->remove_from_helperlist(next);
            ready_enqueue(next);
            if(!r->helper_avail()) {
              r->activate_owner();
              break;
            }
          }
        } else {
          next->set_state(SELECTED);
          enqueue_in_notification_queue(next);
          remove_from_resourcelist(r);
          break;
        }
      }
    }
  } while(activity);
  /* fuer jede Ressource ein Helfer oder
   * Helferliste dafuer leer, fuer eine Ressource
   * lokaler Helfer moeglich, sonst alle Helfer
   * remote */
  deliver_notifications();
  if(local_helper) {
    local_helper->set_state(HELPING);
    switch_to(local_helper);
    return false;
  } else {
    return true;
  }
}
```

Listing 8.14: Hilfsfunktionen für preempt_function

```
void select_next_helper(thread, list) {
  resource = thread->get_resource();
  current_helper = resource->helper();
  current_helper->set_state(DEQUEUED);
  resource->remove_from_helperlist(current_helper);
  ready_enqueue(current_helper);
  if(resource->helper_avail()) {
    next = resource->peek_helperlist();
    /* Benachrichtigung in lokale Liste */
    next->set_state(SELECTED);
    list->add(resource);
  } else {
    resource->activate_owner();
  }
}

void gather_notifications(list) {
  while(this->has_pending_notifications()) {
    notification = head_of_notifierlist();
    ack_notification(notification);
    list->add(notification->resource());
  }
}

void deliver_notifications() {
  /* anpassbare Bedingung um oszilierende
   * Benachrichtigungen zu vermeiden */
  while(should_send_notification()) {
    helper = head_of_notification_queue();
    send_notification(helper);
  }
}
```

B Messwerte

Tabelle 8.1: Geschwindigkeitsgewinn bei der Berechnung der Mandelbrotmenge durch zwei Threads auf einem und zwei Prozessoren im Modell 2, Taktzyklen in Milliarden

Zoomstufe	**Ein Prozessor**	**Zwei Prozessoren**	**Speedup**
1	2,97	1,49	1,99
2	3,02	1,51	2,00
3	3,07	1,54	1,99
4	3,13	1,57	2,00
5	3,19	1,60	1,99
6	3,25	1,63	1,99
7	3,32	1,66	1,99
8	3,38	1,70	1,99
9	3,44	1,73	1,99
10	3,51	1,76	1,99
		Durchschnitt	1,99

Glossar

APIC *Advanced Programmable Interrupt Controller*

IPC *Inter Process Communication*, damit wird die Kommunikation zwischen Threads bezeichnet

IPI Inter-Prozessor-Interrupt, Interrupt der von einem Prozessor auf einem anderen Prozessor ausgelöst werden kann

SMP *Symmetric Multi Processor*

TCB *Thread Control Block*, ist ein Datenstruktur in der Informationen (z.B. Zustand) über einen Thread gespeichert sind

TLB *Translation Lookaside Buffer*, im TLB werden häufig genutzte Adressübersetzungen für einen schnellen Zugriff gespeichert

TSS *Task State Segment*

Literaturverzeichnis

[BBB+90] Baron, R. V., Black, D., Bolosky, W., Chew, J., Draves, R. P., Golub, D. B., Rashid, R. F., Avadis Tevanian, Jr., and Young, M. W. Mach kernel interface manual. Technical report, School of Computer Science, Carnegie Mellon University, `ftp://ftp.cs.cmu.edu/project/mach/doc/unpublished/manual.ps`, 1990. 13

[BH03] Ray Bryant and John Hawkes. Linux scalability for large numa systems. In *Proceedings of the Linux Symposium*, 2003. 16

[ea05] Shekhar Borkar et. al. Platform 2015: Intel processor and platform evaluation for the next decade. Technical report, Intel Corp., 2005. 9

[ea06] Orran Krieger et. al. K42: Building a complete operating system. In *Eurosys 2006*, 2006. 16

[Hei01] Gernot Heiser. *Inside L4/MIPS: anatomy of a high-performance microkernel.* Operating systems group, University of New South Wales, Australia, 2001. 14

[Hoh02] Michael Hohmuth. *Pragmatische nichtblockierende Synchronisation für Echtzeitsysteme.* PhD thesis, TU-Dresden, 2002. 10

[Kau05] Bernhard Kauer. L4.sec implementation - kernel memory management. Master's thesis, TU-Dresden, 2005. 14

[Lie95] Jochen Liedtke. On μ-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, 1995. 13

[Lie96] Jochen Liedtke. Microkernels must and can be small. In *5th IEEE Workshop on Object-Orientation in Operating Systems*, 1996. 13

[Liu00] Jane W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, 2000. 32

[okl] Okl4. `http://www.ok-labs.com/`. 14

[Pet01] Michael Peter. Portierung des Fiasco Micro-Kernel auf SMP-Systeme, 2001. 17

[Pis] Pistachio microkernel. `http://l4ka.org/projects/pistachio/`. 14

[Raw97] F. L. Rawson. Experience with the development of a microkernel-based, multiserver operating system. In *The Sixth Workshop on Hot Topics in Operating Systems 1997*, 1997. 13

[Sch06] Sven Schneider. Multiprocessor Support for the Fiasco Microkernel. Master's thesis, Technische Universität Chemnitz, 2006. 17

[sys] Sysgo p4 mikrokern. `http://www.sysgo.com/services-solutions/industry-solutions/automotive-transportation/`. 14

[Uhl05] Volkmar Uhlig. *Scalability of Microkernel-based Systems*. PhD thesis, Universität Karlsruhe, 2005. 17, 24

[Völ02] Marcus Völp. Prototypical design and implementation of L4-SMP Microkernel mechanisms. Technical report, TU-Karlsruhe, 2002. 17, 21, 23

[YB] V. Yodaiken and M. Barabanov. A real-time linux. 16