Diplomarbeit

zum Thema

Leistungs-Analyse und -Optimierung des L4Linux-Systems

an der

Technischen Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Eingereicht von: Michael Peter
Geboren am: 11. 9. 1976
Geboren in: Leipzig
Matrikel-Nr.: 2479979
Eingereicht am: 30. Juni 2002

Verantwortlicher Hochschullehrer:
Prof. Dr. H. Härtig

Betreuer:
Dipl.-Inf. Michael Hohmuth

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich diese Arbeit nur mit den zugelassenen und aufgeführten Hilfsmitteln und ohne fremde Hilfe erstellt habe.

Dresden, 30. Juni 2002

Michael Peter

Inhaltsverzeichnis

1. Einleitung 1

1.1. Motivation 1
1.2. Gliederung 1
1.3. Danksagung 1

2. Stand der Technik 2

2.1. Prozessor-Implementierungen 2
2.1.1. Cache und Speicherhierarchie 2
2.1.2. Branch Prediction und Pipelines 3
2.1.3. Superskalarität und Out-of-order-Ausführung 5
2.2. Fiasco und Echtzeitfähigkeit 7
2.3. Spezialisierte Funktionen 8
2.4. Implementierung in Assembler 11
2.5. C-Aufrufkonvention 12
2.6. Systemaufrufe 17
2.7. Systemveränderung durch IPC 20
2.8. Unterbrechbare IPC in Fiasco 22

3. Entwurf 28

3.1. Fastpath für Spezialfälle von IPC 28
3.2. Multiprozessor 31
3.2.1. Daten Partitionierung 34
3.2.2. Speicherbasierte Protokolle 35
3.2.3. IPC-Fastpath 38
3.3. IPC-Handshake 38

4. Implementierung 40

5. Ergebnisse und Bewertung 41

5.1. Facts and Figures 41
5.2. Werkzeuge 42
5.3. Bewertung und Ausblick 42

A. Fastpath, Einzelprozessor 44

B. Fastpath, Mehrprozessor 57

Literaturverzeichnis 62

Abbildungsverzeichnis

1. Pipeline . . . 4
2. Superskalare Microarchitektur. . . . 6
3. Pseudo-Code für Threadumschaltung . . . 9
4. Verkettete Frames . . . 14
5. C-Aufrufkonvention, Codebeispiel x86 . . . 16
6. C-Aufrufkonvention, Stack-Layout . . . 18
7. Syscall, C-Implementierung . . . 19
8. Syscall, Assembler-Implementierung . . . 19
9. `call`-IPC, Empfänger bereit . . . 22
10. Rendezvous, IPC-Empfang . . . 23
11. Rendezvous, IPC-Senden . . . 25
12. IPC-Handshake . . . 26
13. Fastpath Flussdiagramm . . . 29
14. `call`-IPC, Empfänger bereit . . . 31
15. Zustände des Thread-Locks für Multiprozessoren . . . 32
16. Zustände des Thread-Locks für Einzelprozessoren . . . 32
17. Flussdiagramm für Mehrprozessor-Threadlock . . . 33
18. Bakery-Algorithmus . . . 36
19. Lamports schneller Algorithmus . . . 37
20. Rendezvous, IPC-Senden . . . 39
21. Multi-Fastpath Flussdiagramm . . . 40

Tabellenverzeichnis

1. Registernutzungskonvention bei *x86* . . . 15
2. `write-read`-Synchronisation . . . 24
3. Kosten für IPC [Takte, alle Werte für Roundtrip]. . . . 27
4. Kosten Scheduling und Threadumschaltung [Takte]. . . . 27
5. Kosten ausgewählter Instruktionen . . . 34
6. Kosten für den Eintritt in den kritischen Abschnitt. . . . 36
7. Metriken der Mehrprozessorvarianten. . . . 41
8. Metriken für die Einzelprozessorvariante. . . . 41
9. Kosten für Kernein- und -austritt. . . . 43

1. Einleitung

1.1. Motivation

Auf Mikrokernen basierte Architekturen sind ein Weg, komplexe Systeme zu strukturieren. Sie dienen als Basis für Systeme mit unterschiedlichsten Eigenschaften. Das am Lehrstuhl entwickelte DROPS-System vereint beispielsweise eine Timesharing-Komponente mit echtzeitfähigen Bestandteilen. In Zukunft zeichnet sich eine Tendenz hin zu sicherheitskritischen Anwendungen ab, deren Integrität auch in der Anwesenheit nicht vertrauenswürdiger Bestandteile garantiert werden muss.

Mikrokerne erreichen diese Flexibilität durch Beschränkung auf unbedingt notwendige Minimalfunktionalität. Im Falle der L4-Mikrokernfamilie sind das Adressräume, Aktivitätsträger und ein sicherer Mechanismus zum Austausch von Nachrichten. Andere Funktionalität kann als Anwendung je nach Bedarf außerhalb des Kerns implementiert werden.

Existierende Implementierungen L4-kompatibeler Kerne sind sehr stark auf ein Anwendungsfeld hin optimiert. So hat der in Dresden entwickelte Fiasco-Kern sehr gute Echtzeiteigenschaften, ist aber der Karlsruher Hazelnut-Implementierung mit Hinblick auf die IPC[1] -Leistung klar unterlegen. Letztere zeigt wiederum ein sehr gutes durchschnittliches Verhalten, hat aber vereinzelt extrem lange Latenzzeiten, was sie für Anwendungen mit Echtzeitanforderungen unbrauchbar macht. Diese Situation ist unbefriedigend, wenn sowohl ¡Leistung wie auch Echtzeitunterstützung benötigt werden.

Das Anliegen dieser Arbeit ist es zu untersuchen, ob gutes Echtzeitverhalten zwangsläufig schlechte Nachrichtenleistung bedingt. Dazu soll die in einem ersten Schritt untersucht werden, welchen Einfluss die Fiasco zugrunde liegende nicht-blockierende Synchronisation auf das IPC-Subsystem hat. Auf den in diesem Schritt gewonnenen Erkenntnissen aufbauend, soll danach eine möglichst leistungsfähige Implementierung für häufig auftretende Fälle von IPC entwickelt werden. Diese Bemühungen resultierten in einer Lösung, welche die originale Implementierung um mehr als den Faktor neun, eine bestehende Optimierung immer noch um den Faktor vier schlugen und mit den ebenfalls auf Leistung optimierten *Hazelnut*-Kernen nahezu gleichzog.

1.2. Gliederung

Die Arbeit ist wie folgt gegliedert: Kapitel 2 gibt im ersten Teil einen Überblick über Optimierungsmöglichkeiten, die auch unabhängig von Fiasco anwendbar sind. Nachfolgend wird die originale IPC-Implementierung näher erläutert.

Kapitel 3 beschreibt den Entwurf der Optimierung und überträgt die Lösung ebenfalls auf die Multiprozessorimplementierung. Ein ausgewähltes Implementierungsdetail wird in Kapitel 4 vorgestellt, bevor in Kapitel 5 die Ergebnisse dargelegt werden.

Im Anhang ist schließlich der Quellcode für den optimierten IPC-Fastpath angefügt.

1.3. Danksagung

An dieser Stelle möchte all diejenigen erwähnen, ohne die diese Arbeit nicht möglich gewesen wäre. Ohne Wertung durch die Reihenfolge der Nennung gilt mein Dank ganz besonders meinen beiden Betreuern Prof. Härtig und Michael Hohmuth, sowie Frank Mehnert, Jean

[1] **I**nter**p**rocess **C**ommunication

Wolter, Sebastian Schönberg und Volkmar Uhlig. Sie alle standen mit bewundernswerte Ruhe jederzeit für alle Arten von Fragen zur Verfügung.

2. Stand der Technik

2.1. Prozessor-Implementierungen

Moderne Prozessorimplementierungen nutzen Caches, tiefe Pipelines und superskalare Rechenwerke, um größtmögliche Leistung zu erbringen. Neben einer kurzen Darstellung der Konzepte wird in den folgenden Abschnitten diskutiert, welche Konstruktionsprinzipien zu performantem Code führen.

Optimierungen für die einzelnen Systemkomponenten sind teilweise konträr. Sie kollidieren zudem oftmals mit Kriterien wie Wartbarkeit oder Erweiterbarkeit.

2.1.1. Cache und Speicherhierarchie

Die von Gordon Moore geprägte Regel, dass Prozessoren ihre Leistungsfähigkeit innerhalb von 18 Monaten verdoppeln, hat in der letzten Dekade erstaunlicherweise ihre Gültigkeit behalten. Die Größe der Hauptspeicher wuchs ebenfalls in nie für möglich gehaltener Geschwindigkeit, während die Speicherzugriffszeiten nahezu konstant blieben. Das Speichersubsystem stellt damit in aktuellen Rechnern eine kritische Komponente dar, die zu einer schwer überwindbaren Leistungsbarriere werden kann. Die aktuellen Entwicklungen lassen mit großer Sicherheit vermuten, dass diese Konstellation — schnelle Prozessoren mit großen, aber nur mit erheblicher Latenz zugreifbaren Speichern — in absehbarer Zukunft vorherrschend sein wird. Fortschritte sind nur in der Speicherzugriffsbandbreite erkennbar.

Programme zeigen häufig ein charakterisches Speicherzugriffsverhalten. Zugriffe erfolgen für begrenzte Zeit auf eng begrenzte Bereiche. Zurückzuführen ist dieses Verhalten auf Programmstrukturen wie Schleifen und logisch zusammengehörige Datenstrukturen. Die Instruktionsfolge einer Schleife wird häufig sehr oft hintereinander ausgeführt. In dieser Zeit werden die abzuarbeitenden Befehle aus dem in seiner räumlichen Ausdehnung begrenzten Schleifenrumpf geladen. Datenstrukturen in nicht-trivialen Programmen bestehen oftmals aus einer Sammlung zusammengehöriger einfacherer Strukturen, die in räumlich nahe zusammenliegenden Speicherbereichen angeordnet sind. Veränderungen an der logischen Struktur werden mittels Modifikation mehrerer einfacher Teilstrukturen realisiert. Im Zuge einer Operation auf einer komplexen Struktur wird somit wiederum auf die zusammenliegenden Bestandteile zugegriffen.

Das oben beschriebene Zugriffsverhalten ermöglicht eine Beschleunigung durch die Vorhaltung von Daten in schnellen Zwischenspeichern, englisch als Cache bezeichnet. Die Zwischenspeicher zeichnen sich durch eine geringe Zugriffslatenz aus, sind aber in ihrer Größe beschränkt. In aktuellen Systemen wird eine Hierarchie von Zwischenspeichern implementiert. Mit der Entfernung zur CPU steigen sowohl ihre Größe wie auch ihre Zugriffsverzögerung an.

Caches werden in Blöcken[2] verwaltet. Daten werden am besten so organisiert, dass zur Ausführung einer logischen Operation auf eine minimale Anzahl von ihnen zugegriffen werden muss.

Ein Programm läuft mit maximaler Geschwindigkeit, wenn alle benötigen Daten im am nächsten zur CPU liegenden Cache liegen. Die Ausführungszeit nimmt erheblich zu, wenn

[2] englisch: cache line, typische Größe: 32-256 Bytes

Daten nicht in diesem Cache gefunden werden und aus niederen Ebenen der Speicherhierarchie geladen werden müssen. In [1] wurden für ein System[3] die Zeiten für das Laden aus verschiedenen Stufen der Speicherhierarchie ausgemessen. Instruktionsfolgen sollten somit so beschaffen sein, dass sie komplett in den Cache passen. Der am häufigsten auszuführende — und damit am meisten zur Gesamtausführungszeit beitragende — Instruktionsstrom wird für maximale Kompaktheit am besten linear angeordnet.

Die Caches sind als Assoziativspeicher ausgelegt. Das System versucht, sich dynamisch durch die Einlagerung von kürzlich benötigten Daten an das aktuelle Programmverhalten anzupassen. Dieser Vorgang ist für das Programm transparent. Er ist auch nur schwer komplett vorhersagbar. Echtzeitsysteme, die genaue Kenntnis über das Ausführungsverhalten von Programmteilen benötigen, stehen vor einem schwerwiegenden Problem. Die radikalste Lösung verzichtet komplett auf Caches. Die Ausführungszeit ist dann exakt bekannt. Sie ist somit nur von der bekannten statischen Instruktionsfolge abhängig. Gleichzeitig sind die Leistungsvorteile von Caches nicht mehr verfügbar.

Andere Techniken zur Verbesserung der Voraussagbarkeit nutzen die Organisationsstruktur von Caches. Daten können in Abhängigkeit von ihrer physischen Adresse nur in bestimmten Teilen eines Caches platziert werden. Man spricht in diesem Zusammenhang auch von einer *set associative* Organisation. Durch die gezielte Verteilung von Daten auf physische Adressen kann erreicht werden, dass die von zeitkritischen Operationen benötigten Daten nicht aus dem Cache verdrängt werden. Auf diesem Prinzip beruhende Verfahren sind als *cache coloring* bekannt.

Die Latenz eines Speicherzugriffs verzögert die Ausführung eines Programmes nicht, wenn soviel Parallelität in der Instruktionsfolge vorhanden ist, dass sie mit anderen Aktivitäten verdeckt werden kann. Das dynamische Laden des Caches erfolgt beim ersten Zugriff auf Daten. Diese Daten werden für den die weitere Abarbeitung benötigt und sind damit nur in den seltensten Fällen durch Parallelität verdeckbar. Notwendig ist hingegen ein Mechanismus, der das Laden von Daten bewirkt, ohne dass diese Daten in Kontrollfluss- oder Datenabhängigkeiten innerhalb der aktuellen Programmausführung in Erscheinung treten. Moderne Prozessoren bieten solche Mechanismen in Form von *prefetching*. Daten, von denen bekannt ist, dass sie in unmittelbarer Zukunft benötigt werden, können so in den Cache geladen werden. Parallel läuft das Programm weiter, die Daten werden erst später benötigt.

2.1.2. Branch Prediction und Pipelines

Abgesehen von preiswerten Mikrokontrollern arbeiten alle aktuellen Prozessoren Instruktionen mit Hilfe einer Pipeline ab. Einzelne Befehle werden in einfache Teilaufgaben aufgeteilt und diese nacheinander abgearbeitet. Eine einzelne Instruktion befindet sich so über mehrere Takte verteilt in verschiedenen Stufen der Abarbeitung. Mehrere Befehle können sich gleichzeitig in unterschiedlichen Phasen der Abarbeitung befinden und damit die Effizienz der Instruktionsabarbeitung durch überlappende Parallelität erhöhen. Abbildung 1 zeigt mehrere Befehle in unterschiedlichen Stufen der Pipeline. Das Beispiel zeigt eine Standardpipeline, wie sie zum Beispiel im MIPS R2000 implementiert ist. Sie besteht aus fünf Stufen: *i*nstruction *f*etch, *i*nstruction *d*ecode, *ex*ecute, *mem*ory access und *w*rite *b*ack.

Die Latenz einer Befehlsabarbeitung erhöht sich auf die Anzahl der Stufen der Pipeline, der Durchsatz pro Takt bleibt konstant. Wegen der einfachen Teilaufgaben kann der Takt und

[3] Dual PIII, 450Mhz

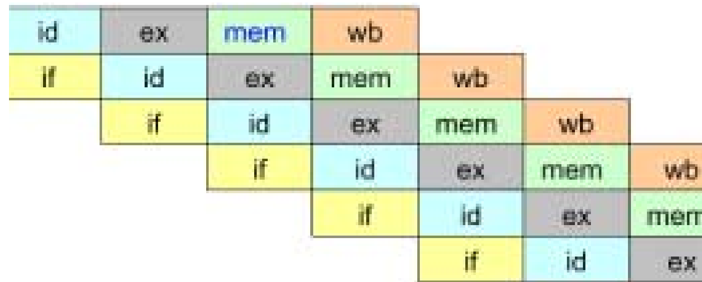


Abbildung 1: Pipeline

damit auch der Durchsatz signifikant erhöht werden. Diese Parallelität kann allerdings nur ausgenutzt werden, wenn durch einen steten Instruktionsstrom kontinuierlich neue Befehle in die Pipeline eingespeist werden. Eine Unterbrechung der Einspeisung führt zu einer in einigen Stufen nicht gefüllten Pipeline.

Bedingte Sprünge verändern den Instruktionsstrom in potentiell nicht voraussagbarer Art und Weise. Zwei Probleme lassen sich darauf zurückführen: Es muss zeitig in der Pipeline erkannt werden, dass es sich um einen Sprung handelt und das Ziel dieses Sprunges sollte frühzeitig mit möglichst hoher Wahrscheinlichkeit vorausgesagt werden.

Ein Sprung sollte zeitig — idealerweise in der ersten Stufen — in der Pipeline erkannt werden, da für den nachfolgenden Befehl in Abhängigkeit der Ausführung des Sprunges zwei Kandidaten zur Verfügung stehen. Im Falle der nichtlinearen Verzweigung wird sonst der falsche, nachfolgende Befehl in die Pipeline eingespeist. Einige Architekturen zeigen dieses Verhalten explizit.[4] In der Implementierungen anderer Architekturen ist das Verzweigen aufgrund des Einlesen eines falschen Befehls mit zusätzlichen Kosten versehen. Mehraufwand in Form von zeitig gewonnenen Dekodierungshinweisen, zum Beispiel im Cache, hilft, die Kosten von ausgeführten Sprüngen zu senken. Sobald ein Sprung und die damit mögliche Kontrollflussverzweigung erkannt ist, kann mittels Sprungvorhersage versucht werden, die Pipeline mit gültigen Befehlen zu füllen.

Potentiell sind Sprünge durch ihre direkten Vorgängerinstruktionen beeinflussbar. Das Ergebnis einer Instruktion liegt aber häufig erst relativ spät in der Pipeline vor. Erst zu diesem Zeitpunkt ist mit Sicherheit das Sprungziel bekannt. Vom Zeitpunkt des Erkennen eines Sprunges bis zur sicheren Kenntnis der Sprungbedingung und damit des Sprungzieles ist das Füllen der Pipeline spekulativ. Möglicherweise werden Befehle eingespeist, die im logischen Programmablauf niemals ausgeführt werden. Die falschen Befehle in der Pipeline dürfen keine Auswirkungen auf den sichtbaren Zustand der Architektur haben — die Befehle in der Pipeline tragen nicht zum Programmfortschritt bei.

Programme zeigen in ihrem Sprungverhalten oftmals charakteristische Eigenschaften. Einzelne Sprünge werden deutlich häufiger oder seltener als 50% aller Fälle ausgeführt. Beispielsweise werden Schleifen typischerweise mehr als einmal durchlaufen. Der abschließende Sprung wird somit bei jedem Durchlauf mit Ausnahme der letzten Iteration ausgeführt. Mit dem Wissen um das wahrscheinliche Ausführungsverhalten eines Sprunges kann die Pipeline mit den am wahrscheinlichsten nachfolgend ausgeführten Befehlen geladen werden. Für den hoffentlich seltenen Fall einer Falschvoraussage muss trotzdem die Sprungbedingung getest

[4]Der wahrscheinlich bekannteste Vertreter ist der *branch delay slot* in der MIPS-Architektur.

werden, bevor die spekulativ geladenen und vielleicht teilweise ausgeführten Instruktionen den sichtbaren Zustand des Prozessors verändern.

Das dynamische Verhalten eines Programms ist in der Mehrheit aller Fälle vom aktuellen Aufführungskontext abhängig. Es ändert sich mit den konkreten Eingaben eines Programmlaufes. Damit ist es notwendig, ebenfalls dynamisch dieses Verhalten vorauszusagen. Aktuelle Prozessorimplementierungen versuchen dies mit komplexen Vorhersagemechanismen aufbauend auf dem bisherigen Ausführungsverhalten zu erreichen.

Prinzipiell sollte versucht werden, datenabhängiges Verhalten ohne bedingte Sprünge zu implementieren. Teilweise gelingt dies durch Instruktionen — außer Sprüngen — die nur bedingt ausgeführt werden. Die von den verschiedenen Architekturen und Architekturvarianten angebotene Unterstützung reicht von äußerst rudimentären Unterstützung (i386)[5] über eingeschränkte[6] bis hin zur vollständigen bedingten Ausführung (Itanium, ARM). Sind bedingte Sprünge aufgrund komplexer Kontrollflussbeeinflussung dennoch notwendig, kann ein weitestgehend vorhersagbares, das heißt sich wenig änderndes und dem Standardverhalten entsprechendes, Verhalten Leistungseinbrüche vermeiden helfen. Beispielsweise werden vorwärtsgerichtete Sprünge in Ermangelung besseren Wissens oftmals als nicht ausgeführt vorhergesagt. Instruktionsfolgen können mit Hinblick auf die Verarbeitung in Pipelines am effizientesten abgearbeitet werden, wenn sie als lineare Folge ohne oder mit nicht ausgeführten Sprüngen vorliegen. Eine Optimierung besteht in der Instrunktionsanordnung für häufige Fälle in dieser Art und Weise.

2.1.3. Superskalarität und Out-of-order-Ausführung

Die im vorangegangenen Abschnitt beschriebene einfache Pipeline führt Befehle gleichzeitig in verschiedenen Stadien der Abarbeitung aus. Leistungssteigerungen können durch eine Verlängerung der Pipeline erreicht werden. Die einzelnen Stufen weisen geringere Schaltverzögerungen auf und laufen somit bei höheren Takten. Pipelines sind aufgrund der begrenzten Unterteilbarkeit der Befehlsabarbeitung in ihrer Länge beschränkt. Typische Werte sind fünf bis 20 Stufen. Mit wachsender Länge steigt zudem die Anfälligkeit für Abhängigkeiten innerhalb der Pipeline. Befehlsabhängigkeiten von gleichzeitig in der Pipeline abgearbeiteten Instruktionen führen zu schlecht ausgelasteten Pipelines.

Die daraus resultierende Weiterentwicklung war die Verwendung mehrerer paralleler Pipelines. Die Tiefe der parallelen Pipelines bleibt moderat und damit auch ihre Anfälligkeit gegenüber Pipelinestillständen nach Abhängigkeiten. Der Instruktionsstrom muss spezielle Eigenschaften haben, um effizient in einer Doppelpipeline abgearbeitet werden zu können. Komplexe Befehle sind selten in allen Pipelines abarbeitbar. Das Scheduling von Instruktionen mit Hinblick auf die Einspeisung in Mehrfachpipelines ist maßgeblich an der Ausführungsgeschwindigkeit beteiligt. Jochen Liedtke hat mit seinen Assembler-L4-Kernen eindrucksvoll die Optimierungspotentiale dieser Technik bewiesen.

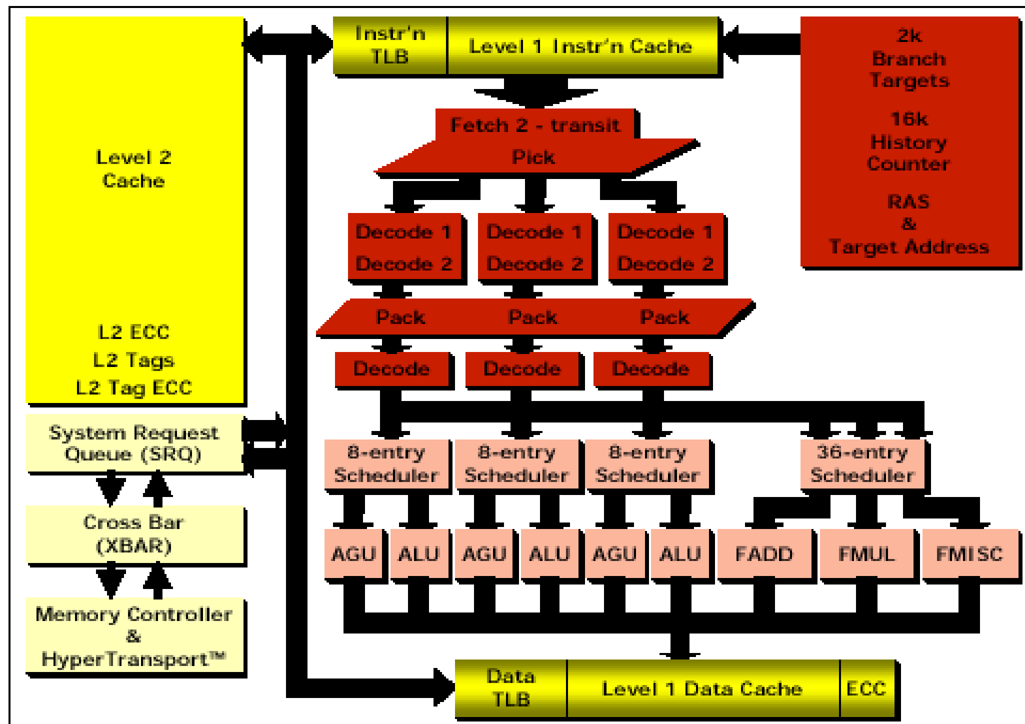
Abhängigkeiten im Befehlsstrom sind oftmals nur lokal. Instruktionen vor und nach den voneinander abhängigen Instruktionen können oftmals unabhängig und damit potentiell parallel ausgeführt werden. Diese Parallelität kann von *in-order-execution*[7]-Implementierungen nicht ausgenutzt werden. Befehle, die nach einer Abhängigkeit folgen, müssen erst die Auflösung

[5] i386 stellt *setcc* zur Verfügung.

[6] i686 bietet spezielle bedingte *mov*-Befehle.

[7] Die Instruktionen werden gemäß ihrer Reihenfolge in die (mehrfach ausgelegte) Pipeline eingespeist und auch strikt in dieser Reihenfolge abgearbeitet.

dieser abwarten. *Out-of-order*-Prozessoren verfolgen dagegen ein datengetriebenes Modell. Instruktionen werden in Puffer eingelesen und ausgeführt, sobald alle Abhängigkeiten aufgelöst wurden. Es ist möglich, dass Instruktionen, für die alle Daten bereitstehen, zeitiger ausgeführt werden als Befehle, die zeitiger im Instruktionsstrom standen, aber noch auf Eingabewerte warten.



Die Stufen der Pipeline sind als horizontal Strukturen erkennbar. In den letzten Stufen halten Puffer (hier als Scheduler bezeichnet) mehrere Instruktionen bereit, die bei einer verfügbaren Verarbeitungsresource (ALU) und allen benötigten Argumenten abgearbeitet werden.

Abbildung 2: Superskalare Microarchitektur.

Der Instruktionsstrom für *out-of-order*-Prozessoren muss nicht so strikten Kriterien genügen, wie dies für *in-order*-Implementierungen der Fall ist. Die dynamische Zuteilung von Befehlen auf Ausführungsresourcen sorgt automatisch für eine gute Auslastung der Rechenwerke. Latenzen — z.B. aufgrund von Cachefehlzugriffen — können mit der Ausführung unabhängiger Befehle besser überbrückt werden. Für den Programmierer ist diese Reihenfolgevertauschung vollkommen transparent. Der Prozessorkern trägt dafür Rechnung, dass die Ergebnisse einer Berechnung erst dann als Architekturzustand sichtbar werden, wenn alle im Instruktionsstrom davor auftretenden Befehle auch abgearbeitet wurden. Spekulative Ausführung von Code, zum Beispiel nach der unsicheren Vorhersage eines Sprunges, lässt sich als Verwerfen der Ergebnisse der falsch voraussgesagten Instruktionen implementieren. Falsch vorhergesagte Sprünge führen aber nichtsdestotrotz zu einer verlangsamten Programmausführung.

Die Ausführung von Instruktionsfolgen in dynamischen Prozessorkernen wird von kom-

plexen Befehlssatzarchitekturen erschwert. Die Prüfung von Abhängigkeiten ist umso aufwendiger, je mehr Möglichkeiten in Form von Adressierungsvarianten, Operandengrößen und Nebeneffekten beachtet werden müssen. Um dennoch die unbestrittenen Vorteile dynamischer Ausführung nutzen zu können, übersetzen moderne Prozessoren die Befehle des Programms in einfachere interne Instruktionen. Diese besitzen wenige, explizit sichtbare Abhängigkeiten und können einfacher unter Beachtung aller notwendigen Beschränkungen ausgeführt werden. Der Befehlsstrom der originalen ISA[8] wird durch spezielle Dekoder in das interne Format umgewandelt. Sind die Dekoder nicht in der Lage genug Instruktionen zu übersetzen, kann der *out-of-order*-Kern nicht die maximale Parallelität des Befehlsstroms ausnutzen. Es existieren Beschränkungen, welche Kombinationen von Original-Befehlen von den Dekodern innerhalb eines Taktes verarbeitet werden können. Der Programmierer hat dafür Sorge zu tragen, dass es aufgrund dieser Einschränkungen nicht zu einem Engpass aufgrund fehlender interner Instruktionen kommt. Das Scheduling von Instruktionen mit Hinblick auf Datenabhängigkeiten ist jedoch nicht mehr notwendig. Der Kern wird automatisch parallel ausführbare Befehle aus dem Instruktionsstrom auswählen. Abbildung 2 zeigt am Beispiel eines modernen AMD-Prozessors[9] die Organisation einer *out-of-order*-Implementierung.

2.2. Fiasco und Echtzeitfähigkeit

Echtzeitfähigkeit ist die Eigenschaft, (Nutzer-) Aktivitäten zu exakten vorhersagbaren Zeiten auszuführen. Die Güte dieser Eigenschaft ist dabei die maximale Abweichung von der zugesicherten Ausführungszeit.

Komplexe Operationen müssen im Kern mit (pseudo-)parallelem Ausführungsmodell synchronisiert werden. In den klassischen Unix-Kernen wurde dies dadurch erreicht, dass Threads, die im Kern Code ausführen, niemals verdrängt werden. Kerndatenstrukturen können somit ohne speziellen Schutz manipuliert werden, aus der Sicht eines Prozesskontextes ist die Veränderung atomar.[10] Die Aktivierung einer User-Task kann somit um die maximale Ausführungszeit eines Kernpfades verzögert werden.

Eine Möglichkeit, dies zu vermeiden, ist ein Kern, der nur Operationen zur Verfügung stellt, die in einer definierten, begrenzten Zeit ausgeführt werden können. Diese Option ist allerdings nur in den seltensten Fällen wählbar. Ist eine Schnittstelle vorgegeben, so muß die darin zugesicherte Funktionalität auch implementiert werden. Die Schnittstelle monolithischer Kerne ist so komplex, dass dort eine Vereinfachung mit dem Bruch sämtlicher Kompatibilität einhergehen würde. Mikrokerne begrenzen die angebotene Funktionalität auf ein notwendiges Minimum. Die L4-Schnittstellenversion V4 wurde unter Berücksichtigung dieser Punkte entworfen. In der von Fiasco implementierten Version 2 sind jedoch auch potentiell lang andauernde Operationen definiert. Das Löschen einer task impliziert zum Beispiel das Löschen aller Subtasks. Die Ausführungszeit steigt mit der Anzahl der zu löschenden Tasks.

Der häufigere Ansatz besteht darin, komplexe Operation zu unterbrechen, um zeitkritische Funktionen ausführen zu können, sobald diese rechenbereit werden. Dabei muss sichergestellt werden, dass es zu keinen Interaktionen zwischen gleichzeitig aktiven Threads in kritischen Abschnitten gibt. Dies ist erreichbar, indem nur an ausgewählten Punkten Unterbrechungen zugelassen werden. Diese Unterbrechungspunkte erfordern zusätzlichen Entwicklungsaufwand.

---

[8] Befehlssatzarchitekture, englisch: *i*nstruction *s*et *a*rchitecture

[9] AMD-Hammer,available from URL: `http://www.amd.com`

[10] Werden Datenstrukturen jedoch aus unterbrechenden Kontexten (Interrupts und Bottom Halves) heraus verändert, so muss dieser Zugriff durch explizite Locks serialisiert werden.

Änderungen in Datenstrukturen wie auch im Kontrollfluss ziehen fehleranfälligen Anpassungen nach sich.

Durch den Einstatz von lock-freier Synchronisation können die Unterbrechungspunkte praktisch über den gesamten Code verteilt werden. Lock-freie Algorithmen sind für einige Datenstrukturen bekannt, neigen aber zu großer Komplexität. Ihre effiziente Implementierung setzt zudem Unterstützung durch die zugrundeliegende Rechnerarchitektur voraus. Lock-freie Synchronisation kann gänzlich ohne Preemptionsverbot implementiert werden, wenn hinreichend sematisch starke atomare Operationen wie beispielsweise *compare-and-swap*[11] zur Verfügung stehen. Fiasco benutzt lock-freie Synchronisation nur zum Schutz einfacher Datenstrukturen wie Zustandswörtern und einfach verketteten Listen.

Einfacher in der Handhabung ist ein für den Entwickler transparenter oder zumindest sematisch einfacher Mechanismus. Eine mögliche Strukturierung kann mittels Locks erfolgen. Ein Lock wird einer Resource zugeordnet und serialisiert Manipulationen auf ihr. Vor dem kritischen Abschnitt wird das Lock gesetzt, nach ihm freigegeben.

Fiasco erreicht Unterbrechbarkeit durch die Verwendung von nichtblockierender Synchronisation. Kritische Abschnitte werden von *helping* Locks geschützt. Das Halten eines Locks verhindert nicht, dass ein Thread in einem kritischen Abschnitt verdrängt wird. Vielmehr signalisiert ein Thread durch das Halten des Locks, dass die dadurch geschützten Resourcen manipuliert werden.Versucht ein weiterer Thread den durch das Lock geschützten Abschnitt zu betreten, so schaltet er zum aktuellen Besitzer um. Er wiederholt dies so oft, bis dieser das Lock freigegeben hat. Dieser *helping* genannte Mechnismus implementiert *priority inheritance*. Es gilt die Invariante des Locks, dass der aktuelle Lockhalter lauffähig und in der Lage sein muss, das Lock unabhängig von äußeren Einflüssen (IPC, etc.) freizugeben. Das schließt freiwilliges Blockieren in Erwartung einer IPC und Seitenfehler aus. Mit dieser Konvention kann sichergestellt werden, dass Helfen nach begrenzter Zeit — der Ausführungszeit für den kritischen Abschnitt — zum Betreten des kritischen Abschnitts führt.

Dieses Schema garantiert, dass kritische Abschnitte, die keine gemeinsamen Ressourcen manipulieren — und somit nicht versuchen, gemeinsame Locks zu setzen — sich gegenseitig nicht behindern. Ein Schutz aller kritischen Abschnitte durch Ausschluss von Preemption kann dies nicht garantieren. Außerdem lässt sch die maximale Verzögerung bis zum Eintritt in einen kritischen Abschnitt sicher berechnen.

2.3. Spezialisierte Funktionen

Die meisten Funktionen in Fiasco sind auf wenige Anwendungsfälle beschränkt. Beispielhaft ist die Funktion `sys_ipc` zum Versenden von IPC, die genau an einer Stelle — nach dem Eintritt eines Systemcalls — aufgerufen wird. Es existieren jedoch auch einige generellere Funktionen wie Threadumschaltung und Scheduling, die vergleichsweise häufig benutzt werden. Funktionen wie `switch_to` (Threadumschaltung, 29 Aufrufe) oder `schedule` (Auswahl des nächsten auszuführenden Threads, 7 Aufrufe) werden in unterschiedlichen Situationen unter unterschiedlichen Bedingungen aufgerufen. Das erwartete Verhalten muss unter allen Umständen richtig implementiert werden.

Die folgende Diskussion geht von einem aktuellen Thread aus, der zu einem Zielthread umschalten möchte. Abbildung 3 zeigt die derzeitige Implementierung in Pseudo-Code. Die-

[11] Ein Wert in einem Register wird mit dem Wert einer Speicherzelle verglichen. Stimmen sie überein, wird die Speicherzelle mit einem weiteren Registerwert überschrieben. Der Austausch des Speicherzellenwertes erfolgt atomar.

```
switch_to(dest_thread):
     if dest_thread  is locked:
         dest_thread = current owner of dest_thread
     if dest_thread is not runnable:
         return failure
     if fpu regs are active:
         save fpu state, mark fpu regs invalid
     if dest_thread is runnable and not in run queue:
         enqueue dest_thread in run queue
     if current thread has different
        address space than dest_thread:
        switch address space

     save context of current thread into current tcb

     load register context of dest_thread from its tcb
```

Abbildung 3: Pseudo-Code für Threadumschaltung

se Operation ist nur erlaubt, wenn der Zielthread gewissen Bedingungen genügt. Er muss lauffähig und darf nicht gelockt sein. Bei der Umschaltung ist es je nach Situation notwendig, weitere Operationen vorzunehmen. Teilweise müssen Adressräume gewechselt werden. Manchmal ist das Einfügen in Verwaltungsstrukturen notwendig.

Ist aufgrund des bisherigen Ausführungspfades Wissen über die aktuelle oder zukünftige Bedingungen vorhanden, kann auf Tests und teilweise aufwendige Operationen verzichtet werden. Nachfolgend sind die wichtigsten Bedingungen für die Threadumschaltung kurz dargestellt.

Eine (teure) Adressraumumschaltung ist nicht notwendig, wenn der Zielthread vor der nächsten Threadumschaltung nicht in den Nutzeradressraum zurückkehrt. Dies ist für Threads, die ein Thread-Lock halten und denen geholfen wird sowie immer für *idle*-Threads der Fall. Bei der Threadumschaltung innerhalb einer Task kann ebenfalls der aktuelle Adressraumkontext[12] beibehalten werden.

Sobald bekannt ist, dass ein Thread nicht gelockt sein kann, ist der Test auf diese Bedingung überflüssig. Diese Situation ergibt sich, wenn der aktuelle Thread den Zielthread selber gelockt hatte und zum Beispiel durch Abschalten von Interrupts sicher weiß, dass zwischenzeitlich keine Threadumschaltung stattgefunden hat. Nur nach einer solchen Threadumschaltung kann ein höher priorisierter Thread Operationen auf dem Threadlock durchgeführt haben.

Die x86-Architektur stellt ab dem *i486* ein integriertes Gleitkommasubsystem zur Verfügung.[13] Die zugehörigen Gleitkommaregister sind für den Anwendungsprogrammierer sichtbar und somit Teil des Ausführungskontextes eines Programms. Bei einer Thread-Umschaltung müssen alle im Nutzer-Modus verfügbaren Ressourcen gesichert werden, damit die Programmausführung später mit genau diesen Werten fortgesetzt werden kann. Die Mehrzahl aller Prozesse in einem klassischen Timesharingsystem beinhaltet keine Gleitkommarechnungen.

[12] TLB-Inhalt

[13] Davor waren Lösungen auf Basis von Koprozessoren erhältlich.

Der Prozessor kann so konfiguriert werden, dass der Zugriff auf Gleitkommaregister eine Exception auslöst. Mit dieser Zugriffsbeschränkung kann auf das Speichern der Gleitkommaregister in den meisten Fällen verzichtet werden. Nach einer Threadumschaltung läuft der neue Thread mit abgeschaltetem Registerzugriff. Ein Zugriff löst eine Exception aus. Der letzte gültige Zustand wird geladen, der Gleitkommaregisterzustand als aktiv markiert und die Programmausführung mit zugreifbaren Registern fortgesetzt. Bei der Umschaltung von einem Thread mit aktiven Gleitkommaregistern werden diese gesichert. Die sofortige Sicherung ist eigentlich nicht zwingend notwendig. Es reichte auch, wenn der Zustand beim nächsten Zugriff gesichert würde, genau vor der Wiederherstellung des dann aktuellen Zustandes. Dieses Verhalten erspart sogar in einigen Fällen das Speichern und Wiederherstellen. Wenn zwischen zwei Aktivierungen eines Threads, der die Gleitkommaregister benutzt, diese nicht anderweitig benutzt wurden und noch ihre alten Werte enthalten, können sie einfach weiter benutzt werden. Gegen diese späte Sicherung spricht, dass zum Zeitpunkt einer Exception als Folge eines Gleitkommaregisterzugriffs nicht unmittelbar klar ist, zu welchem Ausführungskontext die Register gehören und ob sie überhaupt gültige Werte enthalten. Auf Mehrprozessorsystemen muss weiterhin sichergestellt werden, dass der Gleitkommakontext eines Threads nicht noch in den Registern anderer Prozessoren gehalten wird. Das frühzeitige Speichern bei einer Umschaltung von einem Thread mit aktiven Registern vermeidet dieses Szenario. Die Register müssen niemals bei einer Gleitkommaausnahme gesichert werden. Dies geschah, wenn nötig, bereits bei der letzten Threadumschaltung. Ist dem Thread bekannt, dass er seit der Umschaltung zu ihm keine Gleitkommaregister benutzt hat, so kann dieser Test ebenfalls entfallen. Das Wissen resultiert beispielsweise aus einem zeitigeren Test an geeigneter Stelle.

Threadumschaltungen können für die Ausführung explizit — ein auf eine IPC wartender Thread schaltet aufgrund des Fehlens selbiger selbst auf den höchstpriorisierten lauffähigen Thread um — oder implizit erfolgen. Letzterer Fall liegt unter anderem vor, wenn nach Ablauf einer Zeitscheibe umgeschaltet wird. Threads setzen ihre Fortführung nach einer Umschaltung genau nach dieser Stelle fort. Dieses Verhalten ist für die impliziten Fälle der Threadumschaltung erforderlich. Anderenfalls sind transparente Unterbrechungen nicht mehr möglich. Viele Instruktionsfolgen setzen voraus, dass sie vollständig abgearbeitet werden. Unterbrechungen verzögern diese Abarbeitung, brechen sie aber nicht ab.

Bei expliziten Threadumschaltungen ist die Rückkehr an die ursprüngliche Stelle nicht immer unbedingt notwendig. Der Thread befindet sich in einem wohldefinierten Zustand, der gezielte Manipulationen zulässt. Besonders interessant ist der Fall, dass der Zielthread sofort in den Nutzer-Modus zurückkehrt. Der im Kern-Modus bis zur Rückkehr in den Nutzer-Modus auszuführende Code besteht im Wesentlichen aus der Rückkehr aus Funktionen. In der C/C++-Implementierung ist die Rückkehr aus geschachtelt aufgerufenen Funktionen nicht umgehbar.[14] Darin werden keine Veränderungen an seinem Zustand, am Zustand anderer Threads und am Zustand des Systems vorgenommen. Der Thread kehrt aus dem C++-implementierten Teil des Kerns zurück, lädt im Assembler-Stub seinen User-Kontext und setzt die Ausführung im User-Mode fort.

Initiiert ein Thread eine explizite Threadumschaltung, ist in den meisten Fällen sein Zustand bekannt. Es ist insbesondere bekannt, ob ein Thread lauffähig ist. Ist er das nicht, so ist definitiv keine Anpassung der run queue notwendig.[15] Das allgemeine `switch_to`, das auch

[14] Siehe dazu auch den Abschnitt bzgl. C-Aufrufkonvention

[15] Fiasco benutzt eine *lazy queuing policy* wie sie in [2] beschrieben wird. Um häufige Ein- und Auskettoperationen zu vermeiden, wird erlaubt, dass auch nicht lauffähige Threads in der run queue verkettet sein dürfen. Es wird jedoch gefordert, dass alle lauffähigen, momentan nicht ausgeführten Threads über diese

für implizite Threadumschaltungen benutzt wird, testet, ob der aktuelle Thread lauffähig ist, aber nicht in der run queue verkettet. In diesem Fall muss er in diese eingehangen werden.

Für den Multiprozessorfall ist interessant, ob der Zielthread aktuell auf einem anderen Prozessor ausgeführt wird. Wurde er vorher vom Ausgangsthread gelockt, so kann diese Möglichkeit ausgeschlossen werden — der im allgemeinen Fall notwendige Test entfällt.

Für `schedule`, die Funktion, die den nächsten lauffähigen Thread auswählt und zu diesem umschaltet, sind deutlich weniger verschiedene Aufrufszenarien vorhanden. `schedule` implementiert einen Round-Robin-Scheduler mit statischen Prioritäten.

`schedule` wird in zwei unterschiedlichen Szenarien aufgerufen. Im ersten wartet in Thread auf ein Ereignis und kann seine Ausführung bis zum Eintritt dieses Ereignisses nicht weiter ausführen. Er ist zu diesem Zeitpunkt nicht mehr lauffähig. Von allen anderen im System vorhandenen lauffähigen Threads wird einer der höchsten Priorität zum Weiterlaufen ausgewählt. Mit dem Idle-Thread exisitiert immer ein geeigneter Thread.

Im zweiten Fall führte ein Ereignis potentiell dazu, dass ein höher priorisierter Thread lauffähig wurde. Es muss eine Entscheidung getroffen werden, welcher Thread weiter ausgeführt wird. Diese Entscheidung beschränkt sich auf den aktuellen Thread und die in Folge des Ereignisses lauffähig gewordenen. Für den häufigen Fall einer IPC wird der Empfänger der Nachricht lauffähig. Erwartet der Sender seinerseits keine IPC, so sind nach der Zustellung beide — Sender und Empfänger — lauffähig.

Im allgemeinen Fall durchsucht `schedule` eine globale Struktur, um den geeignetsten Thread für die nachfolgende Ausführungsperiode zu finden. Für den Fall, dass nur zwischen zwei Threads entschieden werden muss, ist der Zugriff auf die globale Struktur nicht notwendig.

Allgemein gültige Funktion sind bei der Entwicklung hilfreich. Die aufrufenden Programmteile können einfach gehalten werden und ausdrucksstarke Abstraktionen in anderen Funktionen benutzen. Die Komplexität wird an einer Stelle gebündelt. Zusätzlich vorhandenes Wissen kann jedoch nicht optimal genutzt werden. Es wird Code ausgeführt, der in der speziellen Situation überflüssig ist. Die Alternative dazu sind spezialisierte Funktionen, die nur unter bestimmten Bedingungen das erwartete Verhalten implementieren. Diese Funktionen sind kürzer — Testen weniger Bedingungen — und benötigen einzeln weniger Resourcen (Cache, Branch Predictor). Insgesamt vergrößern sie allerdings das Programm. Zwischen diesen entgegengesetzt wirkenden Effekten existiert ein Optimum. Aufgrund seiner Abhängigkeit von den verschiedenen Ausführungshäufigkeiten ist dieses nur schwer exakt identifizierbar.

2.4. Implementierung in Assembler

Der Fiasco-Mikrokern ist in C++[3] implementiert. Die Nutzung einer Hochsprache bietet alle damit verbundenen Vorteile.

Korrektheit Komplexer Code kann in einer Hochsprache besser realisiert werden, als dies in reinem Assembler der Fall wäre. Sogar Änderungen des Designs sind unter Weiterverwendung nicht direkt betroffener Komponenten möglich.

Erweiterbarkeit Gewünschte Funktionalität kann einfach hinzugefügt werden. Dabei wird nicht notwendigerweise eine tiefe Vertrautheit mit dem System vorausgesetzt. Der Kerndebugger ist dafür ein bezeichnendes Beispiel.

---

queue erreichbar sind.

Wartbarkeit Fehler können in einer Hochsprache einfacher verfolgt und korrigiert werden. Es ist möglich, umfangreiche Protokollierungen vorzunehmen.

Optimierung Ein optimierender Compiler erzeugt für Projekte ab einer gewissen Größe korrekten Code, der handgeschriebenen Code mit Hinblick das Optimierungskriterium übertrifft. Compiler können auf eine Vielzahl umfangreicher Optimierungsalgorithmen zurückgreifen.

Portabilität Für die Portierung des Kerns muss auf dem Zielsystem ein entsprechender Compiler verfügbar sein. Der von Fiasco verwendete gcc ist auf einer großen Anzahl von Systemen vorhanden. Ein geringer Teil des Kerns besteht aus Code, der in Assembler implementiert werden muss. Nur dieser Teil ist neu zu realisieren.

Trotz der schwer wiegenden Vorteile kann handgeschriebener Code vorteilhaft sein.

Reguläre Struktur Compiler-erzeugter Code ist in C in Funktionen als kleinste Einheit organisiert. Diese Struktur, die unter normalen Bedingungen sinnvoll ist, erweist sich in extremem Fällen als nicht vorteilhaft. Der Einsprungpunkt in den Kern ist ein solches Beispiel. Der Code wird nicht nach dem *call-return*-Paradigma[16] organisiert. Die genaue Kenntnis der Bedingungen, unter denen dieser Code ausgeführt wird, gestattet es, Code-Blöcke mittels direkter Sprünge zu verbinden.

Kompakter Code Code, der nur unter bestimmten Bedingungen ausgeführt werden soll, wird in Blöcken organisiert, die mittels bedingten Sprüngen betreten werden. Die Anordnung dieser Blöcke ist prinzipiell frei wählbar. Es wird jedoch häufig eine lineare Hintereinanderreihung in der Folge des Auftretens gewählt. Diese Anordnung bewirkt, dass der Cache nicht optimal ausgenutzt wird. Neuere Compiler gestatten eine gewisse Einflussnahme auf die Anordnung. Sie limitieren die Möglichkeiten insofern, als dass auch sie dem *call-return*-Paradigma folgen.

In Fiasco ist der sogenannte Fast-Path der ausgeführte Code vom Eintritt des Kerns nach Anforderung des Kerndienstes Short-IPC bis zur Rückkehr in den Nutzerkontext. Die Implementierung dieses Pfades erfolgte in optimiertem Assembler. Dabei wird davon ausgegangen, dass in der überwiegenden Zahl der Fälle klar definierte Bedingungen vorliegen.[17] Sollte sie nicht gegeben sein, wird in den bereits vorhanden Code verzweigt, der die teils umfangreichen anderen Fälle in C++ implementiert. Der Code bringt nur Geschwindigkeitsvorteile, wenn der als häufig angenommene Fall auch tatsächlich eintritt.

Die Geschwindigkeitsgewinne stammen von einem einfacheren Ausführungsparadigma, kompakterem Code und damit verbundener besserer Cachenutzung und einer linearen Codeausführung, die falsche Sprungvorhersagen minimiert.

2.5. C-Aufrufkonvention

C/C++-Programme werden in eine Folge von Maschinenbefehlen kompiliert. Diese werden von der CPU direkt ausgeführt. C/C++ bietet ein flexibles Programmiermodell auf mittlerem Abstraktionsniveau. Es stehen strukturierte Datentypen und Kontrollflusselemente zur

[16] Code wird mittels einer *call*-Instruktion angesprungen. Der ausgeführte Code erzeugt einen eigenen Stackframe. Bei Beendigung des Codes kehrt man mittels einer *ret*-Instruktion hinter die aufrufende *call*-Instruktion zurück.

[17] Der Sender sendet nur Wörter in Registern. Der Empfänger ist empfangsbereit.

Verfügung. C++ bietet darüber hinausgehend Unterstützung für objektorientierte Programmierung, parametriersierbare Datentypen und komplexe Kontrollflusskontrolle mittels Ausnahmebehandlung sowie umfangreiche Laufzeitsysteme an. Programme sollen weitesgehend unabhängig von der Hardware und ihren Eigenschaften sein, auf der sie später ausgeführt werden. In Assembler hingegen sind nur einfache Operation wie Addition und Vergleiche auf Speicherzellen ausdrückbar, die zudem an eine konkrete Prozessorarchitektur gebunden sind. Der Programmzustand eines C-Programms — der Wert seiner Variable, der Ort der Ausführung — wird auf Speicher- und Registerwerte abgebildet. Schleifen und vollständig geschachtelte Funktionsaufrufe lassen sich aus bedingten Sprüngen unter Einhaltung von Konventionen nachbilden. Die Abbildung von C-Programmen auf diese niedere Abstraktionsebene kann auf vielfältige Art und Weise geschehen. Zahlreiche Optimierungen für Spezialfälle sind möglich.

Jeder Deklaration ist ein zu jedem Zeitpunkt der Programmausführung aktueller Wert zugeordnet. Dieser Wert wird in einem Register oder Speicher gehalten oder ist als konstant bekannt. Die Zuordnung eines Variablenreferenzen auf die zugeordnete Deklaration und damit auf einen Wert erfolgt statisch zur Kompilationszeit. Prozessoren bieten nur ein begrenzte Anzahl von Registern. Zudem können in Registern nur in der Größe begrenzte Typen gehalten werden. Variable, die nicht in Registern gehalten werden können, müssen im Speicher abgelegt werden. Operationen auf diesen Variablen können direkt im Speicher ausgeführt werden, wenn die Architektur entsprechende Unterstützung in Form von Speichermanipulation bietet. Ist dies nicht der Fall, muss der Wert in ein Register geladen, verändert und zurück in den Speicher geschrieben werden. Die Zuordnung von Variable auf Register oder Speicher wird als Registerallokation bezeichnet. Eine optimale Belegung ist nur unter hohem Aufwand zu finden. Es existieren jedoch gute Heuristiken. Je häufiger auf eine Variable zugegriffen wird, desto vorteilhafter ist es, den Wert dieser Variable in einem Register zu halten. Langsame Speicherzugriffe fallen dann nur noch beim Laden und Speichern an. Zu jedem Zeitpunkt wird damit eine Teilmenge aller Variable in schnell zugreifbaren Registern gespeichert, der Rest im langsameren Speicher. Die Zuordnung einer Varialbe auf Speicher oder Register kann an verschiedenen Stellen des Programms in Abhängigkeit der Nutzungshäufigkeit in diesem Programmteil unterschiedlich sein.

C/C++ kennt als Kontrollflussprimitive Entscheidungen, Schleifen und Funktionsaufrufe. Zusätzlich sind freie Sprünge mittels *goto* möglich, wovon aber aus Wartbarkeitsgründen abgeraten wird. In Assembler stehen nur bedingte und unbedingte Sprünge[18] zur Verfügung. Schleifen und Fallunterscheidungen sind mittels Vergleichen und darauf basierten bedingten Sprüngen leicht implementierbar.

Aus Funktionen können weitere Funktionen aufgerufen werden. Dabei bestehen bezüglich der Tiefe keine Begrenzungen. Funktionsaufrufe können auch direkt oder indirekt rekursiv erfolgen. Das Aufrufmuster kann datenabhängig sein und ist somit zur Kompilationszeit unbekannt. Eine statische Reservierung von Speicher zum Halten von in der Funktion deklarierten Variable ist damit nicht möglich.

Funktionsaufrufe sind in C/C++ vollständig geschachtelt. Nach einem Funktionsaufruf wird in die aufrufende Funktion unmittelbar nach dem Funktionsaufruf zurückgekehrt. Es sind auch andere Ausführungsmodelle denkbar und aus verschiedensten Gründen propagiert worden. Koroutinen als Variante mit der Erhaltung eines Scopes über die aktuelle Aufrufkette hinweg und mehrfachen Funktionseintrittspunkten ist das am weitesten bekannte Beispiel.

[18] In den meisten Architekturen existiert auch eine Sprungvariante, die den aktuellen Programmzähler speichert.

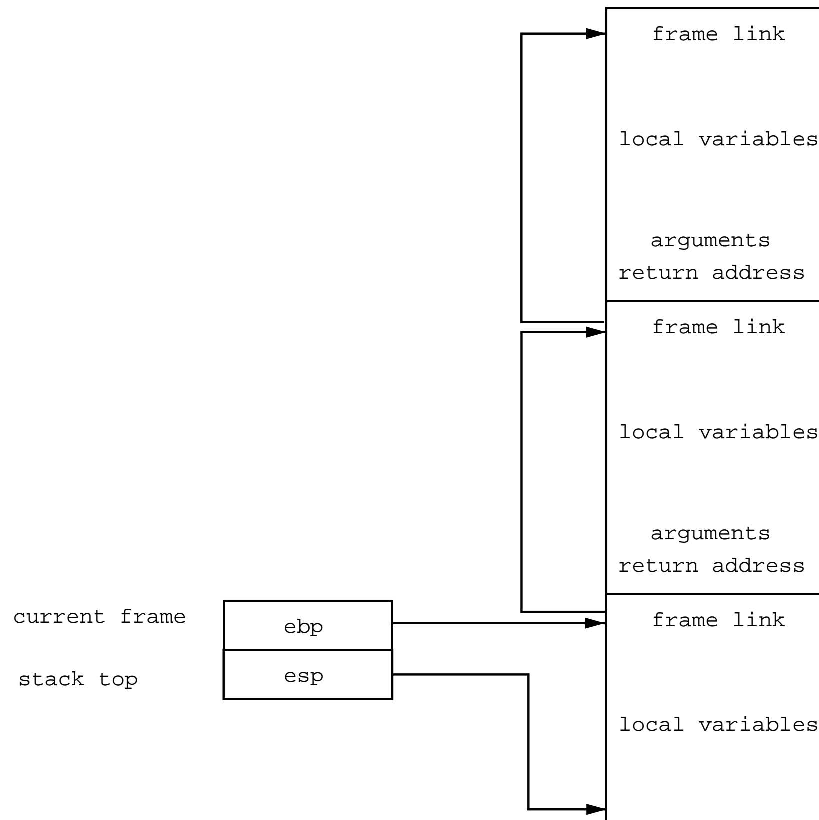


Abbildung 4: Verkettete Frames

Als geeignetste Datenstruktur zur Implementierung des vollständig geschachtelten C/C++-Modells bietet sich ein Stack an. Die Limitierung der Zugreifbarkeit auf das oberste Element entspricht der C/C++-Semantik, in der auch nur der Scope der aktuellen Funktion sichtbar ist. Werte aus dem Namesraum der aufrufenden Funktion müssen explizit als Parameter in den Namensraum der aufgerufenen Funktion exportiert werden.

Beim Aufruf einer Funktion wird für deren Scope auf dem Stack der notwendige Speicher reserviert. Dieser Bereich wird Frame oder Rahmen genannt. Jeder aktiven Funktioninstanz ist ein Frame zugeordnet. Eine Funktion kann in Folge von rekursiven Aufrufen mehrere aktive Funktionsinstanzen besitzen. Der Frame wird beim Verlassen wieder freigegeben. Die Reservierungen der Scopes der übergeordneten Funktionen innerhalb der Aufrufkette bleiben reserviert, sind aber bis zur Rückkehr in die entsprechende Funktion nicht zugreifbar. In Abbildung 4 wird die Situation nach dem Aufruf von zwei Funktionen dargestellt. Die Frames der einzelnen Funktionen sind durch Zeiger in Form einer einfach verketteten Liste miteinander verbunden. Diese Verkettung erleichtert die Analyse der aktuellen Aufrufsituation wie sie Debugger häufig anbieten. Ein Register ist speziell als Zeiger auf den aktuellen Rahmen reserviert. Die Verkettung ist für die Ausführung eines Programms nicht unbedingt notwendig. Die Adressierung von Argumenten und lokalen Variablen kann statt relativ zum Framepointer auch relativ zum Stackpointer erfolgen. Das Framepointerregister ist dann zum allgemeinen Gebrauch verfügbar. Damit entfällt die Möglichkeit, einfach die Aufruffolge bis zur aktuellen Stelle einfach nachzuvollziehen. Die Grösse der Frames muss als externe Information vorliegen

oder durch Codeanalyse aufwendig gewonnen werden.

In der aufrufenden und in der aufgerufenen Funktion können Variable auf die selben Ressourcen abgebildet werden. Bei im Speicher gehaltenen Variablen ist dies kein Problem. Frames werden immer in disjunkten Speicherbereichen reserviert. Prozessorregister können hingegen nicht dupliziert werden. Die aufrufende Funktion erwartet nach der Rückkehr aus dem Funktionsaufruf den Zustand der Variablezuordnung auf Register und Speicher wie davor. Da die aufgerufene Funktion ebenfalls die Register nutzt, muss dafür Sorge getragen werden, dass lebendige[19] Werte aus den Registern vor Funktionsaufruf gesichert und danach wiederhergestellt werden.

Das Speicher und Wiederherstellen der Registerinhalte kann sowohl auf aufrufender wie auch aufgerufener Seite erfolgen. Beide Seiten haben allerdings keine Kenntnisse, welche Register mit lebendigen Werten belegt sind bzw. welche Register von der aufgerufenen Funktion benötigt werden. Das komplette Speichern auf einer Seite bringt damit potentiell vermeidbaren Aufwand in Form von unnötigerweise gesicherten und wiederhergestellten Registern mit sich. Die benutzte Konvention versucht einen Mittelweg zu finden: Einige Register können von der aufgerufenen Funktion benutzt werden, ohne dass ihr Inhalt gesichert werden muss. Hält die aufrufende Funktion lebendige Werte in diesen Registern, muss sie diese selbst vor dem Funktionsaufruf sichern und danach restaurieren. Diese Register werden als *caller saved* bezeichnet. Der Aufrufer kann sich hingegen bei anderen Registern darauf verlassen, dass sie nach dem Funktionsaufruf die selben Werte wie davor besitzen. Benötigt die aufgerufenen Funktion für das Halten von Variablenwerten diese Register — *callee saved* genannt —, muss sie sicherstellen, dass bei der Rückkehr die ursprünglichen Werte enthalten sind. Abbildung 1 zeigt die Zuordnung der x86-Register auf die beiden Gruppen.

Register	caller saved	callee saved	Sonstiges
eax	x		return value
ecx	x		
edx	x		
edi		x	
esi		x	
ebx		x	
ebp		x	frame link
esp			stack pointer

Tabelle 1: Registernutzungskonvention bei *x86*

Mit dem Aufruf einer Funktion wird komplett in deren Scope gewechselt. Die Deklarationen der aufrufenden Funktion sind nicht mehr sichtbar. Die Kommunikation zwischen den beiden Funktionsscopes erfolgt mittels Parameterübergabe beim Aufruf und Rückgabewert bei der Zurückkehr in die aufrufende Funktion. Eine Funktion kann unbegrenzt viele Parameter übernehmen und liefert genau einen Rückgabewert. Werden mehr Rückgabewerte benötigt, so können Zeiger auf Variable im Sichtbarkeitsbereich des Aufrufers als Argumente übergeben werden.[20] Über die Zeiger sind diese Werte als Seiteneffekt der aufgerufenen Funktion

[19]Ein Registerinhalt heißt lebendig, wenn er den aktuellen Wert einer Variable enthält, dieser Wert an keiner anderen Stelle gespeichert ist und er an späterer Stelle noch benötigt wird.

[20]Die Variable ist nur über den übergebenen Zeiger, nicht aber über ihren Namen zugreifbar.

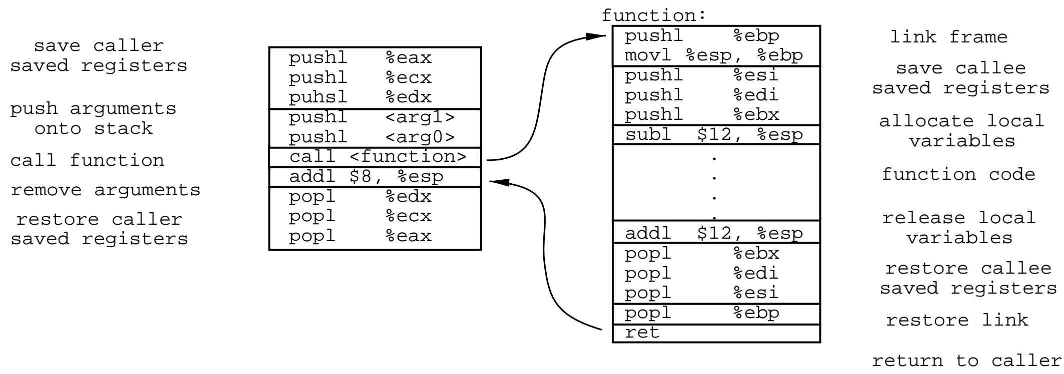


Abbildung 5: C-Aufrufkonvention, Codebeispiel x86

veränderbar. Für die Parameterübergabe scheiden die in begrenzter Anzahl verfügbaren Register alleine aus. Der Stack als nicht in der Größe begrenzte Datenstruktur[21] bietet sich damit auch zur Parameterübergabe an. Die Argumente werden vom letzten beginnend auf dem Stack abgelegt. Es existieren Optimierungen, die eine begrenzte Anzahl von Argumenten in Registern übergeben. Die zur Verzweigung zur Zielfunktion benutzte `call`-Instruktion speichert unterhalb[22] der Argumente die Rückkehradresse. Die aufgerufene Funktion erzeugt die Frameverkettung und allokiert durch Dekrementierung des Stackpointers Platz für lokale Variable. Falls notwendig, speichert sie auch noch *callee saved*-Register. Nach Beendigung der Funktionsausführung wird der einzelne Rückgabewert in ein dafür reserviertes Register geladen. Der beim Funktionseintritt reservierte Platz für lokale Variable wird freigegeben und die ebenfalls beim Funktionseintritt gesicherten Register wiederhergestellt. Der Zeiger für den aktuellen Frame wird auf den nächsthöheren Frame, in den zurückgekehrt wird, angepasst. Der Stack befindet sich zu diesem Zeitpunkt mit der Rückkehradresse gefolgt von den Argumenten in dem Zustand, den er direkt nach der Verzweigung zu der jetzt verlassenen Funktion hatte. Mit der `ret`-Instruktion wird in die aufrufende Funktion direkt nach dem dortigen Funktionsaufruf zurückgekehrt. Diese entfernt die für den Funktionsaufruf auf dem Stack abegelegten Argumente und benutzt den Rückgabewert in `eax` für die weitere Programmausführung.

Abbildungen 5 und 6 zeigen am Beispiel der x86-Architektur die Instruktionsfolge, die einen Funktionsaufruf implementiert, beziehungsweise den Stack vor und nach einem Funktionsaufruf.

Funktionen müssen ebenso wie Variablen vor der ersten Benutzung deklariert werden. Im Unterschied zu Variablen existiert nur ein globaler, nicht schachtelbarer Namensraum.[23] Die Deklaration umfasst einen Namen und eine Signatur. Letztere besteht aus einem Rückgabetyp und einer Folge von Argumenten, denen ein Typ zugeordnet ist. Bei der Kompilation wird überprüft, ob die der Funktionaufruf der deklarierten Funktionssignatur entspricht. Anzahl und Typ der deklarierten Argumente müssen mit Anzahl und Typ der aktuellen Funktionsargumente übereinstimmen. Der Compiler bricht mit einer Fehlermeldung ab, wenn dies nicht der Fall ist. Funktionsdeklarationen können unabhängig von den zugehörigen Defini-

[21] In der konkreten Implementierung wird der Stack natuerlich durch die Größe des verfügbaren Speichers begrenzt.

[22] Es wird an zu niedrigeren Adressen hin wachsender Stack angenommen.

[23] Einige Compiler bieten Erweiterungen für lokale Funktionen.

tionen in C-Programme eingebunden werden. Einzelne Teile eines Programms können damit unabhängig voneinander übersetzt werden und später gelinkt werden. Die Typüberprüfung zur Kompilationszeit und die Aufrufkonvention zur Laufzeit stellen sicher, dass die einzelnen Programmteile korrekt zusammenarbeiten.

Aus Sicht des Assemblers ist eine Funktion ein Symbol, das den Beginn der Implementierung definiert. Das fehlende Typsystem verhindert eine Überprüfung, ob die aufrufende und aufgerufene Funktion mit ihren Argumenten und Funktionssignatur übereinstimmen. Es ist Aufgabe des Programmierers oder Compilers, dafür zu sorgen, dass die aufgerufene Funktion die erwartete Umgebung vorfindet.

C++ verwendet auf Assemblerebene prinzipiell das gleiche Ausführungsmodell und die gleiche Aufrufkonvention wie C. Unterstützung für Objektfuntionen, virtuelle Funktionen, automatische Konstruktur-/Destruktoraufrufe, Ausnahmebehandlung und dynamische Typinferenz erfordern aber umfangreicheren Code zur Implementierung von Funktionsaufrufen.

Für Programme, die ausschließlich in C/C++ implementiert sind, ist die Kenntnis der Implementierungsdetails und das Ausführungsmodell auf Assemblerebene nicht notwendig. Das ändert sich, wenn Assemblercode mit in C/C++ implementierten Programmteilen zusammenarbeiten soll. Der Code muss die Konventionen der Programmausführung erfüllen.

Das Einhalten der Aufrufkonvention ist mit Aufwand verbunden. Dieser Aufwand dient dem Herstellen eines wohldefinierten Ausführungsumfeldes, das sicherstellt, dass unabhängig voneinander entwickelte und übersetzte Programmfragmente den Erwartungen entsprechend miteinander interoperieren. Die Konventionen sind für den durchschnittlichen Anwendungsfall optimiert. In seltenen Fällen kann es sinnvoll sein, die Vorteile dieser Allgemeingültigkeit zum Preis erhöhten Entwicklungsaufwandes gegen hohe Ausführungsgeschwindigkeit einzutauschen. Der in diesen Fällen verwendete Assemblercode ist hochgradig an das entsprechende Umfeld angepasst.

Auch in C/C++-Programmen kann der Mehraufwand für einen Funktionsaufruf eingespart werden. Der Funktionsrumpf von als *inline* deklarierte Funktionen wird nicht mittels Funktionsaufruf ausgeführt, sondern direkt an die Aufrufstelle eingefügt. Eingesparten Instruktionen für einen konkreten Ausführungspfad steht eine Aufblähung des gesamten Programmumfangs entgegen. Inlining ist nur für Funktionen sinnvoll, die an wenigen Stellen sehr häufig aufgerufen werden.

2.6. Systemaufrufe

Der L4-Mikrokern stellt Nutzerapplikationen eine Systemschnittstelle mit 7 Systemaufrufen bereit. Dies Aufrufe sind im Kernadressraum implementiert. Ein Aufruf erfordert damit eine Schnittstelle zum Kerneintritt und kann nicht allein in Form eines C-Funktionsaufrufes erfolgen.

Aus Nutzersicht erscheint ein Systemaufruf wie ein normaler C-Funktionsaufruf. Tatsächlich wird jedoch nur eine Wrapperfunktion aufgerufen, die die gemäß der C-Aufrufkonvention übergebenen Argumente an das Format der Kernschnittstelle anpasst und in den Kern mittels eines geeigneten Mechanismus eintritt. Die Argumentanpassung und der Kerneintritt erfordern speziellen Inline-Assembler. Der Eintritt in den Kern erfolgt mit Hilfe einer `int`- oder `sysenter`-Instruktion.

Im Kern wird die Ausführung in Assembler fortgesetzt. Die Aufgabe des Eingangsassemblercodes besteht in der Konvertierung der in Registern übergebenen Systemrufargumente in auf dem Stack übergebenen Argumente entsprechend der C-Aufrufkonvention und Auf-

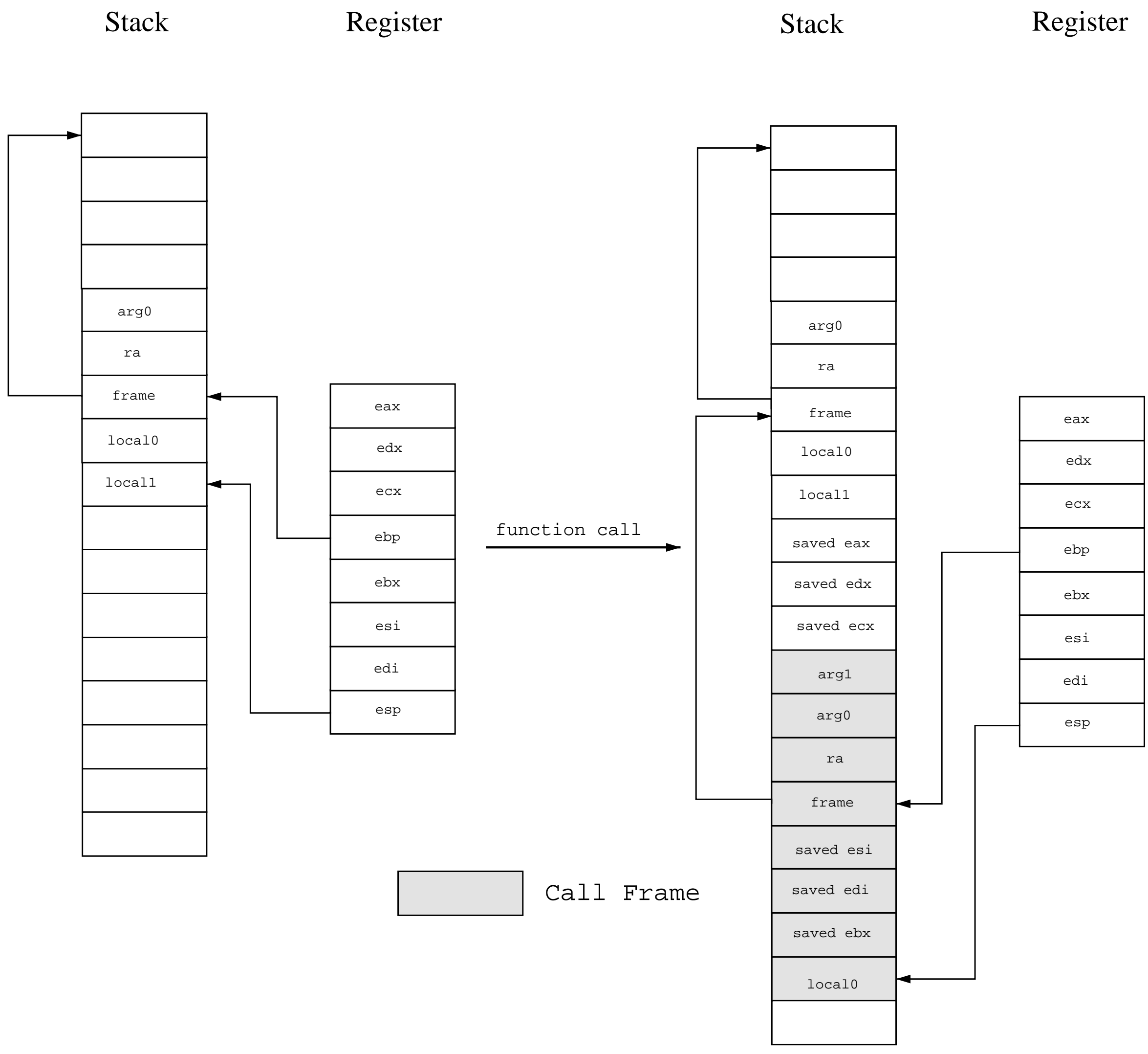


Abbildung 6: C-Aufrufkonvention, Stack-Layout

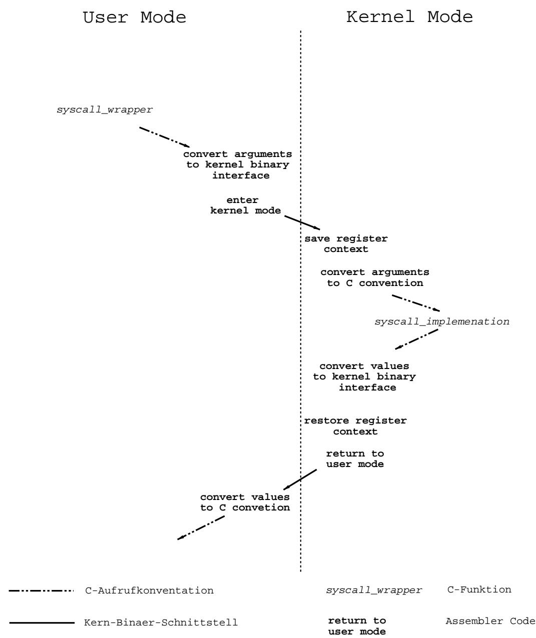


Abbildung 7: Syscall, C-Implementierung

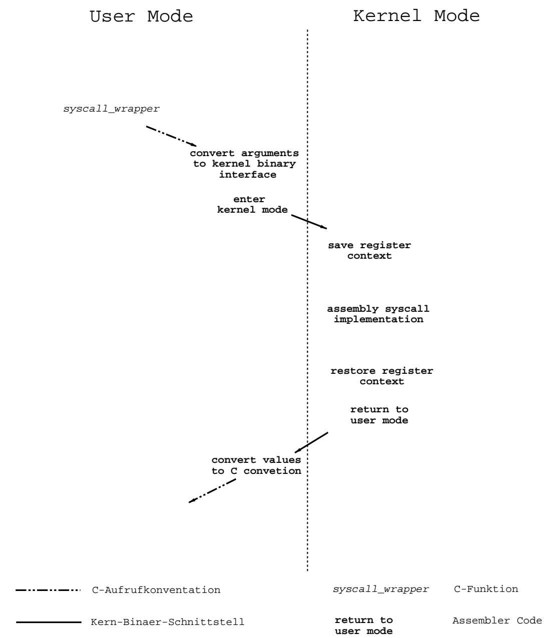


Abbildung 8: Syscall, Assembler-Implementierung

ruf der in C++ implementierten Kernteile. Nach Befriedigung des Systemaufrufes kehrt der Kontrollfluss aus dem C++-Teil zurück. Wieder ist es Aufgabe des Assemblercodes, die Rückgabewerte aus der C-Konvention in das Format der Kernschnittstelle anzupassen und in den Nutzer-Modus zurückzukehren.

Der Assemblerteil der C-Wrapperfunktion im Nutzer-Modus wandelt die Rückgabewerte aus dem Format der Kernschnittstelle in das der C-Bindings für die Systemaufrufe um. Die Wrapperfunktion kehrt danach zurück.

Ein Systemaufruf beinhaltet nach obiger Darstellung vier Konvertierungen zwischen C-Aufrufkonvention und Kernschnittstelle. Abbildung 7 zeigt dies noch einmal. Der usprüngliche Fiasco-Kern war so implementiert.

Beim Entwurf der Kernschnittstelle wurde bewußt auf Effizienz geachtet. Argumente und Rückgabewerte werden soweit wie möglich in Registern übergeben. Bei der sehr häufig aufgerufenen und somit die Leistung des Kerns maßgeblich beeinflussenden IPC-Operation werden die Teile der zu sendenden Nachricht und der empfangenen Nachricht in den selben Registern gehalten. Idealerweise würden diese Register vom Kerneintritt des Senders bis zum Kernaustritt des Empfängers nicht verändert. Zwischenspeicherung der Nachricht, die den Cache-Footprint erhöhen würden, entfielen damit.

Mit einer C/C++-Implementierung ist diese effiziente Lösung nicht erreichbar. Ein Grund dafür ist, dass gemäß der C-Aufrufkonvention Argumente über den Stack transferiert werden. Die C/C++-Implementierung benötigt potentiell Zugriff auf die Werte aller Register zum Zeitpunkt des Kerneintrittes. Diese müssen somit aus den Registern auf den Stack übertragen werden.

Aufgrund der begrenzten Registeranzahl in der x86-Architektur muss davon ausgegangen werden, dass jede nichttriviale C-Funktion alle verfügbaren Register benutzt. Dieses Verhalten ist im Normalfall auch durchaus erwünscht. Es gibt auch keinen Mechanismus, um Register

auf der x86-Architektur für den Compiler als nicht benutzbar zu markieren.

Eine Lösung, die Vorteile aus den Eigenschaften der Kernschnittstelle ziehen soll, muss aus den angedeuteten Gründen in Assembler implementiert werden. Dabei kommen aber nur Kernfunktionen mit beschränkter Komplexität in Frage. Abschnitt 3.1 diskutiert diese Frage detailliert.

2.7. Systemveränderung durch IPC

Die L4-API stellt IPC zur Verfügung. Eine IPC Operation enthält einen Sende- und einen Empfangsteil. Beide Teile sind optional, mindestens einer muss aber vorhanden sein. Mit einer IPC können drei Arten von Nachrichten versandt und empfangen werden:

short IPC Die Nachricht besteht aus zwei Werten, die ohne Speicherzugriff direkt über Register übertragen werden.

long IPC Die Nachricht besteht aus mehreren Werten. Diese werden mittels einer Speicherstruktur beschrieben. Beim Zugriff auf diese Speicherstruktur wie auch auf indirekte Werte kann es zu Seitenfehlern sowohl im Sender- wie auch Empfängeradressraum kommen. Diese Seitenfehler werden wiederum in IPC an den entsprechenden Pager umgesetzt.

flex pages Der L4-Mikrokern bietet nur ein sehr rudimentäres Speichermodell an. Physischer Speicher wird beim Systemstart an die Starttasks zugewiesen. Diese können diesen Speicher an beliebige Tasks weitergeben und auch wieder zurückfordern. Sie besitzen damit einen Mechnismus, der es ihnen erlaubt, eine beliebiges Schema an Speicherzuteilung zu implementieren. Die Weitergabe von Speicher ist in den IPC-Mechanismus integriert.

Es ist möglich, für die verschiedenen Phasen einer IPC-Operation unterschiedliche Timeouts anzugeben. Wird bis zum Verstreichen des Timeouts die Operation nicht beendet, so erfolgt ein Abbruch. Der Rückgabewert der IPC-Operation enthält einen Fehlerkode, der den genauen Grund des Scheiterns näher beschreibt.

Im Laufe einer IPC können nachfolgend aufgelistete Systemzustände verändert werden. Welche Manipulationen notwendig sind, hängt von der Art der IPC ab.

Threadzustand Ein Thread, der auf eine IPC, wartet zeigt dies durch seinen Zustand an. Eine nichtleere IPC hat somit mindestens eine Zustandsübergang zur Folge. Der bisherige Empfänger kann unmittelbar nach der IPC keine weitere IPC annehmen.

Threadkontext Der Threadkontext besteht aus drei Teilen:

Ausführungskontext Der letzte User-Kontext eines Threads bestehend aus *eip*, *esp*, den zugehörigen Segementselektoren und den *flags* muss gespeichert werden, damit die Ausführung später an dieser Stelle fortgesetzt werden kann. Der Ausführungskontext ist für den Thread nicht direkt sichtbar.

Syscall-Kontext Die Argumente für die IPC-Operation werden in Registern übergeben. Ein Teil dieser Argumente wird vom Kern ausgewertet, ein anderer als Nutzerdaten direkt an den Empfänger weitergereicht. Nach der Rückkehr in den User-Mode findet der Thread in seinem Syscall-Kontext die Rückgabewerte der Operation.

Zusatzkontext Neben den *general purpose* Registern weist die x86-Architektur acht weitere Gleitkommakommaregister auf. Es ist möglich, den Zugriff auf diese Register zu kontrollieren. Damit muss ihr Inhalt nur bei Veränderung gesichert werden.

Zustand des Addressraums Long IPC transferiert Daten zwischen Adressräumen. Dabei können sowohl im Sender- wie auch im Empfängeradressraum Seitenfehler auftreten.

Umfang des Addressraums IPC wird in L4 auch zum Einblenden von Speicher benutzt. Daten, die den Adressraumbereich im Adressraum des Senders beschreiben, werden in den Adressraum des Empfängers übertragen.

Timeouts IPC Operation können zeitlich begrenzt werden. Zur Verwaltung dieser zeitlichen Limits werden die Zeitpunkte, nach denen nicht beendete Operationen abgebrochen werden, in entsprechenden Datenstrukturen verwaltet.

Scheduling IPC kann zur Folge habe, dass der aktuelle Ausführungskontext gewechselt wird. Dafür können unterschiedliche Gründe verantwortlich sein:

- Der aktuelle Thread verliert seine Lauffähigkeit, weil der IPC-Partner nicht bereit ist. Der Scheduler wählt den nächsten auszuführenden Thread gemäß der Scheduling-Policy des Systems.
- Ein höher priorisierter Empfänger wird nach einem *send* lauffähig. Die vorherrschende Meinung besagt, dass von Sender und Empfänger der höher priorisierte ausgewählt wird. Insbesondere wird nicht stets zum Empfänger der IPC umgeschaltet.
- Eine spezielle Umgebungssituation wird angenommen. Die IPC-Varianten *call* und *reply-and-wait* deuten auf eine enge Kopplung der kommunizierenden Threads im Rahmen eines Client-Server-Systems hin. Um diese häufig auftretende Situation besonders effizient zu behandeln, wird nicht der reguläre Pfad über den Scheduler gewählt, sondern unabhängig von der Priorität der beteiligten und aller sonst im System vorhandenen Threads zum Empfänger der Nachricht umgeschaltet.

Run Queue Lauffähige Threads sind nicht notwendigerweise in der Run Queue eingekettet.

- Ein auf IPC wartender Thread ist nicht lauffähig. Nach Erhalt der Nachricht ist er bereit, seine Ausführung im Nutzeradressraum fortzuführen. Da nicht zwingend sofort zu ihm umgeschaltet wird, ist es notwendig, ihn in die Run Queue einzuhängen.
- Der Sender einer IPC ist noch lauffähig, der Scheduler entscheidet aber zum Empfänger umzuschalten. Der Sender muss jetzt in die Run Queue eingehangen werden.

Abhängig von der Art der IPC sind nicht alle Veränderungen in jedem Fall tatsächlich notwendig. Die Manipulation des Threadzustandes, des Ausführungskontextes und des Syscall-Kontextes ist immer vorhanden.

Eine IPC-Operation kann unter ungünstigen Umständen sehr lange dauern. Eine long IPC[24] löst potentiell mehrere Seitenfehler aus. Die Konstruktion rekursiver Adressräume erfordert

[24] *long IPC* bezeichnet eine IPC, die Teile der Nachricht aus dem Quelladressraum des Senders in den Zieladressraum des Empfängers überträgt.

komplexe Verwaltungsstrukturen. Es ist damit notwendig, für den allgemeinen, potentiell lang andauernden Fall, Unterbrechungen zuzulassen.

Fiasco implementiert den IPC-Pfad mit Hilfe seiner nicht blockierenden Synchronisation. Die IPC-Operation ist größtenteils unterbrechbar.[25]

2.8. Unterbrechbare IPC in Fiasco

IPC kann in der Fiasco-Implementierung in die drei Phasen Vorbereitung, Rendezvous/Registerübertragung und Speicherübertragung gegliedert werden. Diese Gliederung hat ihre Ursache in der von L4 spezifizierten IPC-Semantik und den von Fiasco angestrebten Echtzeiteigenschaften.

```
 1    sys_ipc
 2      prepare_receive
 3        setup_receiver
 4          r -> r | recv
 5      do_send
 6        state_add
 7          r | recv -> r | recv  | iip | sip | p
 8        sender_enqueue
 9        ipc_send_regs
10          ipc_try_lock
11            lock
12            sender_ok
13            set_partner
14          sender_dequeue
15          state_del
16            r | recv | iip | sip | p -> r | recv | iip
17          state_change
18            <receiver state>r | recv | iip -> r
19          ipc_unlock
20            clear
21              switch_to
22                r | recv | iip -> r
23        state_del
24          r -> r
25     do_receive
```

Abbildung 9: `call`-IPC, Empfänger bereit

Die L4-Spezifikation [4] fordert, dass ein Server auf eine aus Sende- und Empfangsteil bestehende Anfrage mit einer Nachricht mit Null-Timeout antworten können muss.[26] Der Klient schickt eine IPC mit Sende- und Empfangsteil an den Server. Im Sendeteil übergibt er die Argumente, im Empfangsteil erwartet er die Rückgabewerte. Der Server empfängt die Anfrage und errechnet eine Antwort. Diese sendet er an den Klient zurück. Diese Sendeoperation wird abgebrochen, wenn sie nicht sofort durchgeführt werden kann. Anderenfalls sind *denial-of-service*-Angriffe möglich: Anfragen werden an den Server gesandt, der Klient

[25] Es existieren sehr kurze Abschnitte, die trotzdem atomar ausgeführt werden müssen und damit nicht unterbrechbar sind.

[26] Die Nachricht wird sofort zugestellt, wenn der Empfänger bereit ist, oder abgebrochen, falls dies nicht der Fall ist.

nimmt die Antwort aber nicht entgegen. Der Server würde an dieser Stelle warten, bis der Empfänger empfangsbereit oder eine Zeitüberschreitung festgestellt wird.[27] Damit dieser Angriff unmöglich ist, wird vom Klienten gefordert, dass er sofort nach dem Senden seiner Anfrage empfangsbereit ist und die nur sofort zustellbare Antwort entgegennimmt. Fiasco wurde so entworfen, dass der Sende- orthogonal zum Empfangszustand ist. Der Klient bereitet *vor* seiner Sendoperation den Empfangszustand vor. Nach Abschluss des Sendens wird derjenige Teil des Threadzustandwortes zurückgesetzt, der eine aktive Sendeoperation anzeigt. Zurück bleibt ein empfangsbereiter Thread. Abbildung 9[28] zeigt diesen Vorgang. In Zeile 4 wird der Empfangsteil vorbereitet. Zeile 7 fügt den Sendeteil hinzu, der in Zeile 16 wieder entfernt wird. Mit der Freigabe des Threadlocks — dies beendet auch den Sendeteil der IPC — in Zeile 20 befindet sich der Sender wie gefordert in einem empfangsbereiten Zustand.

```
 1   sys_ipc
 2     prepare_receive
 3       setup_receiver
 4         state_add:f0007fe3
 5           r -> r | wait
 6       state_add:f000c1eb
 7           r | wait  -> r | wait | iip
 8     do_receive
 9       state_change_safely
10            r | wait | iip -> r | wait | iip | busy
11       sender_list
12       state_change_safely
13             r | wait | iip | busy -> wait | iip
14       schedule
15         lock
16           cli
17         switch_to
```

Abbildung 10: Rendezvous, IPC-Empfang

Die Vorbereitung des Empfangzustandes benötigt signifikant Zeit. Es müssen unter Umständen Timeouts gesetzt werden. Die Integration vieler Fälle in eine Quelltextbasis bedingt zudem viele schwer voraussagbare Entscheidungen. So vergehen auf einem PIII 450MHz-System über 200 Takte vom Eintritt in die allgemeine IPC-Funktion bis zu dem Punkt, an dem definitiv entschieden wird, ob eine IPC-Operation möglich ist oder der Sender warten muss. Ein Ziel bei der Entwicklung von Fiasco war die Untersuchung von Techniken, die maximale Unterbrechbarkeit erlauben. Deswegen wurde auch an dieser Stelle eine Lösung entwickelt, in der Threads auch während der Durchführung einer IPC so weit wie möglich verdrängbar sind.

Prinzipiell könnten sowohl Sender wie auch Empfänger die Zustandsänderungen im Laufe einer IPC vornehmen. In Fiasco wurde die Entscheidung getroffen, dass nur der Sender im Laufe einer IPC aktiv wird. Der Empfänger signalisiert lediglich die Bereitschaft zum

[27]IPC sind in L4 synchron. Beide Partner blockieren, wenn die Gegenseite nicht bereit ist.
[28]Trace des vom Sender ausgeführten Codes in einer *call*-IPC. Der Empfänger ist empfangsbereit.

initial state : `m[0] = 0, m[1] = 0`
program A : `w0 r1`
program B : `w1 r0`

executed sequence	read values	
	program A	program B
`w0 r1 w1 r0`	0	1
`w0 w1 r1 r0`	1	1
`w0 w1 r0 r1`	1	1
`w1 w0 r1 r0`	1	1
`w1 w0 r0 r1`	1	1
`w1 r0 w0 r1`	1	0

Tabelle 2: `write-read`-Synchronisation

Empfang. Der Sender findet entweder einen empfangsbereiten Empfänger oder wartet darauf, bei Eintritt dieses Zustandes von selbigem benachrichtigt zu werden. Der Empfänger signalisiert über seinen Zustand Empfangsbereitschaft und benachrichtigt, falls vorhanden, wartende Sender. Der Sender führt daraufhin die beim ersten Versuch fehlgeschlagene IPC — der Empfänger war nicht bereit — aus.

Für die Zustellung einer IPC, in deren Verlauf Threads unterbrochen werden können, sind zwei Synchronisationsprobleme zu lösen: Versuchen zwei Threads gleichzeitig einem dritten eine IPC zu senden, so darf nur einer erfolgreich sein. Es darf weiterhin nicht zu der Situation kommen, dass sowohl Sender und Empfänger für eine IPC bereit sind, beide aber aufgrund von ungünstigem Ausführungsverhalten der Meinung sind, dass der jeweils andere dies noch nicht ist.

Zwei Sender synchronisieren sich über das Threadlock des Empfängers. Bei der in Abbildung 9 dargestellten IPC setzt der Sender in Zeile 11 das Threadlock. Bevor er dieses in Zeile 20 wieder freigibt, modifiziert er den Zustand des Empfängers (Zeile 18). Der zweite Sender wird nach Setzen des Threadlocks den Empfänger in einem nicht empfangsbereiten Zustand vorfinden. Diese Situation ist in Abbildung 11 dargestellt. Er setzt das Threadlock (Zeile 8), stellt fest, dass der Empfänger nicht empfangsbereit ist (Zeile 9) und gibt das Threadlock frei (Zeile 10). Das IPC-Rendezvous ist fehlgeschlagen. Der Sender gibt seine Lauffähigkeit auf (Zeile 12) und ruft den Scheduler auf (Zeile 13). Er wartet bis der Empfänger empfangsbereit wird. Dafür hat er sich in dessen Liste aller Threads eingetragen, die IPC senden wollen (Zeile 5).

Die Synchronisation zwischen Sender und Empfänger wird durch in einer geeigneten Reihenfolge ausgeführten Folge von Lese- und Schreibzugriffen erreicht. Dabei ist es ausreichend, wenn ein IPC-Partner das Vorhandensein einer IPC-Situation mit zwei bereiten Partnern erkennt. Der in Abbildung 2 dargestellte Algorithmus erreicht dieses Verhalten. Zwei Aktivitäten — als program A und program B bezeichnet — signalisieren ihre Bereitschaft durch Schreiben eines signifikanten Wertes — hier 1 — in eine ihnen zugeordnete Speicherzelle. Nachfolgend lesen sie die Speicherzelle ihres Partners aus. Unabhängig von der Ausführungsreihenfolge der einzelnen Schritte erkennt immer mindestens ein Partner die Bereitschaft des anderen.[29] In der Abbildung sind alle möglichen Ausführungsreihenfolgen mit den daraus resultierenden Ergebnissen aufgeführt. In keinem Fall lesen beide Prozesse 0. Damit wird sichergestellt, dass

[29] Dies gilt nur für ein sequentielles Speichermodell. Darin wird gefordert, dass eine globale Ordnung für alle Teiloperationen rekonstruierbar ist. Operationen einer Aktivität erscheinen innerhalb dieser Ordnung in der Programmreihenfolge. Für multiprogrammierte Einzelprozessoren ist dies immer der Fall. Dieses Modell wird auch von Intel-Prozessoren unterstützt.[5]

eine IPC-Situation immer erkannt wird.

```
 1    sys_ipc
 2      do_send
 3        state_add
 4           r -> r | poll | sip | iip
 5        sender_enqueue
 6        ipc_send_regs
 7          ipc_try_lock
 8            lock
 9            sender_ok
10            clear
11         state_change_safely
12           r | poll | sip | iip -> poll | sip | iip
13         schedule
14            lock
15              cli
16            switch_to
```

Abbildung 11: Rendezvous, IPC-Senden

In Fiasco tritt eine leicht veränderte Version des oben geschilderten Problems auf: Es müssen nicht nur zwei vorher bekannte Aktivitäten synchronisiert werden. Die Anzahl der Threads, die eine IPC-Operation auf einem Zielthread starten wollen, ist nicht begrenzt. Es muss also nicht nur erkannt werden, *dass* eine IPC-Operation anhängig ist, sondern es ist auch die Identität des Partners herauszufinden. Dementsprechend sind die Lese- und Schreiboperationen asymmentrisch. Während die Sender sich atomar in eine jedem Thread zugeordnete Liste einketten, genügt es für den Empfänger, sein Statuswort zu modifizieren. Abbildungen 11 und 10 zeigen die IPC-Initiierungssequenz für Sender und Emfänger. Der Sender führt in Zeile 5 seine Schreib- und in Zeile 9 seine Leseoperation aus. Der Empfänger tut das in Zeile 7 beziehungsweise 11.

Nachdem in Abbildung 9 der Fall gezeigt wurde, dass ein Sender auf einen bereiten Empfänger trifft, zeigt Abbildung 12 einen Sender mit einer IPC an einen nicht bereiten Empfänger. Der Sender führt in den Zeilen 6 und 10 den Schreib- beziehungsweise Leseteil des Synchronisationsprotokolls durch. Parallel laufende IPC anderer Sender werden durch das Setzen des Threadlocks des Empfängers in Zeile 9 verhindert. Nachdem er den Empfänger als nicht empfangsbereit vorgefunden hat, setzt er seinen Zustand in Zeile 13 auf *nicht lauffähig* und lässt den Scheduler andere lauffähige Threads auswählen (Zeile 14). Der Empfänger führt ebenfalls sein *write-read*-Synchronisationsprotokoll aus (Zeilen 25 und 29). Dabei stellt er fest, dass Sender bereitstehen. Er benachrichtigt einen geeigneten (Zeile 30). Dabei wird dieser wieder lauffähig und überprüft erneut, ob der Empfänger bereit ist (Zeile 50). Dies ist jetzt der Fall, die IPC wird ausgeführt. Nach Beendigung der zugehörigen Manipulationen[30] werden beide IPC-Threadzustände gelöscht (Zeilen 53 und 55). Beide Threads sind lauffähig und können vom Scheduler zur weiteren Ausführung ausgewählt werden.

[30]Es werden user-mode Register übertragen. Im Trace ist das nicht sichtbar.

```
              SENDER                                    RECEIVER

 1   sys_ipc
 2     do_send
 3       set_receiver
 4       state_add
 5          r -> r| sip | iip | poll
 6       sender_enqueue
 7       ipc_send_regs
 8         ipc_try_lock
 9           lock
10           sender_ok
11           clear
12       state_change_safely
13         r| sip | iip | poll -> sip | iip | poll
14       schedule
15         lock
16           cli
17         switch_to


18                                                       sys_ipc
19                                                         prepare_receive
20                                                           setup_receiver
21                                                             state_add
22                                                                 r -> r | wait
23
24                                                          state_add
25                                                        r | wait → r | wait | iip
26                                                         do_receive
27                                                           state_change_safely
28                                                             r | wait | iip -> r | busy | wait | iip
29                                                           sender_list
30                                                           ipc_receiver_ready
31                                                             thread_lock
32                                                             lock_guard_t
33                                                                 test_and_set
34                                                             state_del
35                                                               <sender> sip|iip|poll -> sip |iip
36                                                             state_add:f000be7e
37                                                               <sender> sip | iip -> r | sip | iip
38                                                             ~lock_guard_t
39                                                               clear
40                                                                 switch_to
41                                                                   lock_guard_t
42                                                                      cli

43          clear
44            sti
45        state_add
46           r| sip | iip -> r| sip | iip | poll
47        ipc_send_regs
48          ipc_try_lock
49            lock
50            sender_ok
51          sender_dequeue
52          state_del
53            r| sip | iip | poll -> iip | r
54          state_change
55            <receiver> busy | iip | wait | r -> r
56          ipc_unlock
57            clear
58              switch_to
59                lock
60                  cli
61                                                                   ~lock_guard_t:f0006df0
62                                                                      sti
63                                                           state_change_safely
64                                                                  r -> r
65                                                           state_del
66                                                                  r -> r
```

Abbildung 12: IPC-Handshake

	PIII 450Mhz		P4 1.6GHz	
		late sender_enqueue		late sender_enqueue
ping pong	1850	1663	2892	2532
Rendezvous	845	673	1367	1011
Receiver-Setup	412	313	743	535

Tabelle 3: Kosten für IPC [Takte, alle Werte für Roundtrip].

	PIII 450Mhz		P4 1.6GHz	
	Einzelprozessor	Mehrprozessor	Einzelprozessor	Mehrprozessor
`schedule`	214	643	304	1464
`switch_to`	148	367	220	1012

Tabelle 4: Kosten Scheduling und Threadumschaltung [Takte].

Die in diesem Kapitel beschriebenen IPC-Implementierung ist nahezu vollständig unterbrechbar. Ausnahmen sind hier das Einketten in die Senderlisten und Threadumschaltungen. Die Kosten dafür sind erheblich. Abbilung 3 listet die Zeiten für ein IPC-Roundtrip nebst den Zeiten für Vorbereitung des Empfängers und Vollendung des Rendezvous auf. In Abbildung 9 entsprechen die letzten beiden Zeiten den Verzögerungen bis zum Erreichen von Zeile 9 (Receiver-Setup) beziehungsweise Zeile 15 (Rendezvous). Ist der Empfänger bereits für die IPC bereit, ist das Ausführung des Synchronisationsprotokolls durch den Sender nicht mehr notwendig. Die in diesem Fall redundante Ein- und Auskettung in die Senderliste kann entfallen. Diese Optimierung bringt allerdings nur geringfügige Gewinne, wie Abbildung 3 in der Spalte "late sender_enqueue" entnommen werden kann.

Auch in den häufigen günstigen Fällen, in denen der Empfänger bereits auf den Sender wartet, keine Timeouts beachtet werden müssen und die Nachricht nur aus Registerinhalten besteht wird der Mehraufwand zum Erreichen weitestgehender Unterbrechbarkeit betrieben. Ein weiterer Fakt, der zu den vergleichsweise langsamen Roundtripzeiten führt, ist die Benutzung der Standardimplementation von `switch_to` und `schedule`, welche die in Abbildung 4 angegebenen Verzögerungen mit sich bringen. Nach Empfang einer IPC kehrt ein Thread in den Nutzer-Mode zurück, ohne noch weitere Systemveränderungen vorzunehmen. In der ursprünglichen Fiasco-Implementierung wurde in seinen letzten Kern-Kontext zurückgeschaltet. Dort kehrte er aus allen auf seinem Thread vorhandenen Funktions-Frames zurück, um letztendlich im Kerneintrittscode seinen Nutzer-Register-Kontext wiederherzustellen und in den Nutzermode zurückzukehren. Alle Informationen, um dies zu tun, sind bereits zum Zeitpunkt der Umschaltung zu ihm vorhanden. Eine direkte Rückkehr unter Umgehung einer aufwendigen Threadumschaltung ist damit möglich und wurde in den optimierten Varianten des Fastpaths auch implementiert.

3. Entwurf

3.1. Fastpath für Spezialfälle von IPC

Die vom L4-Mikrokern bereitgestellte Funktionalität wird extrem ungleichmäßig genutzt. Deutlich über 95% aller Operationen sind IPC. Der Großteil davon wiederum entfällt auf short ipc, eine IPC-Operation, die nur Daten in Registern überträgt und keinen Nutzerspeicher referenziert. Die Kosten dieser Operation haben signifikanten Einfluß auf die Gesamtleistung des Systems. Mit guter Näherung kann behauptet werden, dass die Effizienz der IPC-Implementierung die Leistung des Mikrokerns bestimmt.

Short IPC werden im gleichen Ausführungspfad wie die viel aufwendiger long IPC bearbeitet. Es wird beträchtlicher Aufwand zur Ermöglichung von Unterbrechbarkeit betrieben. Bei zeitlich begrenzten Operationen bringt dieser Mehraufwand keine Vorteile in Form geringerer Latenzen. Vorhandene Optimierungspotentiale können so nur schwer erschlossen werden.

Einige Fälle sind mit ihren wenigen Systemmodifikationen einfach implementierbar. Die Implementierung kann in Assembler erfolgen. Die sich daraus ergebenden kurzen Lösungen sind in ihrer Ausführungszeit begrenzt.

Nachfolgend sind die IPC-Varianten nach steigendem Implementierungsaufwand geordnet. Alle Operationen umfassen nur Transfer von Registerwerten und keine Timeouts.

call, replay-and-wait Der einfachste Fall. Es sind keine Queue-Veränderungen notwendig. Der Syscall-Kontext kann teilweise direkt aus den Registern weitergenutzt werden.

send Entscheidung, ob Sender oder Empfänger weiterlaufen soll. Nach der IPC sind beide Threads lauffähig. Ein Thread muss in die Run-Queue eingefügt werden.

receive, wait Ein aufwendiger Scheduling-Zyklus ist notwendig. Aus der IPC-Operation lässt sich nicht der nächste auszuführende Thread ableiten.

Bei den IPC mit Sendeanteil muss geprüft werden, ob

- der Empfänger aus beliebigen Quellen Nachrichten akzeptiert oder
- der Empfänger nur Nachrichten eines Senders akzeptiert und der Sender dies ist.

Kann der Empfänger direkt aus der IPC abgeleitet werden, so kann die Threadumschaltung vereinfacht ausgeführt werden. Der neue ausführende Thread findet einen Teil seines Syscall-Kontextes bereits in den Registern vor und muss diesen nicht aus dem Speicher auslesen.

Ein minimaler Fastpath umfasst mindestens:

- Test, ob ein für den Fastpath geeigneter IPC-Spezialfall vorliegt
- Identifikation der Kernrepräsentationen der beteiligten Threads und Überprüfung, ob diese in den richtigen Zuständen sind
- Veränderung der Thread-Zustände
- Test, ob eine Adressraumumschaltung notwendig ist
- Rückkehr in den Nutzerkontext oder Threadumschaltung.

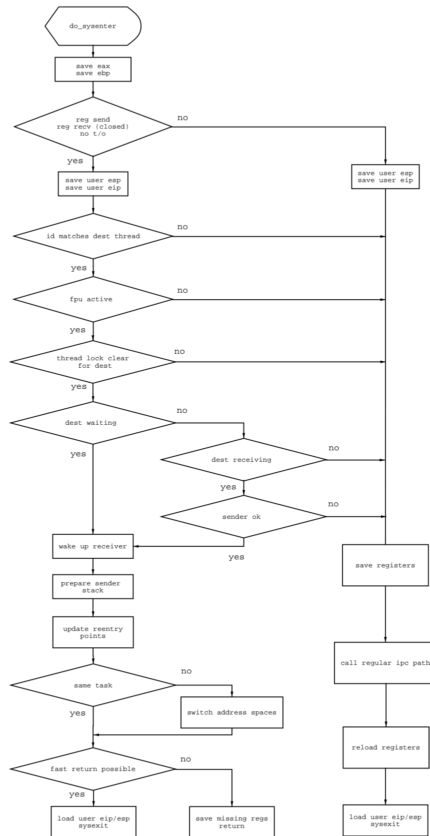


Abbildung 13: Fastpath Flussdiagramm

Das entsprechende Flussdiagramm ist in Abbildung 13 dargestellt.

Der Fastpath muss mit der in Fiasco genutzten Synchronisation kompatibel sein. Es muss der Fall beachtet werden, dass ein Thread im konventionellen Code eine IPC begonnen und das Thread-Lock des Empfängers bereits gesetzt hat, den Zustand des Empfängers aber noch nicht manipulieren konnte. Bei Threads, die den Fastpath benutzen, kann diese Situation niemals auftreten. Sie führen die IPC immer komplett ununterbrechbar aus.

Der Test, ob ein Thread gelockt ist, lässt sich mit der Überprüfung des Wertes des Locks im TCB des Empfängers überprüfen.

Eine im Fastpath ausgeführte IPC-Operation muss sich ebenso wie alle anderen mit dem Empfänger der Nachricht und möglichen anderen Sendern synchronisieren. Die Synchronisation mit den Sendern erfolgt über das Threadlock des Empfängers. Da der Fastpath nicht unterbrochen werden kann, muss das Lock nicht gesetzt werden. Es reicht die Überprüfung, dass es nicht bereits von Thread gesetzt wurde. Die Überprüfung des Zustands des Empfängers stellt sicher, dass dieser ebenfalls für eine IPC bereit ist. Schlägt einer dieser beiden Tests fehl, ist eine optimierte Behandlung der IPC nicht möglich. In diesem Fall wird auf den regulären Code zurückgegriffen.

Setzt ein Thread nach Empfang einer IPC seine Ausführung fort, so verändert er den Systemzustand nicht mehr. Er kehrt aus allen Funktionen, die er vor seiner Deaktivierung aufgerufen hatte, zurück, lädt im Kerneintrittsstub seinen Nutzerkontext und kehrt zur Ausführung in den Nutzermodus zurück. Alle notwendigen Informationen, die zu dieser Rückkehr notwendig sind, stehen bereits im Fastpath zur Verfügung. Eine Optimierung besteht darin, nicht zu dem Thread umzuschalten, sondern ihn direkt in den Nutzermodus zurückkehren lassen. Dieses Verhalten eröffnet außerdem die Möglichkeit, die in Register übergebenen Nachrichtenwerte direkt in den Register zu belassen und nicht erst zu speichern, um sie später wieder in selbige Register zu laden.

Threads können im Kernmodus verdrängt werden, setzen ihre Ausführung aber normalerweise immer hinter der letzten Verdrängung fort. Im Falle, dass der Empfänger einer IPC gleichzeitig Sender war, beispielsweise mit einem `reply-and-wait`, ist dieses Standardverhalten zu beachten. Abbildung 14 zeigt den im Laufe einer IPC-ausgeführten Code. Zu beachten ist hier der Fall, dass ein Thread nach Ausführung von Zeile 16 als empfangsbereit betrachtet wird. Zu diesem Zeitpunkt hält er aber noch das Threadlock des Empfängers seiner IPC, dass er erst in Zeile 20 freigibt. Da die IPC unterbrechbar ist, könnte ein dritter Thread versuchen, ihm auf dem Wege des Fastpaths eine Nachricht zu senden. Die Nachricht würde gesendet werden, der Thread auf direktem Wege in den Nutzermodus und das Threadlock des ersten Empfängers würde niemals freigegeben werden. Dieses Verhalten muss verhindert werden.

Es gibt zwei Lösungen für dieses Problem: Erstens könnte der Code von Zeile 15 bis Zeile 20 atomar ausgeführt werden. Es gäbe somit keine Stelle, an der ein empfangsbereiter Thread ein Threadlock hält. Dies würde die Unterbrechbarkeit des IPC-Pfades geringfügig verschlechtern.

Die zweite Lösung müsste sicherstellen, dass ein per Fastpath sendender Thread erkennt, dass sein Empfänger ein Lock hält und in diesem Fall auf den langsamen Code zurückfällt. Dieser Test ist beispielsweise durch Test auf Lauffähigkeit implementierbar. So lange wie ein Thread ein Threadlock hält, muss er lauffähig sein. Diese Lösung hat zwei Nachteile. Erstens wird im Fastpath ein weiterer Test notwendig. Außerdem schließt der Test auf Lauffähigkeit zu viele Threads aus. Lauffähige Empfänger, die kein Threadlock halten und damit gültige Partner sind, werden im Fastpath abgelehnt. Aus diesen Gründen wurde die atomare Zustandsänderung und Threadlockfreigabe gewählt.

```
1    sys_ipc
2      prepare_receive
3        setup_receiver
4          r -> r | recv
5      do_send
6        state_add
7          r | recv -> r | recv  | iip | sip | p
8        sender_enqueue
9        ipc_send_regs
10         ipc_try_lock
11           lock
12           sender_ok
13           set_partner
14         sender_dequeue
15         state_del
16           r | recv | iip | sip | p -> r | recv | iip
17         state_change
18           <receiver state>r | recv | iip -> r
19         ipc_unlock
20           clear
21             switch_to
22               r | recv | iip -> r
23       state_del
24         r -> r
25     do_receive
```

Abbildung 14: `call`-IPC, Empfänger bereit

3.2. Multiprozessor

Parallele Ausführung auf mehreren Prozessoren kann als fein granulare Unterbrechung von Threads aufgefasst werden. Die bereits für vollständige Unterbrechbarkeit ausgelegte Synchronisationsmethoden des Fiasco-Kerns müssen damit nicht grundlegend verändert werden. Das Synchronisationsschema von Fiasco ist für Multiprozessoren korrekt, wenn die Semantik der der einzelnen Primitive — Threadlock, Helping-Lock,nicht unterbrechbarer Code und Protokolle basierend auf lock-freier Synchronisation — garantiert werden kann.

Alle Synchronisationsprimitive benötigen zu ihrer Implementierung eine atomare *read-modify-write*-Operation. Es muss sichergestellt werden, dass global sichtbare und von mehreren auf verschiedenen Prozessoren ausgeführten Threads manipulierbare Zustände in vorhersagbarer Art und Weise verändert werden können. Unter der Voraussetzung der Verfügbarkeit solcher atomaren Operationen ist die Semantik der höheren Konstrukte auch auf Multiprozessoren implementierbar. Die nachfolgende Diskussion geht von der Verfügbarkeit einer solchen Operation aus.

Die Semantik des Helping-Lock umfasst neben dem Schutz einer Ressource vor gleichzeitiger Manipulation die Zusage, dass Zeit von hoch priorisierten Bewerbern an nieder priorisierte aktuelle Besitzer weitergereicht wird, damit diese das Lock schnellstmöglich freigeben. In Einzelprozessorsystemen gab es für dieses Verhalten eine naheliegende Implementierung. Da immer genau ein Thread zu jedem möglichen Zeitpunkt ausgeführt wurde, konnte der Bewerber stets davon ausgehen, dass er seine Zeit an den Besitzer des Lock weitergeben kann. In Mehrprozessorsystemen kann diese Zeitweitergabe scheitern, wenn der aktuelle Besitzer be-

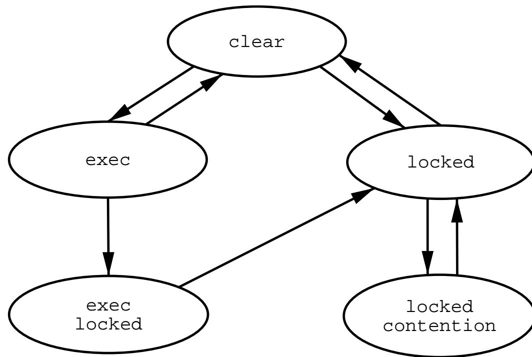


Abbildung 15: Zustände des Thread-Locks für Multiprozessoren

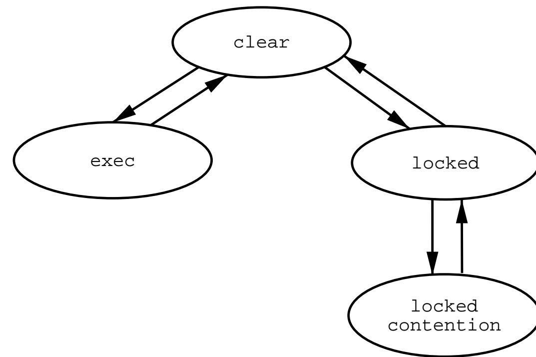


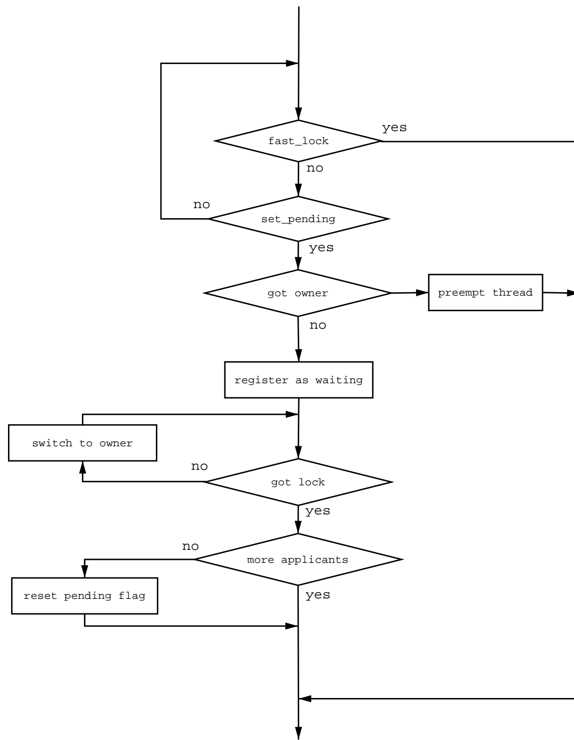
Abbildung 16: Zustände des Thread-Locks für Einzelprozessoren

reits auf einem anderen Prozessor ausgeführt wird. Ein Modell, das unter diesen Bedingungen sinnvoll eingesetzt werden kann, wird in [6] beschrieben.

Threadlocks sind Spezialfälle von Helping-Locks und können somit wie diese mit Hinblick auf *priority inheritance* behandelt werden. Sie unterscheiden sich von Helping-Locks insofern, als dass die geschützte Ressource wiederum ein Thread ist. Ihre Semantik verlangt, dass dieser Thread nicht ausgeführt wird, wenn das Lock gesetzt ist. Wurde auf einem Einzelprozessor ein Threadlock gesetzt, so wurde der damit geschützte Thread zu diesem Zeitpunkt automatisch nicht mehr ausgeführt.[31] Zusammen mit der Zusicherung, dass zu gelockten Threads nicht umgeschaltet wird, ist damit die gewünschte Semantik erreicht. Auch in der Mehrprozessorvariante von Fiasco kann zu gelockten Threads nicht umgeschaltet werden. Im Gegensatz zu zu Einzelprozessoren stimmt jedoch die Eingangsbehauptung nicht mehr. Ein Thread, dessen Threadlock gesetzt wurde, kann auf einem anderen Prozessor gerade ausgeführt werden. Abbildungen 16 und 15 zeigen die Zustände von Threadlocks und möglichen Zustandsübergänge für Einzel- und Mehrprozessoren. Bei der Manipulation des Threadlocks muss den neuen Zuständen und Zustandsübergängen Rechnung getragen werden. Abbildung 17 zeigt die derzeitige Implementierung.

Nicht unterbrechbarer Code konnte auf Einzelprozessorsystemen effizient mittels Sperren von Interrupts implementiert werden. Diese Implementierung ist offensichtlich auf Mehrprozessoren nicht relevant. Das Sperren von Interrupts verhindert die lokale Verdrängung, ist aber wirkungslos gegen Ausführung auf anderen Prozessoren. Eine einfache Lösung besteht in der Nutzung von Spinlocks. Wird ein nicht unterbrechbarer Codeabschnitt auf verschiedenen Prozessoren häufig ausgeführt, so ist der Bus-Traffic, der in Folge der häufigen Lese- und Schreibzugriffe aufgrund des Cache-Koheränzprotokolls entsteht, beachtlich. Eine Lösung dieses Problems besteht in der Zuweisung von Objekten an Prozessoren. Manipulationen auf diesen Objekten sind dann nur noch durch Threads möglich, die auf dem entsprechenden Prozessor ausgeführt werden. Damit wird nicht unterbrechbarer Code mittels Sperrung von Interrupts auf diesem Prozessor wieder möglich. Möchte ein Thread, der nicht auf der dem Objekt zugewiesenem CPU läuft, Veränderungen vornehmen, so muss er auf geeignete Art und Weise, diese Veränderungen auf der entsprechenden CPU zum Beispiel mittels Migration initiieren.

[31] Eine Ausnahme sind Threads, die ihr eigenes Threadlock setzen.



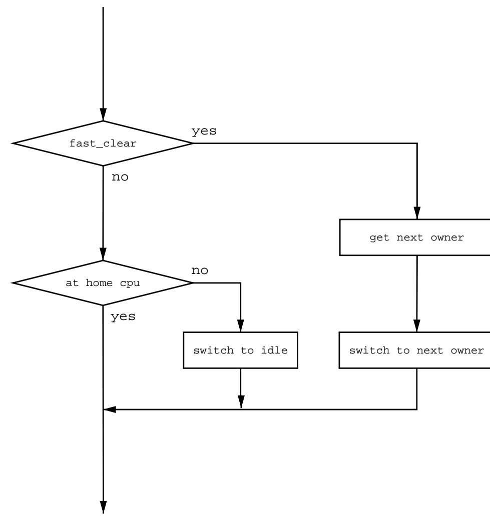


Abbildung 17: Flussdiagramm für Mehrprozessor-Threadlock

Operation	**PIII 450Mhz**		**P4 1.6GHz**	
lock-Präfix	nein	ja	nein	ja
cmpxchg8b_asm	18	39	166	210
cmpxchg_asm	11	35	17	145
xchg_asm	19	19	125	125
add_add	18	37	28	132
xadd_asm	11	29	12	126
btc_asm	6	21	16	125
bts_asm	6	21	16	125
sticli_asm	23		78	
pushf_asm	53		135	

Tabelle 5: Kosten ausgewählter Instruktionen

Atomare Operationen sind in Form lock-freier Synchronisation ohne weitere Zwischenschichten direkt verfügbar. Einzig und allein das Erscheinungsbild kann sich leicht unterscheiden. So kann ein komplexe *compare-and-swap*-Operation mittels eines durch *test-and-set* implementierten Spinlocks und eines damit geschützten kritischen Abschnitt realisiert sein.

Zustandsübergänge können mittels atomarer Operationen auf Assemblerebene implementiert werden. Auch in Mehrprozessorsystemen kann die Semantik bereitgestellt werden. Bei den in dieser Arbeit untersuchten Systemen waren gravierende Unterschiede in der Leistungsfähigkeit dieser Operationen zu beachten. Wie Abbildung 5 zeigt, ist der Unterschied zwischen Instruktionen mit dem *lock*-Präfix und ohne auf dem modernen P4 bedeutend größer als auf dem älteren PIII. Wir vermuten, dass eine Ursache dieses Unterschiedes in der ebenfalls größer gewordenen Differenz zwischen Prozessorkerntakt und Takt des Caches liegt. Die Semantik der atomaren Operation wird mit Hilfe des Cache-Koheränzprotokolls sichergestellt. Ein weitere Grund dürfte der P4-Kern sein, der auf hohe Taktraten und die Verarbeitung vergleichsweiser einfacher Instruktionsströme optimiert wurde. Es ist bekannt, dass andere komplexe Instruktionen der x86-Architektur ebenfalls zu starken Leistungseinbrüchen führen.

Der Einsatz von mit dem *lock*-Präfix versehenen atomaren Instruktionen hätte zur Folge, dass diese überall zur Manipulation entsprechender Datenstrukturen eingesetzt werden müssten. Dies gilt auch für den IPC-Fastpath.

Es gibt mit lokalen und auf Speicherprotokollen basierten atomaren Operationen zwei weitere Implementierungsmöglichkeiten.

3.2.1. Daten Partitionierung

Lokale atomare Operationen könnten die auch auf dem P4 mit vertretbaren Kosten verbunden atomaren Operationen ohne *lock*-Präfix benutzen. Die derzeitigen Implementierung ordnet nur ausgewählte Datenstrukturen wie die Liste aller lauffähigen Threads einem Prozessor zu. Da per Konvention nur von einem Prozessor auf diese Daten zugegriffen werden darf, sind umfangreiche Manipulationen lokal unter Ausschluss von Interrupts und damit Verdrängung effizient möglich.

Die Manipulation von nicht lokalen Objekten ist aufwendig. Da nur die zugeordnete CPU die Veränderung ausführen darf, muss dieser eine entsprechende Anforderung zugeleitet wer-

den. Dies Benachrichtigung erfolgt mit einem IPI, der hohe Latenzen aufweist.[32] Bei mehreren aufeinanderfolgenden Veränderungen ist es unter Umständen sinnvoll, den die Veränderung verursachenden Thread auf die lokale CPU des Objektes zu migrieren. Dort können die Manipulation dann effizient lokal ausgeführt werden. Diese Umgebung ist für häufige Zugriffe mit wenigen Veränderungen pro Zugriff, wie sie auch bei IPC auftreten, schlecht geeignet.

3.2.2. Speicherbasierte Protokolle

Das Problem des gegenseitigen Ausschlusses ist ein Klassiker unter den Informatikaufgabenstellungen. Bereits seit den sechziger Jahren sind Lösungen bekannt, die nur durch atomares Lesen und Schreiben von Speicher Prozesse synchronisieren. Insbesondere werden keine atomaren *read-modify-write*-Operationen benötigt.

Die praktische Relevanz dieser Algorithmen war allerdings begrenzt. In Einzelprozessorsystemen ist gegenseitiger Ausschluss einfach mittels Blockieren von Interrupts erreichbar. Systeme auf kleinen Multiprozessoren benutzen häufig atomare *read-modify-write*-Operationen wie *test-and-set* oder *compare-and-swap*.

Es existieren Szenarien, in denen reine Lese-Schreibe-Zugriffe atomaren Operationen vorzuziehen sind. Moderne RISC-Prozessoren bieten häufig keine direkte Unterstützung für komplexe Operationen.[33]

Dijkstra schlug bereits 1965 eine Lösung [7] für die Synchronisation zweier Prozesse nur mit Hilfe von Lese- und Schreiboperationen vor. Lamport verallgemeinerte die Lösung in [8] auf beliebig viele Prozesse. Dieser Ansatz wurde auch als Bakery-Algorithmus bekannt. Er hat auch ohne konkurrierende Zugriffe O(N)-Zeitkomplexität. In [9] wird vom selben Autor ein Algorithmus vorgestellt, mit dem ohne Konkurrenzsituation das Betreten des kritischen Abschnitts in konstanter Zeit möglich ist. Unter Last verhält sich dieser Algorithmus allerdings zweifelhaft. Es besteht bespielsweise die Möglichkeit von *starvation*.

Anderson schlug mehrere Lösungen vor, um Skalierbarkeit zu erreichen. In einer seiner letzten Arbeiten [10] präsentiert er einen adaptiven Ansatz. Ohne konkurrierenden Zugriff ist der Eintritt in konstanter Zeit möglich. Unter Last schaltet das Lock automatisch in einen skalierenden Modus, wo Techniken wie *local spinning* Skalierbarkeit ermöglichen. Die wesentliche Neuerung besteht in einem Verfahren, wie aus diesem Modus wieder sicher zurück in den lastlosen geschaltet werden kann.

Steht das Primitiv eines kritischen Abschnitts zur Verfügung, ist die Realisierung von atomaren Operationen problemlos möglich. Durch eine Reimplementierung der entsprechenden Funktion in Fiasco wären keine Anpassungen an anderen Stellen des Quelltextes notwendig. Es ist denkbar, die kritischen Abschnitte auch für komplexere Aufgaben wie beispielsweise Teile des IPC-Rendezvous zu verwenden.

In Abbildung 18 und 19 sind der Bakery- und Lamports Fastpath-Algorithmus in Pseudo-Code dargestellt. Aus Abbildung 6 sind die Kosten für den Eintritt in den jeweiligen kritischen Abschnitt entnehmbar. Die Eintrittskosten liegen klar unter denen einer atomare Maschineninstruktion mit *lock*-Präfix.

[32] Ein **I**nter**P**rocessor **I**nterrupt in einem Dual PIII 450MHz System hat eine Latenz von ca. 1200 Takten.
[33] Die oft angebotene Kombination von *load-locked* und *store-conditionally* kann zur Implementierung stärkerer Operationen benutzt werden.

Algorithmus	Bakery	Fastpath
Instruktionen	29	11
Speicherzugriffe	8	7
bedingte Sprünge	6	2
davon ausgeführt	4	2
Takte PIII 450 MHz	18	7.1
Takte P4 1.6 GHz	14.3	9.3

Tabelle 6: Kosten für den Eintritt in den kritischen Abschnitt.

```
bool choosing[N] = {false, false,....,false};
int  number[N]   = {0,0,....,0}

enter_critcal_section(i):
    choosing[i] = true;
    number[i]   = max(number[0]...number[N-1]) + 1;
    choosing[i] = false;
    for j in [0 ... N -1 ]:
        wait while choosing[j];
        wait while number[j] != -1 and (number[j],j) < (number[i],i);

leave_critical_section(i):
    number[i] = -1;
```

N - number of concurrent processes
i - current process
(x,x') < (y,y') $\leftrightarrow$ x < y or (x == y and $x' < y'$)

Abbildung 18: Bakery-Algorithmus

```
bool busy[N] = {false, false, ..., false};
int  x, y = 0;

enter_critcal_section(i):
  start:
    busy[i] = true;
    x = i;
    if y != 0:
       busy[i] = false;
       wait while y != 0;
       goto start;
    y = i;
    if x != i:
       busy[i] = false;
       for j in [0 ... N-1]:
           wait while busy[j];
           if y != i:
               wait while y != 0;
               goto start;

leave_critical_section(i):
    y = 0;
    busy[i] = false;
```

N - number of concurrent processes
i - current process

Abbildung 19: Lamports schneller Algorithmus

3.2.3. IPC-Fastpath

Die Struktur des Pfades stimmt prinzipiell mit der der Einzelprozessorvariante überein.

Der Fastpath muss mit der im übrigen Kern benutzten Synchronisation kompatibel sein und einige Erweiterungen der Locksemantiken und Ausführungseinschränkungen beachten.

Threads sind in Fiasco-SMP an Prozessoren gebunden. Damit soll eine Verschmutzung der Caches durch häufig migrierende Threads unterbunden werden. Gleichzeitig wird damit sichergestellt, dass auf Prioritäten aufbauende Synchronisationsschemas auch im Multiprozessorfall funktionieren. Durch diese Forderung ist Fastpath-IPC nur möglich, wenn sowohl Sender wie auch Empfänger derselben CPU zugeordnet sind. Dies muss durch einen zusätzlichen Test überprüft werden.

Da in einem Multiprozessorsystem zu jeder Zeit parallel Threads ausgeführt werden können, reicht es nicht aus, das Threadlock des Empfängers nur zu testen. Es muss gesetzt werden, damit IPC die von Threads auf anderen Prozessoren initiiert werden tatsächlich ausgeschlossen werden. Aus dem selben Grund muss der Sender einer im Fastpath behandelten IPC seinen Zustand mit den starken atomaren Operationen verändern. Verwendet er wie im Einzelprozessorfall nur Lese- und Schreiboperationen, so können Threads, die ein Threadlock auf ihn setzen wollen, inkonsistente Zustände beobachten.

3.3. IPC-Handshake

Das Threadlock wird für zwei unterschiedliche Arten von Synchronisation benutzt:

1. Sequentialisierung konkurrierender Zugriffe auf einen Thread. Senden beispielsweise zwei Threads einem dritten eine Nachricht, so darf nur eine Nachricht zugestellt werden. Der erste Thread setzt das Thread-Lock und beginnt die Zustellung seiner Nachricht. Dabei kann er unterbrochen werden. Der zweite Thread versucht ebenfalls das Thread-Lock zu setzen, blockiert aber bis der erste seine IPC beendet hat. Nach Erlangen des Threadlocks wird er aber Feststellen müssen, dass der Empfänger aufgrund der bereits abgewickelten IPC nicht mehr für den Empfang einer weiteren Nachricht bereit ist.

2. Fixierung des Zustand eines Threads. Bei der Veränderung eines Threads mit Hilfe des *ex_regs* Systemaufrufes darf der Zustand des Zielthread nicht verändert werden.

Im Einzelprozessorfall ist diese Unterscheidung nahezu bedeutungslos. Ein Thread, dessen Threadlock gesetzt wird, läuft nicht.[34] Wie Abschnitt 3.2 beschrieben, kann ein Threadlock auf einen Thread gesetzt werden, der zu diesem Zeitpunkt auf einer anderen CPU ausgeführt wird. Die Semantik des Locks fordert, dass dieser Thread nicht mehr ausgeführt werden darf. Dazu wird eine Nachricht an die den Thread ausführende CPU gesandt, welche den Thread nach einem Kerneintritt verdrängt. Eine solche Inaktivierung ist aufgrund der hohen Latenz von IPI und den beachtlichen Kernein- und -austrittskosten kostspielig. Auf einem Dual PIII 450MHz System wird die Ausführung eines Threads durch eine Inaktivierung um ca. 8000 Takte unterbrochen.

Für das nochmals in Abbildung 20 dargestellte Send-Rendezvous ist dieses Verhalten sehr ungünstig. Das in Zeile 8 gesetzte Threadlock des Empfängers dient nur zur Synchronisation mit anderen Sendern. Die Überprüfung, ob eine Nachricht akzeptiert wird erfolgt in Zeile 9 und ist vollkommen unabhängig vom Threadlock.

[34] Eine Ausnahme ist das Setzen des eigenen Threadlocks.

```
 1    sys_ipc
 2      do_send
 3        state_add
 4           r -> r | poll | sip | iip
 5        sender_enqueue
 6        ipc_send_regs
 7          ipc_try_lock
 8            lock
 9            sender_ok
10            clear
11         state_change_safely
12           r | poll | sip | iip -> poll | sip | iip
13         schedule
14            lock
15              cli
16            switch_to
```

Abbildung 20: Rendezvous, IPC-Senden

Dieses Verhalten kann in der Realität schwere Auswirkungen haben. Beispielsweise führt in L4Linux ein Serverthread den Kerncode von allen Linux-Prozessen aus. Diese senden dazu IPC. Im Mehrprozessorfall können Linux-User-Prozesse auf verschiedene Prozessoren verteilt werden. Sie senden damit untereinander vollkommen asynchron Anforderungen an den Linux-Kernel-Thread. Dieser wird durch den Versuch eine Linux-Prozess-Threads, eine IPC zu senden, in den (L4-)Kern gezwungen.

Dieser Missstand kann durch die Verwendung einer schwächeren Synchronisation, einer Beschränkung von IPC oder einer Umstrukturierung der bisherigen Implementierung beseitigt werden.

Als schwächere Synchronisation käme ein Helping- oder Spinlock in Frage. Beide gewährleisten gegenseitigen Ausschluss für konkurrierende Threads, bringen aber keine zusätzliche Semantik für das geschützte Objekt. Ein interessanter Punkt wäre die Integration dieses kritischen Abschnittes mit kritischen Abschnitten zur Implementierung von atomaren Operationen.

Ein Lösung mit weitreichenden Folgen ist die Einführung der von Rechte. Unkontrollierte Kommunikation allein auf der Basis von Threadidentifikatoren würde wahrscheinlich durch ein Indirektionsmodell ersetzt werden. Eine Diskussion dieser Frage geht über den Rahmen dieser Arbeit hinaus. Ein Rechteverwaltung ist aber wahrscheinlich das einzige angemessene Mittel, um Beeinträchtigungen von Threads durch andere feindliche Threads vollständig zu unterbinden. Gleichzeitig entfiele mit den freien Kommunikationsmöglichkeiten auch die Verfügbarkeit von *covert channels*. Beide bisher für L4 vorgeschlagenen Modelle [11], [4] konnten sich nicht durchsetzen.

Die vergleichsweise einfachste Lösung besteht in der Prüfung der Bedingung *vor* Setzen des Locks. Da das Lock keinen Einfluss auf die Synchronisation zwischen Sender und Empfänger hat, ist dieser Test problemlos möglich. Wird bereits beim Vortest erkannt, dass keine Empfangsbereitschaft vorliegt, kann auf das teure Setzen des Threadlocks verzichtet werden. Fällt

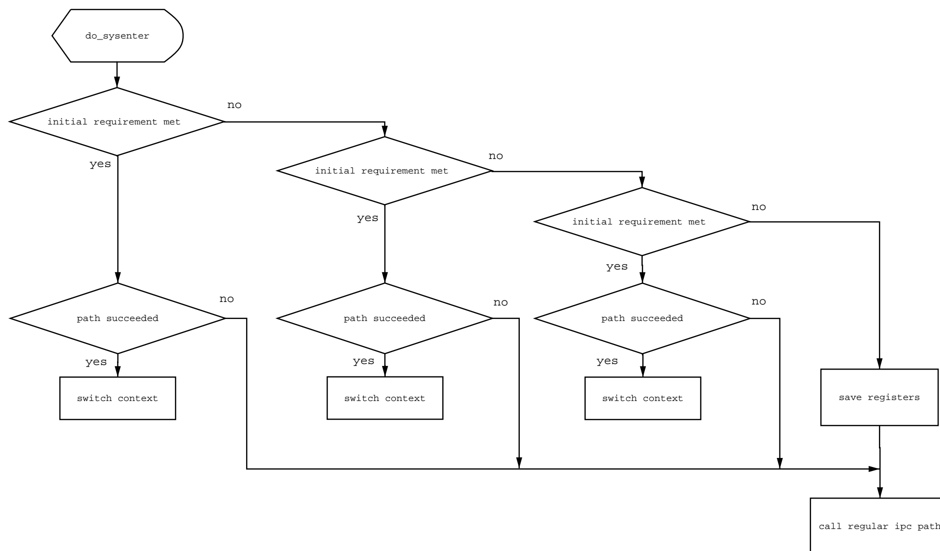


Abbildung 21: Multi-Fastpath Flussdiagramm

der Vortest positiv aus, muss das Threadlock gesetzt werden und nochmals getestet werden. Für die Synchronisation der Sender untereinander ist unbedingt ein Lock erforderlich. Nachteil dieser Lösung ist, dass der IPC-Pfad um einen Test und einen weiteren Vergleich verlängert wird.

4. Implementierung

In Abbildung 13 ist das Flussdiagramm für einen speziellen Fall des Fastpaths dargestellt. Ist eine Bedingung für diesen Fastpath verletzt, fällt er zurück in die konventionelle — langsame — Implementierung. Aufgrund der großen Leistungsunterschiede kann es sinnvoll sein, außer diesem am einfachsten zu implementierenden Pfad noch weitere, komplexere Fälle zu behandeln. Die Realisierung erfolgt mittels einer in Abbildung 4 dargestellten Struktur. In dieser Struktur werden die Implementierung quasi miteinander verkettet. Der Test, ob eine bestimmte IPC, die für die aktuelle Stufe geeignet ist, vorliegt erfolgt sequentiell. Nimmt keine der einzelne Pfade die IPC an, oder kann in einem späteren Test eine Bedingung nicht erfüllt werden, wird die IPC im regulären Pfad behandelt.

In Reihenfolge werden folgende Fälle behandelt: `call`, `reply-and-wait`, `send` und `receive/wait`. In allen Variante dürfen nur short IPC ohne Timeouts vorkommen.

Eine Unterstützung für ausschließlich den `call`-Fall brächte in einer Produktionsumgebung mit Klienten und Servern nur die Hälfte aller IPC in den Fastpath. `call` wird nur von Klienten bei Anfragen an ihren Server benutzt.

Die Server wiederum benutzen wieder `reply-and-wait`, um nach der Beantwortung einer Anfrage sofort die nächste von einem anderen Klienten annehmen zu können. Die Implementierung dieses Falles ist aufwendiger als das reine `wait`. Es muss zusätzlich geprüft werden, ob für den einkommenden Sender seinerseits ein Sender wartet. Da sich `replay-and-wait` nicht auf den Empfänger seiner Nachricht als Sender der Antwort festlegt, muss dieser Fall behandelt werden. Ebenso wie bei `call` sind in keinem Fall Einketteoperationen notwendig.

Mehrprozessor	**shortcut**	**shortcut classic**	**slow path**
Befehle	92	733	1153
Ausführungszeit PIII	292/566	2385/2659	3425/3705
Ausführungszeit P4	975/1895	6828/8982	9135/11352
Speicherzugriffe	37	329	580
Speicherverbrauch[35]	14	38	58
cli	0	4	6
pushf	0	5	5
cmpxchg	1	7	9
cmpxchg8b	0	4	7
bedingte Sprünge	13	89	113
ausgführte bedingte Sprünge	0	37	49

Tabelle 7: Metriken der Mehrprozessorvarianten.

Einzelprozessor	**shortcut**	**shortcut classic**	**slow path**
Befehle	61	277	728
Ausführungszeit PIII	188/453	814/1059	1857/2124
Ausführungszeit P4	614/1323	1562/3192	2813/5132
Speicherzugriffe	28	132	361
Speicherverbrauch	14	28	46
cli	0	2	8
pushf	0	3	7
cmpxchg	0	0	6
cmpxchg8b	0	0	0
bedingte Sprünge	9	27	85
ausgführte bedingte Sprünge	0	17	44

Tabelle 8: Metriken für die Einzelprozessorvariante.

Die Entscheidung `send` und `receive/wait` zu unterstützen viel aufgrund deren Verwendung bei der Implementierung von Semaphoren in L4Env. Die resultierenden Implementierung sind auch bedeutend umfangreicher und fehlerträchtiger als die beiden ersten Fälle. Die häufige Benutztung von Semaphoreprimitiven und die langsame Standard-IPC gaben dann doch den Ausschlag für die Unterstützung.

5. Ergebnisse und Bewertung

5.1. Facts and Figures

In den Abbildungen 7 und 8 sind detaillierte Zahlen zum Verhalten des IPC-Fastpaths aufgeführt. Zum Vergleich sind dazu die ursprüngliche Implementierung und eine Optimierung in C++ aufgeführt.

5.2. Werkzeuge

Während der zu dieser Arbeit parallel laufenden Entwicklungen entstanden einige sehr nützliche Werkzeuge. Diese sind durchaus allgemeiner Natur und werden hoffentlich auch bei künftigen Arbeiten und Untersuchungen Anwendung finden.

Mit Hilfe des *kernel level tracers* besteht die Möglichkeit, instruktionsgenau Ausführungspfade zu verfolgen. Fragen nach der Anzahl von ausgeführten/ nicht ausgeführten Sprüngen, von Speicherzugriffen, aufwendigen Instruktionen oder einfach Pfadlängen lassen sich erst mittels dieser automatischen Unterstützung mit vertretbarem Aufwand beantworten. Etliche Unregelmäßigkeiten, die in den Traces auffielen, ließen sich auf versteckt wirkende Mechanismen wie beispielsweise automatische Konstruktor-/Destruktoraufrufe zurückführen. Erst die instruktionsgenaue Offenlegung gibt ein Gefühl für die tatsächlichen Kosten eines einfach wirkenden Konstruktes wie beispielsweise eines Funktionsaufrufes.

Die Möglichkeit, gezielt im Kern Code auszuführen gestattet es erst, Programmteile aussagekräftig zu testen. Spezielle Testthreads wurden genau für diese Aufgabe in den Kern integriert. Durch ein einfaches Klasseninterface lassen sich mit geringem Aufwand neue Tests bauen. Alle Messungen dieser Arbeit wurden von Testthreads in Zusammenarbeit mit dem *kernel level tracer* durchgeführt.

Eine Reihe von Python-Skripten korreliert die numerischen Adressen der Traces mit symbolischen Namen des Programms. Aus verwirrenden Zahlenkolonnen werden so aussagekräfige Aufrufgraphen.

5.3. Bewertung und Ausblick

Das Ergebnis der Arbeit hat gezeigt, dass Echtzeitfähigkeit und schnelle IPC sich keinesfalls ausschließen. Der eingeschlagene Weg, aufwendige Kernsynchronisierungsmittel bei vertretbaren Latenzen zu umgehen, wird in Zukunft weiter untersucht werden müssen. Vollkommen unterbrechbare Kerne sind nicht notwendig. Wenn Kerneintrittszeiten wie aus Tabelle 9 entnehmbar im Bereich von Hunderten Takten liegen, dann kann diese Größe durchaus auch als Richtlinie für die Kernpreemptierbarkeit dienen. Nicht blockierende Synchronisation wird auch in Zukunft eine wichtige Technik bleiben, um komplexe Operationen mit geringem Aufwand unterbrechbar zu machen. Es wird aber stärker hinterfragt werden, ob man mit Hinblick auf größere Leistungsfähigkeit nicht zu anderen Synchronisationsmethoden wechseln sollte. Ein nächster Schritt könnte der Neuentwurf des IPC-Pfades sein, wobei in der Anbahnungsphase Unterbrechbarkeit nicht mehr die heutige Bedeutung haben wird.

Die Entwicklung hin zu Mehrprozessormaschinen mit stark segmentierten Speichersubsystemen verlangt nach Ansätzen, die deren Ausführungscharakteristika Rechnung tragen. Die Nutzung von speicherbasierten Synchronisationsschemas ist ein Ansatz, der in Zukunft weiterverfolgt werden sollte. Stärker skalierbare Techniken werden in Mikrokernen Einzug halten müssen, sollen weiterhin eine Vielzahl von möglichen Anwendungsfeldern — darunter solche, die Skalierbarkeit benötigen — unterstützt werden. Geschieht dies nicht, könnte dies Mikrokernidee als erwiesenermaßen leistungsunfähig wieder verschwinden. Die Geschichte der Mikrokerne der ersten Generation sollte als Warnung dienen.

Die erreichte Leistung ist ein Indiz, dass die am Anfang der Arbeit angenommenen Optimierungskriterien richtig sind. Gleichzeitig sind aber auch Widersprüchlichkeiten zu beobachten: So *reduzierte* das Einfügen redundanter bedingter Sprünge die Ausführungszeit um ca. 5%. Auch die Verringerung von Speicherreferenzen zu Lasten arithmetischer Operationen auf Re-

	PIII 450Mhz	P4 1.6GHz
sysenter	52	172
sysexit	57	252
int	114	700
iret	149	844

Tabelle 9: Kosten für Kernein- und -austritt.

gistern verschlechterte die Ausführungszeit. Über die Gründe kann nur spekuliert werden: Die Implementierungsdetails heutiger hochoptimierter Prozessorkerne sind vielfältiger als dies bei oberflächlicher Betrachtung erscheinen mag. Viele Größen und Algorithmen sind nicht dokumentiert und können nur aufgrund von experimentellen Ergebnissen abgeschätzt werden. Die Einfluss nehmenden Komponenten bei diesen Experimenten sind nicht immer klar erkennbar. Schon die Werte in den Tabellen 9 und 8 verdeutlichen das Problem. Allein die Kerneintrittkosten summieren sich zu einem Wert größer als die gesamte Roundtrip-Zeit. Die Ursache für diesen offensichtlichen Widerspruch liegt wahrscheinlich in der Messmethodik für die Kerneintrittskosten. Das möglicherweise noch vorhandene Parallelisierungspotential ging durch die komplexen, zur Zeitmessung verwendeten Instruktionen verloren. Einfachere Befehle wären vermutlich parallel ausgeführt worden.

Diese ernüchternden Erfahrungen legen nahe, dem Befehlsscheduling in Zukunft weniger Aufmerksamkeit zu widmen. Stattdessen erscheinen Modelle für die systemweit sinnvolle Nutzung der Speicherhierarchie erfolgversprechender. Es sind für den Fastpath keine großen Leistungsverbesserungen zu erwarten. Erstens bewegen sich die unvermeidbaren Kosten für den Kerneintritt bereits in der Größenordnung der Fastpathausführungszeit, zweitens kann für alle Instruktionen im Fastpath eine Begründung für ihre Existenz an dieser Stelle gegeben werden. Eine genauere Abschätzung wie in [2] erscheint aufgrund von Neuerungen wie globalen TLB-Einträgen und den oben beschriebenen Schwierigkeiten ebenfalls unwahrscheinlich.

Die möglichen Folgen ungewollter Interaktion und der Mangel an Mitteln, dies zu unterbinden, deuten auf erheblichen Entwicklungsaufwand in dieser Richtung. Neben den sicherheitstechnischen Erwägungen kann durchaus auch Leistung eine Rolle spielen. Wird beispielsweise IPC auf Threads eines Prozessors beschränkt, wäre es denkbar, dass zur Ausführung dieser IPC spezialisierte Funktionen zum Einsatz kommen, die nur lokal korrekt — dafür aber schnell — synchronisieren. Ohne explizites Wissen ist die Erkennung eines solchen Falles bedeutend schwieriger.

A. Fastpath, Einzelprozessor

```
#define ASSEMBLER_IPC_SHORT_CUT
#ifdef ASSEMBLER_IPC_SHORT_CUT

#define ASSEMBLER

#include <flux/x86/paging.h>
#include <flux/x86/seg.h>
#include <flux/x86/proc_reg.h>
#include <flux/x86/asm.h>

#include "config_gdt.h"
#include "config_tcbsize.h"
#include "shortcut.h"
#include "../tcboffset.h"

#define L4_IPC_RECANCELED               0x40
#define L4_IPC_RETIMEOUT                0x20

#define SYSENTER_DEBUG 0
#define STATISTICS     0
#define SET_SEG_SEL    0
#define FAST_TEST      0
#define THREADID_TEST  0
#define SYSENTER_STACK_OFFSET              8

#define RESET_KERNEL_SEGMENTS                   \
        movw    $ (GDT_DATA_USER|SEL_PL_U), %cx ;\
        movl    %ds,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%ds                        ;\
1:      movl    %es,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%es                        ;\
1:

#define RESET_USER_SEGMENTS                     \
        movw    $ (GDT_DATA_USER|SEL_PL_U), %cx ;\
        movl    %fs,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%fs                        ;\
1:      movl    %gs,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%gs                        ;\
1:

#define RESET_THREAD_STATE_AT(reg)              \
        andl    $~Thread_cancel, (reg)

#define SAVE_STATE                              \
        pushl   %ebp                            ;\
        pushl   %ebx                            ;\
        pushl   %edi                            ;\
        pushl   %esi                            ;\
        pushl   %edx                            ;\
        pushl   %ecx

#define RESTORE_STATE                           \
        popl    %ecx                            ;\
        popl    %edx                            ;\
        popl    %esi                            ;\
        popl    %edi                            ;\
        popl    %ebx                            ;\
        popl    %ebp                            ;\

#define REG_ECX
#define REG_EDX (1*4)
#define REG_ESI (2*4)
#define REG_EDI (3*4)
#define REG_EBX (4*4)
#define REG_EBP (5*4)
#define REG_EAX (6*4)
#define REG_EIP (7*4)
#define REG_ESP (10*4)

#define OFS__THREAD__EIP 0x7ec
#define OFS__THREAD__ESP 0X7f8
#define OFS__THREAD__EBX 0X7e0
#define OFS__THREAD__EDX 0X7d0

// ipc entry point for int 0x30
        .data
        .p2align 2
        .global shortcut_failed_cnt
```

```
shortcut_failed_cnt:
        .long  0

        .p2align 2
        .global se_shortcut_failed_cnt
se_shortcut_failed_cnt:
        .long  0
        .previous


        .data
        .global se_ipc_cnt
se_ipc_cnt:
        .long 0

        .global se_path2_cnt
se_path2_cnt:
        .long 0


        .global se_path3_cnt
se_path3_cnt:
        .long 0
        .previous


        .align  16
        .globl  do_sysenter
do_sysenter:
#if STATISTICS
        incl    se_ipc_cnt
#endif
        movl    (%esp), %esp
#if SYSENTER_STACK_OFFSET != 48
        // eflags, eip, esp, 7 * regs
        subl    $(48 - SYSENTER_STACK_OFFSET), %esp

#endif
        /* use kmem::kernel_esp() instead of dedicated pointer */
/*      subl    $48, %esp       // eflags, eip, esp, 7 * regs   */


#if SET_SEG_SEL
        movl    $(GDT_CODE_USER|SEL_PL_U), 32(%esp) // set user cs
        movl    $(GDT_DATA_USER|SEL_PL_U), 44(%esp) // set user ss
#endif

        movl    %eax, REG_EAX(%esp)
        movl    %ebp, REG_EBP(%esp)


        orl     %ebp, %eax       // eax = snd desc or rcv desc
        orl     4(%ecx), %eax    // eax = snd desc or rcv desc or timeout

        jne     fast_path2

        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip



#define DEST_ebp %ebp
#define CURR_eax %eax

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax

        testl   $Thread_fpu_active, OFS__THREAD__STATE(CURR_eax)
        jne     se_shortcut_call_failed_fpu

        // calculate tcb from thread id
        leal    (%esi,%esi), DEST_ebp

        andl    $0x1ffff800, DEST_ebp
        orl     $0xc0000000, DEST_ebp           // dest = dst_id.lookup

        cmpl    %esi,OFS__THREAD__ID(DEST_ebp)  // do thread and id match
        jne     se_shortcut_call_failed_id_mismatch

        // dest->thread_lock()->test()
        cmpl    $0, OFS__THREAD__THREAD_LOCK__SWITCH_LOCK__LOCK_OWNER (DEST_ebp)
        jne     se_shortcut_call_failed_lock            // dest is locked

        // read dest state
        movl    OFS__THREAD__STATE (DEST_ebp), %ecx

        andl    $(Thread_receiving | Thread_waiting | \
                  Thread_send_in_progress | Thread_ipc_in_progress ), %ecx
```

```
        // ipc_state == (Thread_receiving | Thread_ipc_in_progress)?
        cmpl    $(Thread_receiving | Thread_ipc_in_progress), %ecx
        jne     test_on_wait

        // partner() == sender?
        leal    OFS__THREAD__ID (CURR_eax), %ecx// (sender_t*)this
        cmpl    %ecx, OFS__THREAD__PARTNER (DEST_ebp)

        jne     se_shortcut_call_failed_wrong_sender

se_sender_ok:
        .data
        .global se_sender_ok_cnt
se_sender_ok_cnt:
        .long 0
        .previous

#if STATISTICS
        incl se_sender_ok_cnt
#endif

        // wake up receiver
        movl OFS__THREAD__STATE(DEST_ebp), %ecx
        andl $~(Thread_ipc_receiving_mask | Thread_ipc_in_progress), %ecx
        orl  $Thread_running, %ecx
        movl %ecx, OFS__THREAD__STATE(DEST_ebp)

        xorl    $(Thread_receiving | Thread_ipc_in_progress | Thread_running) , OFS__THREAD__STATE(CURR_eax)

        leal    OFS__THREAD__ID (DEST_ebp), %ecx// (sender_t*)this
        movl    %ecx, OFS__THREAD__PARTNER (CURR_eax)

        leal    0x7d0(%eax), %ecx
        movl    %ecx, OFS__THREAD__RECEIVE_REGS(CURR_eax)

        // no valid ebp available
        pushl   $0
        // push restart address on old stack
        pushl   $se_ret_switch

        movl    %esp, OFS__THREAD__KERNEL_SP (CURR_eax)

        // *(kmem::kernel_esp()) = reinterpret_cast<vm_offset_t>(regs() + 1);
        movl    _4kmem.tss, %ecx
        leal    THREAD_BLOCK_SIZE (DEST_ebp), %edi
        movl    %edi, 4 (%ecx)  // x86_tss.esp0

        // sysenter_stack = reinterpret_cast<vm_offset_t>(&regs()->esp);
        leal    THREAD_BLOCK_SIZE - SYSENTER_STACK_OFFSET (DEST_ebp), %ecx
        movl    %ecx, EXT(sysenter_stack)

        movl    CURR_eax, %ecx
        xorl    DEST_ebp, %ecx
        testl   $0x1ffc0000, %ecx              // same task ?
        jne     fast_path_switch_address_space
address_space_switched:

        movl    OFS__THREAD__KERNEL_SP(DEST_ebp), %ecx
        cmpl    $se_ret_switch, (%ecx)
        jne     fast_path_slow_switch


        movl    OFS__THREAD__ID(CURR_eax), %esi
        movl    4+OFS__THREAD__ID(CURR_eax), %edi
        movl    %ebp, %eax
        movl    %edx, %ebp          // copy msg.w
        movl    OFS__THREAD__EIP(%eax), %edx   // load eip
        movl    OFS__THREAD__ESP(%eax), %ecx   // load esp
        movl    $0x4000, %eax        // set msg dope

        sti
        sysexit


test_on_wait:
        // ipc_state == (Thread_waiting | Thread_ipc_in_progress)?
        cmpl    $(Thread_waiting | Thread_ipc_in_progress), %ecx
        je      se_sender_ok                    // open wait
        jmp     se_shortcut_call_failed_no_rcv          // !dest->sender_ok

        .globl fast_path_slow_switch
fast_path_slow_switch:
        movl    OFS__THREAD__ID(CURR_eax),         %esi
        movl    4+OFS__THREAD__ID(CURR_eax),         %edi
        movl    OFS__THREAD__KERNEL_SP (DEST_ebp), %eax

        testl   %eax, %eax                      // check new stack pointer
        jz      se_no_switch_pop                // fail
```

```
        movl    %eax, %esp                      // load new stack pointer
        movl    OFS__THREAD__RECEIVE_REGS(DEST_ebp), %eax // load receive regs

        movl    %ebx,    REG_EBX(%eax)
        movl    %edx,    REG_EDX(%eax)
        movl    $0x4000, REG_EAX(%eax)
        movl    %esi,    REG_ESI(%eax)
        movl    %edi,    REG_EDI(%eax)

        ret

fast_path_switch_address_space:
        movl    OFS__THREAD__SPACE_CONTEXT (DEST_ebp), %ecx
        subl    $physmem, %ecx
        movl    %ecx, %cr3
        jmp address_space_switched


#if STATISTICS
#define SHORTCUT_FAILED(reason)                         \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_failed_##reason##_cnt        ;\
se_shortcut_failed_##reason##_cnt:                      ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_failed_##reason##:                          ;\
        incl se_shortcut_failed_##reason##_cnt          ;\
        jmp se_shortcut_failed
#else
#define SHORTCUT_FAILED(reason)                         \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_failed_##reason##_cnt        ;\
se_shortcut_failed_##reason##_cnt:                      ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_failed_##reason##:                          ;\
        jmp se_shortcut_failed
#endif

#if STATISTICS
#define SHORTCUT_CALL_FAILED(reason)                     \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_call_failed_##reason##_cnt   ;\
se_shortcut_call_failed_##reason##_cnt:                 ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_call_failed_##reason##:                     ;\
        incl se_shortcut_call_failed_##reason##_cnt     ;\
        movl    $0, REG_EAX(%esp)                       ;\
        movl    $0, REG_EBP(%esp)                       ;\
        jmp se_shortcut_failed
#else
#define SHORTCUT_CALL_FAILED(reason)                     \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_call_failed_##reason##_cnt   ;\
se_shortcut_call_failed_##reason##_cnt:                 ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_call_failed_##reason##:                     ;\
        movl    $0, REG_EAX(%esp)                       ;\
        movl    $0, REG_EBP(%esp)                       ;\
        jmp se_shortcut_failed
#endif



SHORTCUT_FAILED(wrong_sender)
//SHORTCUT_FAILED(running)
SHORTCUT_FAILED(no_rcv)
SHORTCUT_FAILED(lock)
SHORTCUT_FAILED(irq_or_sender)
SHORTCUT_FAILED(fpu)
SHORTCUT_FAILED(ready_list)

SHORTCUT_CALL_FAILED(no_rcv)
SHORTCUT_CALL_FAILED(lock)
SHORTCUT_CALL_FAILED(wrong_sender)
SHORTCUT_CALL_FAILED(fpu)
SHORTCUT_CALL_FAILED(id_mismatch)

        .align  8
fast_path2:
#if STATISTICS
        incl    se_path2_cnt
#endif
```

```
        subl    $1, %ebp              // open wait ?
        orl     REG_EAX(%esp), %ebp

        jne     fast_path3

        // reply and wait

        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax

        RESET_THREAD_STATE_AT(%eax)

        testl   $Thread_fpu_active, OFS__THREAD__STATE(CURR_eax)
        jne     se_shortcut_failed_fpu


        // irq associated or sender waiting ?
        movl    OFS__THREAD__IRQ(CURR_eax), %ecx
        orl     OFS__THREAD__SENDER_FIRST(CURR_eax), %ecx
        jne     se_shortcut_failed_irq_or_sender

        // calculate tcb from thread id
        leal    (%esi,%esi), DEST_ebp

        andl    $0x1ffff800, DEST_ebp
        orl     $0xc0000000, DEST_ebp           // dest = dst_id.lookup

        // dest->thread_lock()->test()
        cmpl    $0, OFS__THREAD__THREAD_LOCK__SWITCH_LOCK__LOCK_OWNER (DEST_ebp)
        jne     se_shortcut_failed_lock         // dest is locked

        // read dest state
        movl    OFS__THREAD__STATE (DEST_ebp), %ecx

        andl    $(Thread_receiving | Thread_waiting | \
                  Thread_send_in_progress | Thread_ipc_in_progress ), %ecx

        // ipc_state == (Thread_waiting | Thread_ipc_in_progress)?
        cmpl    $(Thread_waiting | Thread_ipc_in_progress), %ecx
        je      se_sender_ok2                   // open wait

        // ipc_state == (Thread_receiving | Thread_ipc_in_progress)?
        cmpl    $(Thread_receiving | Thread_ipc_in_progress), %ecx
        jne     se_shortcut_failed_no_rcv               // !dest->sender_ok

        // partner() == sender?
        leal    OFS__THREAD__ID (CURR_eax), %ecx// (sender_t*)this
        cmpl    %ecx, OFS__THREAD__PARTNER (DEST_ebp)
        jne     se_shortcut_failed_wrong_sender // !dest->sender_ok

se_sender_ok2:
        .data
        .global se_sender_ok2_cnt
        .p2align 2
se_sender_ok2_cnt:
        .long 0
        .previous
#if STATISTICS
        incl se_sender_ok2_cnt
#endif
        // wake up receiver
        andl    $~(Thread_ipc_receiving_mask | \
                   Thread_ipc_in_progress), OFS__THREAD__STATE (DEST_ebp)
        orb     $Thread_running, OFS__THREAD__STATE (DEST_ebp)

        xorl    $(Thread_ipc_in_progress | Thread_waiting | Thread_running),\
                OFS__THREAD__STATE(CURR_eax)
        movl    $0x4000, REG_EAX(%esp)


        // no valid ebp available
        pushl   $0
        // push restart address on old stack
        pushl   $se_ret_switch

        movl    %esp, OFS__THREAD__KERNEL_SP (CURR_eax)

        // *(kmem::kernel_esp()) = reinterpret_cast<vm_offset_t>(regs() + 1);
        movl    _4kmem.tss, %ecx
        leal    THREAD_BLOCK_SIZE (DEST_ebp), %edi
        movl    %edi, 4 (%ecx)  // x86_tss.esp0


        // sysenter_stack = reinterpret_cast<vm_offset_t>(&regs()->esp);
```

```
        leal    THREAD_BLOCK_SIZE - SYSENTER_STACK_OFFSET (DEST_ebp), %ecx
        movl    %ecx, EXT(sysenter_stack)


        movl    CURR_eax, %ecx
        xorl    DEST_ebp, %ecx
        testl   $0x1ffc0000, %ecx              // same task ?
        jne     fast_path2_switch_address_space
address_space_switched2:

        movl    OFS__THREAD__KERNEL_SP(DEST_ebp), %ecx
        cmpl    $se_ret_switch, (%ecx)
        jne     fast_path2_slow_switch

        movl    OFS__THREAD__ID(CURR_eax), %esi
        movl    %ebp, %eax
        movl    %edx, %ebp          // copy msg.w
        movl    OFS__THREAD__EIP(%eax), %edx   // load eip
        movl    OFS__THREAD__ESP(%eax), %ecx   // load esp
        movl    $0x4000, %eax        // set msg dope

        sti
        sysexit

fast_path2_switch_address_space:
        movl    OFS__THREAD__SPACE_CONTEXT (DEST_ebp), %ecx
        subl    $physmem, %ecx
        movl    %ecx, %cr3
        jmp address_space_switched2

fast_path2_slow_switch:

        movl    OFS__THREAD__ID(CURR_eax),         %esi
        movl    4+OFS__THREAD__ID(CURR_eax),         %edi
        movl    OFS__THREAD__KERNEL_SP (DEST_ebp), %eax

        testl   %eax, %eax                      // check new stack pointer
        jz      se_no_switch_pop                // fail

        movl    %eax, %esp                      // load new stack pointer
        movl    OFS__THREAD__RECEIVE_REGS(DEST_ebp), %eax // load receive regs

        // save registers
        movl    %ebx,    REG_EBX(%eax)
        movl    %edx,    REG_EDX(%eax)
        movl    $0x4000, REG_EAX(%eax)
        movl    %esi,    REG_ESI(%eax)
        movl    %edi,    REG_EDI(%eax)

        ret

        .align  8
fast_path3:
#if STATISTICS
        incl    se_path3_cnt
#endif

        addl    $2, %ebp              // no receive, ebp
                                      // was decremented in fast_path2
        orl     REG_EAX(%esp), %ebp
        jne     fast_path4

        // snd only

        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax

        RESET_THREAD_STATE_AT(%eax)

        testl   $Thread_fpu_active, OFS__THREAD__STATE(CURR_eax)
        jne     se_shortcut_failed_fpu


        // calculate tcb from thread id
        leal    (%esi,%esi), DEST_ebp

        andl    $0x1ffff800, DEST_ebp
        orl     $0xc0000000, DEST_ebp           // dest = dst_id.lookup

        // dest->thread_lock()->test()
        cmpl    $0, OFS__THREAD__THREAD_LOCK__SWITCH_LOCK__LOCK_OWNER (DEST_ebp)
        jne     se_shortcut_failed_lock         // dest is locked

        // read dest state
        movl    OFS__THREAD__STATE (DEST_ebp), %ecx
```

```
        andl    $(Thread_receiving | Thread_waiting | \
                  Thread_send_in_progress | Thread_ipc_in_progress ), %ecx

        // ipc_state == (Thread_waiting | Thread_ipc_in_progress)?
        cmpl    $(Thread_waiting | Thread_ipc_in_progress), %ecx
        je      se_sender_ok3                   // open wait

        // ipc_state == (Thread_receiving | Thread_ipc_in_progress)?
        cmpl    $(Thread_receiving | Thread_ipc_in_progress), %ecx
        jne     se_shortcut_failed_no_rcv               // !dest->sender_ok

        // partner() == sender?
        leal    OFS__THREAD__ID (CURR_eax), %ecx// (sender_t*)this
        cmpl    %ecx, OFS__THREAD__PARTNER (DEST_ebp)
        jne     se_shortcut_failed_wrong_sender // !dest->sender_ok


        .data
        .global se_sender_ok3_cnt
        .p2align 2
se_sender_ok3_cnt:
        .long 0
        .previous

se_sender_ok3:
#if STATISTICS
        incl se_sender_ok3_cnt
#endif
        // wake up receiver
        andl    $~(Thread_ipc_receiving_mask | \
                   Thread_ipc_in_progress), OFS__THREAD__STATE (DEST_ebp)
        orb     $Thread_running, OFS__THREAD__STATE (DEST_ebp)

        xorl    %ecx, %ecx
        movw    OFS__THREAD__SCHED__PRIO(CURR_eax), %cx
        cmpw    OFS__THREAD__SCHED__PRIO(DEST_ebp), %cx

        jl      se_path3_switch

        // copy words and sender id to receiver regs
        movl    OFS__THREAD__RECEIVE_REGS(%ebp), %ebp



        movl    OFS__THREAD__ID(CURR_eax),    %esi
        movl    4+ OFS__THREAD__ID(CURR_eax), %edi

        movl    %ebx, REG_EBX(%ebp)
        movl    %edx, REG_EDX(%ebp)

        movl    %esi, REG_ESI(%ebp)
        movl    %edi, REG_EDI(%ebp)

        // dest in ready list
        andl    $~(THREAD_BLOCK_SIZE-1), %ebp        // get rid of regs offset
        cmpl    $0, OFS__THREAD__READY_NEXT(%ebp)
        je      dest_enqueue
dest_enqueued:

        // prepare for sysexit
        movl    OFS__THREAD__EIP(%eax), %edx   // load eip
        movl    OFS__THREAD__ESP(%eax), %ecx   // load esp

        movl    $0x4000, %eax        // set msg dope
        sysexit

dest_enqueue:
        xorl    %edx, %edx
        movzwl  OFS__THREAD__SCHED__PRIO (DEST_ebp), %esi

        cmpl    %edx, _9context_t.prio_first (, %esi, 4)
        jne     1f
        movl    DEST_ebp, _9context_t.prio_first (, %esi, 4)
1:      cmpl    %edx, _9context_t.prio_next (, %esi, 4)
        jne     1f
        movl    DEST_ebp, _9context_t.prio_next (, %esi, 4)
1:      cmpl    _9context_t.prio_highest, %esi
        jbe     1f
        movl    %esi, _9context_t.prio_highest

1:      // i = (i+1) % 255
        .globl enqueue_marker2
enqueue_marker2:

        incl    %esi
        andl    $255, %esi

        // edx = sibling = prio_first[i]
        movl    _9context_t.prio_first (, %esi, 4), %edx
```

```
        testl   %edx, %edx
        jz      1b

        movl    OFS__THREAD__READY_PREV (%edx), %ecx
        // edx = sibling
        // ecx = sibling->ready_prev

        // ready_next = sibling
        movl    %edx, OFS__THREAD__READY_NEXT (DEST_ebp)

        // ready_prev = sibling->ready_prev
        movl    %ecx, OFS__THREAD__READY_PREV (DEST_ebp)

        // sibling->ready_prev = this
        movl    DEST_ebp, OFS__THREAD__READY_PREV (%edx)

        // sibling->ready_prev->ready_next = this
        movl    DEST_ebp, OFS__THREAD__READY_NEXT (%ecx)


        jmp dest_enqueued

se_path3_switch:

        movl    $0x4000, REG_EAX(%esp)

        // no valid ebp available
        pushl   $0
        // push restart address on old stack
        pushl   $se_ret_switch

        movl    %esp, OFS__THREAD__KERNEL_SP (CURR_eax)

        // *(kmem::kernel_esp()) = reinterpret_cast<vm_offset_t>(regs() + 1);
        movl    _4kmem.tss, %ecx
        leal    THREAD_BLOCK_SIZE (DEST_ebp), %edi
        movl    %edi, 4 (%ecx)  // x86_tss.esp0

        // sysenter_stack = reinterpret_cast<vm_offset_t>(&regs()->esp);
        leal    THREAD_BLOCK_SIZE - SYSENTER_STACK_OFFSET (DEST_ebp), %ecx
        movl    %ecx, EXT(sysenter_stack)

        movl    CURR_eax, %ecx
        xorl    DEST_ebp, %ecx
        testl   $0x1ffc0000, %ecx             // same task ?
        jne     fast_path3_switch_address_space
address_space_switched3:

        // current in ready list ?
        cmpl    $0,OFS__THREAD__READY_NEXT(CURR_eax)
        je      curr_enqueue
curr_enqueued:
        movl    OFS__THREAD__KERNEL_SP(DEST_ebp), %ecx
        cmpl    $se_ret_switch, (%ecx)
        jne     fast_path3_slow_switch

        movl    OFS__THREAD__ID(CURR_eax), %esi
        movl    4+OFS__THREAD__ID(CURR_eax), %edi


        movl    %ebp, %eax
        movl    %edx, %ebp          // copy msg.w
        movl    OFS__THREAD__EIP(%eax), %edx   // load eip
        movl    OFS__THREAD__ESP(%eax), %ecx   // load esp
        movl    $0x4000, %eax        // set msg dope

        sysexit

curr_enqueue:
        xorl    %edx, %edx
        movzwl  OFS__THREAD__SCHED__PRIO (CURR_eax), %esi

        movl    _9context_t.prio_first (, %esi, 4),%edi
        cmpl    %edx, %edi
        cmove   CURR_eax, %edi
        movl    %edi, _9context_t.prio_first (, %esi, 4)


        movl    _9context_t.prio_next (, %esi, 4),%edi
        cmpl    %edx, %edi
        cmove   CURR_eax, %edi
        movl    %edi, _9context_t.prio_next (, %esi, 4)


        movl    _9context_t.prio_highest, %edi
        cmpl    %edi, %esi
        cmova   %esi, %edi
        movl    %esi, _9context_t.prio_highest

1:      // i = (i+1) % 255
```

```
        .globl enqueue_marker1
enqueue_marker1:
        incl    %esi
        andl    $255, %esi

        // edx = sibling = prio_first[i]
        movl    _9context_t.prio_first (, %esi, 4), %edx
        testl   %edx, %edx
        jz      1b

        movl    OFS__THREAD__READY_PREV (%edx), %ecx
        // edx = sibling
        // ecx = sibling->ready_prev

        // ready_next = sibling
        movl    %edx, OFS__THREAD__READY_NEXT (CURR_eax)

        // ready_prev = sibling->ready_prev
        movl    %ecx, OFS__THREAD__READY_PREV (CURR_eax)

        // sibling->ready_prev = this
        movl    CURR_eax, OFS__THREAD__READY_PREV (%edx)

        // sibling->ready_prev->ready_next = this
        movl    CURR_eax, OFS__THREAD__READY_NEXT (%ecx)

        jmp curr_enqueued

fast_path3_slow_switch:
        movl    OFS__THREAD__ID(CURR_eax),         %esi
        movl    4+OFS__THREAD__ID(CURR_eax),         %edi
        movl    OFS__THREAD__KERNEL_SP (DEST_ebp), %eax

        testl   %eax, %eax                      // check new stack pointer
        jz      se_no_switch_pop                // fail

        movl    %eax, %esp                      // load new stack pointer
        movl    OFS__THREAD__RECEIVE_REGS(DEST_ebp), %eax // load receive regs

        movl    %ebx,    REG_EBX(%eax)
        movl    %edx,    REG_EDX(%eax)
        movl    $0x4000, REG_EAX(%eax)
        movl    %esi,    REG_ESI(%eax)
        movl    %edi,    REG_EDI(%eax)

        ret

fast_path3_switch_address_space:
        movl    OFS__THREAD__SPACE_CONTEXT (DEST_ebp), %ecx
        subl    $physmem, %ecx
        movl    %ecx, %cr3
        jmp address_space_switched3


        .data
        .globl se_path4_cnt
        .p2align 2
se_path4_cnt:
        .long 0
        .previous

        .align  8
fast_path4:
#if STATISTICS
        incl    se_path4_cnt
#endif

        // no timeouts
        cmpl   $0, 4(%ecx)
        jne    fast_path4_failed

        cmpl   $0xffffffff, REG_EAX(%esp)  // send involved ?
        jne     fast_path4_failed

        cmpl   $1, REG_EBP(%esp)

        // ebp == 1
        je se_wait
        // ebp == 0
        jc se_rcv

        jmp     fast_path4_failed


se_wait:
        .data
        .global se_wait_cnt
se_wait_cnt:
        .long 0
        .previous
```

```
#if STATISTICS
        incl se_wait_cnt
#endif


        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax

        // irq associated or sender waiting ?
        movl    OFS__THREAD__IRQ(CURR_eax), %ecx
        orl     OFS__THREAD__SENDER_FIRST(CURR_eax), %ecx
        jne     se_shortcut_failed_irq_or_sender


        andl    $~(Thread_ipc_mask | Thread_running), OFS__THREAD__STATE(CURR_eax)
        orl     $(Thread_ipc_in_progress | Thread_waiting), OFS__THREAD__STATE(CURR_eax)
//      xorl    $(Thread_ipc_in_progress | Thread_waiting | Thread_running), OFS__THREAD__STATE(CURR_eax)
        movl    $0, OFS__THREAD__PARTNER(CURR_eax)
        movl    %esp, OFS__THREAD__RECEIVE_REGS(CURR_eax)


        movl    OFS__THREAD__READY_PREV(CURR_eax), %ebp
        testl   %ebp, %ebp
        je      do_schedule
//      jmp     do_schedule
        .globl search_prev
search_prev:
        cmpl    $0xc0000000, %ebp
        je      search_next
        testl   $Thread_running, OFS__THREAD__STATE(%ebp)
        jne     do_wait_switch
        movl    OFS__THREAD__READY_PREV(%ebp), %ebp
        testl   %ebp, %ebp
        je      do_schedule
        jmp     search_prev
        .globl search_next
search_next:
        movl    OFS__THREAD__READY_NEXT(CURR_eax), %ebp
        testl   %ebp, %ebp
        je      do_schedule
        .globl search_next2
search_next2:
        testl   $Thread_running, OFS__THREAD__STATE(%ebp)
        jne     do_wait_switch
        movl    OFS__THREAD__READY_NEXT(%ebp), %ebp
        testl   %ebp, %ebp
        je      do_schedule

        jmp     search_next2

        .globl do_wait_switch
do_wait_switch:
        // no valid ebp available
        pushl   $0
        // push restart address on old stack
        pushl   $se_ret_switch

        movl    %esp, OFS__THREAD__KERNEL_SP (CURR_eax)
        // *(kmem::kernel_esp()) = reinterpret_cast<vm_offset_t>(regs() + 1);
        movl    _4kmem.tss, %ecx
        leal    THREAD_BLOCK_SIZE (DEST_ebp), %edi
        movl    %edi, 4 (%ecx)  // x86_tss.esp0
        // sysenter_stack = reinterpret_cast<vm_offset_t>(&regs()->esp);
        leal    THREAD_BLOCK_SIZE - SYSENTER_STACK_OFFSET (DEST_ebp), %ecx
        movl    %ecx, EXT(sysenter_stack)
        movl    CURR_eax, %ecx
        xorl    DEST_ebp, %ecx
        testl   $0x1ffc0000, %ecx              // same task ?
        jne     fast_path4_switch_address_space
address_space_switched4:

        movl    OFS__THREAD__KERNEL_SP (DEST_ebp), %edx

        movl    %esp, last_esp4
        movl    %edx, %esp                      // load new stack pointer
        ret

        .globl last_esp4
        .data
last_esp4:
        .long 0
        .previous

fast_path4_switch_address_space:
        movl    OFS__THREAD__SPACE_CONTEXT (DEST_ebp), %ecx
```

```
        subl    $physmem, %ecx
        movl    %ecx, %cr3
        jmp address_space_switched4



do_schedule:
        pushl   %eax
        call    schedule__9context_t
        movl    $0,(%esp)
        movl    4+REG_EAX(%esp), %eax

        jmp se_ret_switch

se_rcv:
        .data
        .global se_rcv_cnt
se_rcv_cnt:
        .long 0
        .previous
#if STATISTICS
        incl se_rcv_cnt
#endif

        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax

        // irq associated or sender waiting ?
        movl    OFS__THREAD__IRQ(CURR_eax), %ecx
        orl     OFS__THREAD__SENDER_FIRST(CURR_eax), %ecx
        jne     se_shortcut_failed_irq_or_sender


        andl    $~(Thread_ipc_mask | Thread_running), OFS__THREAD__STATE(CURR_eax)
        orl     $(Thread_ipc_in_progress | Thread_receiving), OFS__THREAD__STATE(CURR_eax)

//      xorl    $(Thread_ipc_in_progress | Thread_receiving | Thread_running), OFS__THREAD__STATE(CURR_eax)

        leal    (%esi,%esi), DEST_ebp
        andl    $0x1ffff800, DEST_ebp
        orl     $(0xc0000000|OFS__THREAD__ID), DEST_ebp

        movl    DEST_ebp, OFS__THREAD__PARTNER(CURR_eax)
        movl    %esp, OFS__THREAD__RECEIVE_REGS(CURR_eax)

        pushl   %eax
        call    schedule_wrapper
        movl    $0, (%esp)
        movl    4+REG_EAX(%esp), %eax
        jmp se_ret_switch


fast_path4_failed_early:
        // restore eax
        xorl    %ebp, %eax
        xorl    4(%ecx), %eax
        movl    %ebp, REG_EBP(%esp)
        movl    %eax, REG_EAX(%esp)

fast_path4_failed:
        .data
        .global se_path4_fail_cnt
        .p2align 2
se_path4_fail_cnt:
        .long 0
        .previous
#if STATISTICS
        incl    se_path4_fail_cnt
#endif

        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax


se_shortcut_failed:

#if STATISTICS
        incl se_shortcut_failed_cnt
#endif
```

```
#if SYSENTER_DEBUG
        pusha
        push $4
        call log_sysenter_enter
        addl $4, %esp
        popa
#endif
        // shortcut failed, execute normal ipc C++ - pass

        // save register not saved yet
        movl    %ebx, REG_EBX(%esp)
        movl    %edx, REG_EDX(%esp)
        movl    REG_ESP(%esp), %ecx
        movl    -4(%ecx), %ecx
        movl    %esi, REG_ESI(%esp)
        movl    %edi, REG_EDI(%esp)
        movl    %ecx, REG_ECX(%esp)

        sti

        // pass pointer to regs structure as arg
        pushl   %esp                        // pass pointer to regs
        pushl   CURR_eax                    // pass pointer to "this"

        call *EXT(syscall_table) +4


        cli

        addl    $12, %esp                   // pop args, skip ecx
        popl    %ebp                        // ebp => edx
        popl    %esi
        popl    %edi
        popl    %ebx
        addl    $8,%esp                     // skip eax -- already loaded
                                            // with ret value
                                            // skip ebp

        movl    (%esp), %edx                // user eip
        movl    12 (%esp), %ecx             // user esp


        sti
        sysexit

se_flush_pdir:
        movl    %eax, %cr3
        ret

se_no_switch_pop:
        addl    $4, %esp

        .globl se_ret_switch
se_ret_switch:
        popl    %ebp

se_no_switch:
        .data
        .globl se_ret_switch_cnt
se_ret_switch_cnt:
        .long 0
        .previous
#if STATISTICS
        incl se_ret_switch_cnt
#endif
        // shortcut success

        // state_change_dirty (~Thread_ipc_mask, 0)
        movl    %esp, %edx
        andl    $~(THREAD_BLOCK_SIZE -1), %edx
        andl    $~Thread_ipc_mask, OFS__THREAD__STATE (%edx)

        // direct load registers instead of pop

        addl    $4, %esp                    // skip ecx
        popl    %ebp                        // ebp => edx
        popl    %esi
        popl    %edi
        popl    %ebx
        addl    $4, %esp                    // skip ebp
        popl    %eax

        movl    (%esp), %edx
        movl    12(%esp), %ecx

        sti
        sysexit

#endif //ASSEMBLER_IPC_SHORT_CUT
```

B. Fastpath, Mehrprozessor

```
#ifndef ASSEMBLER_IPC_SHORT_CUT
#define ASSEMBLER_IPC_SHORT_CUT
#endif
#ifdef ASSEMBLER_IPC_SHORT_CUT

#define ASSEMBLER

#include <flux/x86/paging.h>
#include <flux/x86/seg.h>
#include <flux/x86/proc_reg.h>
#include <flux/x86/asm.h>

#include "config_gdt.h"
#include "config_tcbsize.h"
#include "shortcut.h"

#include "../tcboffset.h"

#define L4_IPC_RECANCELED               0x40
#define L4_IPC_RETIMEOUT                0x20

#define SYSENTER_DEBUG                     0
#define STATISTICS                         0
#define SET_SEG_SEL                        0
#define FAST_TEST                          0
#define LOG_IPC                            0

#define SYSENTER_STACK_OFFSET              8
/* keep consistent with CPU_ALIGN in cpu.cpp */
#define CPU_ALIGN                       4096

#define RESET_KERNEL_SEGMENTS                  \
        movw    $ (GDT_DATA_USER|SEL_PL_U), %cx ;\
        movl    %ds,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%ds                        ;\
1:      movl    %es,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%es                        ;\
1:

#define RESET_USER_SEGMENTS                    \
        movw    $ (GDT_DATA_USER|SEL_PL_U), %cx ;\
        movl    %fs,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%fs                        ;\
1:      movl    %gs,%edx                        ;\
        cmpw    %cx,%dx                         ;\
        jz      1f                              ;\
        movl    %ecx,%gs                        ;\
1:

#define RESET_THREAD_STATE_AT(reg)             \
        andl    $~Thread_cancel, (reg)

#define SAVE_STATE                             \
        pushl   %ebp                            ;\
        pushl   %ebx                            ;\
        pushl   %edi                            ;\
        pushl   %esi                            ;\
        pushl   %edx                            ;\
        pushl   %ecx

#define RESTORE_STATE                          \
        popl    %ecx                            ;\
        popl    %edx                            ;\
        popl    %esi                            ;\
        popl    %edi                            ;\
        popl    %ebx                            ;\
        popl    %ebp                            ;\

#define REG_ECX
#define REG_EDX (1*4)
#define REG_ESI (2*4)
#define REG_EDI (3*4)
#define REG_EBX (4*4)
#define REG_EBP (5*4)
#define REG_EAX (6*4)
#define REG_EIP (7*4)
#define REG_ESP (10*4)

#define OFS__THREAD__EIP 0x7ec
#define OFS__THREAD__ESP 0X7f8
#define OFS__THREAD__EBX 0X7e0
```

```
#define OFS__THREAD__EDX 0X7d0

// ipc entry point for int 0x30
        .data
        .p2align 2
        .global shortcut_failed_cnt
shortcut_failed_cnt:
        .long  0

        .p2align 2
        .global se_shortcut_failed_cnt
se_shortcut_failed_cnt:
        .long  0
        .previous


        .data
        .global se_ipc_cnt
se_ipc_cnt:
        .long 0

        .global se_path2_cnt
se_path2_cnt:
        .long 0


        .global se_path3_cnt
se_path3_cnt:
        .long 0
        .previous


        .align  16
        .globl  do_sysenter
do_sysenter:
#if STATISTICS
        incl    se_ipc_cnt

#endif
        movl    (%esp), %esp



#if SYSENTER_STACK_OFFSET != 48
        // eflags, eip, esp, 7 * regs
        subl    $(48 - SYSENTER_STACK_OFFSET), %esp

#endif


#if SET_SEG_SEL
        movl    $(GDT_CODE_USER|SEL_PL_U), 32(%esp) // set user cs
        movl    $(GDT_DATA_USER|SEL_PL_U), 44(%esp) // set user ss
#endif

        movl    %eax, REG_EAX(%esp)
        movl    %ebp, REG_EBP(%esp)

        orl     %ebp, %eax       // eax = snd desc or rcv desc
        orl     4(%ecx), %eax    // eax = snd desc or rcv desc or timeout

        jne     fast_path4_failed // no short ipc

        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip



#define DEST_ebp %ebp
#define CURR_eax %eax

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax


        testl   $Thread_fpu_active, OFS__THREAD__STATE(CURR_eax)
        jne     se_shortcut_call_failed_fpu

        // calculate tcb from thread id
        leal    (%esi,%esi), DEST_ebp

        andl    $0x1ffff800, DEST_ebp
        orl     $0xc0000000, DEST_ebp           // dest = dst_id.lookup

        // check if dest id and dest thread match
        cmpl    %esi,OFS__THREAD__ID(DEST_ebp)  // do thread and id match
        jne     se_shortcut_call_failed_id_mismatch
```

```
        // read dest state
        movl    OFS__THREAD__STATE (DEST_ebp), %ecx
        testl   %ecx, %ecx
        je      se_shortcut_call_failed_lock

        testl   $Thread_locked, %ecx
        je      se_shortcut_call_failed_lock
        testl   $(Thread_executing | Thread_enqueued), %ecx
        jne     se_shortcut_call_failed_lock

        // only cpu local, otherwise we cannot return via sysexit

        movl    (%eax), %edi
        shr         $2, %edi              // align exec and home cpu mask
        xorl    %ecx, %edi
        testl   $Thread_home_cpu_mask , %edi

        jne     se_shortcut_call_failed_lock

        // change state atomically
        movl    %eax, %edi                      // save eax
        movl    %ecx, %eax
        andl    $~Thread_locked, %ecx
        lock cmpxchgl %ecx, (DEST_ebp)          // eax is implicit operand
        movl    %edi, %eax                      // does not change flags
        jne     se_shortcut_failed_clear_lock   // dest is locked

        andl    $(Thread_receiving | Thread_waiting | \
                  Thread_send_in_progress | Thread_ipc_in_progress ), %ecx

        // ipc_state == (Thread_waiting | Thread_ipc_in_progress)?
        cmpl    $(Thread_waiting | Thread_ipc_in_progress), %ecx
        je      se_sender_ok                    // open wait

        // ipc_state == (Thread_receiving | Thread_ipc_in_progress)?
        cmpl    $(Thread_receiving | Thread_ipc_in_progress), %ecx
        jne     se_shortcut_failed_clear_lock           // !dest->sender_ok

        // partner() == sender?
        leal    OFS__THREAD__ID (CURR_eax), %ecx// (sender_t*)this
        cmpl    %ecx, OFS__THREAD__PARTNER (DEST_ebp)

        jne     se_shortcut_failed_clear_lock

se_sender_ok:
        .data
        .global se_sender_ok_cnt
se_sender_ok_cnt:
        .long 0
        .previous

#if STATISTICS
        incl se_sender_ok_cnt
#endif

        // wake up receiver
        movl OFS__THREAD__STATE(DEST_ebp), %ecx
        andl $~(Thread_ipc_receiving_mask |       \
                Thread_ipc_in_progress |          \
                Thread_exec_cpu_mask), %ecx
        movl OFS__THREAD__STATE(CURR_eax), %edi
        andl $Thread_exec_cpu_mask, %edi
        orl  $(Thread_running) , %ecx
        orl  %edi, %ecx
        movl %ecx, OFS__THREAD__STATE(DEST_ebp)



        movl  OFS__THREAD__STATE(CURR_eax), %ecx
        orl   $(Thread_exec_cpu_mask|Thread_receiving|Thread_ipc_in_progress), %ecx
        andl  $~Thread_running , %ecx
        movl  %ecx, OFS__THREAD__STATE(CURR_eax)

set_partner_rcvregs:
        leal    OFS__THREAD__ID (DEST_ebp), %ecx      // (sender_t*)this
        movl    %ecx, OFS__THREAD__PARTNER (CURR_eax)

        leal    0x7d0(%eax), %ecx
        movl    %ecx, OFS__THREAD__RECEIVE_REGS(CURR_eax)

        // no valid ebp available
        pushl   $0
        // push restart address on old stack
        pushl   $se_ret_switch
set_kernel_esp:
        movl    %esp, OFS__THREAD__KERNEL_SP (CURR_eax)

        // *(kmem::kernel_esp()) = reinterpret_cast<vm_offset_t>(regs() + 1);
        movl    _4kmem.tss, %ecx
        leal    THREAD_BLOCK_SIZE (DEST_ebp), %edi
```

```
        movl    %edi, 4 (%ecx)  // x86_tss.esp0

        // sysenter_stack[node::id()] = reinterpret_cast<vm_offset_t>(&regs()->esp);
        leal    THREAD_BLOCK_SIZE - SYSENTER_STACK_OFFSET (DEST_ebp), %ecx

        //
        movl    OFS__THREAD__STATE(DEST_ebp), %esi
        andl    $Thread_exec_cpu_mask, %esi
        shr     $(Thread_exec_cpu_offset-2), %esi
load_sysenter_stack:
        movl    %ecx, EXT(sysenter_stack)(%esi)

        movl    CURR_eax, %ecx
        xorl    DEST_ebp, %ecx
        testl   $0x1ffc0000, %ecx              // same task ?
        jne     fast_path_switch_address_space
address_space_switched:

        movl    OFS__THREAD__ID(CURR_eax), %esi
        movl    4+OFS__THREAD__ID(CURR_eax), %edi

        andl    $~Thread_executing, OFS__THREAD__STATE(CURR_eax)


        orl     $(Thread_executing| Thread_locked), OFS__THREAD__STATE(DEST_ebp)
        // no stack access after CURR_eax has been marked not executing

        movl    OFS__THREAD__KERNEL_SP(DEST_ebp), %ecx
        cmpl    $se_ret_switch, (%ecx)
        jne     fast_path_slow_switch

        movl    %ebp, %eax
        movl    %edx, %ebp          // copy msg.w

        movl    OFS__THREAD__EIP(%eax), %edx   // load eip
        movl    OFS__THREAD__ESP(%eax), %ecx   // load esp

        movl    $0x4000, %eax        // set msg dope

        sti
        sysexit

fast_path_slow_switch:


        movl    OFS__THREAD__KERNEL_SP (DEST_ebp), %ecx
        movl    %ecx, %esp                      // load new stack pointer

        movl    OFS__THREAD__RECEIVE_REGS(DEST_ebp), %ecx // load receive regs

        movl    %ebx,    REG_EBX(%ecx)
        movl    %edx,    REG_EDX(%ecx) // already saved
        movl    $0x4000, REG_EAX(%ecx)
        movl    %esi,    REG_ESI(%ecx)
        movl    %edi,    REG_EDI(%ecx)

        ret

fast_path_switch_address_space:
        movl    OFS__THREAD__SPACE_CONTEXT (DEST_ebp), %ecx
        subl    $physmem, %ecx
        movl    %ecx, %cr3
        jmp address_space_switched


#if STATISTICS
#define SHORTCUT_FAILED(reason)                         \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_failed_##reason##_cnt        ;\
se_shortcut_failed_##reason##_cnt:                      ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_failed_##reason##:                          ;\
        incl se_shortcut_failed_##reason##_cnt          ;\
        jmp se_shortcut_failed
#else
#define SHORTCUT_FAILED(reason)                         \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_failed_##reason##_cnt        ;\
se_shortcut_failed_##reason##_cnt:                      ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_failed_##reason##:                          ;\
        jmp se_shortcut_failed
#endif
```

```
#if STATISTICS
#define SHORTCUT_CALL_FAILED(reason)                     \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_call_failed_##reason##_cnt   ;\
se_shortcut_call_failed_##reason##_cnt:                 ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_call_failed_##reason##:                     ;\
        incl se_shortcut_call_failed_##reason##_cnt     ;\
        SAVE_SYSCALL_CTXT                               ;\
        jmp se_shortcut_failed
#else
#define SHORTCUT_CALL_FAILED(reason)                     \
        .data                                           ;\
        .p2align 2                                      ;\
        .globl se_shortcut_call_failed_##reason##_cnt   ;\
se_shortcut_call_failed_##reason##_cnt:                 ;\
        .long 0                                         ;\
        .previous                                       ;\
se_shortcut_call_failed_##reason##:                     ;\
        SAVE_SYSCALL_CTXT                               ;\
        jmp se_shortcut_failed
#endif



SHORTCUT_FAILED(wrong_sender)
//SHORTCUT_FAILED(running)
SHORTCUT_FAILED(no_rcv)
SHORTCUT_FAILED(lock)
SHORTCUT_FAILED(irq_or_sender)
SHORTCUT_FAILED(fpu)
SHORTCUT_FAILED(ready_list)

SHORTCUT_CALL_FAILED(no_rcv)
SHORTCUT_CALL_FAILED(lock)
SHORTCUT_CALL_FAILED(wrong_sender)
SHORTCUT_CALL_FAILED(fpu)
SHORTCUT_CALL_FAILED(id_mismatch)

        // pre:  ebp - destination thread
        //             destination thread is locked
        //       eax - current
        //       ecx - state of dest
        // post:       destiantion thread is not longer locked
se_shortcut_failed_clear_lock:
        movl OFS__THREAD__STATE(DEST_ebp), %ecx
        movl %eax, %edi
        movl %ecx, %eax
        orl $Thread_locked, %ecx

        lock cmpxchgl %ecx, (DEST_ebp)
        movl %edi, %eax
        jz 1f
        int3
1:
        testl $Thread_locked, OFS__THREAD__STATE(DEST_ebp)
        jnz 1f
        int3
1:
        jmp se_shortcut_failed

fast_path4_failed:
        .data
        .global se_path4_fail_cnt
        .p2align 2
se_path4_fail_cnt:
        .long 0
        .previous
#if STATISTICS
        incl    se_path4_fail_cnt
#endif

        movl    (%ecx), %eax
        leal    8(%ecx), %ebp
        movl    %ebp, REG_ESP(%esp)   // save user esp
        movl    %eax, REG_EIP(%esp)   // save user eip

        movl    %esp,%eax
        andl    $~(THREAD_BLOCK_SIZE -1), %eax  // current() => eax

se_shortcut_failed:

#if STATISTICS
        incl se_shortcut_failed_cnt
#endif

        // shortcut failed, execute normal ipc C++ - pass
```

```
        // save register not saved yet
        movl    %ebx, REG_EBX(%esp)
        movl    %edx, REG_EDX(%esp)
        movl    REG_ESP(%esp), %ecx
        movl    -4(%ecx), %ecx
        movl    %esi, REG_ESI(%esp)  // already saved upon entry
        movl    %edi, REG_EDI(%esp)
        movl    %ecx, REG_ECX(%esp)

        sti

        // pass pointer to regs structure as arg
        pushl   %esp                            // pass pointer to regs
        pushl   CURR_eax                        // pass pointer to "this"

        call *EXT(syscall_table) +4

        cli

        addl    $12, %esp                       // pop args, skip ecx
        popl    %ebp                            // ebp => edx
        popl    %esi
        popl    %edi
        popl    %ebx
        addl    $8,%esp                         // skip eax -- already loaded
                                                // with ret value
                                                // skip ebp

        movl    (%esp), %edx                    // user eip
        movl    12 (%esp), %ecx                 // user esp

        sti
        sysexit


        .data
        .global sysexit_eip
        .global sysexit_esp
sysexit_eip:
        .long 0
sysexit_esp:
        .long 0
        .previous


se_flush_pdir:
        movl    %eax, %cr3
        ret

se_no_switch_pop:
        addl    $4, %esp

        .globl se_ret_switch
se_ret_switch:
        popl    %ebp

se_no_switch:
        .data
        .globl se_ret_switch_cnt
se_ret_switch_cnt:
        .long 0
        .previous
#if STATISTICS
        incl se_ret_switch_cnt
#endif
        // shortcut success

        // state_change_dirty (~Thread_ipc_mask, 0)
        movl    %esp, %edx
        andl    $~(THREAD_BLOCK_SIZE -1), %edx
        andl    $~Thread_ipc_mask, OFS__THREAD__STATE (%edx)

        // direct load registers instead of pop

        addl    $4, %esp                        // skip ecx
        popl    %ebp                            // ebp => edx
        popl    %esi
        popl    %edi
        popl    %ebx
        addl    $4, %esp                        // skip ebp
        popl    %eax

        movl    (%esp), %edx                    // prepare for sysexit
        movl    12(%esp), %ecx

        sti
        sysexit

#endif
```

Literatur

[1] M. Völp, "Prototypical design and implementation of l4-smp microkernel mechanisms," Master's thesis, University of Karlsruhe, 2002. URL: `http://i30www.ira.uka.de/~teaching/thesisdocuments/voelp-marcus_study-thesis_mv_xp_ipc.ps`.

[2] J. Liedtke, "On μ-kernel construction," in *15th ACM Symposium on Operating System Principles (SOSP)*, (Copper Mountain Resort, CO), pp. 237–250, Dec. 1995.

[3] B. Stroustrup, *Die C++-Programmiersprache*. Addision-Wesley, 1998.

[4] J. Liedtke, "L4 reference manual (486, Pentium, PPro)," Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, Sept. 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[5] Intel Corp., *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999.

[6] M. P. M. Hohmuth, Claude-Joachim Hamann and H. Härtig, "Wait-free object sharing with the multiprocessor priority-inheritance protocol." unpublished, 2002.

[7] E. W. Dijkstra, "Solutions of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, p. 569, Sept. 1965.

[8] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, pp. 453–455, Oct. 1974.

[9] L. Lamport, "A fast mutual exclusion algorithm," *ACM Transactions of Computer Systems*, vol. 5, no. 1, pp. 1–11, 1987.

[10] Anderson and Kim, "A new fast-path mechanism for mutual exclusion," *DISTCOMP: Distributed Computing*, vol. 14, 2001.

[11] J. L. K. Elphinstone, T. Jaeger and V. Penteelenko, "Flexible access control using ipc redirection," 1999.