Diplomarbeit

Implementierung eines Echtzeit-IPC-Pfades mit Unterbrechungspunkten für L4/Fiasco

René Reusner

Juli 2005

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dr.-Ing. Michael Hohmuth

Erklärung

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

Dresden, den 31. Juli 2005

René Reusner

Inhaltsverzeichnis

1 Aufgabenstellung **11**

2 Einleitung **13**
2.1 Gliederung . . . 14
2.2 Danksagung . . . 14

3 Hintergrund **15**
3.1 Fiasco . . . 15
3.2 Aufbau von Fiasco . . . 15
3.3 Synchronisation in Fiasco . . . 18
3.4 IPC-Architektur . . . 20

4 Analyse der Verzögerungszeiten **25**
4.1 Hardware . . . 25
4.2 Software . . . 27
4.3 Zusammenfassung . . . 30

5 Entwurf **33**
5.1 Kernspeicher . . . 33
5.2 Synchronisation . . . 36
5.3 Senderwarteschlange mit Prioritäten . . . 38
5.4 Aufbau IPC-Pfades . . . 42
5.5 Probleme des bisherigen IPC-Pfades . . . 46

6 Implementierung **49**
6.1 Timeouts . . . 49
6.2 Senderwarteschlange . . . 50
6.3 Optimierung des IPC-Pfades . . . 52

7 Auswertung **57**
7.1 Mikro-Benchmarks . . . 57
7.2 Echtzeiteigenschaften im Vergleich . . . 61
7.3 Auswertung . . . 63

8 Zusammenfassung und Ausblick **67**

Glossar **69**

Literaturverzeichnis **71**

Abbildungsverzeichnis

3.1 Aufteilung des Kernspeicher. 16
3.2 Übersicht über die benötigten Klassen und deren Beziehungen. 17

5.1 TCB Zugriff. 34
5.2 Indirekter TCB Zugriff. 35
5.3 List als Senderwarteschlange. 39
5.4 Senderwarteschlange-Trie. 40
5.5 Senderqueue-Heap. 41
5.6 IPC-Zustandsdiagramm. 43
5.7 Prioritätsinversion. 47
5.8 Timeout-Fehler. 48

6.1 Timeout-Liste. 50
6.2 Implementierte Senderwarteschlange. 51
6.3 Kernstack. 53

7.1 Abhängigkeit der Kosten von der Größe der Senderwarteschlange 58
7.2 Durchschnittliche Zeit zum Einketten von Sendern unterschiedlicher Priorität 59
7.3 Durchschnittliche Kosten zum Einketten eines Timeouts 60
7.4 Geschwindigkeitsvergleich. 62
7.5 Cache-Flooder 63
7.6 hoher IPC-Durchsatz 63
7.7 IPC + Cache-Flooder 64
7.8 Dope + Cache-Flooder 64
7.9 Abhängigkeit der IPC-Performance von Unterbrechungspunkten 65
7.10 Abhängigkeit der Echtzeit-Eigenschaften von Unterbrechungspunkten 65

Tabellenverzeichnis

4.1 Ausführungszeiten von speziellen Maschineninstruktionen 26
4.2 Hardware Verzögerungszeiten 28
4.3 Verzögerungszeiten von Kernoperationen 31

5.1 Sende- und Empfangszustände 42

7.1 Geschwindigkeitsvergleich bisheriger und neuer IPC-Pfad 61

1 Aufgabenstellung

2 Einleitung

Mikrokerne sind kleine Betriebssystemkerne, die nur die notwendigen Grundfunktionen bereitstellen. Die höheren Funktionen des Betriebssystems werden durch Server bereit gestellt, welche sich in separate Adressräumen befinden. Es soll dadurch eine größere Modularität, Flexibilität und Sicherheit erreicht werden.

Ein Mikrokern der 1. Generation war Mach [ABB+86]. Mach entstand, indem man aus einem monolithischen System immer mehr Funktionen in Nutzerprogramme auslagerte und nur noch diese Funktionen im Kernel lies, welche nicht sicher auf Nutzerebene implementiert werden können. Mikrokerne der ersten Generation zeigten erhebliche Geschwindigkeitsdefizite, so dass in späteren Versionen wieder Funktionen zurück in den Kern verlagert wurden.

Mikrokerne der 2. Generation wurden unter Berücksichtigung von Geschwindigkeit [Lie93] und Minimalismus entworfen. Es wird dem Kern nur das hinzugefügt, dass nicht sicher im Nutzerbereich implementiert werden kann. Die Mikrokerne der 2. Generation zeigten, dass man mit ihnen auch effiziente Systeme bauen kann.

Der L4-Kern [Lie96] ist ein Mikrokern der 2. Generation. Er ist ein Nachfolger des L3-Systems [Lie88]. Der L4 Mikrokern stellt folgende Mechanismen zur Verfügung.

- **Adressräume:** Adressräume bilden die Schutzdomäne. Sie enthalten einen oder mehrere Threads. Adressräume werden rekursiv durch Pager konstruiert, mit Sigma0 als initialen Adressraum. Bei einem Seitenfehler in einem Adressraum, generiert der Kern eine Nachricht an seinen Pager. Dieser kann dann in diesen Adressraum eine Seite einblenden, um den Seitenfehler aufzulösen. Damit ist es möglich, eine Hierarchie von Adressräumen rekursiv aufzubauen.
- **Threads:** Threads sind die Aktivitätsträger. Die Threads werden entsprechend ihrer Prioritäten vom Scheduler eingeplant und ausgeführt. Alle bisherigen L4-Implementationen verwenden Kernthreads, die dem Kern, im Gegensatz zu Threads auf Nutzerebene, bekannt sind. Die Threads werden im Kern durch eine Kontrollstruktur, dem Thread-Kontroll-Block (TCB), repräsentiert.
- **IPC:** Threads können miteinander mittels Interprozesskommunikation, engl. Inter Process Communication, kurz IPC, kommunizieren. Die L4-IPC bietet als Operationen das Senden *send*, das Empfangen *receive* und *wait*, und das kombinierte Senden und Empfangen, *reply_wait* und *call* an. Alle IPC-Operationen sind synchron, d.h. die Nachrichtenübertragung findet erst statt, wenn Sender und Empfänger bereit sind. Um nicht erfolgreiche IPC-Operationen nach einer bestimmten Zeit abzubrechen, können Timeouts gesetzt werden. Nachrichten können aus einfachen Registerwerten, Speicherinhalten und Flexpages[1] [HWL96] bestehen. Für die Behandlung von IPCs ist der IPC-Codepfad des Kerns zuständig.

Ein L4-Mikrokern-System besteht aus dem L4-Kern, mehreren Servern, welche die Dienste wie Dateisysteme, Netzwerkdienste und grafische Oberfläche zur Verfügung stellen, sowie den Nutzerprogrammen.

[1] Flexpages sind Speicherseiten, welche im Adressraum des Empfängers eingeblendet werden können.

Die Nutzerprogramme kommunizieren über IPC mit diesen Servern, wenn sie einen bestimmten Dienst in Anspruch nehmen wollen. Daher ist die IPC-Performance entscheidend für ein effizientes System.

Heute bezeichnet L4 eine Mikrokernfamilie. Es existieren verschiedene L4-Schnittstellen. Es gibt die originale Schnittstelle, **V2** genannt, und die experimentellen Schnittstellen **X.0** [Lie99] und **X.2** [Tea05]. Weiterhin gibt es noch eine hoch experimentelle Schnittstelle **L4.sec** [Kau05], die besonders Rechte- und Ressourceprobleme adressiert.

2.1 Gliederung

In dieser Arbeit wird ein neuer IPC-Pfad für den L4-kompatiblen und echtzeitfähigen Mikrokern Fiasco implementiert. Im nachfolgendem Kapitel gebe ich einen Überblick über den Fiasco-Kern. Insbesondere umfasst dies alle Aspekte des Kerns, welche für die IPC-Performance und Echtzeit entscheidend sind. Eine kurze Erläuterung der vorhandenen IPC-Implementation wird auch gegeben. Im 3. Kapitel gehe ich auf die Eigenschaften von Hardware und Software, im Hinblick auf Echtzeit, ein. Das 4. Kapitel beschreibt den Entwurf des neuen IPC-Pfades und geht dabei auf verschiedene Probleme ein. Die Implementierung und einige damit verbundene Probleme werden im 5. Kapitel kurz erläutert. Die Auswertung im Hinblick auf Performance und Echtzeitfähigkeiten, im Vergleich zum bisherigen Kern, erfolgt im 6. Kapitel. Im 7. Kapitel erfolgt eine kurze Zusammenfassung, welche Ziele wurden erreicht und welche Probleme bestehen noch.

2.2 Danksagung

Hier möchte ich mich bei Prof. Dr. Hermann Härtig für die Möglichkeit bedanken, in der Betriebssystemgruppe zu arbeiten. Mein besonderer Dank gilt auch meinem Betreuer, Dr.-Ing. Michael Hohmuth. Weiterhin möchte ich mich bei Jean Wolter, Dr.-Ing. Frank Mehnert, Dietrich Clauß, Bernhard Kauer und Adam Lackorzynski bedanken.

3 Hintergrund

3.1 Fiasco

Fiasco [Hoh98, Hoh02a] ist ein L4-kompatibler und echtzeitfähiger Mikrokern, entworfen und implementiert von Michael Hohmuth. Es war der erste L4-Kern, welche in einer Hochsprache geschrieben wurde und bei dessen Entwurf besonders Wert auf sehr gute Echtzeiteigenschaften gelegt wurde. Der Fiasco-Kern implementiert die V2, X0 und X.2-Schnittstelle. Es gibt auch eine stark abgeänderte Version, welche die L4.sec Schnittstelle implementiert.

Fiasco läuft auf IA-32, IA-64 [War02] und ARM [War03]. Eine x86-64 und Power-PC Portierung sind in Arbeit. Desweiteren Fiasco als Nutzerprogramm auf einem Linuxsystem [Ste02] verfügbar. Fiasco ist in C++ implementiert. Michael Peter implementierte zusätzlich einen IPC-Shortcut in Assembler [Pet02].

Fiasco ist ein Echtzeitkern und garantiert kurze und begrenzte Verzögerungszeiten bei der Zustellung von Ereignissen. Um dies zu erreichen, ist der Kern sehr oft unterbrechbar, nur kurze kritische Abschnitte werden durch das Sperren von Interrupts geschützt. Für größere kritische Abschnitte nutzt Fiasco nichtblockierende Synchronisation, deren Details in [Hoh02b] erläutert werden.

Einen anderen Ansatz als die L4-Kerne, insbesondere Fiasco geht der Fluke-Mikrokern. Der Fluke-Kern bietet nur atomare Operationen an, längere Systemaufrufe werden in mehrere atomare Teiloperationen aufgespalten [BF98]. Der Zustand zwischen diesen Operationen wird vollständig zum Nutzer exportiert, so dass der Kern sich keine Zustände im Kern merken muss.

3.2 Aufbau von Fiasco

Adressräume

Adressräume bilden die Schutzdomäne und enthalten einen oder mehrere Threads, die Aktivitätsträger. Die V2-Schnittstelle unterstützt maximal 128 Threads pro Adressraum, die V4-Spezifikation setzt keine solche Grenze. Für das Einblenden und Entfernen von Flexpages in anderen Adressräumen, verwaltet der Kern eine Mapping-Datenbank, wo für jede physische Speicherseite ein Mapping-Baum angelegt wird. Wird eine Flexpage entfernt, wird über diesen Baum iteriert, und die Seite wird aus betroffenen Unterbäumen entfernt, oder deren Rechte modifiziert.

Der Adressraum ist auf der IA-32 Architektur in einen, drei Gigabyte großen, Nutzeradressraum und in einen, ein Gigabyte großen, Kernadressraum aufgeteilt. Der Kernadressraum ist für den Nutzer nicht zugreifbar und ist in jedem Adressraum eingeblendet.

Er enthält den TCB-Bereich und die IO-Bitmap, sowie die Region für den verfügbaren physischen Kernspeicher, siehe Abb. 3.1. In einigen unbenutzten Einträgen des Seitenverzeichnis, werden auch Task-spezifische Variablen eingetragen.

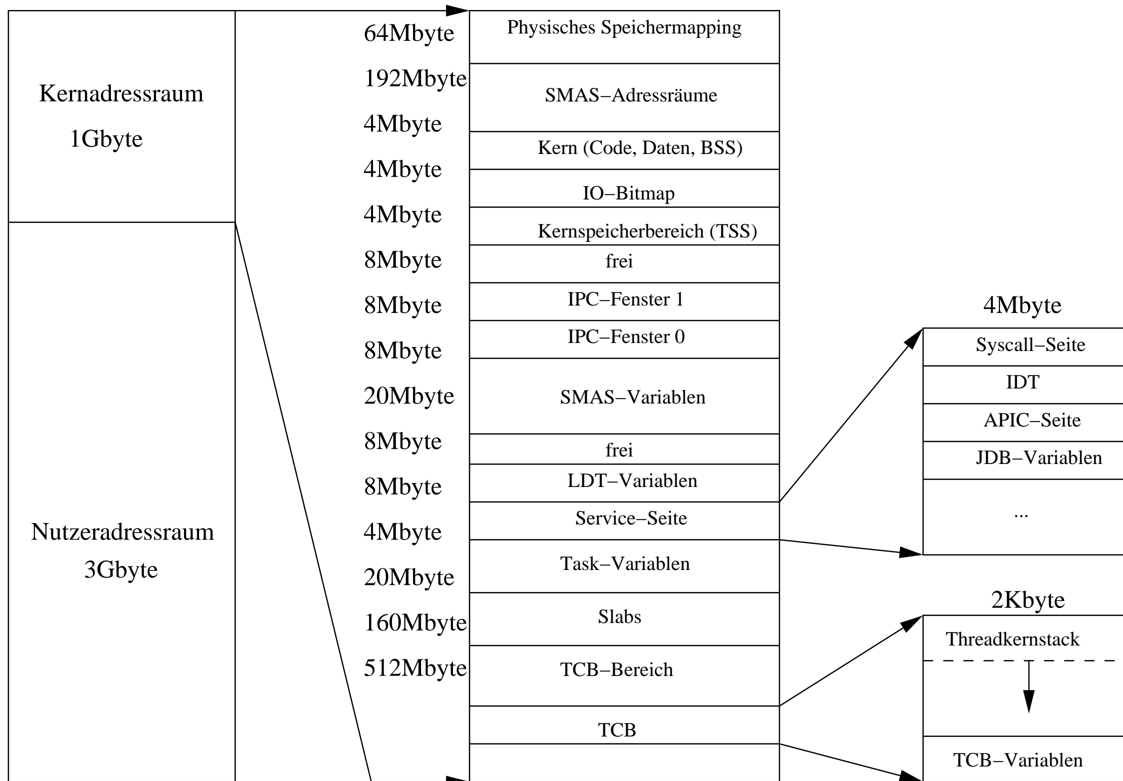

Abbildung 3.1: Aufteilung des Kernspeicher.

Threads

Threads sind die Aktivitätsträger, und können erzeugt, verändert und gelöscht werden. In der V2 und X0-Spezifikation erfolgt dies über den *thread_ex_regs*-Systemaufruf. In der X.2-Spezifikation dient *thread_ex_-regs* nur zum Modifizieren von Threads, Erzeugen und Löschen erfolgt durch *Thread_control*-Systemaufruf.

Jeder Thread wird durch seinen TCB repräsentiert, welcher im TCB-Bereich des Kernadressraumes zu finden ist. Der TCB besteht aus einem *Thread*-Objekt, welches Thread-spezifische Variablen enthält, und dem Kernstack dieses Threads.

Fiasco implementiert das “Prozessmodell”, d.h. jeder Thread besitzt einen Kernstack. Darauf wird der aktuelle Zustand des Threads automatisch gesichert, ein Umschalten der Threads erfolgt durch Umschalten des Kernstacks. Dadurch ist unterbrechbarer Kerncode sehr einfach zu implementieren, da der Zustand des Threads jederzeit auf dem Stack gespeichert ist. Der Gegensatz dazu ist das “Interruptmodell“. Hier gibt es nur einen Kernstack pro CPU. Der Zustand eines Threads muss beim Kontextwechsel explizit in einer *Continuation*[1] gesichert werden. Daher kann dieser Thread nur an dafür vorbereiteten Punkten unterbrochen werden.

Die Klasse *Thread* ist von den Klassen *Receiver* und *Sender* abgeleitet, siehe auch Abb. 3.2. Diese beiden Klassen entsprechen der Empfänger- und Senderrolle einer IPC-Operation. Die Klasse *Receiver* ist weiterhin von *Context* abgeleitet. Die Klasse *Context* ist der Ausführungskontext. Sie stellt den den Scheduler (*schedule*)und Funktionen zum Umschalten des Ausführungskontextes (*switch_to*, *switch_exec*) zur Verfügung. Details sind in [Hoh02a] zu finden.

Ein Thread kann sich in verschiedenen Zuständen befinden, diese werden durch eine Kombination von Flags im Zustandswort der *Context*-Klasse beschrieben.

[1]Continuation: Eine Datenstruktur, welche die benötigte Informationen enthält, um den Zustand des Threads wiederherzustellen und die Operation fortzusetzen.

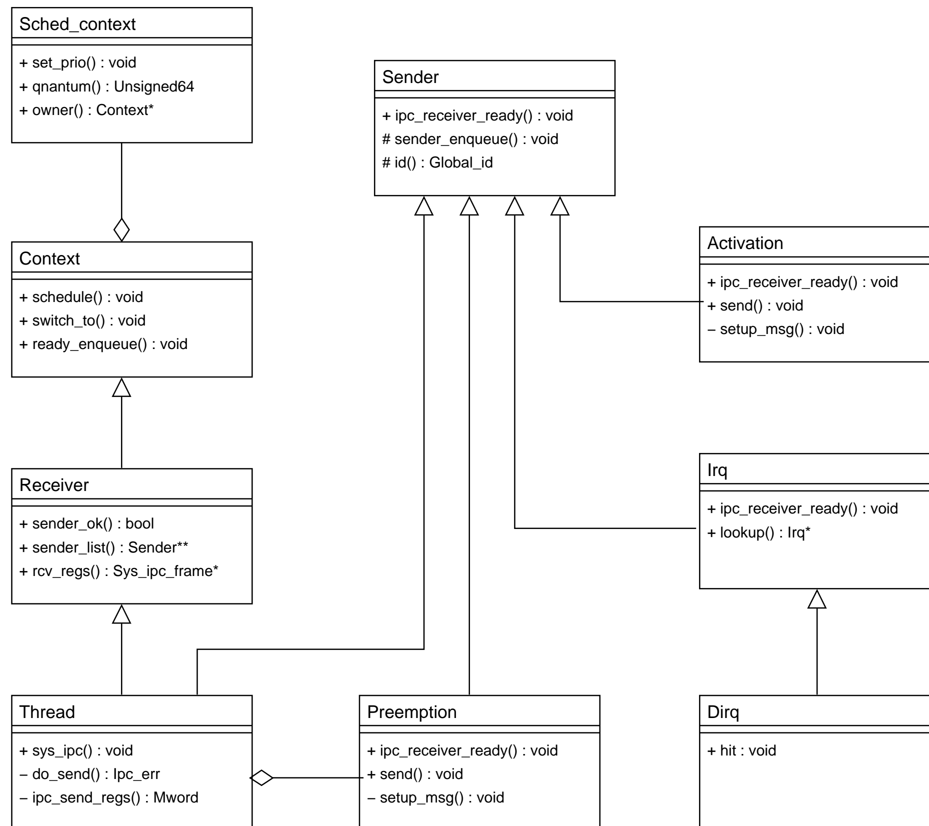

Abbildung 3.2: Übersicht über die benötigten Klassen und deren Beziehungen.

Interrupts

Interrupts werden in L4 auf IPC abgebildet. Ein Thread kann sich mit einem Interrupt assoziieren. Er bekommt eine Nachricht zugestellt, wenn dieser Interrupt ausgelöst wurde. Unterbrechungen werden durch Interruptdeskriptoren, die Klasse *Irq*, dargestellt Die Klasse *Irq* ist von *Sender* abgeleitet, und besitzt damit auch die Fähigkeit, IPC-Nachrichten zu verschicken. Die Klasse *Dirq* leitet von *Irq* ab, und ihre *hit*-Methode ist für die Zustellung der Nachricht zuständig, welche aufgerufen wird, wenn ein Interrupt auftritt.

Scheduling

Scheduling umfasst den Bereich des Kerns, welcher sich mit der Auswahl und Ausführung von Threads befasst. Der Scheduler wird durch die *schedule*-Funktion implementiert. Die Scheduling-Informationen wie Priorität, Zeitdauer und Besitzer dieser Zeitscheibe, sind im Scheduling-Kontext, repräsentiert durch die Klasse *Sched_context*, getrennt vom TCB abgelegt.

Neben dem normalen Scheduling-Kontext, der im TCB des Threads aggregiert ist, kann ein Thread über zusätzliche Real-Time-Scheduling-Kontexte verfügen. Ein Zeiger im TCB zeigt stets auf den aktiven *Sched_context* dieses Threads. Details dazu können in [Ste04] nachgelesen werden. Wenn ein Real-Time-Scheduling-Kontext abgelaufen ist, wird eine Preemption-IPC an den Preempter des Besitzer-Threads generiert.

Es werden 256 Prioritäten zur Verfügung gestellt. Der Scheduler wählt immer den Thread mit der höchsten Priorität aus. Threads mit gleicher Priorität werden nacheinander, mit einer Round-Robin-Strategie, ausgeführt.

Zeitgeber

Die schedule() Funktion wird am Ende einer Zeitscheibe durch den Zeitgeber aufgerufen, um einen neuen Thread auszuwählen. Der Zeitgeber kann auf IA-32 drei verschiedene Quellen nutzen, den PIT[2], die CMOS Echtzeituhr (RTC) und die Local-APIC[3] der CPU. Der APIC-Timer kann als periodischer oder als *One-Shot-Timer* konfiguriert werden. Bei den periodischen Zeitgebern, beträgt die Frequenz 1000Hz (PIT) oder 1023Hz (RTC), so dass die Zeitgeber-Routine rund jede Millisekunde aufgerufen wird. Bei dem *One-Shot-Timer* wird immer nur ein Interrupt ausgelöst. Der Zeitgeber wird dann auf das nächste Ereignis neu programmiert. Dies kann das Ende der neuen Zeitscheibe oder der Ablauf eines Timeouts sein. Der "One-Shot-Timer" bietet eine Genauigkeit für Timeouts im Bereich von Mikrosekunden an. Man kann die Timeouts auf eine Mikrosekunde spezifizieren, jedoch begrenzen die Verzögerungszeiten des Kerns die erreichbare Genauigkeit.

Ready-Liste

Die Bereitwarteschlange (Ready-Liste) enthält alle ausführbaren Threads. Es gibt jedoch eine Ausnahme, der aktuell aktive Thread muss nicht in der Ready-Liste enthalten sein. Ferner können in der Ready-Liste auch nichtbereite Threads enthalten sein. Sobald der Scheduler beim Iterieren über die Ready-Liste einen solchen blockierten Thread entdeckt, wird dieser nachträglich aus dieser Liste entfernt.

Der Zweck dieser Optimierung (Lazy-Scheduling) ist, dass IPC-Operationen das Ein- und Ausketten von Threads aus der Ready-Liste einsparen können. Bei einer kombinierten Sende- und Empfangsoperation wird sofort zum Empfänger umgeschaltet, ohne diesen in die Ready-Liste einzutragen. Der alte Sender wird auch nicht aus der Ready-Liste entfernt, obwohl er sich im IPC-Wartezustand befindet und nicht mehr lauffähig ist. Die Aktualisierung der Ready-Liste erfolgt erst nachträglich im Scheduler. Diese Optimierung beschleunigt besonders die Client-Server Kommunikation, wo viele kombinierten Sende- und Empfangsoperationen, *call* und *reply_wait*, verwendet werden. Wenn zwei Threads miteinander Pingpong spielen, wird so das Aktualisieren der Ready-Liste bei IPC-Operationen komplett eingespart.

Implementiert wird die Ready-Liste durch ein Array, dessen 256 Elemente doppelt verkettete Listen sind, welche alle Threads derselben Priorität enthalten. Die Synchronisation von Änderungen der Ready-Liste erfolgt durch das Sperren der Interrupts.

3.3 Synchronisation in Fiasco

Kritische Abschnitte in Fiasco werden nichtblockierend synchronisiert Dies bietet den Vorteil der vollen Unterbrechbarkeit und Prioritätsinversion wird vermieden. Nichtblockierende Synchronisation wird in sperrfreie, engl. *lock-free* und wartefreie, engl. *wait-free*, Synchronisation unterteilt.

- **Wartefreie Synchronisation** ist ähnlich Sperren, wobei das Blockieren durch Helfen ersetzt wird. Wenn ein hoch priorisierter Thread A einen Konflikt mit einem niedrig priorisierten Thread B entdeckt, hilft A, dass B seinen kritischen Abschnitt beendet. Dies wird dadurch erreicht, dass A seine CPU-Zeit und seine Priorität an B weitergibt. Sobald B den kritischen Abschnitt verlässt, beginnt A den kritischen Abschnitt. Dieser Vorgang lässt sich transitiv fortsetzen.

  Um die Prioritätsvererbung sicher zustellen, darf ein Thread in einem kritischen Abschnitt nie blockieren. Dies umfasst z.B. auch Zugriffe auf den Nutzeradressraum, da diese Seitenfehler auslösen können. Der Kern setzt generiert eine Seitenfehler-IPC und dieser Thread blockiert.

[2] PIT: Programmable Interrupt Timer
[3] Local advanced Interrupt Controller

- **Lockfreie Synchronisation** kommt ohne Sperren aus. Threads berechnen die Ergebnisse lokal, und versuchen sie dann mittels atomarer Speicheroperationen auf die gemeinsamen globalen Daten zu übertragen. Schlägt dies fehl, wird die Operation wiederholt. Fiasco verwendet als atomare Maschineninstruktion die *compare-and-swap* Operation (CAS). Diese Operation vergleicht zuerst, ob der übergebene alte Wert noch mit dem Speicherinhalt übereinstimmt. Wenn ja, wird der Speicher mit dem neuen Ergebnis überschrieben. Wenn der Vergleich fehlschlägt, wird die Operation in einer *Retry-loop* wiederholt. Diese teuren Maschineninstruktionen, die *Retry-loop* und die notwendigen Prüfungen sind ein Grund, warum der bisherige IPC-Pfad sehr langsam ist.

Atomare Speicheroperationen

Atomare Speicheroperationen sind das normale CAS, das Doppelwort compare-and-swap (CAS2), welches CAS auf zwei Wörter erweitert, oder die load-locked(LL) und store-conditional(SC) Instruktionen, welche auf RISC-CPUs zu finden sind. Die x86-Architektur bietet nur CAS und andere einfache atomare Operationen, wie Inkrement und Dekrement an. Für Multiprozessorsysteme, können diese Instruktionen mit einem speziellen Lockprefix versehen werden, welche den Bus sperren und somit die Konsistenz auch bei mehreren CPUs sicher stellen.

Diese Instruktionen sind nur bei Modifikation von einzelnen Worten verwendbar. Um etwas komplexere Datenstrukturen und mehrere Worte atomar zu verändern, wie zum Beispiel doppelt verkettete Listen, werden für sehr kurze Zeit die Interrupts gesperrt. Das Sperren der Interrupts reicht auf Multiprozessorsystemen, nur für lokale Daten aus, für globale Daten sind zusätzlich Spin-Locks notwendig. Obwohl die Interrupts gesperrt werden, sind die Zeiten dennoch sehr kurz, und die Verzögerungszeiten verschlechtern sich nur unwesentlich.

Sperren der Interrupts

Das Sperren der Interrupts erfolgt bei x86-Prozessoren durch Löschen des Flags für Interrupts mittels der CLI-Instruktion. Das Freigeben der Interrupts erfolgt durch die STI-Instruktion. Diese sind erheblich schneller als das Sperren der Unterbrechungen am Interruptcontroller. Bei Fiasco-UX erfolgt dies auch durch die CLI oder STI-Operation. Sobald eine dieser Instruktionen auftritt, generiert der darunterliegende Linuxkern ein Signal, dessen Signalfunktion die Interrupts sperrt. Die Klasse *Cpu_lock* stellt für das Sperren und Freigeben der Interrupts eine allgemeine Lock-Schnittstelle zur Verfügung. Intern verwendet diese Klasse die CLI- und STI-Operationen.

Im aktuellen Fiasco-Kern sind nach einem Kerneintritt die Interrupts noch gesperrt. Die Freigabe der Interrupts erfolgt, nach dem Sichern der Register dem Kernstack, sofort durch den Kern.

Locks

Neben der *Cpu_lock* Klasse gibt es weitere Locks zur wartefreien Synchronisation. Wenn bei dem Betreten des kritischen Abschnittes ein solches Lock schon gesperrt ist, wird dem Lock-Besitzer durch Leihen der CPU-Zeit und Priorität geholfen, seinen kritischen Abschnitt zu beenden. Innerhalb der Lock-Methoden erfolgt die Synchronisation durch Sperren der Interrupts.

Im Detail, sieht das folgendermaßen aus: Wenn der Thread versucht, sich das Lock zu holen und feststellt, dass es bereits durch einen anderen Thread gesperrt ist, schaltet er zu dem Lock-Besitzer mittels *switch_to* um, ohne jedoch den Scheduling-Kontext umzuschalten. Es wird vorher noch ein Hinweis im TCB des Lock-Besitzers gesetzt und der alte Thread bleibt weiter rechenbereit. Wenn sein Scheduling-Kontext abgelaufen ist, und der Scheduler diesen Thread erneut auswählt, prüft er erneut, ob das Lock frei ist, und wenn nicht,

wird der Vorgang wiederholt. Bei der Freigabe eines Locks wird anhand des Hinweises im eigenen TCB nachgeschaut, ob dem Thread geholfen wurde, den kritischen Abschnitt zu beenden. Wenn ja, wird zu dem Helfer umgeschaltet, da in diesem Fall der Helfer eine höhere Priorität besitzt.

Lock-Klassen

Es gibt neben dem *Cpu_lock* weitere Klassen von Locks in Fiasco. Das *Switch_lock* bildet die Grundlage, auf dem das *Helping_lock* und *Thread_lock* aufbauen. Das *Helping_lock* entspricht dem *Switch_lock*, jedoch wird zusätzlich der Fall des nicht initialisierten Scheduling-System betrachtet. Dies ist beim Booten des Kerns wichtig.

Die Klasse *Thread_lock* dient zum Sperren von anderen Threads. Gesperrte Threads, werden bis auf eine Ausnahme, nie vom Scheduler ausgewählt. Wenn zu einem gesperrten Thread umgeschaltet werden soll, wird automatisch zu seinem Lock-Besitzer umgeschaltet. Eine Ausnahme bildet das Zerstören von Threads durch *Thread::kill*. Dort läuft ein gesperrter Thread solange, bis er alle seine gehaltenen Locks freigegeben hat. Ein Zähler im TCB wird inkrementiert, wenn dieser Thread ein Lock in Besitzt nimmt, und dekrementiert, sobald dieses Lock freigegeben wird. Dadurch kann *Thread::kill* feststellen, ob der zu zerstörende Thread noch Locks besitzt. Es wird solange zu dem zerstörendem Thread umgeschaltet, bis dieser alle Locks freigegeben hat.

Die Lock-Klassen, inklusive *Cpu_lock*, werden nie direkt verwendet, sondern über einen Wächter, den *Lock_-guard*. Der Konstruktor dieser Klasse nimmt ein Zeiger auf ein Lock-Objekt entgegen, und greift dann das Lock im Konstruktor. Im Destruktor wird dieses Lock dann wieder freigeben. Der *Lock_guard* behandelt auch das verschachtelte Sperren und Freigeben ein- und desselben Locks.

3.4 IPC-Architektur

Als IPC-Operationen werden das Senden einer Nachricht zu einem bestimmten Thread, kurz *send* und das Warten auf eine Nachricht angeboten. Es kann auf eine Nachricht von einem beliebigen Thread, das offene Warten (*open-wait*), und eines bestimmten Threads, das geschlossene Warten (*closed-wait*), gewartet werden.

Ferner gibt es noch das kombinierte Senden und Empfangen, welches aus einer Sendeoperation gefolgt vom Empfangen einer Antwort vom gleichen Thread (*call*), oder offenes Warten (*reply_wait*) umfasst. Diese kombinierten Operationen garantieren ein atomares Umschalten zwischen dem Sende- und Empfangszustand. Dadurch kann ein Server dem Client mit Timeout Null antworten, ohne dass die IPC mit einem Timeout-Fehler abgebrochen wird.

Timeouts bilden ein eigenes Untersystem im Fiasco-Kern. Auf Timeouts werden IPC-Timeouts, Timeslice-Timeouts und Deadline-Timeouts abgebildet. Der IPC-Timeout dient dazu, Zeitbeschränkungen für IPC-Operationen durchzusetzen. Der Timeslice-Timeout bildet Zeitscheiben und deren Ablauf auf Timeouts ab. Deadline-Timeouts dienen zum Durchsetzen von Perioden und Deadlines.

Sende- und Empfangsrolle

Für die Senderolle ist die Klasse *Sender* zuständig, für die Empfangsrolle, die Klasse *Receiver*. Jeder Sender, also Threads, Interrupts, Preemptions und Activations [Cla05], leitet sich von der *Sender*-Klasse ab. Da Threads auch Nachrichten empfangen können, leitet sich die Klasse *Thread* auch von *Receiver* ab. Die Sender-Klasse enthält die Absenderadresse (Absender-ID), und implementiert die Senderwarteschlange. Sie stellt mit der *ipc_receiver_ready* Methode eine Schnittstelle bereit, welche alle Sender implementieren müssen. Die Klasse *Receiver* enthält den IPC-Zustand, den Kopf der Senderwarteschlange und den IPC-Partner. In Fiasco

ist der Sender das aktive Element in einer IPC-Operation, der Empfänger bleibt bis auf wenige Ausnahmen inaktiv.

Passive Sender

In Fiasco gibt es neben Threads als aktive Sender, auch passive Sender. Sie sind nicht lauffähig und implementieren nur die Schnittstelle der *Sender*-Klasse, die *ipc_receiver_ready* Methode. Der Empfänger ist in diesem Fall das aktive Element der IPC, der sich die Daten direkt von dem Sender holt. Beispiele für passive Sender sind Interruptdeskriptoren, Preemptions und Activations.

Passive Sender überschreiben die *ipc_receiver_ready* Methode der *Sender*-Klasse. Der Empfänger ruft diese Methode auf, wenn er einen Sender in seiner Senderwarteschlange findet. Ein aktiver Sender (Thread) schaltet in dieser Methode zum Sender-Thread um, welcher den Datentransfer durchführt. Bei einem passiven Sender erfolgt der Transfer der Nachricht jedoch direkt in dieser Methode, die auf in dem Kontext des Empfängers ausgeführt wird. Die Nachricht wird aus gespeicherten Informationen innerhalb des passiven Senders konstruiert, und in die Empfangsregister geschrieben.

Weiterhin hat die Aufteilung in aktive und passive Sender zur Folge, dass der Empfänger keine Threads, sondern nur noch Sender kennt. Ein Vorteil dieser Aufteilung ist die hohe Flexibilität. Es kann damit problemlos fast jedes Ereignis auf eine IPC-Nachricht abgebildet werden. Ein Nachteil sind die notwendigen virtuellen Methoden in der Senderklasse, da jeder Sender über andere Attribute verfügt.

Nachrichtenformat

Der Nachrichteninhalt kann aus einfachen Registerinhalten, Speicherseiten (Flexpages), und Speicherinhalten bestehen. Da die Kommunikation synchron ist, müssen die Nachrichten nicht im Kern zwischengespeichert werden.

Man spricht von einer *Short-IPC*, wenn kein Zugriff auf den Nutzeradressraum notwendig ist. Sie umfasst den einfachen Transfer von Registerwerten und Flexpages in Registern. Unter *Long-IPC* werden die IPC-Operationen zusammengefasst, bei denen ein Zugriff auf den Nutzeradressraum erfolgt. Hier können Seitenfehler auftreten. In diesem Fall wird laufende IPC-Operation suspendiert, und der Kern generiert eine verschachtelte Seitenfehler-IPC an den Pager des betroffenen Threads, um den Seitenfehler aufzulösen.

- **Die Registerwerte-IPC** dient zum Übertragen von Registerinhalten und ist sehr schnell, da kein Zugriff auf den Nutzerspeicher erforderlich ist. Die Werte können gleich in den Registern gelassen werden. Zwischen zwei Adressräumen werden zusätzliche TLB-Misses vermieden, da kein Zugriff auf Nutzerspeicher erforderlich ist. Seitenfehler können hier nicht auftreten, so dass keine Maßnahmen zu ihrer Behandlung notwendig sind.
- **Flexpages** dienen dazu, um Speicherseiten von dem Senderadressraum in den Adressraum des Empfängers einzublenden. Ein Pager kann auf eine Seitenfehler-IPC mit einer Flexpage antworten, um den Seitenfehler aufzulösen. Flexpages können in Registern oder in speziellen Nachrichtenstrukturen einer Long-IPC enthalten sein.
- **Long-IPC** umfasst direkte und indirekte Zeichenketten (Strings), sowie Flexpages. In der V2-Schnittstelle wird die Nachricht durch eine spezielle Struktur, dem *Message-Dope* beschrieben. Diese kann Flexpages, mehrere Speicherwörter und Zeiger mit Größenangabe zu den indirekten Zeichenketten enthalten. In der X.2-Schnittstelle wird die Nachricht durch die Nachrichtenwörter in den UTBCs[4] beschrieben.

[4] Die UTCBs sind ein Teil des TCBs, welcher für den Nutzer zugreifbar im Adressraum eingeblendet ist.

Die darin enthaltenen Wörter und die Inhalte der indirekten Zeichenketten werden durch den Kern in den Empfängeradressraum kopiert. Um doppeltes Kopieren vom Sender zum Kern und vom Kern zum Empfänger zu vermeiden, nutzt der L4 Kern das IPC-Fenster. In den Senderadressraum wird ein Teil des Empfängeradressraumes, das IPC-Fenster, eingeblendet. Der Sender kann nun direkt die Speicherinhalte von seinem Adressraum in das IPC-Fenster kopieren. Seitenfehler können im Senderadressraum und im IPC-Fenster auftreten und müssen entsprechend unterschiedlich behandelt werden.

Aufbau einer IPC-Operation

Die Sende und Empfangsoperationen lassen sich in mehrere Teile, das Setup, Rendezvous, Datentransfer und Abschluss aufteilen.

- **Setup & Rendezvous:** Im Setupteil werden aus den übergebenen IDs die Sender und Empfänger bestimmt. Der Zustand wird auf Senden bzw. Empfangen gesetzt. Beim Sender wird sich zusätzlich in die Senderwarteschlange des Empfängers eingekettet. Bei einer kombinierten Sende- und Empfangenoperation, wird das Setup der Empfangsphase vor der Sendeoperation ausgeführt, damit später atomar zwischen dem Sende- und Empfangszustand umgeschaltet werden kann. Dadurch kann ein Server seinem Client mit Timeout Null antworten, um DoS-Angriffe zu vermeiden.

  Beim Rendezvous versucht der Sender mit *ipc_send_regs*, und der Empfänger mit *ipc_receiver_ready* die IPC zu beginnen. Wenn dies nicht möglich ist, blockiert der Thread solange, bis sein IPC-Partner das Rendezvous erneut versucht, oder der Timeout überschritten wird.

- **Datentransfer:** Der Sender ist während des Datentransfers der aktive Teil, der Empfänger bleibt passiv. Ausnahmen sind die passiven Sender und die Behandlung von Seitenfehlern im IPC-Fenster bei einer Long-IPC. In diesem Fall wird der Empfänger vom Sender aufgeweckt, damit dieser eine Seitenfehler-IPC an seinen Pager senden kann. Wenn diese vom Pager des Empfängers beantwortet wird, weckt der Empfänger den Sender wieder mit *ipc_continue* auf. Der Sender übernimmt wieder die aktive Rolle, und führt die Nachrichtenübertragung fort.

- **Abschluss:** Im der Abschlussphase weckt der Sender den Empfänger auf, indem er die IPC-Zustandsbits im Zustandswort des Empfängers löscht und ihn lauffähig setzt. Ferner kann eine IPC auch durch eine Timeoutüberschreitung, oder durch den Abbruch der IPC-Operation mittels *thread_ex_regs* beendet werden. Wenn eine laufende IPC mit *thread_ex_regs* abgebrochen wird, wird die IPC beendet und der Sender und Empfänger kehren zum Nutzer mit einem Fehler zurück.

Assemblershortcut

Der Assemblershortcut [Pet02] ist eine Abkürzung im IPC-Pfad, welcher nur den einfachen Registerwertetransfer ohne Flexpages behandelt. Es werden als IPC-Operationen das Senden und das kombinierte Senden und Empfangen unterstützt. Als Timeouts sind nur Timeout Null oder unendlich möglich. Es ist die schnellste IPC-Implementation von Fiasco. In den anderen Fällen erfolgt die Nachrichtenübertragung durch den normalen IPC-Pfad. Der Shortcut läuft mit gesperrten Interrupts.

Obwohl nur der einfache Registerwertetransfer unterstützt wird, profitieren viele Applikationen, z.B. L4Linux davon, da oftmals nur sehr wenige Wörter übertragen werden. Oft wird IPC zur Synchronisation innerhalb eines Adressraums verwendet, so dass hier der Assemblershortcut sehr hilfreich ist.

Da der Kernadressraum verzögert zwischen den Adressräumen synchronisiert wird, kann der Fall eintreten, dass der IPC-Shortcut, bei dem Zugriff auf einem nicht eingeblendeten Empfänger-TCB, einen Seitenfehler

auslöst. In diesem Fall setzt die Routine zur Behandlung des Seitenfehlers nur ein Flag, und kehrt sofort zurück, ohne jedoch die Unterbrechungen freizugegeben. Der Assemblershortcut testet nach diesem ersten Zugriff dieses Flag, und bricht die IPC ab, wenn es gesetzt ist. Er ruft den generischen IPC-Pfad auf, welcher die IPC dann behandelt. Dies ist notwendig um größere Verzögerungszeiten durch die Seitenfehlerbehandlung zu vermeiden, damit mit gesperrten Interrupts keine Seiten alloziert werden müssen.

Time-Slice-Donation

Nach Abschluss der IPC wird im Normalfall immer vom Sender zum Empfänger umgeschaltet. Es wird zwar der Ausführungskontext umgeschaltet, der Scheduling-Kontext jedoch nicht. So wird, ähnlich dem Helfen bei Locks, dem Empfänger die verbleibende Zeit der aktuellen Zeitscheibe geschenkt. Dies ist besonders bei Client-Server-Szenarios sehr nützlich, der Client schenkt dem Server seine Zeit, damit dieser die Anfrage möglichst schnell beantworten kann, und Prioritätsinversion wird so vermieden

Weil das Umschalten zum Empfänger und die Weitergabe der Zeitscheibe nicht immer gewünscht ist, wurde der *Deceite-Bit-Hack* eingeführt. Hier wird ein unbenutztes Bit, das *Deceite-Bit*[5] genutzt, als Information für den Kern, dass er nach der IPC-Operation nicht automatisch zum Empfänger umschalten soll. Es erfolgt damit keine Weitergabe der Zeitscheibe.

[5] Deceite-Bit: ein Bit in dem Sende-Deskriptor, welches für das *Clan & Chief* Sicherheitsmodell von Bedeutung war. Da dieses Modell zu unflexibel ist, wird es jedoch in fast keinem L4-Kern implementiert

4 Analyse der Verzögerungszeiten

Um Aussagen über die notwendige Häufigkeit und Verteilung von Unterbrechungspunkten, an denen der IPC-Pfad vom Zeitgeber und Interrupt unterbrochen werden kann, zu erhalten, wird der Zeitbedarf von Operationen der Hardware und Funktionen des Fiasco-Kerns ausgemessen.

4.1 Hardware

Die Hardwarekosten für verschieden Operationen unterscheiden sich auf modernen Prozessoren um mehrere Größenordnungen. Ein einfaches Ausnullen eines Registers ist erheblich schneller als der Zugriff auf einen IO-Port. Dies muss bei der Platzierung von Unterbrechungspunkten berücksichtigt werden, damit die Verzögerungszeiten nicht zu groß werden. Als Testrechner wird 1.6Ghz Pentium IV verwendet, auf dem verschiedene Maschineninstruktionen ausgemessen werden, um einen Überblick über die Kosten zu gewinnen.

Bis auf wenige Ausnahmen sind die meisten Instruktionen sehr schnell. Ausnahmen sind zum Beispiel , der Kernein- und Kernaustritt, sowie bestimmte Systembefehle, welche nur dem Betriebssystem erlaubt sind. Tabelle 4.1 und 4.2 gibt einen Überblick darüber. Besonders teure Instruktionen sind der Kernein und -austritt mittels INT+IRET und Zugriffe auf IO-Ports (IN8, OUT8). SYSENTER wird nur bei IPC-Systemaufrufen genutzt, alle anderen Systemcalls nutzen die INT-Instruktion zum Kerneintritt. Aus Kompatibilitätsgründen ist auch der IPC-Systemaufruf mittels INT-Kerneintritt möglich. Dies muss berücksichtigt werden, da diese Instruktion signifikant zur Laufzeit des IPC-Pfades beiträgt.

Die Routinen von Zeitgeber und zur Interruptbehandlung führen IO-Zugriffe aus, um die Unterbrechung am Interruptcontroller zu quittieren. Fiasco-x86 betreibt den Interruptcontroller (PIC[1]) im “Special Fully Nested Mode”, um auch die Interruptprioritäten des Slave-Interruptcontrollers zu berücksichtigen. Dies erzwingt ein besonderes Protokoll zur Bestätigung von Interrupts, die vom Slave-PIC ausgelöst werden. Eine Bestätigung eines Interrupts, welcher durch den Slave-PIC ausgelöst wurde, kann bis zu drei IO-Ausgabe- (OUT8) und eine IO-Eingabeoperation (IN8) erfordern.

Bei dem Kontextwechsel spielt besonders die INVLPG-Instruktion, zum Invalidieren eines TLB-Eintrags, und das Neuladen des CR3-Registers[2] eine große Rolle. Wenn zwischen zu einem Threads innerhalb eines Adressraumes umgeschaltet wird, welcher eine eine Long-IPC-Operation durchführt, muss das IPC-Fenster im TLB mit INVLPG ungültig gemacht werden, damit jeder Thread stets das richtige IPC-Fenster vom Adressraums des Empfängers besitzt.

Wenn zwischen Adressräumen umgeschaltet wird, muss der komplette TLB durch Neuladen des CR3-Registers invalidiert werden. Das Invalidieren des IPC-Fensters erfolgt dabei implizit. Da der Fiasco-Kern zum größtem Teil global markierte Seiten im Kernadressraum verwendet, welche bei einem Löschen des TLBs nicht entfernt werden, halten sich die zusätzlichen Kosten der nachfolgenden Zugriffe im Kernadressraum in Grenzen.

[1]Programmable Interrupt Controller

[2]Dieses Maschinenregister zeigt auf das aktuelle Seitenverzeichnis, welches den Adressraum beschreibt

Weiterhin muss der Fliesskommaprozessor (FPU) mit CLTS gesperrt werden, wenn von einem Thread, für den die FPU freigeschalten ist, weggeschaltet wird. Dadurch erfolgt das verzögerte Sichern und Wiederherstellen des FPU-Kontextes.

Ein Unterbrechungspunkt besteht aus Sperren (CLI) und Freigeben (STI) der Interrupts, sowie einer Nulloperationen dazwischen. Bei mehreren Unterbrechungspunkten summieren sich diese Kosten, so dass sie durchaus mehr als zehn Prozent der benötigten Kosten für eine IPC ausmachen. Die CAS-Instruktion kann eingespart werden, da bei deaktivierten Interrupts, die normalen Instruktionen ausreichen.

Maschineninstruktion	Taktzyklen	Zeit
Neuladen CR3	252	158ns
invlpg	516	323ns
in8	1491	0.93μs
out8	1392	0.87μs
int + iret	1364	0.85μs
sysenter + sysexit	167	104ns
cli (34 Takte) + sti (44 Takte)	78	49ns
clts	268	168ns
cmpxchg(CAS)	26	17ns
cmpxchg(CAS) + lock-Prefix	147	92ns

Tabelle 4.1: Ausführungszeiten von speziellen Maschineninstruktionen

Cache

Die Lücke zwischen CPU-Geschwindigkeit und Geschwindigkeit beim Speicherzugriff wird immer größer. Die heutigen Speicher verfügen zwar über eine Bandbreite von mehreren Gigabyte in der Sekunde, aber die Zugriffszeit auf zufällige Speicherwörter hat sich nur unwesentlich verbessert. Es dauert immer noch sehr lange, vom Beginn des Zugriffs, bis das erste Datenwort gültig ist. Caches dienen dazu, den schnellen Prozessor vom langsamen Speicher zu entkoppeln. Sie machen aber die genaue Voraussage über Ausführungszeiten im schlechtesten Fall sehr schwer.

Bei einem Burstzugriff werden gleich mehrere Datenwörter übertragen, zum Beispiel zum Füllen oder Schreiben einer Cachezeile. Bei SD-RAM ist ein üblicher Burstmode 5-1-1-1, dies heißt, das erste Datenwort steht nach fünf Bustakten zur Verfügung, die nachfolgenden Datenworte nach einem Takt. DDR-RAM verbessert zwar den Durchsatz bei einem Burstzugriff erheblich, die Zugriffszeit verbessert sich aber nur unwesentlich.

Während ein Cache-Miss im L1-Cache rund 20 Takte (Tabelle 4.2) beträgt, ist ein Miss im L2-Cache signifikant teurer. Wenn bei einer Schreiboperation im L2-Cache ein Cache-Miss auftritt, lädt die verwendete CPU die Cachezeile in den Cache, auch *Write-Allocation* genannt. Dadurch profitieren nachfolgende Leseoperationen, wenn sie von der gleichen Cachezeile lesen. Ferner verfügt die CPU über mehrere Schreibpuffer, welche begrenzt die zusätzlichen Kosten für einen Schreib-Cache-Miss abfangen können.

Ein weiterer Cache ist der TLB, welcher die Umsetzung von virtuellen in physische Speicheradressen beschleunigt. Der verwendete Pentium IV Prozessor besitzt einen getrennten TLB für Daten und für Code, beide mit 64 Einträgen. Wenn für eine virtuelle Adresse kein passender Eintrag im TLB gefunden wird (TLB-Miss), muss der Prozessor die Seitentabelle traversieren und den neuen Eintrag in den TLB eintragen. Wenn sich die benötigten Einträge der Seitentabellen im Cache befinden, kostet ein TLB-Miss 40 Takte. Bei kalten Caches, können bei dem Lesen der Einträge im Seitenverzeichnis und Seitentabelle zwei Cache-Misses auftreten, was in der Summe dann rund 400 Takte benötigt.

Die Kosten für einen Cache-Miss zeigen deutlich, dass der Cache nicht ignoriert werden kann. Die Ausführungszeiten können im schlechtesten Fall ein oder zwei Größenordnungen über dem durchschnittlichen Fall liegen.

Seitenfehler- und Interruptlatenz

Seitenfehler

Weitere Kosten entstehen durch Seitenfehler, welche im Kern durch die verzögerte Synchronisation der TCBs oder bei Long-IPC-Operationen auftreten können. Mit einem kalten Cache kann es mehr als 2000Takte dauern, bis nach einem Seitenfehler die Routine zur seiner Behandlung angesprungen wird.

Die Bestimmung der Kosten durch einen Seitenfehler erfolgte durch Auslesen des Time-Stamp-Counters (TSC[3]), Auslösen eines Seitenfehlers, und erneutes Auslesen des TSCs und Bildung der Differenz. Als Ergebnis wurden über 2200 Takte für einen Lesezugriff und 2000 Takte für einen Schreibzugriff ermittelt.

Der Assemblercode am Einsprungspunkt der Funktion zur Seitenfehlerbehandlung sichert die benötigten Register, darunter auch das CR2 Register, welches die Adresse des Seitenfehlers enthält. Dieser Code läuft mit gesperrten Interrupts, um ein Überschreiben des CR2-Registers durch einen möglichen Kontextwechsel zu verhindern, denn bei einem Kontextwechsel wird dieses Register nicht gesichert.

Es muss also stets berücksichtigt werden, dass ein Zugriff auf einen TCB in wenigen Fällen einen Seitenfehler, zur Synchronisation der TCBs, auslösen kann.

Interrupts

Weiterhin wurden die Verzögerungszeiten zwischen dem Auslösen eines Interrupts und dem Aufruf der Interrupt-Routine bestimmt. Zum Ermitteln der benötigten Zeit liest ein Nutzerthread in einer Schleife ständig den TSC aus. Nach dem Auftreten des Interrupts werden in der Interrupt-Routine die benötigten Register gesichert, der TSC erneut ausgelesen und die Differenz gebildet.

Als Interruptquelle diente der Interrupt der seriellen Schnittstelle. Die gemessenen Zeiten beinhalten also das Erkennen des Interrupts durch den Controller der seriellen Schnittstelle, das Auslösen des Interrupts am PIC und die Zustellung zur CPU, die dann die passende Routine aufruft.

Die Ergebnisse zeigen, wenn die CPU, nach einem STI, Interrupts erneut entgegen nimmt, dauert es mitunter über 2000 Takte, bis die Interrupt-Routine angesprungen wird.

4.2 Software

Es werden verschiedenste Operationen und Funktionen von Fiasco analysiert, um eine Aussage über die zu erwartenden Kosten und erreichbaren Verzögerungszeiten zu erhalten. In Fiasco laufen bestimmte Operationen mit gesperrten Interrupts ab, um kurze kritische Abschnitte zu synchronisieren. Darunter fallen der Kontextwechsel, zum Teil die *schedule*-Funktion, die Timer-Interrupt-Routine und der Code zur Interruptzustellung.

[3] Time-Stamp-Counter: Dieser Zähler wird bei jedem Taktzyklus inkrementiert. Der TSC bietet die beste und genaueste Möglichkeit die Kosten für verschiedenste Operationen auszumessen

Ereignis	Taktzyklen	Zeit
Cache Miss(L1)	23	14ns
Cache Miss(L2) Lesen	197	124ns
Cache Miss(L2) Schreiben	275	172ns
TLB Miss Daten (4k)	44	28ns
TLB Miss Code (4k)	40	25ns
Seitenfehler Lesen	2204	1.3us
Seitenfehler Schreiben	1972	1.2us
Latenzzeiten von Interrupts	2424	1.5us

Tabelle 4.2: Hardware Verzögerungszeiten

Kontextwechsel

Bis auf Ausnahmen ist der Kontextwechsel sehr schnell. Ausnahmen sind zum Beispiel das Löschen des IPC-Fenster oder die Umschaltung des Adressraumes. Beim Löschen des IPC-Fensters wird zweimal die INVLPG-Instruktion aufgerufen. Beim Neuladen des CR3-Registers für das Umschalten des Adressraumes spielen erstens die Kosten für das Neuladen eine Rolle. Die weitaus teureren Kosten entstehen jedoch durch die nachfolgenden TLB-Misses. Obwohl die meisten TLB-Seiten als global markiert sind, und bei einem Neuladen des CR3-Registers nicht aus dem TLB entfernt werden, können sie doch von anderen TLB-Einträgen verdrängt worden sein. So können zu den Kosten für das Neuladen noch zusätzliche Kosten entstehen, wenn nach dem Umschalten des Adressraumes auf Seiten zugegriffen wird, dessen Adressen nicht mehr im TLB enthalten sind.

Eine Besonderheit ist beim Kontextwechsel zu beachten. Die Thread-Lock Semantik erfordert, dass keine gesperrten Threads eingeplant werden, Ausnahme ist *Thread::kill*. Um dies zu erreichen, wird beim Umschalten zu einem gesperrten Thread, stattdessen zu seinem Lock-Besitzer umgeschaltet. Dieser Vorgang ist transitiv, wenn der neue Thread auch wieder gesperrt ist, wird zu seinem Lock-Besitzer umgeschaltet. Dieses Traversieren der Lock-Besitzerkette mit gesperrten Interrupts kann potentiell sehr lange dauern. Wenn die TCB durch Seitenfehler nachträglich eingeblendet werden, verzögert sich das Durchlaufen dieser Kette noch weiter. Vermeiden kann man dies nur, wenn die Synchronisation im Fiasco-Kern komplett überarbeitet wird, welches aber nicht im Blickpunkt dieser Arbeit liegt.

Scheduler

Die *schedule*-Funktion ist der Scheduler und dieser wählt unter Beachtung der Prioritäten einen neuen aktiven Thread aus. Es muss berücksichtigt werden, dass in der Ready-Liste Threads enthalten sind, welche nicht rechenbereit sind. Diese werden dann nachträglich ausgekettet. *Schedule* iteriert absteigend von der maximalen Priorität im System, über die Listen von Threads gleicher Priorität. Das Iterieren über die Elemente einer dieser Listen erfolgt mit Unterbrechungspunkten und endet sobald ein lauffähiger Thread gefunden wird. Nicht bereite Threads werden aus der Liste ausgekettet und nach einem Unterbrechungspunkt wird zum nächsten Element übergegangen.

Wenn der aktuelle Thread bei dem Aufruf von *schedule* noch lauffähig ist, muss er nachträglich in die Ready-Liste eingekettet werden. Wenn er nicht mehr lauffähig ist, sein Ready-Flag ist gelöscht, wird er dagegen aus der Ready-Liste entfernt. Dies ist der Fall, wenn ein Thread blockiert, indem er sein Ready-Flag in seinem Threadzustand löscht und *schedule* aufruft. Selbst wenn sofort ein neuer Thread gefunden wird, und kein Unterbrechungspunkt notwendig ist, ist der Aufwand signifikant. Es wird der aktuelle Thread in die Ready-Liste

eingekettet und zu dem nächsten Thread umgeschaltet. Dieser Kontextwechsel umfasst auch ein Umschalten des Scheduling-Kontextes.

Timer-Interrupt

Die Zeitgeberroutine gehört neben der normalen Interruptbehandlung zu den Programmteilen, welche für einen Großteil der Verzögerungszeiten verantwortlich sind [Hoh02b]. Die Timer-Interrupt-Routine läuft die meiste Zeit mit gesperrten Interrupts. In der Timer-Interrupt-Routine erfolgt die Bestätigung der Unterbrechung am PIC und gegebenenfalls am Zeitgeber. Ferner werden die Timeouts behandelt. Zum Schluss erfolgt bei Bedarf ein Aufruf von *schedule*.

Die benötigte Zeit zum Bestätigen des Interrupts hängt stark vom verwendeten Zeitgeber ab. Die APIC und der PIT sind nicht zeitaufwendig. Bei Verwendung der RTC als Zeitgeber, ist die Bestätigung des Interrupts sehr teuer. Es muss der RTC-Interrupt am Slave-PIC und Master-PIC bestätigt werden, welches bis zu drei OUT8 und eine IN8 Instruktion erfordern kann. Zweitens ist auch das Bestätigen des Interrupts an der RTC sehr teuer. Dies erfolgt zweimal, um Inkompatibilitäten mit verschiedener Hardware zu vermeiden.

Nach dem Bestätigen des Interrupts erfolgt die Behandlung der Timeouts. Dort wird über die sortierte Liste der Timeouts iteriert, die betroffenen Timeouts werden behandelt und ausgekettet. Das Behandeln der Timeouts umfasst zum Beispiel Änderung eines Threadszustandes und Einketten in die Ready-Liste bei einem IPC-Timeout. Zum Schluss wird bei *One-Shot-Timern* der Zeitgeber neu programmiert, damit er den nächsten Interrupt auslöst. Das Behandeln der Timeouts kann, ähnlich wie das Traversieren der Lock-Besitzer-Kette bei dem Kontextwechsel, zu erheblichen Verzögerungszeiten führen. Die Funktion zum Behandeln der Timeouts liefert zurück, ob ein Aufruf von *schedule* notwendig wird. Beispiele sind der Ablauf eines Timeslice-Timeouts, oder der Ablauf eines IPC-Timeouts eines Threads mit höherer Priorität als der aktuelle Thread.

Interupt-Routine

Die Interrupt-Routine dient dazu, für aufgetretene Interrupts eine IPC-Nachricht zu generieren und dem Nutzer zustellen. Dies ist neben der Timer-Interrupt-Routine einer der längsten Abschnitte im Programmcode, welcher mit geschlossenen Interrupts ausgeführt wird. Durch eine Konfigurationsoption kann beim Kompilieren des Fiasco-Kerns weiter festgelegt werden, ob der Kern oder der Nutzer den Interrupt am PIC bestätigen muss.

Es muss eine IPC-Nachricht für diesen Interrupt generiert werden. Der Interruptdeskriptor wird dazu in die Senderwarteschlange des assoziierten Empfängers einkettet. Der Zustand des Empfängers wird auf lauffähig gesetzt, und wenn der Empfänger eine höhere Priorität als der aktuelle Thread besitzt, erfolgt ein Kontextwechsel, welcher auch das Umschalten des Scheduling-Kontext umfasst. Dies erfolgt alles mit gesperrten Interrupts. Es gibt auch einen Interrupt-Shortcut, welcher den Kernstack des Empfängers modifiziert und direkt zu ihm umschaltet. Durch das Modifizieren des Kernstacks wird erreicht, dass der Empfänger sofort zum Nutzer zurück kehrt, ohne dass er vorher alle Stackrahmen[4] auf dem Kernstack aufgeräumt werden müssen.

Weitere Operationen

Lock-Operationen: Die Synchronisation innerhalb der Lock-Methoden erfolgt durch Sperren und Freigeben der Interrupts. Wenn das Lock nicht frei ist, wird ein Kontextwechsel zu dem Lock-Besitzer ausgeführt. Bei

[4] Der Stackrahmen beinhaltet die Rücksprungadresse und lokale Variablen der aktuellen Funktion und wird auf dem Stack angelegt. Bei verschachtelten Funktionsaufrufen stapeln sich dann ihre Stackrahmen auf dem Stack

dem Freigeben von Locks kann auch ein Kontextwechsel notwendig werden, wenn dem bisherigen Lock-Besitzer durch einen höher priorisierten Thread geholfen wurde. Bei Thread-Locks ist besonders bei dem Freigeben des Locks noch ein weiterer Kontextwechsel, oder ein Aktualisieren der Ready-Liste notwendig, wenn der gesperrte Thread lauffähig ist. Tabelle 4.3 gibt dazu einen Überblick über die Kosten.

Da der IPC-Code zum größten Teil mit geschlossenen Interrupts läuft, ist bei den Lock-Operationen die Überprüfung, ob die Interrupts gesperrt sind, nicht notwendig. Deshalb werden dafür optimierte Funktionen bereitgestellt. Im Durchschnitt sinkt dadurch die Zeit um ein Thread-Lock zu holen und freizugeben von 274 auf 167 Takte.

Listenoperationen: Für die Ready-Liste, die Present-Liste[5] und die Senderwarteschlange werden doppelt verkettete Listen verwendet. Die Synchronisation erfolgt durch Sperren der Interrupts. Die Listenelemente sind TCBs, welche die *prev*- und *next*-Zeiger enthalten.

Da diese Zeiger nebeneinander und damit oftmals innerhalb einer Cachezeile liegen, tritt nur ein Cache-Miss beim Zugriff auf den *prev* und *next*-Zeiger eines Elementes auf. Zwar wird der einzukettende TCB im Cache enthalten sein, die angrenzenden Elemente dieser Liste jedoch nicht, so dass im ungünstigen Fall zwei Cache-Misses und zwei TLB-Misses auftreten koennen. Die Kosten für das Einketten kann dadurch rund 480 Takte (2x200 Takte für den Cache-Miss und 2x40 Takte für den TLB-Miss) betragen. Wenn sich die benötigten Einträge der Seitentabelle und des Seitenverzeichnis nicht mehr im Cache befinden, fallen die Kosten noch höher aus. Schon eine einfache Beispielrechnung zeigt, dass der Cache erheblichen Einfluss auf die Verzögerungszeiten besitzt.

Es zeigt sich auch hier der Nachteil der verzögerten Synchronisation des Kernadressraumes. Im schlechtesten Fall können die angrenzenden TCBs noch nicht im aktuellen Kernadressraum eingeblendet sein. In diesem Fall kommen noch die Kosten für einen Seitenfehler und dessen Behandlung dazu. Obwohl dessen Behandlung sehr kurz ist, es werden nur zwei Einträge im Seitenverzeichnis kopiert, können dadurch die Kosten noch erheblich ansteigen.

IPC-Shortcut: Den IPC-Shortcut gibt es in der Assembler- und in der C++-Version. Der Shortcut läuft mit gesperrten Interrupts. Er behandelt den einfachen Registerwertetransfer und wird mittels SYSENTER und INT betreten. Die Kosten mit heißen und kalten Caches für den Shortcut sind in Tabelle 4.3 zu finden. Ferner kann noch ein Seitenfehler beim Zugriff auf den Empfänger-TCB auftreten, falls dieser TCB nicht eingeblendet ist. Deshalb soll nach einem Abbruch des Shortcuts ein Unterbrechungspunkt folgen, bevor der generische IPC-Pfad aufgerufen wird.

4.3 Zusammenfassung

Der Zeitbedarf für bestimmte Maschineninstruktionen und Fiasco-Kernoperationen hat einen sehr großen Einfluss auf die Verzögerungszeiten. Bestimmte Verzögerungszeiten können sich dazu noch addieren. Zum Beispiel, wenn der IPC-Shortcut aktiv ist, muss ein anderer Interrupt am PIC solange warten, bis die CPU die Interrupts wieder zulässt. Dazu kommen die Kosten für die Zustellung und die Behandlung des neu ausgelösten Interrupts. Selbst auf aktuellen CPUs entspricht dies Verzögerungszeiten im Bereich von mehreren Mikrosekunden.

Schon das einfaches Laden und Schreiben eines Registers in den Speicher kann im ungünstigen Fall sehr lange dauern. Selbst wenn man nur TLB- und Cache-Misses durch den Datenzugriff betrachtet, können bei dem Lesezugriff auf eine richtig angeordnete Adresse, ein TLB- und drei Cache-Misses auftreten, wodurch die Kosten auf 640 Takte steigen. Der Fall, dass die Adresse des Speicherworts falsch angeordnet ist, und der Zugriff damit noch länger dauern kann, tritt im Fiasco-Kern nicht auf.

[5]Die Present-Liste enthält alle Threads im System

Operation	Taktzyklen	Zeit
Holen und Freigeben eines Thread-Locks		
Kalter Cache	1242 (577+665)	0.7μs
Heißer Cache	274	172 ns
Kontextwechsel		
innerhalb eines Adressraums	214	134ns
zwischen Adressräumen	557	349ns
Kontextwechsel + Umschalten der Zeitscheibe		
innerhalb eines Adressraums	426	267ns
zwischen Adressräumen	720	450ns
Assembler-Shortcut (heißer Cache)		
innerhalb eines Adressraums	320	200ns
zwischen Adressräumen	720	450ns
Assembler-Shortcut (kalter Cache)		
innerhalb eines Adressraums	$\approx$1800	$\approx$1.1μs
zwischen Adressräumen	$\approx$3100	$\approx$1.9μs

Tabelle 4.3: Verzögerungszeiten von Kernoperationen

Um die Verzögerungszeiten zu minimieren, sollten oft Unterbrechungspunkte gesetzt werden, besonders vor längeren Kernoperationen. Diese erhöhen jedoch die Laufzeit des IPC-Pfades signifikant. Wenn man zu wenig Unterbrechungspunkte setzt, besteht aber das Problem höherer Verzögerungszeiten im ungünstigen Fall. Ein möglicher Ausweg ist das verzögerte Sperren von Interrupts, welches später kurz beschrieben wird.

Anzumerken ist, bei einem IPC-Pfad, welcher mit gesperrten Interrupts ausgeführt wird, treten Cache- und TLB-Misses nur bei den ersten Zugriffen auf, danach liegen die benötigten Werte im Cache. Deshalb reicht schon eine mittlere Anzahl von Unterbrechungspunkten aus, um kleine Verzögerungszeiten zu erhalten. Das inoffizielle Ziel jedoch, eine Latenz zu erreichen, deren Kosten in der gleichen Größenordnung wie für die längste verwendete Maschinenoperation liegen, ist, wenn man den Cache berücksichtigt, nicht erreichbar,

5 Entwurf

Der ursprüngliche IPC-Pfad gliedert sich in mehrere Teile. Die Hauptteile sind der Kerneintritt, das grundlegende Setup, eine mögliche Sende- und Empfangsoperation, sowie der Abschluss. Den größten Teil der IPC-Operationen stellen *call* und *reply_wait* Operationen, welche immer den Sende- und Empfangsteil umfassen und atomar zwischen diesen beiden Teilen umschalten. Diese IPC-Operation gilt es daher besonders zu optimieren.

Die Sende- und Empfangsoperationen untergliedern sich weiter in eine Setup-, Rendezvous-, Datentransfer- und Abschlussphase. Der Datentransfer kann nochmals in Registerwertetransfer und Long-IPC-Transfer aufgeteilt werden.

Bisher wird nur der Kerneintritt und gegebenenfalls der IPC-Shortcut mit geschlossenen Interrupts ausgeführt, die anderen Teile sind voll unterbrechbar. Die Idee ist es, bis auf den Long-IPC-Transfer, alle Teile nicht unterbrechbar zu machen, um so die Kosten für die Synchronisation einzusparen und die Geschwindigkeit zu maximieren. Um geringe Latenzzeiten zu garantieren, müssen an bestimmten Stellen Unterbrechungspunkte eingefügt werden, damit dort der aktive Thread verdrängt werden kann.

Der Long-IPC-Transfer umfasst das Kopieren von Speicherinhalten und das Einblenden von Flexpages in den Adressraum des Empfängers. Das Kopieren von Speicherinhalten ist sehr zeitaufwendig, besonders bei *dirty* Cachezeilen, weil vorher der alte Inhalt der Cachezeile in den Speicher zurückgeschrieben werden muss. Durch das Sperren der Interrupts wird nur die Zeit für Setup der Long-IPC-Operation geringfügig besser. Deshalb wird der Nachrichtentransfer bei Long-IPC-Operationen mit freigegebenen Interrupts ausgeführt. Ferner ist die Behandlung von Flexpages durch die Änderung der Mappingdatenbank teuer. Wenn also bei einem Registerwertetransfer die Register Flexpages enthalten, werden Interrupts zugelassen.

Folgende Phasen werden mit geschlossenen Interrupts ausgeführt, Kerneintritt, Setup, Rendezvous, Registertransfer, sofern er keine Flexpages umfasst, und die Abschlussphase. Innerhalb und zwischen diesen Phasen werden Unterbrechungspunkte eingefügt, um geringe Verzögerungszeiten zu erreichen.

5.1 Kernspeicher

Im aktuellen Fiasco wird aus der Thread-ID die Adresse des zugehörigen TCBs mittels einfacher Bitoperationen errechnet, Grafik 5.1 zeigt dies für die V2-Schnittstelle. Ein Hauptproblem mit dem bisherigen Verfahren, ist die verzögerte Aktualisierung des TCB-Bereichs im Kernadressraum. Wie schon erwähnt, müssen die Änderungen im TCB-Bereich durch neu angelegte Threads, allen Adressräumen bekannt gemacht werden. Dies erfolgt nachträglich bei der Behandlung von Seitenfehlern im TCB-Bereich.

Jedesmal wenn ein Zugriff auf einen nicht eingeblendeten TCB erfolgt, wird durch die CPU ein Seitenfehler ausgelöst. Die Seitenfehler-Routine kopiert dann aus dem Kernhauptverzeichnis den passenden Eintrag in das aktuelle Seitenverzeichnis, und kehrt anschließend zum ursprünglichen Kontext zurück. Die Interrupts bleiben dabei gesperrt. Erst wenn neue Seiten alloziert werden müssen, oder eine Seitenfehler-IPC aufgesetzt wird, werden die Interrupts in der Seitenfehler-Routine freigegeben.

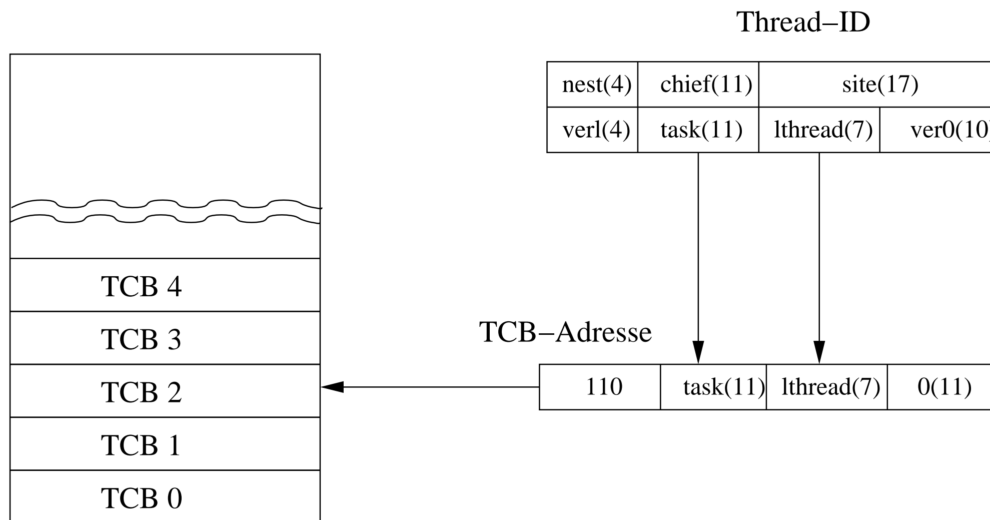

Abbildung 5.1: TCB Zugriff.

Dieses Verfahren verschwendet keinen unnötiger Speicher, aber aufgrund der Kosten zur Seitenfehlerbehandlung, erhöhen sich die Verzögerungszeiten. Die Messungen ergaben über 2000 Takte bis zum Aufruf der Seitenfehler-Routine. Dazu kommen noch die Kosten der Seitenfehlerbehandlung. Im ungünstigen Fall können diese Seitenfehler eine große Rolle spielen.

In vielen Listen sind die Elemente TCBs, zum Beispiel die Ready-Liste, die Senderwarteschlange, die *Present*-Liste und die Timeout-Liste. So kann beim Einketten eines Threads in die Ready-Liste ein Seitenfehler, beim Aktualisieren der *prev* und *next*-Zeiger, auftreten, wenn die angrenzenden TCBs in der Liste nicht eingeblendet sind. Das Traversieren der Lock-Besitzer-Kette beim Kontextwechsel in *switch_to* ist auch davon betroffen.

Die meisten Systemaufrufe, die andere Threads modifizieren, überprüfen zuerst den Zustand des Ziel-Threads. Der erste Zugriff auf den TCB liefert einen ungültigen Threadzustand, wenn der TCB mit der Nullseite hinterlegt ist. Die Operation wird dann mit einem Fehler abgebrochen. Ein Seitenfehler tritt bei einem Lesezugriff nur dann auf, wenn der TCB mit keiner Seite hinterlegt ist.

Da die Nullseite schreibgeschützt ist, treten beim Schreiben auf den TCB Seitenfehler auf. Dies spielt jedoch nur beim Erzeugen eines Threads eine Rolle. Andere Systemaufrufe lesen vorher den Zustand des Threads, und ermitteln einen ungültigen Zustand. Sie brechen dann mit einem Fehler ab.

Keine spezielle TCB Region

Dieser Ansatz verzichtet komplett auf einen speziellen TCB-Bereich im Kernadressraum. TCBs werden nur in der 1:1 Abbildung des physischen Kernspeichers angelegt. Diese Abbildung wird beim Booten des Kerns angelegt, so dass neue Adressräume schon über diese Einträge im Seitenverzeichnis verfügen. Dadurch werden Seitenfehler beim Zugriff auf TCBs vermieden.

Auf die TCBs wird indirekt über eine Tabelle, siehe Abb. 5.2, zugegriffen. Diese Tabelle enthält dann die Adressen der TCB, welche direkt in der 1:1 Abbildung des physischen Kernspeichers liegen.

Dies erfordert zusätzliche Kosten bei der Bestimmung der TCB-Adresse aus einer gegebenen Thread-ID, da jetzt die TCBs nicht mehr an festen Adressen liegen. Diese zusätzlichen Kosten bestehen im größten Teil durch eine zusätzliche Indirektion. Es kann aber der bisherige dedizierte TCB-Bereich im Kernadressraum eingespart werden. Dort können dann weitere kleine Adressräume (SMAS) oder ein vergrößerter Nutzeradressraum liegen.

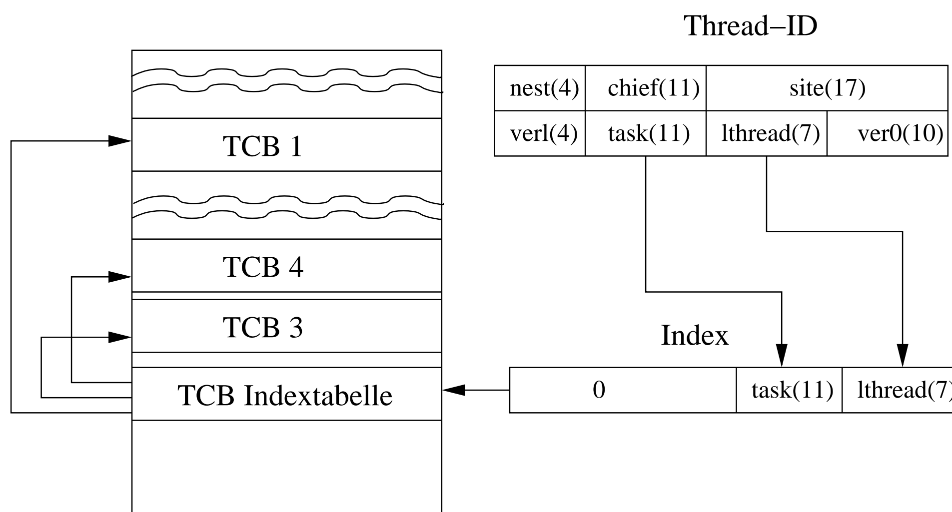

Abbildung 5.2: Indirekter TCB Zugriff.

Ein weiterer Nachteil ist die Größe der Tabelle. Lösungen hierfür sind eine mehrstufige Tabelle oder Hashes. Bei mehreren Stufen kann die erste Stufe in die Seitentabelle integriert werden, so dass die MMU[1] einen großen Teil der Arbeit abnimmt. Hier wird eine virtuelle Speicherregion von der Größe der benötigten Tabelle reserviert. Bei 2048 Adressräumen mit je 128 Threads, wobei pro Eintrag vier Bytes notwendig sind, besitzt die Tabelle eine Größe von einem Megabyte. Die hierfür notwendige gemeinsame Seitentabelle kann beim Booten gleich reserviert werden. Leere Bereiche werden mit der schreibgeschützen Nullseite hinterlegt. Änderungen in dieser Tabelle sind durch die gemeinsame Seitentabelle sofort in allen Adressräumen sichtbar. Es treten Seitenfehler nur beim Anlegen eines Threads auf, wenn die Tabelle modifiziert werden muss, und der TCB-Zeiger in einen Bereich eingetragen wird, der mit der schreibgeschützten Nullseite hinterlegt ist.

Allozieren der benötigten Seitentabellen beim Booten

Hier werden beim Booten alle notwendigen Seitentabellen alloziert, um den 512MByte großen TCB-Bereich im Kernspeicher abzudecken. Dies erfordert 512KByte Kernspeicher für die benötigten Seitentabellen. Da diese Seitentabellen gemeinsam genutzt werden, können “leeren Stellen” im TCB-Bereich, schon mit der schreibgeschützten Nullseite hinterlegt werden. Es treten Seitenfehler nur beim Erzeugen von Threads auf, da andere Systemaufrufe beim ersten Lesen einen ungültigen Thread ermitteln und mit einem Fehlercode den Systemaufruf abbrechen.

Diese Lösung hat den Nachteil, dass kostbarer Kernspeicher verschwendet wird, da oft nie die maximale Anzahl von Threads genutzt wird. Aufgrund der Verschwendung von Kernspeicher, wird von dieser Lösung Abstand genommen.

Aktualisierung des aktuellen Seitenverzeichnisse vor dem Zugriff

Eine andere Lösung ist, vor dem ersten Zugriff auf TCB-Adressen, welche Seitenfehler auslösen können, den passenden Eintrag des aktuellen Seitenverzeichnis mit den Werten aus dem Master-Seitenverzeichnis des Kerns zu aktualisieren. Dies erfordert eine zusätzliche Kopieroperation vor dem ersten Zugriff.

Diese Lösung ist jedoch nur bei bereits bestehenden Threads erfolgreich, da die dafür benötigten Seitentabellen im Master-Verzeichnis eingetragen sind. Bei nicht existierenden Threads existiert im Hauptverzeichnis

[1] MMU: Hardwareeinheit zum Umsetzen von virtuellen Adressen in physische Adressen

auch kein entsprechender Eintrag, es kommt weiterhin zu einem Seitenfehler. Die Routine zur Seitenfehlerbehandlung hinterlegt bei einem Lesezugriff diesen TCB mit der Nullseite. Bei einem Schreibzugriff wird eine neue Seite alloziert und an dieser Stelle eingeblendet.

Diese Lösung besitzt den Nachteil der recht teuren Aktualisierung des aktuellen Seitenverzeichnisses. Ferner treten weiterhin Seitenfehler bei dem Zugriff auf den TCBs von nicht existenten Threads auf, zum Beispiel wenn der Nutzer eine ungültige Thread-ID bei der IPC-Sendeoperation angibt. Die Lösung kann bei verschiedenen Listenoperationen Seitenfehler vermeiden, da diese Listen nur bestehende Threads enthalten.

Zusätzliche Unterbrechungspunkte

Um die mögliche erhebliche Verzögerungszeit von Seitenfehlern abzufangen, wird vor jedem Zugriff auf einen TCB, der einen Seitenfehler auslösen kann, ein zusätzlicher Unterbrechungspunkt eingefügt. Diese Lösung benötigt keinen zusätzlichen Kernspeicher, geht jedoch auf Kosten der Laufzeit.

Das Freigeben der Interrupts kann auch verzögert in der Seitenfehler-Routine erfolgen. Dieses Verfahren hat jedoch etwas schlechtere Verzögerungszeiten zur Folge, da sich die Kosten für das Auslösen des Seitenfehlers zu den Verzögerungszeiten addieren. Die Seitenfehler-Routine gibt dann die Interrupts sofort wieder frei und behandelt den Seitenfehler. Deshalb ist der erste Zugriff auf den TCB wie ein Unterbrechungspunkt anzusehen. Der IPC-Code muss danach prüfen, ob der IPC-Zustand des aktuellen Threads noch gültig ist.

Eine Ausnahme bilden Zugriffe auf existierende TCBs, die noch nicht im aktuellen Adressraum eingeblendet sind. In diesem Fall wird nur das aktuelle Seitenverzeichnis mit dem Werten aus dem Hauptverzeichnis aktualisiert. Die Interrupts bleiben weiterhin gesperrt. Dies ist notwendig, da viele Listenoperationen zum Modifizieren der Ready-Liste, *Present*-Liste und Senderliste die Interrupts zur Synchronisation explizit sperren.

Ich habe mich aus folgenden Gründen für diese Lösung entschieden:

- Es ist keine Aktualisierung des Seitenverzeichnisses vor dem Zugriff notwendig
- Es sind nur wenig zusätzliche Unterbrechungspunkte im IPC-Pfad notwendig
- Die Ermittlung der TCB-Adresse aus der Thread-ID ist weiterhin schnell möglich
- Es wird kein zusätzlicher Kernspeicher benötigt
- Die ausgereifte Implementierung der Kernspeicherverwaltung kann beibehalten werden

Etwas längere Verzögerungszeiten werden in Kauf genommen, da ihre Kosten akzeptabel sind. Dieses Problem besteht zum Teil auch im bisherigen IPC-Pfad, so dass die Verzögerungszeiten nur unwesentlich schlechter ausfallen dürften.

5.2 Synchronisation

Im bisherigen IPC-Pfad werden zur Synchronisation verschiedene Primitive genutzt.

Der passive IPC-Partner wird, bis auf seinen Threadzustand, durch ein Thread-Lock geschützt. Der TCB des aktiven Threads wird jedoch sperrfrei mittels CAS-Operationen synchronisiert. Dies erfordert auch die schon erwähnten zusätzlichen *Retry-loops*, um die Konsistenz sicherzustellen, da in der Zwischenzeit, durch einen anderen Thread, Änderungen erfolgen können.

Synchronisation des Threadzustandes

Der Threadzustand wurde bisher durch atomare Maschineninstruktionen, wie *compare-and-swap*, konsistent gehalten. Mit deaktivierten Interrupts ist dies auf Uni-Prozessorsystemen nicht mehr notwendig, hier kann der Threadzustand direkt verändert werden. Es entfallen die bisher notwendigen *Retry-loops* im IPC-Pfad. Es kann sogar bis zum nächsten Unterbrechungspunkt, das Setzen eines gültigen Threadzustandes verzögert werden.

Nur nach einem Unterbrechungspunkt muss geprüft werden, ob der aktuelle IPC-Zustand noch gültig ist. Neben den expliziten Unterbrechungspunkten, gibt es weitere Operationen, welche unterbrochen werden können. Dazu gehört auch der erste Zugriff auf den Empfänger-TCB, und die Freigabe eines Locks, wenn andere Threads auf dieses Lock warten.

Thread-Locks

Da der neue IPC-Pfad die Interrupts sperrt, stellt sich die Frage, ob Thread-Locks zur Synchronisation der Zugriffe auf TCBs weiterhin verwendet werden sollen.

Die Interrupts können nur kurz gesperrt werden, d.h. bei längeren Operationen müssen Unterbrechungspunkte eingefügt werden. Vor jedem Unterbrechungspunkt muss der Zustand des passiven IPC-Partners in einen konsistenten Zustand gebracht werden, und nach dem Unterbrechungspunkt muss geprüft werden, ob der Zustand des IPC-Partners noch gültig ist. Die Verwendung von Versionsnummern, welche bei jeder Modifizierung inkrementiert werden, kann in diesem Fall sehr hilfreich sein, da ein Vergleich ausreicht, der überprüft, ob der Zustand des IPC-Partners noch gültig ist.

Bei SMP wäre es möglich die Threads als CPU-lokale Datenstrukturen aufzufassen, wobei Änderungen nur durch diese CPU erfolgen dürfen. Threads sind also an eine CPU gebunden. IPC zwischen Threads auf verschiedenen CPUs werden mittels IPI[2] und Proxy-Threads erfolgen. Dadurch würde garantiert, dass Threads nur von der lokalen CPU modifiziert werden.

Die Abschaffung von Thread-Locks und die nachfolgende Anpassung der Synchronisationstrategie würden sich durch den ganzen Kern ziehen, jeder Systemaufruf, welcher bisher Thread-Locks verwendet, muss angepasst werden. Die Codekomplexität steigt, da nach jedem Unterbrechungspunkt geprüft werden muss, ob der Zustand des Partners noch gültig ist. Längere kritische Abschnitte müssen in kleinere Abschnitte aufgebrochen werden. Es ist nicht erkennbar, ob damit die Echtzeitfähigkeit und Effizienz weiterhin gewährleistet werden kann. Aus diesen Gründen werden die Thread-Locks weiter verwendet. Es werden jedoch einige Optimierungen verwendet, um die Geschwindigkeit zu erhöhen.

Bei Locks ist zu beachten, dass zur Vermeidung von Deadlocks bei dem Holen von mehreren Locks eine bestimmte Reihenfolge eingehalten werden muss. Die Adressräume werden bei *sys_task_new* synchronisiert, indem der erste Thread des Adressraumes gesperrt wird. In der V2-Spezifikation müssen beim Löschen eines Chiefs alle seine Untertasks gelöscht werden. Der Systemaufruf *sys_task_new*, welche die Tasks löscht, sperrt also rekursiv alle ersten Threads der betroffenen Adressräume, wenn er einen Chief löscht. Dadurch wird es sehr schwer im restlichen Kerncode gleichzeig mehr als ein Thread-Lock zu holen, da die Gefahr eines Deadlocks besteht. Problematisch ist dies im IPC-Pfad beim kombinierten Senden und Empfangen, wenn neben dem alten Empfänger auch der wartende Sender berücksichtigt werden muss. Um dies zu vermeiden, besitzt der IPC-Pfad zu jeder Zeit höchstens ein Thread-Lock. In der X.2-Schnittstelle ist das Löschen von Tasks ein nichtrekursiver Prozess, so dass dies dort kein Problem ist.

[2] IPI: Interprozessor-Interrupt, damit kann eine CPU einen Interrupt auf einer anderen CPU auslösen, um ihr eine Nachricht zuzustellen

Optimierungen

Zur Optimierung wird das Thread-Lock nur gegriffen, wenn der entsprechende TCB schon gesperrt ist, oder der aktuelle Codeabschnitt unterbrechbar ist. Es wird also das Holen von Locks solange verzögert, bis Interrupts wieder zugelassen werden.

Diese Optimierung erlaubt es, eine Short-IPC durchzuführen, ohne ein Thread-Lock zu greifen, falls der betroffene IPC-Partner nicht gesperrt ist. Erst bei komplexeren Nachrichten, welche Unterbrechungspunkte erfordern, wird der IPC-Partner gesperrt.

Desweiteren wird der der Umstand ausgenutzt, dass der TCB nur durch das Thread-Lock geschützt wird, der Threadzustand und die verschiedenen Listen werden getrennt synchronisiert. Deshalb kann man den Threadzustand und die Warteschlangen auch ohne Halten des Thread-Locks modifizieren. Es muss nur sichergestellt werden, dass der Zustand des Threads noch gültig ist.

5.3 Senderwarteschlange mit Prioritäten

Die bisherige Senderwarteschlange garantiert keine Prioritäten, dies ist ein Problem, wenn Threads mit unterschiedlichen Prioritäten, Anfragen an den selben Server stellen.

Die Implementierung ist eine unsortierte, doppelt verkettete Liste. Neue Sender ketten sich stets am Ende dieser Liste ein, und Empfänger betrachtet immer nur den Kopf dieser Liste. Dadurch wird eine einfache FIFO[3]-Strategie implementiert. Da die Prioritäten nicht berücksichtigt werden, besteht das Problem der Prioritätsinversion. Um dies zu vermeiden, werden im folgenden verschiedene Ansätze vorgestellt, Prioritäten bei der Senderwarteschlange durchzusetzen.

Die Elemente der Senderwarteschlange sind Threads, Interrupts, Preemptions und Activations, welche die Senderschnittstelle implementieren. Das Ein- und Ausketten ist sehr schnell und in konstanter Zeit möglich. Das Ausketten von Sendern ist in jeder Position schnell durchführbar. Dies ist wichtig, wenn eine Sendeoperation durch einen Timeout abgebrochen wird.

Sortieren der Liste nach Prioritäten

Die Liste wird einfach nach Prioritäten sortiert. Sie wird solange beim Einketten durchlaufen, bis die richtige Position gefunden ist. Das Einketten erfolgt mit linearen Aufwand, das Ausketten ist weiterhin mit konstanten Aufwand möglich.

Als Optimierung kann man Sender von gleichen Prioritäten zu Blöcken zusammenfassen, so dass beim Durchlaufen der Liste nur einzelne Blöcke betrachtet werden müssen. Wenn der passende Block gefunden ist, wird der Sender am Ende dieses Blocks eingekettet. Dadurch reduziert sich der Aufwand für das Einketten im schlechtesten Fall auf die maximale Anzahl von Prioritäten im System. Der Sender mit der höchsten Priorität befindet sich dann am Kopf dieser Liste.

Da das Einketten Durchlaufen sehr zeitaufwendig ist, kann es nicht mit deaktivierten Unterbrechungen erfolgen. Dadurch wird jedoch eine Synchronisation notwendig, welche über ein Lock erfolgt, da die IA-32-Architektur über kein MWCAS[4] verfügt.

[3] First-In, First-Out

[4] MWCAS: Multi Word Compare nd Swap, Eine Maschineninstruktion, welche atomar mehrere Worte vergleichen und vertauschen kann

Eine bessere Lösung sind so genannte *Skip Lists* [Pug89]. Dies sind mehrfach verkettete und sortierte Listen siehe Abb. 5.3, welche im Durchschnitt einen logarithmischen Aufwand zum Einketten eines Elements anbieten. Sie sind oftmals schneller als ausgeglichene sortierte Bäume, da keine Rotationen durchgeführt werden müssen. Im schlechtesten Fall ist jedoch weiterhin ein linearer Aufwand notwendig.

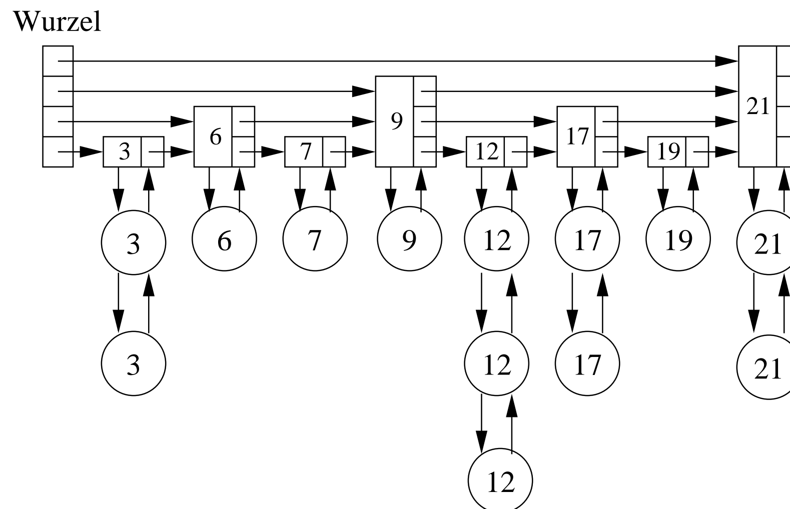

Abbildung 5.3: List als Senderwarteschlange.

Sortierter Binärbaum

Die Implementation der Senderwarteschlange erfolgt mit einem sortierten Binärbaum. Ein unausgeglichener Binärbaum ist im schlechtesten Fall nicht besser als eine sortierte Liste. Er bringt nur Vorteile im Durchschnitt, wo das Einketten mit logarithmischem Aufwand erfolgt. Im schlechtesten Fall degeneriert der Baum zu einer Liste und, das Einketten erfordert einen linearen Aufwand.

Bei einem ausgeglichenen Baum wird der Baum bei jedem Ein- und Ausketten ausbalanciert. Sie garantieren auch im schlechtesten Fall logarithmischen Aufwand zum Ein- und Ausketten. Das Ein- und Ausketten, kann trotzdem sehr aufwendig werden, wenn teure Rotationen zum Ausgleich notwendig sind. Das Bestimmen des maximalen Elements ist weiterhin in konstanter Zeit möglich.

Es ist möglich, Sender mit gleicher Priorität als ein Baumelement zu betrachten. Es entsteht ein Baum, dessen Elemente wieder doppelt verkettete Listen von Sendern gleicher Priorität sind. Dadurch verkürzt sich die Zeit zum Einketten von $O(ln(N))$ auf maximal $O(ln(256))$.

Beim Einketten von mehreren Sendern gleicher Priorität, wird zuerst die entsprechende Liste gesucht und der neue Sender am Ende dieser Liste eingekettet. Dadurch wird auch die Fairness bezüglich Sendern gleicher Priorität garantiert.

Tries

Tries sind sortierten Binärbäumen ähnlich. Sie basieren jedoch nicht auf dem Vergleich des Schlüssels. Stattdessen machen sie sich die digitale Natur des Schlüssels, hier die Priorität, zu Nutze. Sie betrachten den Schlüssel als Beschreibung, wie der Baum zu durchlaufen ist, um die passende entsprechende Stelle zu finden. Ein Beispiel für Tries sind mehrstufige Seitentabellen.

Tries garantieren Einfügen und Ausketten mit logarithmischem Aufwand Auch hier kann die Optimierung, die Zusammenfassung von Sendern gleicher Prioritäten, eingesetzt werden, siehe Abb. 5.4. Es wird dann

die Fairness von Sendern gleicher Priorität garantiert. Die maximale Tiefe von Tries hängt stark von der verwendeten Struktur ab. Die Tiefe eines binären Tries ist identisch zur Anzahl der Bits in dem verwendeten Schlüssel. Bei 256 Prioritäten sind dies maximal acht Ebenen ($ld(256)$).

Im Gegensatz zu ausgeglichenen Bäumen sind keine aufwendigen Operationen beim Ein- und Ausketten notwendig, jedoch sind die Tries im Durchschnitt weniger gut ausgeglichen.

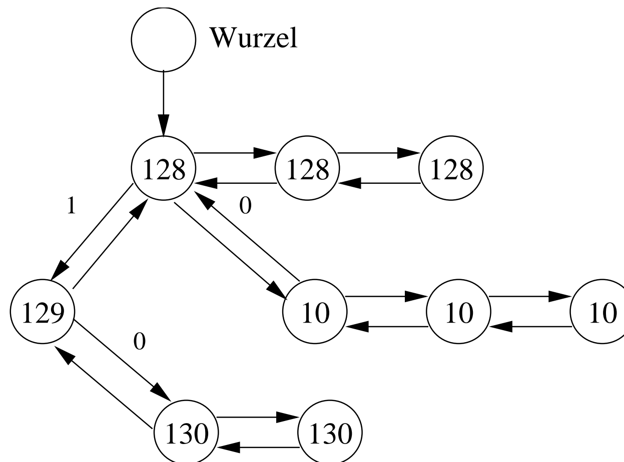

Abbildung 5.4: Senderwarteschlange-Trie.

Heaps

Der Heap ist als die Prioritätswarteschlange bekannt und entspricht topologisch auch einem Baum.

Sie besitzen die Eigenschaft, dass der Schlüssel des Elternelements stets größer ist, als der Schlüssel der Kinder. Dies gilt rekursiv für alle Unterbäume. Das Ein- Ausketten von Elementen in Heaps ist viel effizienter als bei ausbalancierten Bäumen, da keine teuren Ausgleichoperationen notwendig sind.

Es ist nicht möglich, anstelle von einzelnen Sendern, Blöcke von Sendern mit der gleichen Priorität zu betrachten. Es muss stets jeder Sender einzeln eingekettet und ausgekettet werden, da der Heap kein effizientes Suchen von Elementen anbietet. Es ist somit unmöglich, die passende Liste der benötigten Priorität schnell zu finden. Ferner muss die Fairness von Sendern gleicher Priorität durch einen anderen Ansatz sichergestellt werden.

Die meisten Implementierungen nutzen Felder für die einzelnen Elemente. Dadurch ist der Heap immer ausgeglichen und es kann die logarithmische Komplexität garantiert werden. Eine Implementierung durch ein Feld ist aber im diesem Fall nicht möglich.

Eine Möglichkeit zur Implementierung ist es, die Knoten des Baums mit Zählern zu versehen, um ein ausgeglichenes Ein- und Ausketten zu erreichen. Die Fairness kann dadurch garantiert werden, dass beim Einketten eine aufsteigende Nummer vergeben wird. Der Vergleich bezieht dann die Priorität und diese Nummer mit ein.

Da Heaps nur $O(ln(N))$ und nicht $O(ln(Prio))$ anbieten, können die Zeiten im schlechtesten Fall signifikant steigen, so dass der ausgeglichene Baum oder der Trie vorzuziehen ist.

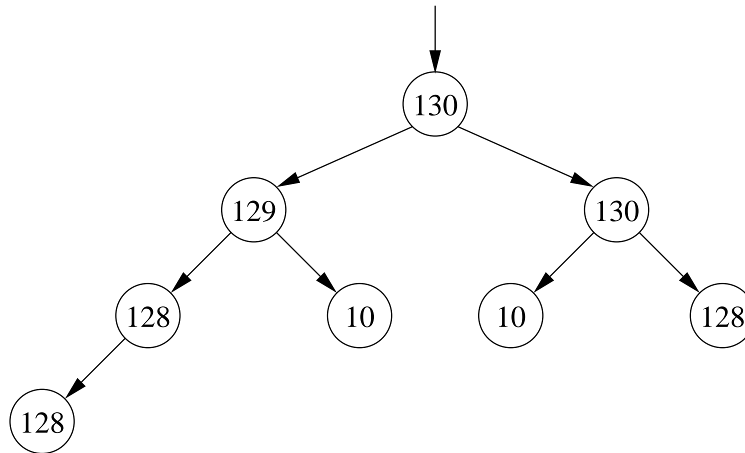

Abbildung 5.5: Senderqueue-Heap.

Zusammenfassung

Eine ideale Senderwarteschlange sollte nach Prioritäten sortiert sein und es erlauben die Elemente mit geringem und konstantem Aufwand einzufügen und zu entfernen. Die einfache Liste garantiert $O(1)$ Aufwand beim Ein- und Ausketten, aber sie setzt keine Prioritäten durch.

Wenn Prioritäten gewünscht sind, kommen nur der ausgeglichene Baum und der Trie in Frage. Um den Aufwand auch im ungünstigen Fall möglichst gering zu halten, werden alle Sender gleicher Priorität zusammengefasst. Im schlechtesten Fall, bei 256 Prioritäten, garantiert der binäre Trie acht und ein Rot-Schwarz Baum 16 Iterationen, um ein Element einzufügen.

Ferner ist die Synchronisation ein limitierender Faktor. Bisher wurden die Interrupts gesperrt, um die Zugriffe auf die Warteschlange zu synchronisieren. Es ist für einige Codeteile unmöglich Locks zu holen. Dies betrifft besonders den Interruptcode zur Zustellung von Interrupt-IPCs.

Im Rahmen dieser Diplomarbeit wurde neben der bereits vorhandenen Lösung, auch eine Warteschlange implementiert, welche die Prioritäten garantiert. Die verwendete Datenstruktur ist ein binärer Trie, dessen Elemente doppelt verkettete Listen sind. Diese Listen enthalten alle Sender gleicher Priorität.

Konsistenz der Senderwarteschlange

Eine inkonsistente Senderwarteschlange enthält auch Sender, welche keine IPC mehr durchführen. Bei einer konsistenten Warteschlange ist garantiert, dass sie nur Sender enthält, welche die IPC noch durchführen können.

Bei Threads als Sendern können Inkonsistenzen entstehen, wenn der IPC-Timeout des Threads überschritten wird, oder die IPC durch ein *thread_ex_regs* abgebrochen wird. Die Konsistenz der Warteschlange kann bei *Thread::kill* und beim Abbrechen der IPC durch *thread_ex_regs* sichergestellt werden, indem der betroffene Sender aus der Warteschlange ausgekettet wird.

Bei einem Timeout ist dies auch möglich, hier ist aber folgendes zu beachten. Die Behandlung von Timeouts erfolgt mit deaktivierten Interrupts in der Zeitgeber-Routine. Sie durchläuft die Liste von Timeouts und kettet die betroffenen Timeout-Objekte aus. Um die Verzögerungszeiten gering zu halten, ist das Ausketten von Sendern aus der Wartschlange nur möglich, wenn diese Operation sehr schnell ist. Dies wird nur durch die doppelt verkettete Liste erreicht.

Bei Activations können bestimmte Ereignisse neutralisieren, so dass auch hier keine IPC mehr durchgeführt werden soll. Ein ähnliches Problem sind Unterbrechungspunkte zwischen dem Ausketten des Senders und dem Start des Rendezvous. Dies können implizite Unterbrechungspunkte, wie das Holen eines Thread-Locks, oder explizit gesetzte, z.B. vor einem Kontextwechsel sein, um die Verzögerungszeiten klein zu halten. Dies ist besonders bei Preemption-IPC ein Problem, da mehrere Locks in ihrer *ipc_receiver_ready* Methode gegriffen werden.

Eine Möglichkeit ist es, die Empfangsoperation mit einem Fehler abzubrechen, wenn der ausgekettete Sender die IPC nicht mehr durchführen kann. Eine andere Lösung ist, solange durch die Warteschlange zu iterieren, bis ein gültiger Sender gefunden wird, oder es keinen Sender mehr gibt. Bei einer Iteration über die Warteschlange müssen dann Unterbrechungspunkte gesetzt werden, damit die Verzögerungszeiten gering bleiben.

5.4 Aufbau IPC-Pfades

IPC Zustände

Sender und Empfänger durchlaufen während der IPC verschiedene Zustände, welche sich in vier grundlegende Teile, Setup, Rendezvous, Datentransfer und Abschluss, umfassen. Die einzelnen Zustände werden durch verschiedene Bitmuster, siehe Tabelle 5.1, im Zustandsword dargestellt. Der Sender übernimmt während der IPC die aktive Rolle, der Empfänger wird nur aktiv, wenn bei einer Long-IPC-Operation Seitenfehler im IPC-Fenster auftreten. Ein Zustandsdiagramm für die einzelnen IPC-Zustände ist in Abb. 5.6 zu sehen.

Zustand	ready	ipc	recv	transfer	send	poll	busy long	poll long	rcv long	cancel
Senderzustände										
send prepared	+	+	-	-	+	-	-	-	-	?
sleep prepared	+	+	-	-	+	+	-	-	-	?
sleep	-	+	-	-	+	+	-	-	-	?
data transfer	+	+	-	+	+	-	-	-	-	?
page fault in IPC window	+	+	-	+	+	-	-	+	-	?
page-in wait	-	+	-	+	+	-	-	+	-	?
ipc finished	+	-	-	-	-	-	-	-	-	?
Empfängerzustände										
receiving	+	+	+	-	-	-	-	-	-	?
try handshake	+	+	+	-	-	-	-	-	-	?
waiting	-	+	+	-	-	-	-	-	-	?
receiving data	-	+	+	+	-	-	-	-	-	?
in long IPC	-	+	-	+	-	-	-	-	+	?
page in	+	+	-	+	-	-	+	-	+	?
receiving end	+	-	-	-	-	-	-	-	-	?
Fehlerzustände										
timeout	+	-	?	?	?	?	?	?	?	?
cancel	+	-	?	?	?	?	?	?	?	+
kill	-	-	-	-	-	-	-	-	-	-

+ = Flag gesetzt, - = Flag gelöscht, ? = Flag kann gesetzt oder gelöscht sein

Tabelle 5.1: Sende- und Empfangszustände

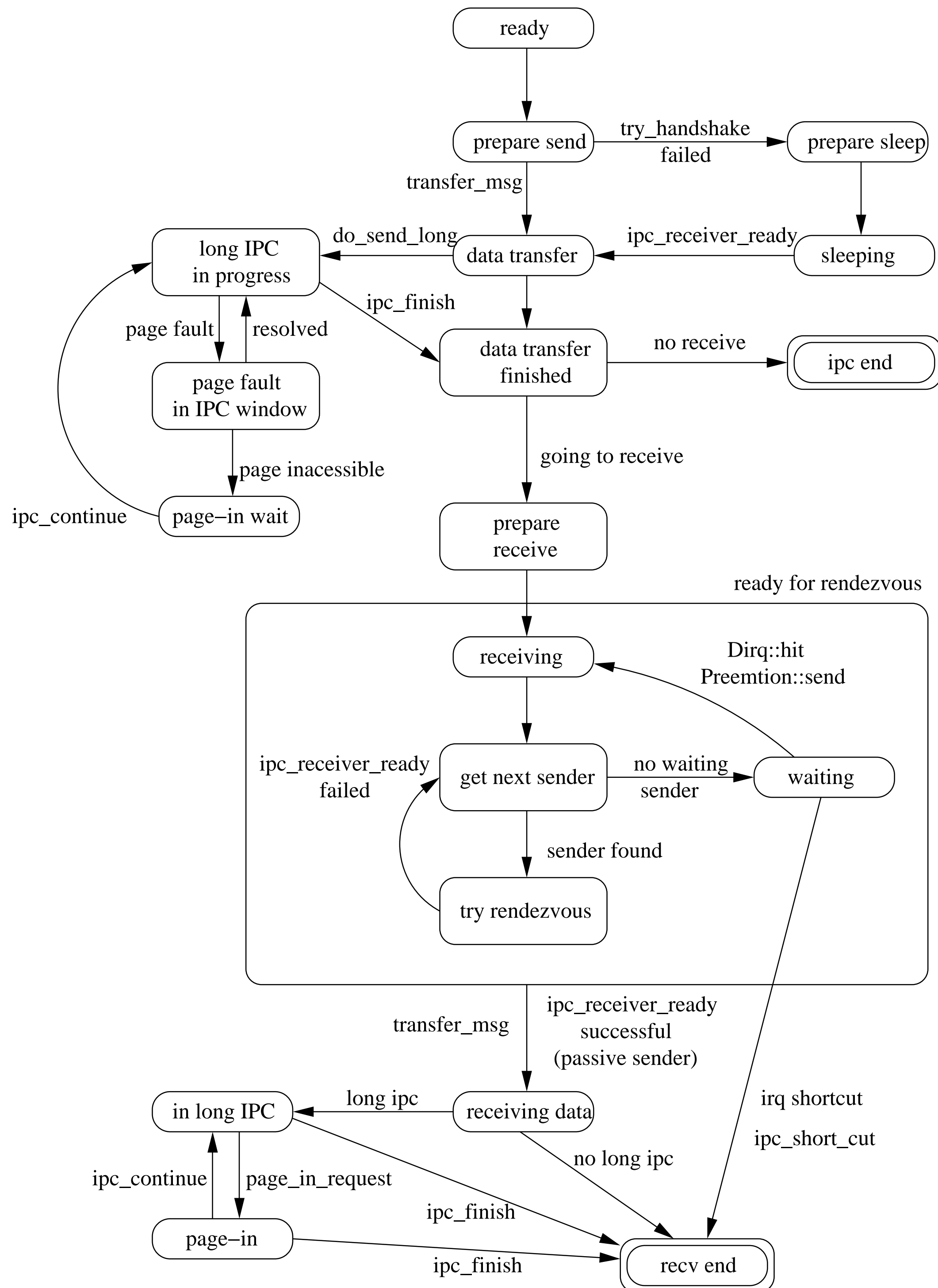

Abbildung 5.6: IPC-Zustandsdiagramm.

Setup

Bei einem Kerneintritt im V2-Kern wird zuerst geprüft, ob es sich um eine normale IPC, oder um eine Interrupt-Operation handelt. Bei einer normalen IPC erfolgt die Umrechnung der Thread-ID zu TCB-Adressen, welche dann der generischen IPC-Funktion, *do_ipc*, übergeben werden.

Weiterhin umfasst das Setup das Setzen des Sende- und Empfangszustandes. Bei einer kombinierten Sende- und Empfangsoperation, erfolgt das Setzen des Empfangszustandes vor dem Abschluss der Sendeoperation, um das atomare Umschalten vom Senden zum Empfangen zu garantieren. Im bisherigen IPC-Pfad wird dies schon vor dem Beginn der Sendeoperation durch *prepare_receive* gemacht, im neuen erfolgt dies erst am Ende der Sendeoperation, um mögliche Fehler wie Prioritätsinversion zu vermeiden. Dafür wurden die früheren *do_send* und *do_receive* Funktionen zu einer einzigen Funktion, *do_ipc* zusammen gefasst.

Nachdem das Setup erfolgt ist, kann das Rendezvous durch den neuen Sender erfolgen.

Nach dem Kerneintritt und vor dem Aufruf der IPC-Operation, wird ein Unterbrechungspunkt gesetzt. Dadurch wird bei einem Kerneintritt mittels INT-Instruktion die Verzögerungszeit nicht zu groß.

Rendezvous

Wenn der Empfänger schon von einen anderen Thread gesperrt ist, versucht der IPC-Pfad sich das Lock zu holen, sonst ist Sperren des Empfängers-TCBs nicht notwendig. Der Sender prüft nach, ob der Empfänger empfangsbereit ist. Ist dies nicht der Fall, bricht er das Rendezvous ab und kettet sich erst jetzt in die Senderwarteschlange des Empfängers ein. Er setzt, wenn nötig, einen Timeout, aktualisiert seinen Zustand und blockiert, bis er vom Empfänger durch *ipc_receiver_ready* geweckt wird. Im Gegensatz zum bisherigen IPC-Pfad beginnt *ipc_receiver_ready* das Rendezvous, so dass der aufgeweckte Sender sofort die Nachricht übertragen kann. Desweiteren kann er auch durch den Timeout und *thread_ex_regs* wieder aktiviert werden, welche die IPC-Operation abbrechen können. Der Sender muss diese Bedingungen prüfen, wenn er nach dem Blockieren wieder aktiv wird.

Der Empfänger versucht nach dem Setup einen wartende Sender mit *ipc_receiver_ready* aufzuwecken. Wenn dieser Sender die IPC nicht mehr durchführen kann, wird er ausgekettet, und der Empfänger versucht das Rendezvous nach einem Unterbrechungspunkt mit dem nächsten Sender.

Wenn kein Sender wartet, setzt der Empfänger einen optionalen Timeout, löscht sein Ready-Flag und blockiert solange, bis ein neuer Sender das Rendezvous versucht. Der Empfänger kann auch durch den Ablauf eines IPC-Timeouts oder durch *thread_ex_regs* wieder aktiviert werden.

Datentransfer

Der Datentransfer gliedert sich in zwei Teile: Registerwertetransfer und Long-IPC. Der Registerwertetransfer kann neben dem Kopieren der Registerinhalte auch das Einblenden von Flexpages in den Adressraum des Empfängers umfassen. Das Kopieren der Registerinhalte passiert mit gesperrten Interrupts, während die Behandlung von Flexpages mit aktivierten Interrupts erfolgt, wobei der Sender vor dem Freigeben der Interrupts mit einem Thread-Lock gesperrt wird.

Long-IPC umfasst das Kopieren von Speicherinhalten und das Einblenden von Flexpages. Da das Kopieren von Speicherinhalten aus dem Nutzeradressraum blockieren kann, muss dies ohne Lock-Schutz erfolgen. Nur bei der Behandlung von Flexpages und Abschluss der Long-IPC wird der Empfänger gesperrt.

Der Empfänger ist während dieser Zeit inaktiv. Wenn jedoch Seitenfehler im IPC-Fenster vom Adressraum des Senders auftreten, weckt die Seitenfehler-Routine den Empfänger auf und setzt den Sender in den *page-in*

wait-Zustand. Der nun aktive Empfänger setzt eine IPC zur Seitenfehlerbehandlung auf. Nach der Antwort weckt der Empfänger den Sender mit *ipc_continue* auf, so dass dieser die IPC fortführen kann.

Bei passiven Sendern wird der IPC-Transfer vom Empfänger durchgeführt. Die passiven Sender überschreiben *ipc_receiver_ready*, so dass diese Funktion die Nachricht in den Registerpuffers des Empfängers schreibt. Wenn der passive Sender nicht mehr an einer IPC interessiert ist, liefert *ipc_receiver_ready* falsch zurück. Der Empfänger versucht dann das Rendezvous mit dem nächsten Sender.

Abschluss

Der Abschluss der IPC wird durch ein gelöschtes *Thread_ipc_in_progress*-Flag gekennzeichnet. Die IPC kann durch einen Timeout, Abbruch der IPC mittels *thread_ex_regs* und durch *Thread::kill* beendet werden. Die Empfangsoperation wird am Ende der IPC-Operation auch durch den Sender abgeschlossen.

Wenn einer Sendeoperation keine Empfangsoperation mehr folgt, werden alle IPC-Flags inklusive des *Thread_-ipc_in_progress* gelöscht. Falls es ein kombiniertes Senden und Empfangen ist, wird nach dem Abschluss des Sendens in den Empfangszustand übergegangen, das *Thread_ipc_in_progress*-Flag bleibt dabei gesetzt.

Der Empfängerm kehrt bei einer erfolgreichen IPC sofort zurück. Erst bei einer nicht erfolgreichen IPC erfolgt die aufwändige Auswertung der Fehlerbedingungen. Wenn dies der Fall ist, wird vor der Rückkehr zum Nutzer ein Fehlercode gesetzt.

Asynchrone Ereignisse

Bedingt durch Unterbrechungspunkte können während der IPC asynchrone Ereignisse die IPC-Operation beeinflussen. Aus diesem Grund muss nach einem Unterbrechungspunkt geprüft werden, ob der IPC-Zustand noch gültig ist. Folgende Ereignisse können eintreten, Timeout, Abbruch der IPC durch *thread_ex_regs*, Löschen des Threads und Zustandsänderungen durch passive Sender wie Interrupts, Preemptions und Activations.

Timeouts

Wenn ein Timeout eintritt, löscht die *expire*-Funktion das *Thread_ipc_in_progress*-Flag, setzt das *Thread_-ready*-Bit und kettet ihn in die Ready-Liste ein. Wenn dieser Thread eine größere Priorität als der aktuelle Thread besitzt, wird im Anschluss der Scheduler aktiv. Der betroffene Thread kann dann an dem gelöschten *Thread_ipc_in_progress*-Flag einen abgelaufenden Timeout erkennen. Er kehrt mit einem Fehlercode zum Nutzer zurück.

IPC-Abbruch

Durch *thread_ex_regs* kann die IPC an jedem beliebigen Unterbrechungspunkt abgebrochen werden. Da *thread_ex_regs* sich das Thread-Lock greift, kann ein passiver IPC-Partner nicht abgebrochen werden, wenn er vom IPC-Code gesperrt ist. Zum Abbruch der IPC löscht *thread_ex_regs* das *Thread_ipc_in_progress*-Flag, setzt das *Thread_cancel*-Bit, sowie das *Thread_ready*-Bit und kettet den Thread erneut in die Ready-Liste ein. Es wird nur ein Thread durch *thread_ex_regs* abgebrochen. Bei einer begonnenen IPC muss dieser Thread auch seinen IPC-Partner abbrechen. Sender und Empfänger kehren in diesem Fall mit einem Fehlercode zum Nutzer zurück.

Long-IPC verwendet einen Trick, um nicht immer auf Abbruch der IPC zu prüfen. Bei dem Kontextwechsel wird das IPC-Fenster gelöscht, und da die IPC nur durch einen anderen Thread abgebrochen werden kann, ist

dieser Test nur am Anfang der Long-IPC und in der Seitenfehler-Routine notwendig. Sobald zu einem Thread umgeschaltet wird, welcher die gerade suspendierte Long-IPC-Operation abbricht, wird der IPC-Zustand ungültig. Wenn der Sender die IPC-Operation fortsetzt, wird durch die Kopieroperation ein Seitenfehler ausgelöst. Deshalb reicht es aus, dass der Test auf Abbruch der IPC in in der Seitenfehler-Routine erfolgt.

Thread::Kill

Wenn ein Thread zerstört wird, wird sein kompletter Zustand gelöscht, und er wird aus der Bereitwarteschlange entfernt. Sobald der zu löschende Thread alle Locks freigeben hat, stoppt er die Ausführung. *Thread::kill* muss daher alle anderen Ressourcen freigeben, gesetzte Timeouts löschen und den Thread aus einer Senderwarteschlange ausketten. Es muss der IPC-Partner einer begonnenen IPC-Operation, erkennbar am gesetzten *Thread_transfer*-Flag, abgebrochen werden.

Passive Sender

Passive Sender werden in die Senderwarteschlange des Empfängers eingekettet und der Empfänger wird wieder aufgeweckt. Der Empfänger muss dann erneut das Rendezvous mit dem neuen Sender versuchen. Auch nach Unterbrechungspunkten muss der Empfänger beachten, dass in der Senderwarteschlange neue Sender enthalten sein können.

5.5 Probleme des bisherigen IPC-Pfades

Prioritätsinversion

Die L4-Spezifikation verlangt bei einer kombiniertem Sende- und Empfangsoperation, dass der Übergang vom Senden zum Empfangen atomar erfolgen muss. Es ist auch der Fall, dass sich ein Sender in der Warteschlange befindet, zu beachten.

Wenn dieser eingekettete Sender die höchste Priorität besitzt, muss er anstelle des alten Empfängers aktiv werden, um eine Prioritätsinversion zu vermeiden. In einem nicht unterbrechbaren IPC-Pfad ist dies problemlos durchsetzbar, der bisherige IPC-Pfad zeigte hier Fehler.

In dem folgenden Szenario, siehe 5.7, hat *C* die höchste Priorität, *B* eine mittlere Priorität und *A* die geringste Priorität. *A* bearbeitet einen Auftrag von *B*, und *B* wartet auf eine Antwort von *A*. *C* wird aktiv und möchte eine IPC zu *A* durchführen. Da *A* nicht empfangsbereit ist, blockiert *C* solange, bis er von *A* aufgeweckt wird. *A* beendet den Auftrag, schickt diesen mit einer *reply_wait*-Operation zurück an B und geht gleichzeitig ins offene Warten über, damit er neue Aufträge entgegen nehmen kann.

Im bisherigen IPC-Pfad, sperrt A bei seiner Antwort den Empfänger *B*, kopiert die Nachricht zu *B*, setzt sich empfangsbereit und *B* auf lauffähig, und gibt das Lock von *B* frei. Durch die Freigabe des Locks von *B*, wird sofort zu *B* umgeschaltet, weil *B* eine höhere Priorität als *A* besitzt.

Wenn nun *B* in eine Endlosschleife eintritt, wird *A* nie wieder aktiv. *A* kann damit auch *C* nicht aufwecken. Obwohl *C* eine höhere Priorität als *B* besitzt und *C* nun die IPC zu *A* beginnen könnte, wird *C* niemals vom Scheduler ausgewählt.

Bei dem kombinierten Senden und Empfangen müssen auch wartende Sender berücksichtigt werden, um solche Fehler zu vermeiden. Ein einfaches atomares Setzen des IPC-Zustandes auf Empfangsbereit reicht nicht aus.

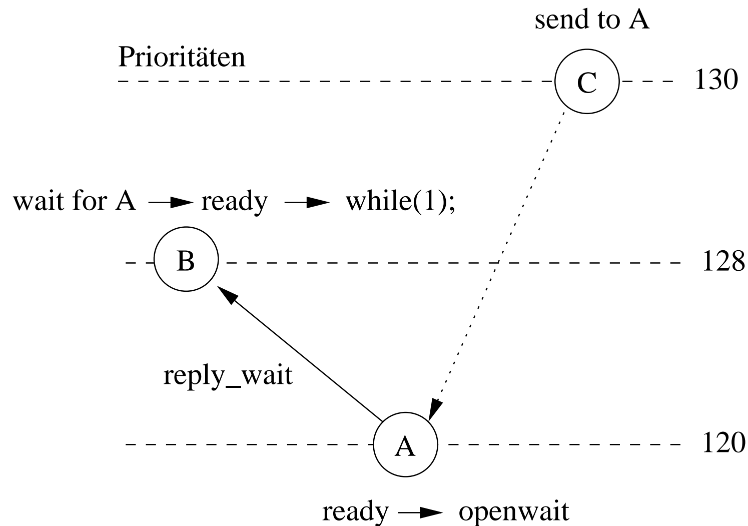

Abbildung 5.7: Prioritätsinversion.

Die neue Implementation umgeht dies, indem nach Abschluss der Sendeoperation nicht sofort zu *B* umschaltet wird. Es wird vorher die Priorität von *B* mit dem eingeketteten Sender *C* verglichen. Wenn der einkettete Sender eine höhere Priorität besitzt, wird zu ihm umgeschaltet, der alte Empfänger *B* wird nur in die Ready-Liste eingekettet.

Empfangstimeouts

Die L4-Spezifikation sagt nichts darüber aus, ob der Empfangstimeout sofort gültig sein muss, oder erst wenn der Empfänger blockiert. Der bisherige IPC-Pfad nutzt dies aus, und setzt den Empfangstimeout bei dem kombinierten Senden und Empfangen verzögert. Dadurch trifft im günstigen Fall die Antwort der IPC vor dem Ablauf der aktuellen Zeitscheibe und vor dem Setzen des Timeouts ein, welches dadurch komplett eingespart werden kann. Der neue IPC-Pfad nutzt die gleiche Optimierung, um so dass Setzen von IPC-Empfangstimeouts gegebenenfalls einzusparen. Es ist jedoch zu beachten, dass dies Probleme mit sehr langen Zeitscheiben verursachen kann, hier kann das Timeoutende noch innerhalb der aktuellen Zeitscheibe liegen. Wenn solche Szenarios wichtig sind, muss der Timeout vorher gesetzt werden.

Weiter können im bisherigen IPC-Code Timeout-Fehler bei Empfangstimeouts eintreten, obwohl es wartende Sender gibt. Das folgende Szenario, Abb. 5.8, verdeutlicht dies. Thread *A* hat hier eine höhere Priorität als Thread *B*. Thread *B* wartet darauf, eine IPC an *A* zu schicken, und ist dafür in der Senderwarteschlange von *A* eingekettet. Thread *A* ist wieder aktiv und führt nach einer bestimmten Zeit eine Empfangsoperation mit Timeout Null durch.

A tritt in den Kern ein und setzt seinen IPC-Zustand auf Empfang. *A* findet *B* in seiner Sendewarteschlange und schaltet zu *B* mittels *ipc_receiver_ready* um, um das Rendezvous zu beginnen. *A* bleibt jedoch noch lauffähig. Es wird nur der Ausführungskontext umgeschaltet, der Scheduling-Kontext nicht. *B* wird aktiv, und will das Rendezvous beginnen. Bevor *B* jedoch *A* durch ein Thread-Locks in *ipc_send_regs* sperren kann, läuft die aktuelle Zeitscheibe von *A* ab.

Der Scheduler wird wieder aktiv und wählt *A* aufgrund seiner höheren Priorität erneut aus. Dies ist möglich, da *A* zu diesem Zeitpunkt noch lauffähig ist. *A* untersucht erst jetzt den Empfangstimeout, und stellt fest, der Timeout ist Null. Er kehrt dann zum Nutzer mit einem Timeout-Fehler zurück, obwohl *B* ihm eine Nachricht senden kann.

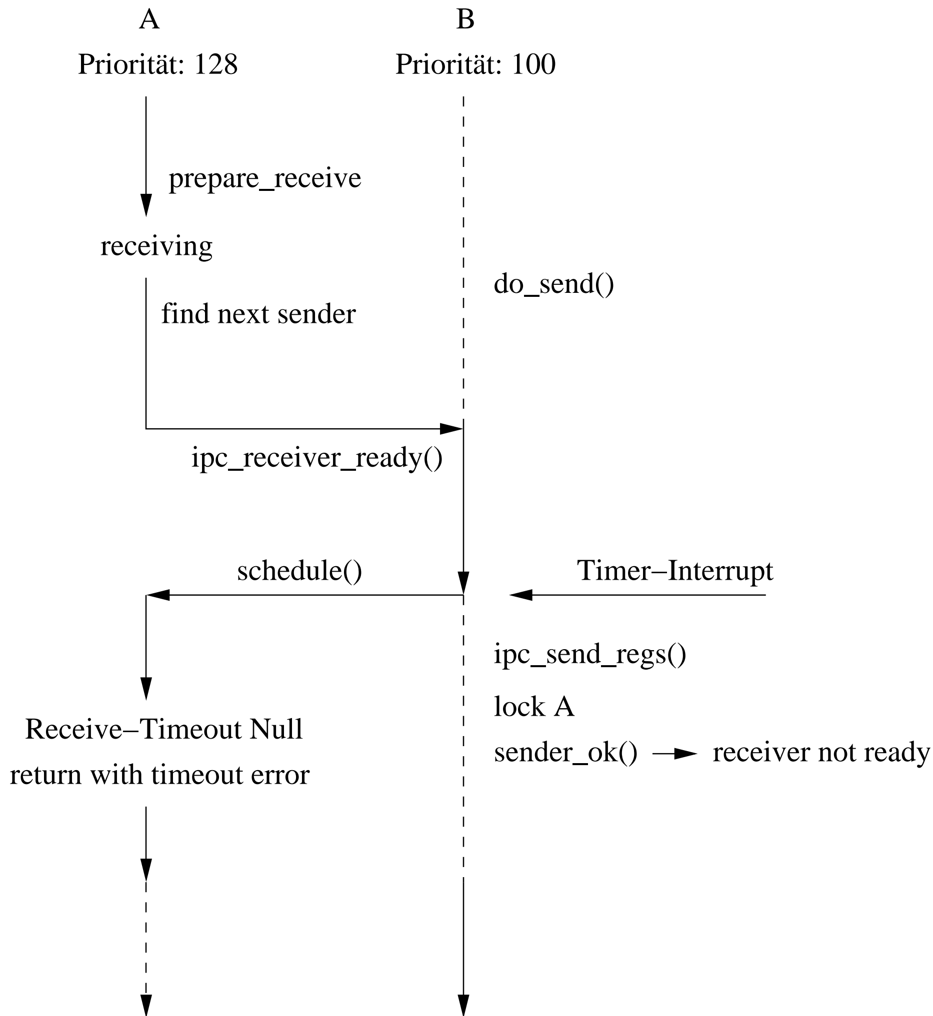

Abbildung 5.8: Timeout-Fehler.

Um diese Fehler zu vermeiden, wird das Rendezvous schon durch den Empfänger in *ipc_receiver_ready* begonnen. Ein neues Flag im Zustandsword zeigt an, ob das Rendezvous schon vollzogen wurde. Ein Setzen des Timeouts ist nicht mehr notwendig, weil die IPC erfolgreich begonnen wurde.

Durch dieses Flag können auch Fehler wie *Cancel* und *Abort* genauer unterschieden werden. Weiterhin erkennt der IPC-Partner durch dieses Flag eine begonnene IPC, und kann dann auch den anderen IPC-Partner abbrechen, wenn seine IPC in der Zwischenzeit mit *thread_ex_regs* abgebrochen wurde.

6 Implementierung

In diesem Kapitel wird kurz auf verschiedene Aspekte der Implementierung eingegangen. Weiterhin werden einige Optimierungen erläutert, um die Geschwindigkeit des IPC-Pfades zu maximieren.

6.1 Timeouts

Fiasco bildet jedes zeitliches Ereignis auf einen Timeout ab, die Klasse *IPC_timeouts* implementiert Zeitgrenzen in IPC-Operationen, *Timeslice_timeouts* stößt nach Ablauf der Zeitscheibe den Scheduler an, und *Deadline_timeouts* implementiert Perioden und Deadlines für Echtzeit-Threads. Jedesmal, wenn durch den Zeitgeber die entsprechende Routine zur Behandlung von Timeouts angestoßen wird, durchläuft diese eine Liste der Timeout-Objekte, und ruft für alle Objekte, deren Zeitindex kleiner als der aktuelle ist, ihre virtuelle *expire* Methode auf.

- **IPC_timeout:** Die Klasse *IPC_timeout* dient dazu, erfolglose IPC-Operationen, nach einer vom Nutzer bestimmbaren Zeit abzubrechen. Bisher wurden IPC-Timeout-Objekte auf dem Kernstack des jeweiligen Threads angelegt. Dadurch wird auch jedesmal ihr Konstruktor und Destruktor aufgerufen. Obwohl der Compiler sehr gut optimiert, konnte eine Zeiteinsparung dadurch erreicht werden, dass IPC-Timeouts direkt im TCB aggregiert werden. Die virtuelle *expire* Methode des IPC-Timeouts setzt den Thread lauffähig, löscht die IPC-Flags und kettet ihn in die Ready-Liste ein.
- **Timeslice_timeout:** Der *Timeslice_timeout* dient dazu die verschiedensten Scheduling-Kontexte eines Threads umzuschalten. Beim Ablauf eines Echtzeit-Scheduling-Kontextes, wird dem Preempter dieses Threads eine Preemption-IPC zugestellt, und es erfolgt die Umschaltung des Scheduling-Kontextes. Ferner dient dieser Timeout dazu, periodisch den Aufruf der *schedule* Funktion zu triggern, die einen neuen Thread aus der Menge der rechenbereiten Threads auswählt. Es befindet sich dadurch immer ein *Timeslice_timeout*-Objekt in der Timeout-Liste.
- **Deadline_timeouts:** Mit dem *Deadline_timeout* können sich Echtzeithreads mit ihrer Periode synchronisieren. Sie blockieren solange durch eine spezielle IPC, bis sie am Anfang ihrer Periode durch den *Deadline_timeout* wieder aufgeweckt werden. Mit dem gleichen Mechanismus werden auch Überschreitungen der Deadline erkannt. Dies ist der Fall, wenn der betroffene Thread nach dem Ablauf eines Deadline-Timeouts nicht auf seine nächste Periode wartet. In diesem Fall wird sein Preempter mit einer Deadline-Miss-IPC informiert.

Timeout-Liste

Im bisherigen Kern wird die Timeout-Liste durch eine sortierte, doppelt verkettete Liste implementiert. Jedesmal wenn ein neuer Timeout gesetzt wird, wird die entsprechende Stelle in dieser Liste gesucht und der Timeout dort eingekettet. Dieses Suchen und Einketten passiert mit geschlossenen Interrupts und kann daher auch hohe Verzögerungszeiten verursachen.

Die Funktion, *do_timeouts*, welche die Timeouts behandelt, durchläuft diese Liste und überprüft, ob der Zeitindex des Timeout-Objektes kleiner als der aktuelle Zeitindex ist und behandelt diese. Sobald ein Objekt gefunden wird, dessen Zeitindex größer als der aktuelle Zeitindex ist, wird das Durchlaufen der Liste abgebrochen.

Um diesen Vorgang zu beschleunigen, wird die bisherige Liste auf mehrere sortierte Timeout-Listen aufgeteilt, siehe Abb. 6.1. Die Auswahl der entsprechenden Liste erfolgt, indem der Zeitindex durch den Zeitabstand der einzelnen Listen geteilt wird, und der Rest dieser Division die entsprechende Liste spezifiziert. Wenn der Abstand der Timeout-Listen eine Größe von 2^n besitzt, ist dies durch einfache Bitoperationen möglich. Dadurch werden die Timeouts im Durchschnitt auf mehrere Listen verteilt, und bei dem Einketten werden damit kürzere Listen durchlaufen.

Die Anzahl der Listen und der zeitliche Abstand zwischen zwei Listen ist einstellbar und ist aktuell auf acht Listen und 4ms Abstand eingestellt. Da die durchschnittliche Zeitscheibe 10ms beträgt, und somit aller 10ms die *Timeslice*-Timeouts behandelt werden, müssen im Schnitt nur drei Listen betrachtet werden.

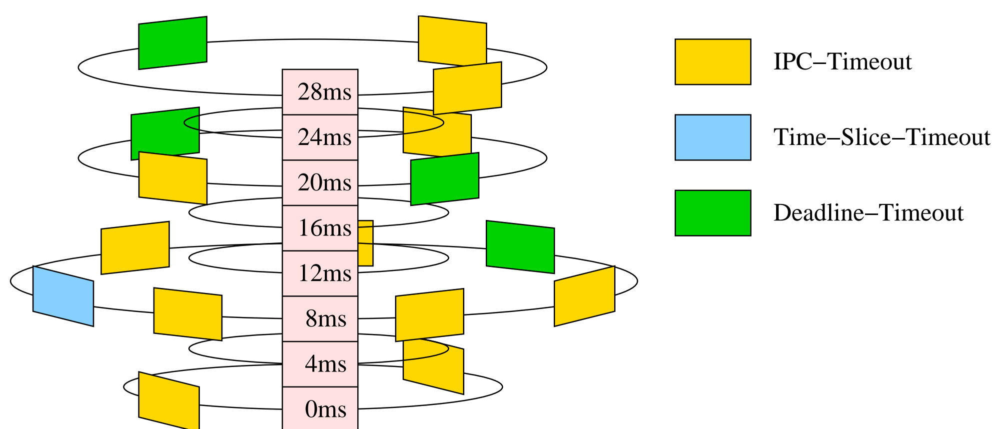

Abbildung 6.1: Timeout-Liste.

Obwohl im Durchschnitt die einzelne Timeout-Liste kürzer wird, wird die Sortierung der Listen beibehalten, damit bei *One-Shot-Timern* das Finden des nächsten Timeouts mit wenig Aufwand möglich ist.

Bei kurzem Listenabstand und bei *One-Shot-Timern* ist es möglich, dass der Zeitraum zwischen zwei Kerneintritten mehrere Listen umfasst. Dadurch müssen stets alle Listen, die innerhalb dieses Zeitraumes liegen, durchlaufen werden. Wenn ein Überlauf entdeckt wird, z.B. wenn bei One-Shot-Timern die CPU erst nach sehr langer Zeit wieder in den Kern eintritt, werden alle Listen behandelt.

6.2 Senderwarteschlange

Die Senderwarteschlange wird mit einem binären Trie implementiert. Dadurch hat bei 256 Prioritäten der Trie eine maximale Tiefe von acht Ebenen. Der Zeitaufwand für das Ein- und Ausketten ist dadurch auch im schlechtesten Fall sehr begrenzt. Möglich sind auch Tries mit vier oder acht Kindern. Dadurch wird zwar die Tiefe weniger, es sind jedoch zusätzliche Zeigeroperationen notwendig, so dass effektiv keine Zeit eingespart wird.

Die Elemente des Tries sind doppelt verkettete Listen von Sendern gleicher Priorität. Der Kopf einer Liste bildet gleichzeitig ein Strukturelement des Tries, Abb. 6.2. Die TCB enthalten dafür zwei Zeiger für die doppelt verkettete Liste von Sendern gleicher Priorität, zwei Zeiger für den rechten und linken Knoten und einen Zeiger für das Elternelement.

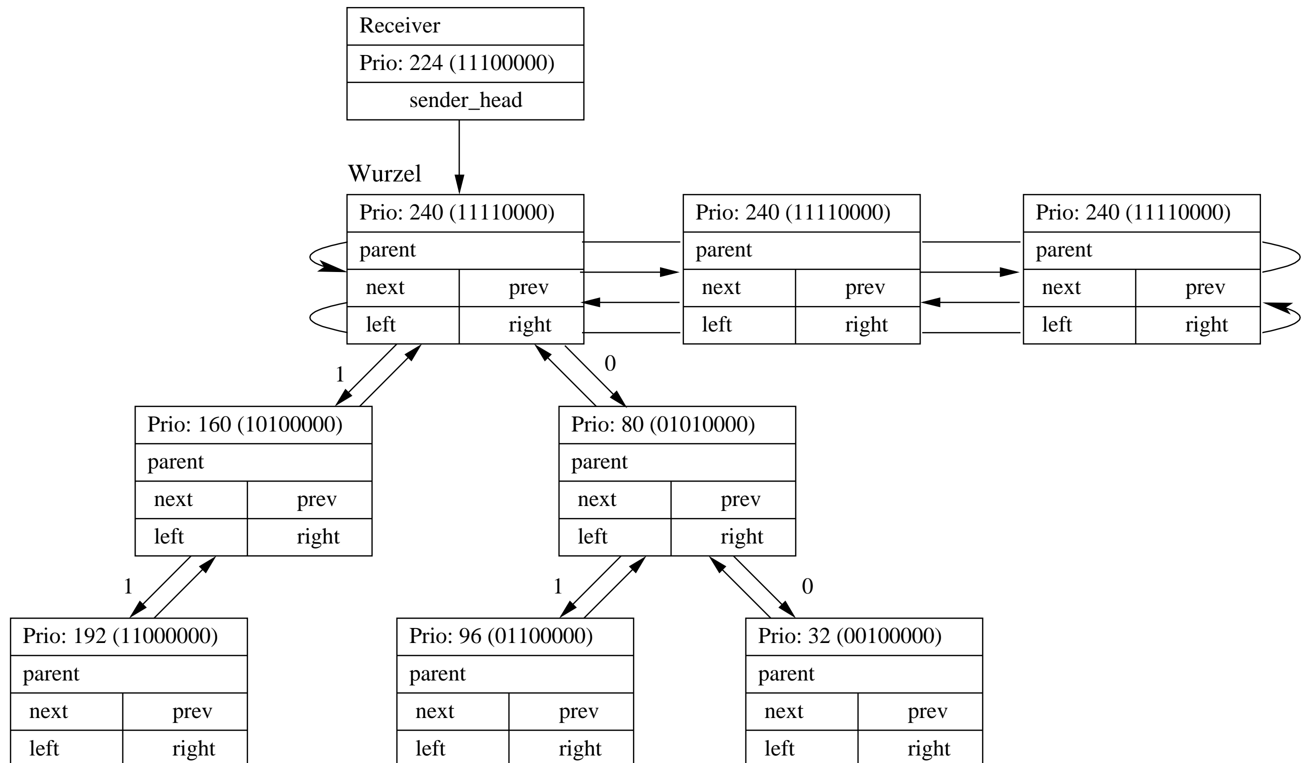

Abbildung 6.2: Implementierte Senderwarteschlange.

Beim Ein- und Ausketten wird sichergestellt, dass der Sender mit der größten Priorität die Wurzel des Baumes bildet. Dadurch ist das Auffinden des Senders, welcher als nächstes aufgeweckt werden muss, sehr schnell möglich.

Einketten: Beim Einketten eines Senders, wird der Baum entsprechend des Schlüssels solange durchlaufen, bis die Liste mit der gewünschten Priorität oder das Ende dieses Zweiges erreicht wird. Bei einem gesetzten Bit wird der linke Zweig genommen, bei einem gelöschten Bit der rechte Zweig. Begonnen wird mit dem MSB des Schlüssels. Dadurch wird erreicht, dass der Sender mit der größten Priorität stets am linken Rand des Tries zu finden sind.

Wenn eine passende Liste erreicht wird, wird der Sender am Ende dieser Liste eingekettet. Dadurch wird auch eine FIFO-Strategie bei Sendern mit gleicher Priorität implementiert, um die Fairness zu gewährleisten. Wenn keine Liste mit dieser Priorität existiert, wird der Sender als neuer Blattknoten am Ende des Zweiges angefügt.

Der Fall, wenn der aktuell einzufügende Sender eine höhere Priorität als die Wurzel besitzt, wird gesondert behandelt. In diesem Fall wird die alte Wurzel mit dem neuen Sender getauscht, und die alte Wurzel wird erneut in den Trie eingekettet. Dadurch ist stets sichergestellt, dass die Sender mit der höchsten Priorität, die Wurzel bilden.

Ausketten: Bei dem Ausketten von Sendern wird unterschieden, ob sie ein Element des Tries sind, oder nur in einer doppelt verketteten Liste von Sendern gleicher Priorität enthalten sind.

Wenn Sender ausgekettet werden, welche in der Struktur des Tries enthalten sind, muss der Sender durch seinen Nachfolger gleicher Priorität im Trie ersetzt werden. Wenn es keinen solchen Nachfolger gibt, wird der komplette Knoten entfernt. An die freiwerdende Position wird ein Blattknoten gesetzt.

Das Entfernen der Wurzel muss gesondert behandelt werden, hier ist eine neue Wurzel zu bestimmen, welche die nächst höhere Priorität besitzt. Der Trie wird von der Wurzel entlang des linken Randes bis zu dem Blattknoten durchlaufen. Das Einketten stellt sicher, dass der Sender mit der nächst höheren Priorität in diesem

Zweig zu finden ist. Die neue Wurzel wird anstelle der alten Wurzel in den Baum eingefügt. Der Blattknoten vom Ende dieses Zweiges wird an die alte Stelle der neuen Wurzel gesetzt. Dadurch stellt das Ausketten sicher, dass der Sender mit der höchsten Priorität die Wurzel bildet.

Synchronisation der Senderwarteschlange

Um die Warteschlange konsistent zu halten, müssen Zugriffe synchronisiert werden. Eine Lösung sind Locks. Dies kann entweder das Thread-Lock des zugehörigen Empfängers oder ein eigenes Lock für die Warteschlange sein.

Es ist an einigen Stellen nicht möglich, ein Lock zu greifen, da nicht blockiert werden darf, falls das Lock von einem anderen Thread gehalten wird. Ein Beispiel ist der Code in der Interrupt-Routine zum Zustellen einer Interrupt-IPC. Genauso kann beim Einketten von Preemption-IPCs bei einem Deadline-Miss keine Lock gegriffen werden, weil die Zeitscheibe schon zu Ende sein könnte. Ähnlich sieht es bei den Activation-IPCs aus.

Interrupts können auf Kernthreads abgebildet werden. Diese Threads laufen auf ihrem eigenen Kernstack und können deshalb Locks greifen, so dass diese Kernthreads auch blockieren können. Die eigentliche Interrupt-Routine weckt nur noch diesen Thread auf, welcher dann die eigentliche IPC-Nachricht verschickt. Bei ARM führt dies jedoch zu Problemen, da es dort bis zu 256 Interrupts geben kann. 256 Kernelthreads würden jedoch zu viel Kernspeicher verbrauchen. Ferner können Preemption-IPCs und Activation-IPCs nicht auf extra Kernthreads abgebildet werden.

Deshalb wird die Senderliste weiterhin mit gesperrten Interrupts synchronisiert. Da aber das Ein- und Ausketten auch im schlechtesten Fall nie mehr als acht Iterationen umfasst, sind die Verzögerungszeiten begrenzt.

6.3 Optimierung des IPC-Pfades

Synchronisation

Der bisherige IPC-Pfad sperrt den IPC-Partner mittels eines Thread-Locks. Das Greifen eines Thread-Locks ist teuer, so dass viel Zeit durch Optimierung eingespart werden kann. Da der IPC-Pfad zum größten Teil mit gesperrten Interrupts läuft, kann er auf einem Uniprozessorsystem nicht unterbrochen werden. Deshalb ist das Sperren des IPC-Partners überflüssig, wenn dieser noch nicht von einem anderen Thread gesperrt ist. Sobald jedoch wieder Unterbrechungen zugelassen werden, muss der aktuelle Thread das Lock nachträglich greifen.

Im Falle, dass der IPC-Partner gesperrt ist, versucht der IPC-Pfad sich das Thread-Lock auf den Partner zu greifen, um die Konsistenz der Daten zu gewährleisten. Es wird zusätzlich ein Unterbrechungspunkt eingefügt, da das Greifen von Locks im ungünstigen Fall sehr kostenintensiv ist. Die Funktion zur Freigabe des Locks wird modifiziert, um diese Optimierung zu erkennen und extra zu behandeln.

Ferner wird eine spezielle Funktion zur Lockfreigabe hinzugefügt, *clear_dirty_dont_switch*, welche nicht automatisch zum gesperrten Thread umschaltet, auch wenn dieser eine höhere Priorität als der aktuelle Thread besitzt. Dies ist notwendig um den Fehler in Abb. 5.7 beim kombinierten Senden und Empfangen zu vermeiden. Erst wenn die Priorität vom wartenden Sender ausgewertet ist, wird zu dem Thread mit der höchsten Priorität umgeschaltet.

Wenn das *Deceite-Bit* gesetzt ist, wird der alte Empfänger bei einer Sendeoperation nur in die Ready-Liste eingekettet. Es wird nur zu ihm umgeschalten, wenn er eine höhere Priorität als der aktuelle Thread besitzt.

Schnelle Rückkehr zum Nutzer

Nachdem die IPC-Operation abgeschlossen wird, wird der IPC-Empfänger lauffähig gesetzt und zu ihm umgeschaltet. Dann läuft noch ein Stück Kerncode auf Empfängerseite. Dieser Code ist dafür zuständig, dass die einzelnen Stackrahmen nacheinander abgebaut werden, und bei einer nicht erfolgreichen IPC-Operation ein Fehlercode für den Nutzer gesetzt wird. Zum Schluss erfolgt das Laden der Register mit dem Werten aus dem Stack und die Rückkehr zum Nutzer.

Als Optimierung kann nach dem Ende einer erfolgreichen IPC der Kernstack des Empfängers überschrieben werden. Es sind auf dem Kernstack nur die IPC-Register und die Rückkehradresse zu einem kleinen Stück Assemblercode abgelegt. Sobald der Empfänger aktiv wird, wird dieser Code angesprungen. Dieser lädt dann die Register mit dem Inhalten aus dem Kernstack und kehrt zum Nutzer zurück, siehe Abb. 6.3.

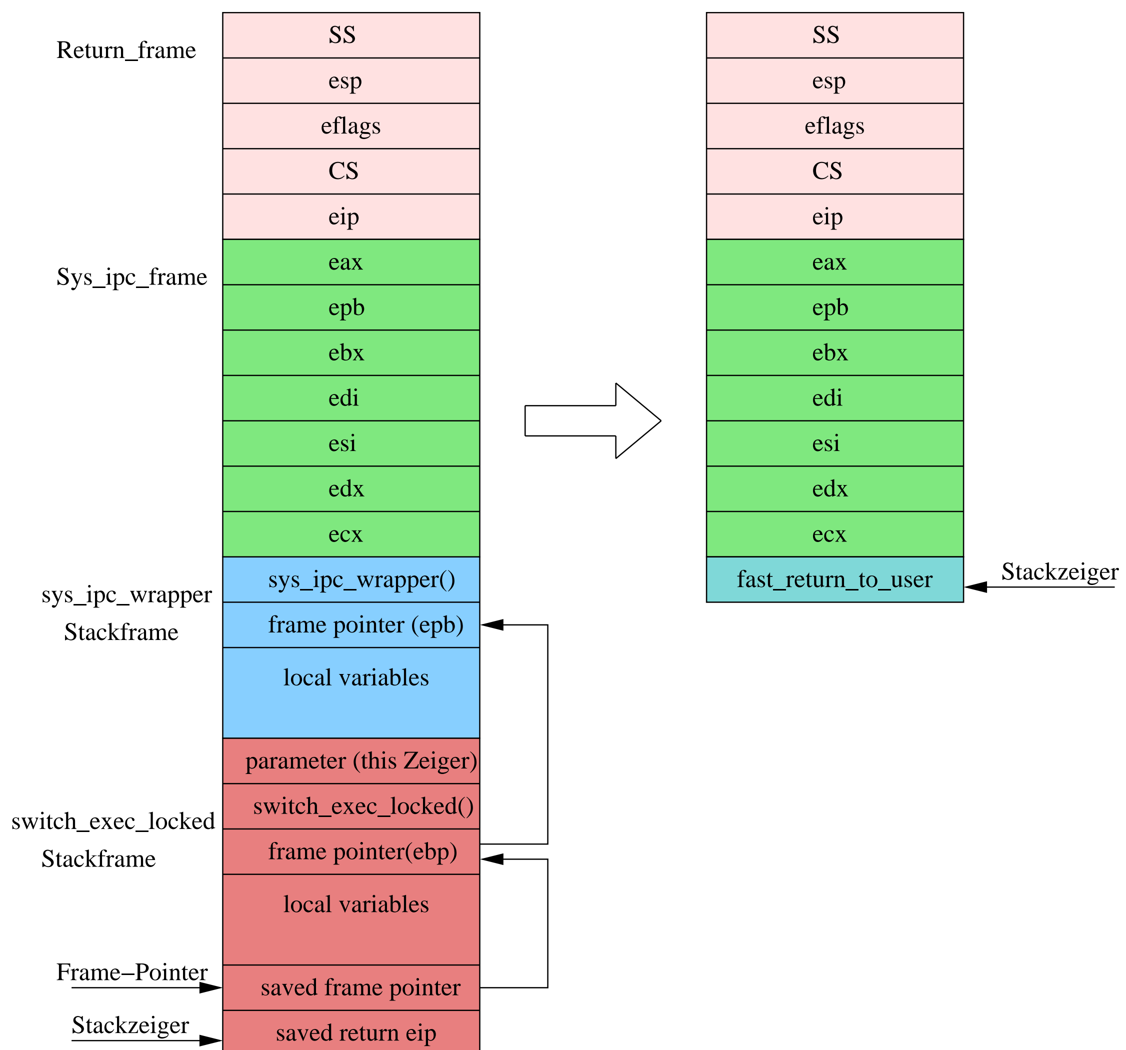

Abbildung 6.3: Kernstack.

Obwohl auf den ersten Blick dieser Ansatz vielversprechend aussieht, waren die Ergebnisse enttäuschend und langsamer als im normalen Fall. Zum ersten liegt es daran, dass fast alle Funktionen inline[1] sind. Daher liegen auf dem Empfängerstack nur sehr wenige Stackrahmen.

[1]inline-Funktionen werden nicht aufgerufen, sondern ihr Code wird an die Stelle des Aufrufs gesetzt

Die IPC-Operation ist oft ein *call* oder ein *reply_wait*. Nach dem Kerneintritt wird vom Assemblercode zuerst die Funktion *sys_ipc_wrapper* aufgerufen. Diese ruft die inline *do_ipc* Funktion auf. Diese nutzt weitgehend inline Funktionen, nur die Kontextwechsel-Funktion ist nicht inline. Es befinden sich in diesem Fall nur zwei Stackrahmen auf dem Kernstack, so dass hier kaum Verbesserungen zu erreichen sind. Weiterhin besitzt auch der eigentliche IPC-Code genug Abkürzungen, um eine schnelle Rückkehr zum Nutzer zu ermöglichen.

Verzögertes Setzen von Empfangs-Timeouts

Diese Optimierung ist nur bei *call* und *reply_wait* IPC-Operationen mit einem Empfangs-Timeout sinnvoll.

Bei dieser Optimierung wird zuerst die Sendeoperation durchgeführt. Im Anschluss erfolgt das Setzen des Zustandes auf Empfangsbereit, ohne das Ready-Flag zu löschen. Dann erfolgt die Umschaltung zu dem Empfänger der IPC-Sendeoperation. Wenn dieser Empfänger sofort eine IPC-Antwort zurück schickt, bevor die aktuelle Zeitscheibe abläuft, erkennt der Thread, dass die IPC schon abgeschlossen ist, und kehrt zum Nutzer zurück. Das Setzen und Löschen eines Timeouts wird eingespart.

Wenn bis zum Ablauf der Zeitscheibe keine IPC erfolgt und der Thread wieder vom Scheduler aktiviert wird, programmiert er seinen Timeout und blockiert endgültig, bis er wieder durch eine IPC oder durch den Timeout aufgeweckt wird.

Ein Nachteil dieser Optimierung ist, es passieren unnötige Kontextwechsel zu Threads, welche dann sofort blockieren und die CPU mittels *schedule* abgeben. Da aber oftmals bei Client-Server die Zeit zum Bearbeiten der Aufträge recht klein ist, lohnt sich diese Optimierung.

Verzögertes Sperren der Interrupts

Die Kosten für einen Unterbrechungspunkt sind erheblich. Sie umfassen den Aufwand für ein Freigeben der Interrupts, eine Nulloperationen und wieder das Sperren der Interrupts. Auf der ausgemessenen CPU waren pro Unterbrechungspunkt 74 Takte notwendig. Wenn man im IPC-Pfad zwei oder mehr Unterbrechungspunkte im gechwindigkeitskritischen Bereich setzt, zum Beispiel sofort nach dem Kerneintritt oder vor dem Kernaustritt, erhöht sich die benötigte Zeit für eine IPC signifikant.

Um diese Kosten zu vermeiden, läuft der IPC-Pfad mit freigegebenen Interrupts, die Freigabe erfolgt sofort nach dem Kerneintritt. Wenn eine Unterbrechung eintritt, sperrt die zugehörige Funktion nachträglich die Interrupts. Dies wird auf IA-32 erreicht, indem die gesicherten Prozessorflags auf dem Kernstack modifiziert werden, und dort das Interrupt-Flag gelöscht wird.

Die Interrupt-Routine merkt sich die aufgetretene Unterbrechung in einer globalen Variable, ohne diese zu behandeln. Sie kehrt dann zum IPC-Pfad zurück.

Der IPC-Pfad läuft nun weiter mit geschlossenen Interrupts bis zum nächsten Unterbrechungspunkt. Ein Unterbrechungspunkt besteht dann aus einem Lesezugriff und einen bedingten Sprung. Es wird einfach anhand der globalen Variable überprüft, ob ein Interrupt aufgetreten war. Wenn dieses nicht gesetzt ist, wird der IPC-Code weiter ausgeführt.

Ist diese Variable gesetzt, wird der gespeicherte Interrupt nachträglich behandelt. Außerdem werden die Interrupts erneut freigegeben, weil diese von der Interrupt-Routine gesperrt wurden. Dadurch werden auch weitere anhängige Unterbrechungen behandelt.

So können weiterhin die einzelnen Operationen zum Sperren und Freigeben der Interrupts beschleunigt werden. Diese Setzen und Löschen einfach ein weiteres globales Flag, welches durch die Interrupt-Routine ausgewertet wird. Wenn dieses Flag gesetzt ist, werden die Interrupts nachträglich gesperrt, der aufgetretene

Interrupt gespeichert und die Interrupt-Routine kehrt sofort zurück. Ist dieses Flag gelöscht, wird der Interrupt normal behandelt. So werden die teuren Maschinenoperationen zum Sperren und Freigeben der Interrupts eingespart. Der größte Nachteil ist der Aufwand, um Race-Conditions zu vermeiden. Die Anpassung des Fiasco-Kerns ist auf dieser tiefen Ebene komplex, so dass die Implementierung nur einen experimentellen Zustand besitzt.

7 Auswertung

Als Testrechner stand ein x86-PC mit einem 1.6Ghz Pentium IV (Willamette) und 256MB DDR-Ram zur Verfügung. Die CPU besitzt getrennte L1-Caches für Code und Daten. Es handelt sich um einen 8KByte großen L1-Cache für Daten und einen L1-Cache für Code, welcher bis zu 12000 so genannte Mikroops speichern kann. Im Gegensatz zu anderen CPUs, wo der L1-Cache für den Code komplette x86-Maschinenbefehle enthält, werden in diesem L1-Cache schon dekodierte Maschinenbefehle abgelegt.

Der L1-Datencache ist physisch markiert und virtuell indiziert. Weiterhin werden neben den physischen Tags noch "virtuelle Hinweise", die *Vhints*, in der Cachezeile abgelegt. welche als Hinweis dienen, welcher Weg (Cachezeile) aus der Menge der möglichen Cachezeilen genommen werden soll. Der L1-Datencache ist vierfach assoziativ, und die Cachezeile ist 64Byte groß. Es werden fünf Bits der Adresse zur Adressierung der Cachezeile und sechs Bits zur Adressierung des Bytes innerhalb der Cachezeile verwendet. Diese Adressinformation kann komplett aus dem 12 Bit großen Offset der Adresse gewonnen werden, so dass hier die Unterscheidung zwischen virtuell und physisch indiziertem Cache irrelevant ist. Wenn zur Adressierung die Bis mit dem niedrigsten Stellenwert genutzt werden, werden Adressen, die sich um 2KByte unterscheiden, auf die selbe Menge von Cachezeilen abgebildet. In diesem Fall hilft dann nur noch die vierfache Assoziativität, um Konflikte zu vermeiden. Die TCB-Größe beträgt 2KByte, so dass hier Konflikte bei dem Zugriff auf den L1-Cache auftreten können.

Der L2-Cache ist ein gemeinsamer Code- und Datencache. Die Größe des L2-Caches beträgt 256KByte und die Organisation ist achtfach assoziativ. Adressen, deren Abstand ein Vielfaches von 32KByte betragen, werden somit auf die gleiche Menge von Cachezeilen abgebildet.

Ein Datum kann sowohl im L1 wie auch im L2-Cache enthalten sein. Eine Garantie gibt es hierfür nicht. Es kann der Fall eintreten, dass eine L1-Cachezeile und ihre entsprechenden L2-Cachezeilen unterschiedliche Daten enthalten und beide *dirty* sind, so dass beide zurückgeschrieben werden müssen, bevor ein neuer Inhalt gelesen werden kann.

Es wurden in verschiedenen Experimenten Aspekte des modifizierten Kerns ausgemessen und mit dem originalen Kern verglichen. Zuerst werden die einzelnen IPC-Operationen in Mikro-Benchmarks ausgemessen. Dann werden die Echtzeiteigenschaften unter verschiedensten Lasten untersucht. Zum Schluss erfolgt eine kurze Bewertung und Analyse.

7.1 Mikro-Benchmarks

Senderwarteschlange

Die Performance der neu implementierten Senderwarteschlange im Vergleich mit der alten Senderwarteschlange wird untersucht. Während die bisherige Implementierung, mit einer Liste nur eine FIFO-Strategie ohne Berücksichtigung der Prioritäten bietet, beachtet die neue Lösung die Prioritäten. Die Warteschlange wurde mit Hilfe eines *Tries* implementiert, und selbst im ungünstigen Fall ist die Zeit zum Ein- und Ausketten eines Elements auf acht Iterationen begrenzt.

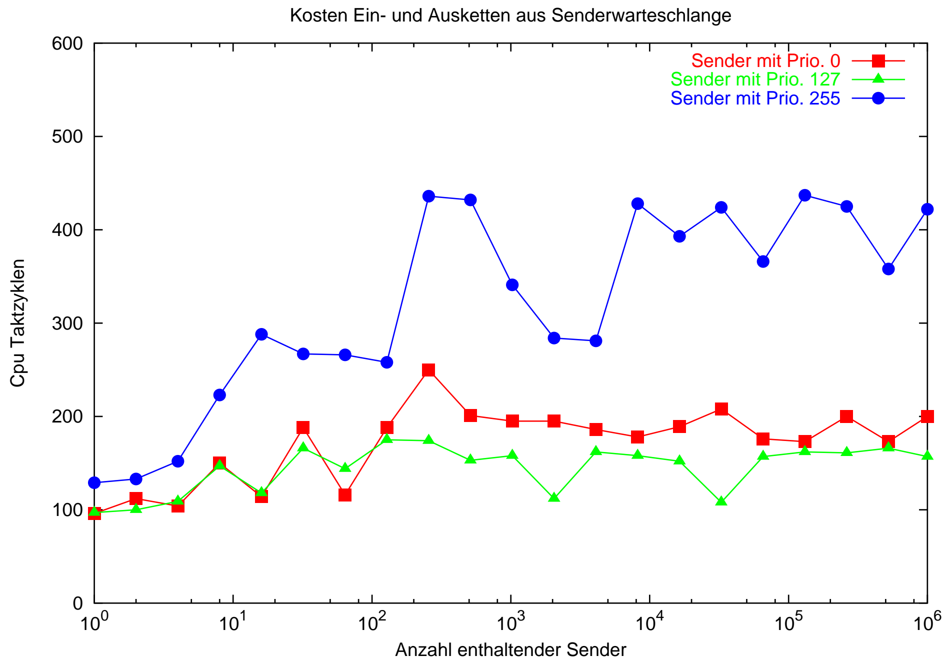

Abbildung 7.1: Abhängigkeit der Kosten von der Größe der Senderwarteschlange

Es wurde die Zeit zum Ein- und Ausketten eines Elements in eine Senderwarteschlange verschiedenster Größe ausgemessen, Abb. 7.1. Die Warteschlangen enthielten Sender zufälliger Priorität im Bereich von 1-254. Es wurde einmal ein Sender mit der Priorität Null, ein Sender mit der Priorität 127 und ein Sender mit der Priorität 255 ein- und ausgekettet. Bei dem Ausketten des Senders mit der Priorität 255 ist jedesmal eine Bestimmung des neuen Maximums notwendig. Selbst bei 10^6 Sendern beträgt die Summe der Zeit zum Ein- und Ausketten bei heißem Cache nicht mehr als 440 Takte. Diagramm 7.2 zeigt die durchschnittliche Zeit für das Ein- und Ausketten von Sendern verschiedener Priorität in Warteschlangen, welche einmal 1, 100 und 1000 Elemente enthalten.

Die gemessenen Werte erfolgten alle mit heißen Caches. Bei kalten Caches sind die Zeiten höher. Die einfache Liste ist in diesem Fall besser. Bei dem *Trie* sind diese Zeiten schlechter, da das Ein- und Ausketten zum großen Teil aus vielen Speicheroperationen besteht. Um die Verzögerungszeiten nicht zu groß werden zu lassen, wird jedoch vor dem Ein- und nach dem Ausketten ein Unterbrechungspunkt gesetzt.

Der alte IPC-Pfad kettete den Sender in die Warteschlange ein, bevor er das Rendezvous mit dem Empfänger versuchte. Durch ein späteres Ein- und Ausketten, kann im Durchschnitt viel Zeit eingespart werden.

Synchronisation

Ein weiterer Grund, warum der bisherige IPC-Pfad langsam ist, ist die Synchronisation. Der neuen IPC-Pfad wird im besten Fall durchlaufen, ohne dass sich ein Lock geholt werden muss. Der bisherige IPC-Code sperrte dagegen jedesmal vor dem Rendezvous den Empfänger und gab ihn nachher wieder frei. Rund 200 Takte werden dadurch eingespart.

Weiterhin wurden die vielen *Retry-loops* zur Synchronisation entfernt, da nur noch nach einem Unterbrechungspunkt auf einem gültigen IPC-Zustand des aktiven Threads geprüft werden muss.

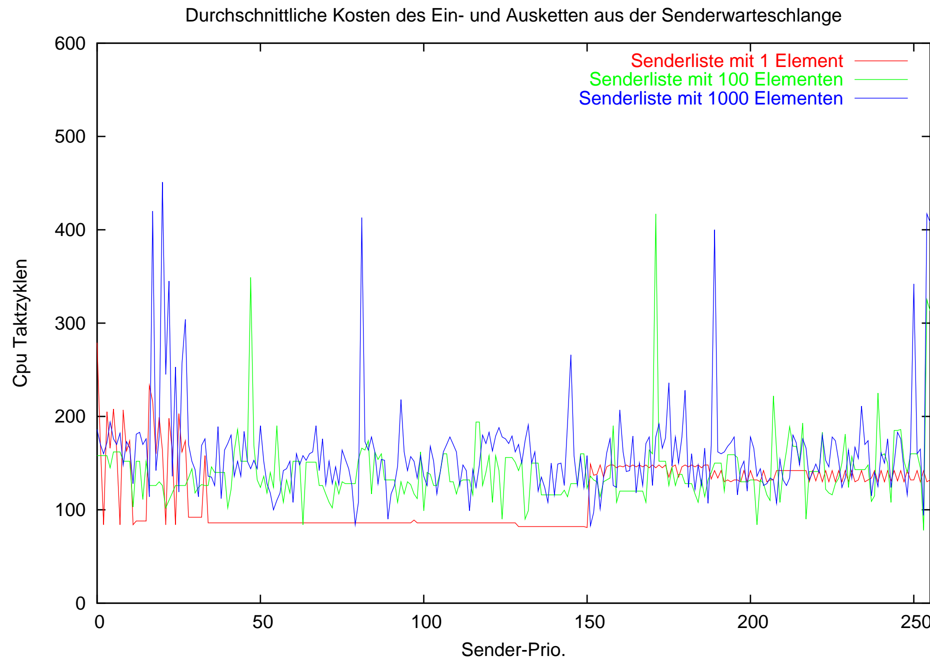

Abbildung 7.2: Durchschnittliche Zeit zum Einketten von Sendern unterschiedlicher Priorität

Timeouts

Das sortierte Aufspalten der Timeout-Liste in mehrere Listen beschleunigt den durchschnittlichen Fall zum Einketten eines Timeout-Objektes, siehe Abb. 7.3. Selbst wenn wenig Timeouts eingekettet sind, wird bei acht Listen in eine kurze Liste eingekettet, so dass nicht lange nach der passenden Position gesucht werden muss. Bei dem Pingpong-Benchmark ist kein Gewinn festzustellen, da die Empfangs-Timeouts verzögert gesetzt werden und die IPC schon abgeschlossen ist, bevor die IPC-Timeouts eingekettet werden müssen.

Im schlechtesten Fall befinden sich alle Timeouts in einer Liste, und es muss die ganze Liste durchlaufen werden, um die richtige Position zum Einketten zu finden. Dieser Fall lässt sich durch diese Lösung nicht beschleunigen. Um auch dort eine Verbesserung der Latenzzeit zu erreichen, müsste das Ein- und Ausketten mit Unterbrechungspunkten erfolgen. Dann wird jedoch die Suche nach der passenden Position sehr schwer, da sich die Liste beim Durchlauf ändern kann. Ferner ist zu beachten, dass die Zeitgeber-Routine auf dem Stack eines Threads mit geringer Priorität ausgeführt werden kann, die Timeoutliste jedoch Threads mit hoher Priorität enthalten kann. Eine unsortierte Timeout-Liste würde hier helfen, die zusätzlichen Kosten fallen dann in der Zeitgeber-Routine an.

Die Synchronisation mittels Locks ist im vorhandenen Kern bisher nicht möglich. Die Zeitgeber-Routine durchläuft und modifiziert die Timeoutliste, jedoch ist das Greifen von Locks in Interrupt-Routinen im bisherigen Kern unmöglich.

Eine andere Möglichkeit, ist die Behandlung der Timeout-Liste durch einen extra Kernthread, welcher unterbrochen werden kann. Dann stellt sich jedoch die Frage, auf welcher Priorität soll dieser Kernthread laufen.

Es gab Überlegungen, endliche Timeouts abzuschaffen und nur unendliche Timeouts und keine Timeouts zuzulassen. In den meisten vorhandenen Systemen werden, bis auf Timeout Null und unendlichen Timeout, selten endliche Timeouts genutzt. Endliche Timeouts dienen oftmals nur dafür, um eine bestimmte Zeit zu blockieren. Ein Beispiel ist die Implementierung des Zeitgeberinterrupts in L4Linux.

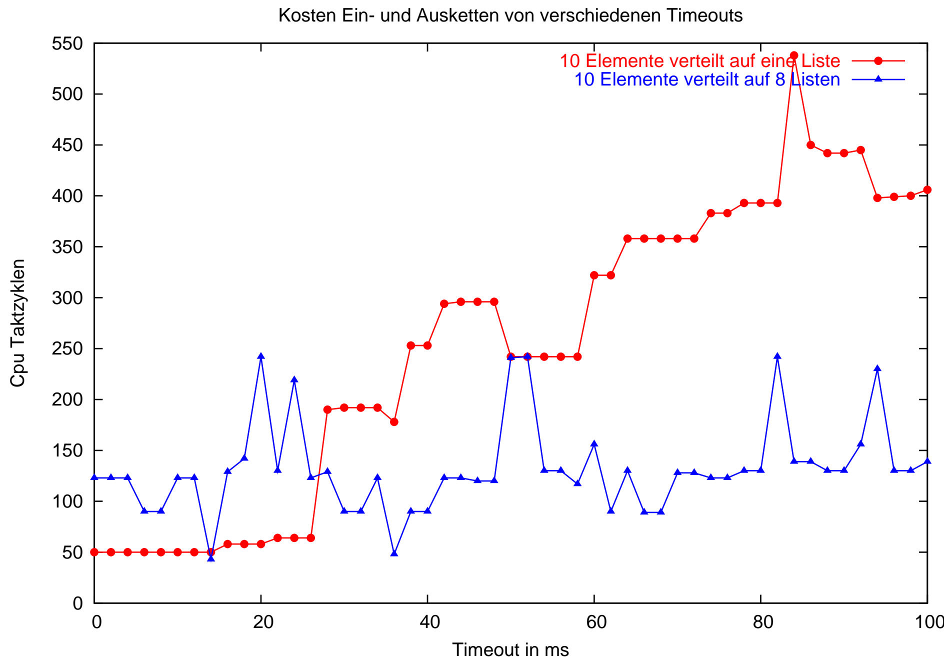

Abbildung 7.3: Durchschnittliche Kosten zum Einketten eines Timeouts

IPC-Performance

Tabelle 7.1 und das Diagramm 7.4 geben einen Überblick über die Kosten verschiedener IPC-Operationen im Vergleich mit dem bisherigen IPC-Pfad. Die Werte umfassen die Kosten für einen kompletten Zyklus, also zwei IPC-Operationen. Beide Kerne wurden mit größtmöglicher Optimierung compiliert. Der Aufruf des Assembler-Shortcuts wurde jedoch unterbunden, so dass direkt der generische IPC-Pfad angesprungen wird.

Ein großer Teil des Geschwindigkeitsgewinns resultiert daraus, dass der neue IPC-Pfad Fall auf viele teure Operationen verzichtet bzw. diese soweit wie möglich zurückstellt. Dazu gehören das Holen von Locks, CAS-Operationen und das frühe Ein- und Ausketten aus der Senderwarteschlange.

Weiterhin tragen das Ausfaktorisieren und die Verschiebung von unkritischem Code, die Verwendung von inline-Funktionen und verzögertes Setzen des Threadzustandes zum Geschwindigkeitsgewinn bei. Ein Teil der Optimierungen wäre auch mit vollständiger Unterbrechbarkeit möglich gewesen, jedoch nicht das Vermeiden von Locks und der CAS-Operationen.

Im Short-IPC Fall, innerhalb eines Adressraumes, wird die doppelte Geschwindigkeit erreicht. Auch kurze Long-IPC Operationen werden schneller, obwohl bei Long-IPC-Operationen nur das Setup und Rendezvous von dem neuen IPC-Pfad profitiert.

Der bisherige Long-IPC Code enthielt einen Bug, welcher sich bei dem Benchmark in Seitenfehlern bei jeder Long-IPC-Operation im IPC-Fenster zeigte. Mit diesem Fehler kostet eine Long-IPC in beiden Kernen rund 2000 Takte mehr.

Bei den Messungen der IPC-Geschwindigkeit, stellte sich eine Eigenart der Pentium IV Caches heraus. Sobald sich die virtuellen Adressen der TCBs von beiden Pingpong-Threads um ein Vielfaches von 64KByte unterschieden, verschlechterte sich die IPC-Performance signifikant. Eine einfache Short-IPC innerhalb eines Adressraumes kostet dann doppelt soviel Zeit. Die physischen Adressen der TCBs spielten in diesem Fall keine Rolle. Ein Grund für diese Effekte können die *Vhints* der Caches sein.

Operation	Neuer IPC-Pfad CPU-Taktzyklen	Bisheriger IPC-Pfad CPU-Taktzyklen	Gewinn
Short-IPC, innerhalb eines Adressraumes			
Heißer Cache	1028	2275	121%
Kalter Cache	13612	18734	37%
Short-IPC, zwischen Adressräumen			
Heißer Cache	2426	3815	57%
Kalter Cache	15341	20298	32%
Long-IPC (4 Wörter)			
Heißer Cache	3641	5525	51%
Kalter Cache	26643	29227	9%
Long-IPC (256 Wörter)			
Heißer Cache	6131	8215	33%
Kalter Cache	28776	33021	14%
Short-Flexpage-Map (4Kbyte)	3798	5210	37%
Seitenfehler zwischen Adressräumen	6751	8046	19%

Tabelle 7.1: Geschwindigkeitsvergleich bisheriger und neuer IPC-Pfad

7.2 Echtzeiteigenschaften im Vergleich

Um die Echtzeiteigenschaften des IPC-Pfades zu bestimmen, wird die Interrupt-Latenzzeit unter ungünstigen Bedingungen ausgemessen. Die Ergebnisse werden dann mit den Werten des bisherigen IPC-Pfad verglichen.

Als periodische Interruptquelle dient der Local-APIC der CPU. Diese Interrupts werden vom Fiasco-Kern durch IPC an ein Nutzerprogramm zugestellt. Es liest den Zeitzähler des Local-APIC aus, und bestimmt so die Zeit zwischen dem Auslösen des Interrupts bis zur Aktivierung des Interrupt-Threads. Der Nutzerthread, welcher mit dem Local-APIC-Interrupt assoziiert ist, besitzt die höchste Priorität im System, und lief in einem eigenen Adressraum.

Der Fiasco-Kern verwendet als Zeitgeber den PIT. Der PIT wird in einem periodischen Modus betrieben, dadurch muss der Interrupt nur am Interruptcontroller bestätigt werden. Eine teure Bestätigung wie bei der RTC ist nicht notwendig. Der Zeitgeber-Interrupt besitzt im Normalfall die höchste Priorität im System. Es wurden zur Bestimmung der “Worst-case”-Zeiten, alle IPC- und Interrupt-Shortcuts deaktiviert. Der Assembler-Shortcut läuft zwar auch mit gesperrten Unterbrechungen, aber bei kaltem Cache ist seine Laufzeit geringer als die des neuen IPC-Pfades, welcher auch mit geschlossenen Interrupts läuft. Daher wird auf dem Assembler-Shortcut verzichtet.

Mit aktivierten Interrupt-Shortcut liegen gemessenen Verzögerungszeiten erheblich geringer, selbst bei hoher Last sind sie unter 10μs.

Es werden verschiedene Lasten verwendet um die Echtzeit-Eigenschaften der Kerne auszumessen und zu vergleichen:

- **Cache-Flooder:** Neben dem Thread zur Ermittlung der Verzögerungzeit lief in der restlichen Zeit ein Cache-Flooder, um die Caches der CPU zu invalidieren. Die Ergebnisse zeigt Diagramm 7.5. Die Zeit für den schlechtesten Fall beträgt für den bisherigen IPC-Pfad 13μs und für den neuen 15μs. Im Schnitt unterscheiden sich die durchschnittlichen Zeiten um 1μs.

- **IPC-Flooder:** Als Last dienten hier mehrere Threads in verschiedenen Adressräumen, welche sich gegenseitig IPCs zustellen. Solange nicht der Interrupt-Thread aktiv ist, werden ständig IPC-Operationen

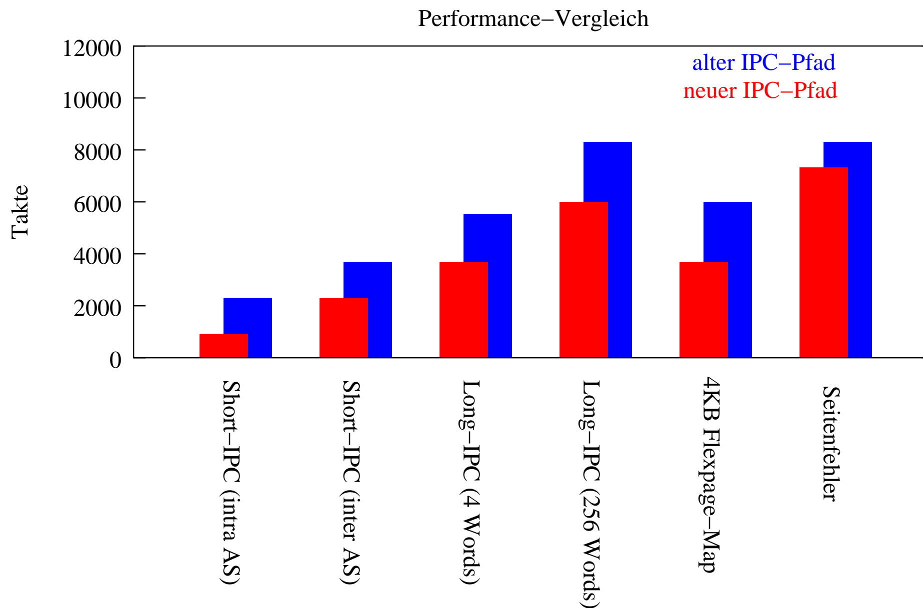

Abbildung 7.4: Geschwindigkeitsvergleich.

ausgeführt. Die Ergebnisse zeigt Abb. 7.6. Die "Worst-Case"-Zeit beträgt für den neuen IPC-Pfad 8μs, für den bisherigen IPC-Pfad 7μs bei hoher IPC-Last mit warmen Cache. Wieder sind ein wenig schlechtere Verzögerungszeiten für den neuen IPC-Pfad erkennbar.

- **IPC-Last und Cache-Flooder:** Hier wird vor jeder IPC der Cache invalidiert. Dadurch ist die IPC-Last geringer als im vorherigen Experiment und es wird sichergestellt, dass der IPC-Pfad einen kalten Cache vorfindet. Die Ergebnisse in Abb. 7.7 unterscheiden nur unwesentlich gegenüber dem ersten Experiment, wo nur der Cache-Flooder aktiv war.

- **DOpE + Cache-Flooder:** Neben dem Cache-Flooder, wird die DOpE-Umgebung [FH03] als Last verwendet. Es laufen der DOpE-Server, weitere benötigte Server und mehrere aktive DOpE-Applikationen, welche für eine hohe IPC-Last sorgen. Die Eingabe- (libinput) und die Semaphorebibliothek (libsemaphore) sind modifiziert, so dass deren Threads nicht mehr die höchste Priorität verwenden. Damit wird sichergestellt, dass der Thread für den Local-APIC-Interrupt die höchste Priorität im System besitzt. Grafik 7.8 stellt die Ergebnisse dar. Hier ist die maximale Zeit für den neuen IPC-Pfad erstmals mit 18μs größer gegenüber dem bisherigen IPC-Pfad, dessen Zeit beträgt 14μs. Ferner ist an der Verteilung ein besseres durchschnittliches Verhalten für den bisherigen IPC-Pfad zu erkennen.

Besonders im letzten Experiment ist die Zeit im ungünstigen Fall für den neuen IPC-Pfad, gegenüber dem bisherigen Pfad, um 4μs höher. Die Experimente zeigen auch, dass der Cache den größten Einfluss auf die Verzögerungszeiten besitzt. Als Ziel muss daher gelten, bei nicht unterbrechbarem Kerncode die Belastung des Caches möglichst gering zu halten. Andere Hardware-Kosten spielen dagegen kaum eine Rolle. Es besteht die Möglichkeit den Cache zu partitionieren [LHH97], damit ein Teil des Caches für den Kern reserviert bleibt. Messungen mit aktivierten Assembler- und IRQ-Shortcut ergaben in beiden Fällen erheblich kleinere Zeiten. Da diese Optimierungen in ungünstigen Fällen nicht genommen werden, wurde in den Experimenten darauf verzichtet.

Es können, auf Kosten der Geschwindigkeit, weitere Unterbrechungspunkte gesetzt werden, um den Abstand zwischen dem neuen und bisherigen IPC-Pfad zu verkleinern. Die Diagramme 7.9 und 7.10 zeigen die Abhängigkeit der IPC-Performance und Verzögerungszeiten unter Last bezüglich der Anzahl der Unterbre-

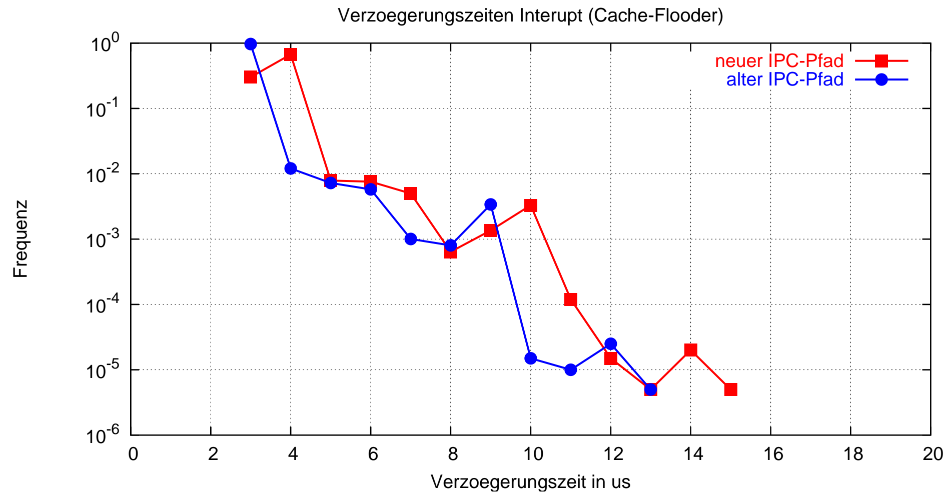

Abbildung 7.5: Cache-Flooder

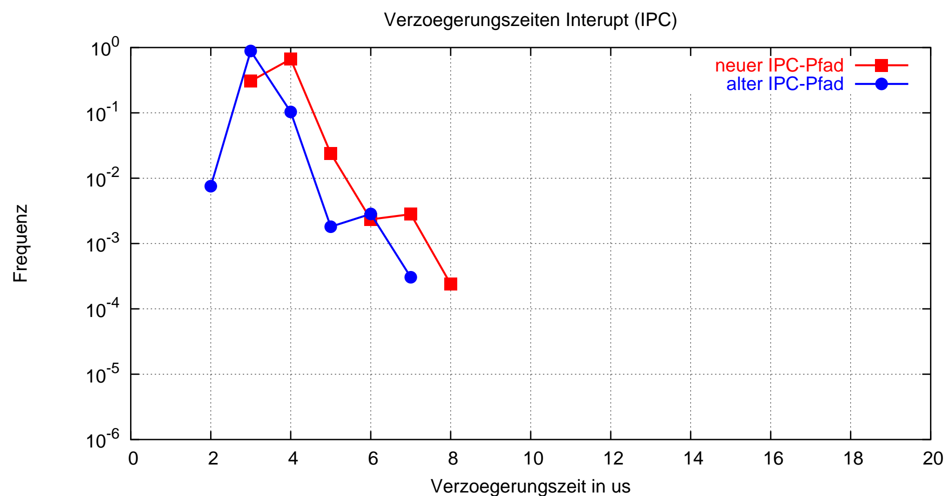

Abbildung 7.6: hoher IPC-Durchsatz

chungspunkte im geschwindigkeitskritischen Programmteil. Wenn die Optimierung, das verzögerte Sperren von Interrupts, verwendet wird, unterscheiden sich die Performancewerte nicht.

7.3 Auswertung

Implementierung und Optimierung

Die Anzahl der Unterbrechungspunkte ist zu minimieren, um die Geschwindigkeit zu optimieren. Dem Gegenüber steht der Assemblershortcut, welcher einen großen Teil der Short-IPC-Operation behandelt und der generische IPC-Pfad selten genommen wird. Dieser behandelt dann nur Long-IPCs, das Einblenden von Flexpages, Timeouts.

In L4Linux 2.6 [Lac04] wird eine Ausnahme-IPC, mittels des implementierten IPC-Pfads, an den Linuxserver zugestellt. Die Antwort durch den Linuxserver erfolgt mit IPC-Shortcut. Hier ist ein geringer Gewinn feststellbar.

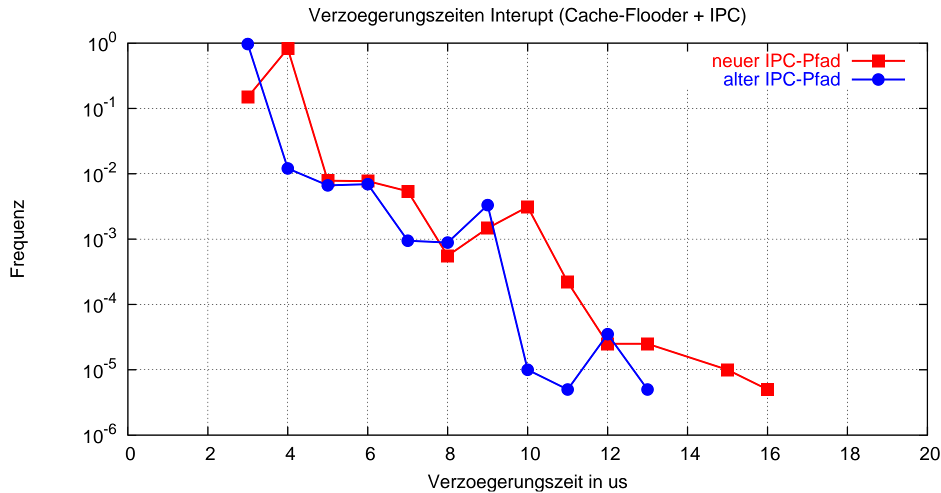

Abbildung 7.7: IPC + Cache-Flooder

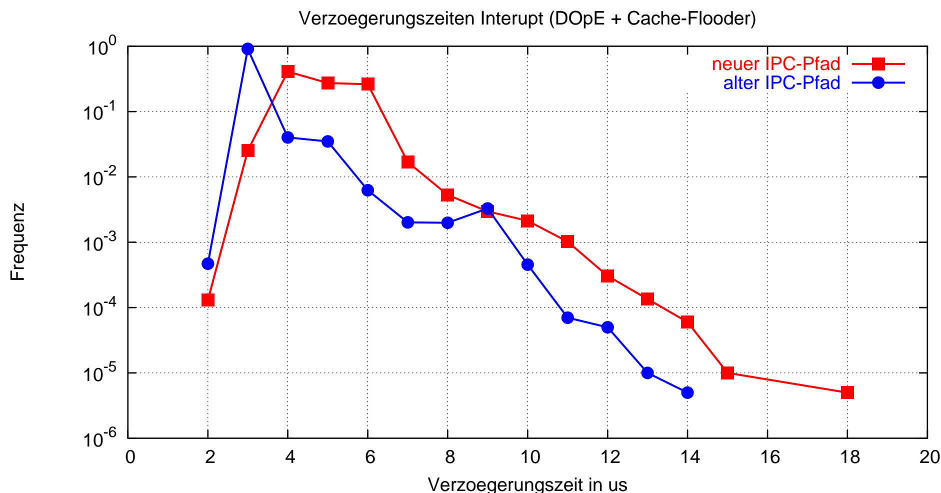

Abbildung 7.8: Dope + Cache-Flooder

Daher ist es besser, mehr Unterbrechungspunkte zu setzten, da die wirklich schnellen IPCs vom Shortcut behandelt werden und nur die langsameren IPC-Nachrichten mit dem normalen IPC-Pfad übertragen werden, wo der Gewinn prozentual geringer ausfällt.

Es werden im neuen IPC viele Inline-Funktionen verwendet, die einige Takte einsparen. Wenn diese Funktion jedoch mehrfach benötigt wird, vergrößert sich der Kerncode. Besonders die IPC-Funktion wird an mehreren Stellen im Kern verwendet. Beispiele sind der IPC-Systemaufruf, Seitenfehlerbehandlung und Exception-IPCs. Für große Funktionen, wie die *do_ipc* Funktion, lohnt sich das Inlining in der Praxis daher kaum.

Weiterhin stellt sich die Frage von Optimierungen, welche man mehr als "Hack" bezeichnen kann. Zum Beispiel das verzögerte Sperren der Interrupts, oder das Überschreiben des Kernstacks, damit der Empfänger möglichst schnell zum Nutzer zurückkehrt. Diese Optimierungen sind schlecht wartbar, und da der Shortcut viele IPC-Operationen schon schnell behandelt, spielen diese nur bei komplexen IPCs eine Rolle, wo ihr Gewinn prozentual sehr niedrig ist. Ferner sind diese Optimierungen oft sehr plattform-spezifisch und unportabel. Es wurden zwar einige dieser Optimierungen umgesetzt, da der absolute Gewinn im späteren Kern jedoch minimal war, wurden einige dieser Optimierungen aufgrund der Wartbarkeit wieder fallen gelassen.

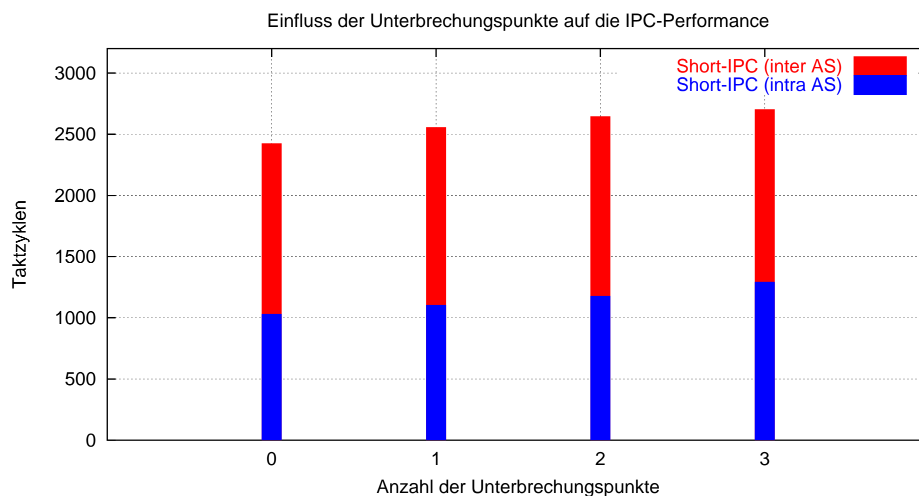

Abbildung 7.9: Abhängigkeit der IPC-Performance von Unterbrechungspunkten

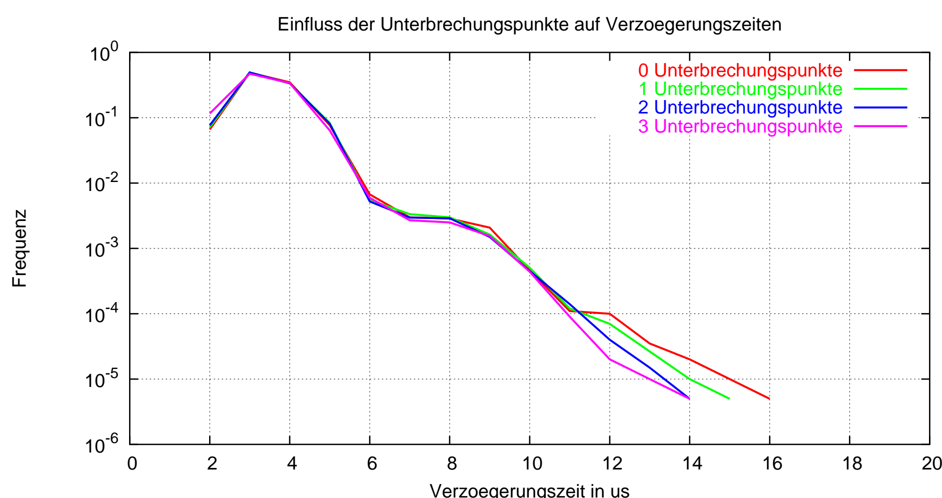

Abbildung 7.10: Abhängigkeit der Echtzeit-Eigenschaften von Unterbrechungspunkten

Ergebnis

In den Messergebnissen sind deutliche Unterschiede zwischen dem bisherigen, unterbrechbaren IPC-Pfad und dem neuen, größtenteils nicht unterbrechbaren IPC-Pfad, festzustellen. Der letztere ist zwar schneller, die Latenzzeiten sind jedoch auch größer. Es werden nur das Rendezvous und der Transfer von Short-IPC mit gesperrten Interrupts ausgeführt. Kopieroperationen für Speicherinhalte, welche im Normalfall erheblich länger dauern, mit gesperrten Interrupts durchzuführen, lohnt sich dagegen nicht. Des Weiteren ist hier auch der Cache zu beachten, bei *dirty* Cachezeilen, können solche Kopieroperationen sehr lange dauern.

Im Fiasco-Kern werden noch andere kritische Abschnitte mit geschlossenen Interrupts, z.B. die Behandlung des Zeitgebers und der Interrupts, ausgeführt, so dass im Vergleich dazu die Latenzzeiten des neuen IPC-Pfades gegenüber dem bisherigen akzeptabel sind. Wenn geringere Verzögerungszeiten benötigt werden, können weitere Unterbrechungspunkte hinzugefügt werden. Ein nicht voll unterbrechbarer IPC-Pfad ist somit für einen Echtzeit-Mikrokern geeignet, wenn genügend Unterbrechungspunkte vorhanden sind.

Die Kosten für die Synchronisation sind im bisherigen IPC-Pfad sehr groß. Eine Umstellung der Synchronisation für kleinere und mittlere kritische Abschnitte, auf das Sperren von Interrupts, z.B. der Zugriff auf

TCBs würde diese Kosten minimieren. Größere kritische Abschnitte können in mehrere kleinere Abschnitte aufgebrochen werden.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein neuer IPC-Pfad entworfen und implementiert, der signifikant schneller als der bisherige IPC-Pfad ist. Obwohl er zum größten Teil mit gesperrten Interrupts ausgeführt wird, ist die Echtzeitfähigkeit durch Unterbrechungspunkte sichergestellt. Die Verzögerungszeiten sind im ungünstigen Fall etwas höher als bei dem bisherigen IPC-Pfad.

Zum Beginn der Arbeit wurden die Kosten für verschiedene Hardware Ereignisse und Instruktionen und besonders kritische Kernoperationen wie Kontextwechsel und Interruptcode analysiert. Hier zeigte sich die negative Wirkung des Caches auf die Verzögerungszeiten. Das anfänglich gesetzte Ziel, Latenzzeiten zu erreichen, welche nicht wesentlich größer als der Zeitaufwand für den längsten Maschinenbefehl sind, zeigte sich schon dort unerreichbar, sobald der Cache mit einbezogen wird. Selbst ein einfaches Laden eines Registers kann im schlechtesten Fall mehrere hundert Nanosekunden betragen. Ferner gibt es auch durch die PC-Architektur Beschränkungen. Das Bestätigen eines Interrupts im *Special fully Nested Mode* kann auf der Testplattform mehr als 5000 Takte kosten.

Nicht unterschätzt werden darf, dass sich die Verzögerungszeiten von bestimmten Ereignissen und Operationen addieren. Wenn die CPU einen kritischen Abschnitt mit gesperrten Interrupts bearbeitet, muss ein ausgelöster Interrupt am Interruptcontroller warten. Erst wenn die CPU die Interrupts freigibt, erfolgt die Zustellung des Interrupts vom Interruptcontroller zum Prozessor. Dieser führt dann die Interrupt-Routine aus. Bis der Interrupt dem Nutzer zugestellt ist, ist im ungünstigen Fall eine Zeit vergangen, welche der Summe der benötigten Zeiten für diese drei Operationen entspricht.

Ein Hauptproblem in dieser Arbeit, war das atomare Umschalten vom Sende- in den Empfangszustand. Der bisherige IPC-Code besitzt hier Fehler, zum Beispiel Timeoutfehler oder Prioritätsinversion. Der neue IPC-Pfad behebt diese Probleme.

Bei der Implementierung wurde nur ein Teil des bisherigen IPC-Pfades übernommen, ein großer Teil wurde neu implementiert. Die meiste Zeit wurde für das Ausmessen und Verfeinern des IPC-Pfades benötigt, um die Geschwindigkeit zu maximieren. Heutige CPUs reagieren durch die verschiedensten Caches und Puffer sehr unterschiedlich auf kleine Codeänderungen. Des weiteren wurden in dieser Arbeit noch kleinere Fehler im bisherigen IPC-Code entdeckt und behoben.

Es bleiben einige Probleme offen, der neue und der bisherige IPC-Pfad führen den Nachrichtentransfer mit der Priorität des Senders durch. Korrekterweise müsste man die Priorität des Empfängers berücksichtigen und den Transfer mit dem Maximum der beiden Prioritäten durchführen. Weiterhin besteht das Problem, dass es zu Prioritätsinversion kommt, wenn Threads mit hoher und geringer Priorität, um einen Server mit geringer Priorität konkurrieren. In [US04] wird auf dieses Problem eingegangen und eine Lösung vorgestellt.

Glossar

CLI Clear Interrupt Flag

DOpE Desktop Operating Environment

DoS Denial of Service

DROPS Dresden Realtime Operating System

FIFO First-In, First-Out

IA-32 Intel x86 32-Bit Architektur

ID Bezeichner, engl. Identifier

IPC Inter Process Communication

IRQ Interrupt

KIP Kernel Information Page

Local-APIC Local Advanced Interruptcontroller

PIC Programmable Interrupt Controller

PIT Programmable Interval Timer

RTC Real Time Clock

STI Set Interrupt Flag

TLB Translation Lookaside Buffer

TSS Taskstatesegment

Literaturverzeichnis

[ABB+86] ACCETTA, M. J., R. V BARON, W. BOLOSKY, D. B. GOLUB, R. F. RASHID, A. TEVANIAN und M. W. YOUNG: *Mach: A New Kernel Foundation for Unix Development*. In: *USENIX Summer Conference*, Seiten 93–113, Atlanta, GA, Juni 1986.

[BF98] B. FORD, M.HIBLER, JAY LEPREAU: *Interface and Execution Models in the Fluke Kernel*. Technischer Bericht, University of Utah, August 1998.

[Cla05] CLAUSS, DIETRICH: *Ivestigation of Mechanisms to Support User-Level Thread Packages on Top of the L4-Fiasco Microkernel*. Diplomarbeit, TU Dresden, Februar 2005.

[FH03] FESKE, NORMAN und HERMANN HÄRTIG: *Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems*. In: *24th IEEE Real-Time Systems Symposium (RTSS)*, Seiten 74–77, Cancun, Mexico, Dezember 2003.

[Hoh98] HOHMUTH, MICHAEL: *The Fiasco Kernel: Requirements Definition*. Technischer Bericht TUD–FI–12, TU Dresden, Dezember 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`.

[Hoh02a] HOHMUTH, MICHAEL: *The Fiasco Kernel: System Architecture*. Technischer Bericht TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[Hoh02b] HOHMUTH, MICHAEL: *Pragmatic nonblocking synchronization for real-time systems*. Doktorarbeit, TU Dresden, Fakultät Informatik, September 2002.

[HWL96] HÄRTIG, H., J. WOLTER und J. LIEDTKE: *Flexible-Sized Page-Objects*. In: *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, Seiten 102–106, Seattle, WA, Oktober 1996.

[Kau05] KAUER, BERNHARD: *L4.sec Implementation*. Diplomarbeit, TU Dresden, Mai 2005.

[Lac04] LACKORZYNSKI, ADAM: *L^4Linux Porting Optimizations*. Diplomarbeit, TU Dresden, März 2004.

[LHH97] LIEDTKE, J., H. HÄRTIG und M. HOHMUTH: *OS-Controlled Cache Predictability for Real-Time Systems*. In: *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, Seiten 213–223, Montreal, Canada, Juni 1997.

[Lie88] LIEDTKE, J.: *An Overview on the L3 Operating System*. In: *ISMM International Symposium on Mini and Microcomputers and Their Applications*, Seite 407, Barcelona, Juni 1988.

[Lie93] LIEDTKE, J.: *Improving IPC by Kernel Design*. In: *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, Seiten 175–188, Asheville, NC, Dezember 1993.

[Lie96] LIEDTKE, J.: *L4 Reference Manual (486, Pentium, PPro)*. Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[Lie99] LIEDTKE, JOCHEN: *L4 Version X in a Nutshell*. Unpublished manuscript, August 1999.

[Pet02] PETER, MICHAEL: *Leistungs-Analyse und -Optimierung des L4Linux-Systems*. Diplomarbeit, TU Dresden, Juni 2002.

[Pug89] PUGH, WILLIAM: *Skip Lists: A Probabilistic Alternative to Balanced Trees*. In: *Workshop on Algorithms and Data Structures*, Seiten 437–449, 1989.

[Ste02] STEINBERG, U.: *Fiasco Microkernel User-mode Port*, 2002.

[Ste04] STEINBERG, UDO: *Quality-Assuring Scheduling in the Fiasco Microkernel*. Diplomarbeit, TU Dresden, März 2004.

[Tea05] TEAM, L4KA: *L4 experimental Kernel Reference Manual, Version X.2*. University of Karlsruhe, 6st Auflage, 2005.

[US04] U. STEINBERG, J. WOLTER, H. HÄRTIG: *Fast Component Interaction for Real-Time Systems*, Dezember 2004.

[War02] WARG, A.: *Portierung von Fiasco auf IA-64*, 2002.

[War03] WARG, A.: *Software Structure and Portability of the Fiasco Microkernel*, 2003.