TECHNISCHE UNIVERSITÄT DRESDEN

Großer Beleg

Implementierung von Local IPC auf L4/Fiasco

Institut für Betriebssysteme
Fakultät für Informatik
Technische Universität Dresden

Rene Reusner

November 2004

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Mikrokerne sind seit längerem Gegenstand der Forschung auf dem Gebiet der Betriebssysteme. Ein Mikrokern stellt nur minimale Kernprimitive zur Verfügung, alles andere muss von Nutzerprogrammen erbracht werden. Mikrokerne der 1. Generation wie Mach entstanden, indem man von einem monolithischen Kern anfing, Teile aus dem Kern in den Nutzerbereich zu verlagern und so den Kern immer weiter zu minimalisieren. Mikrokerne der ersten Generation zeigten aber auch die Geschwindigkeitsprobleme von mikrokernbasierten Systemen auf.

Mikrokerne der 2. Generation, wie L4, Eros, K42 wurden unter einem anderen Gesichtspunkt entworfen. Hier wird nur das dem Kern hinzugefügt, was nicht sicher im Nutzerbereich implementiert werden kann. Sie zeigten außerdem, dass auch mit Mikrokernen effiziente Systeme möglich sind.

Ein Mikrokern stellt Adressräume, welche den Schutzbereich bilden, Threads, die Aktivitätsträger, Kommunikation zwischen Threads, engl. Inter Process Communication, kurz IPC und einen einfachen Scheduler zur Verfügung. Ein mikrokernbasiertes System besteht aus dem Kern, Systemservern und den Anwendungsprogrammen, welche in verschiedensten Adressräumen ablaufen. Die Systemserver stellen verschiedenste Betriebssystemdienste, wie beispielsweise Dateisystem und Netzwerkstack den Nutzerprogrammen zur Verfügung. Die Kommunikation zwischen diesen Programmen erfolgt durch IPC.

Ein Server oder ein Anwendungsprogramm, welches in einem Adressraum läuft, besteht oft aus mehreren Threads. Die Synchronisation zwischen diesen Threads wird häufig auf IPC abgebildet, da der Mikrokern keine weiteren Synchronisationsprimitive anbietet. Häufig bestehen Server auch aus einem Verteilerthread, welcher anfallende Aufträge an die Arbeitsthreads mittels IPC weiterreicht. Ein generisches Problem dieser Struktur sind häufige Adressraumwechsel und ein hoher IPC Durchsatz zwischen den einzelnen Adressräumen. Deshalb bestimmt die IPC Leistung maßgeblich die Geschwindigkeit des Gesamtsystems.

In dieser Arbeit wird eine Möglichkeit untersucht, Kommunikation von Kernthreads innerhalb eines Adressraumes erheblich zu beschleunigen. Dies wird dadurch erreicht, das kurze IPCs mitsamt Threadumschaltung komplett im Nutzermodus ermöglicht werden, ohne die Vorteile von Kernthreads aufzugeben. Dadurch werden der Kerneintritt sowie

die Synchronisation der Kerndaten eingespart. Die eigentliche Aktualisierung der Kerndatenstrukturen erfolgt nachträglich.

1.1 Organisation dieser Arbeit

In dem nächsten Kapitel gehe ich auf L4/Fiasco ein, und stelle deren Grundprimitive wie Adressräume, Threads und Interprozesskommunikation vor. Ich werde die Unterschiede zwischen Kern- und Nutzerthreads beschreiben, und kurz die L4 IPC vorstellen. Im 3. Kapitel werde ich die eigentliche Local IPC, kurz LIPC, Grundidee vorstellen, sowie kurz ihre Probleme und Lösungsvorschläge anreißen. Die Anforderungen und der Entwurf wird im 4. Kapitel vorgestellt. Das Kapitel 5. beschreibt die eigentliche Implementierung auf dem Fiasco Mikrokern. Messergebnisse und die Analyse erfolgt im 6. Kapitel.

2. Hintergrund

2.1 L4

L4, [Liedtke 1996], entwickelt von J. Liedtke ist ein Mikrokern der zweiten Generation. L4 zeichnet sich durch Minimalität bezüglich der Kernschnittstelle aus, wodurch der Kern sehr klein und effizient implementiert werden kann. L4 stellt Adressräume, Threads, IPC, und einen einfachen Scheduler zur Verfügung.

Heute bezeichnet L4 eine Mikrokernfamilie, welche die L4 Schnittstelle bereit stellt. Es gibt heute verschiedene L4 Schnittstellen. Die **V2** Schnittstelle ist die erste und stabile Version, **X.0** [Liedtke 1999] ist eine experimentelle Version, welche V2 erweitert und **X.2** [Dannowski u. a. 2004] eine zweite experimentelle Version, welche Schwächen der V2 Schnittstelle beheben soll.

2.2 Fiasco

Fiasco [Hohmuth 1998; 2002] ist ein L4-kompatibler Mikrokern, welcher die V2, X.0 und die X.2 Schnittstelle implementiert. Fiasco läuft auf IA-32, IA-64 [Warg 2002] und ARM, [Warg 2003] Ebenfalls existiert eine Nutzermodus Portierung [Steinberg 2002] auf Linux unter IA-32, wo der Kern als Linux Applikation auf einem Linuxkern läuft. Fiasco ist in C++ geschrieben und es existiert ein kurzer Assembler IPC Pfad [Peter 2002] zur Optimierung der Geschwindigkeit von einfachen IPC Nachrichten.

Fiasco ist ein Echtzeit-Mikrokern. Dies heißt, dass die Latenzzeiten zwischen Ereignissen und ihrer Behandlung müssen sehr kurz und voraussagbar sein. Um dies zu ermöglichen, ist Fiasco sehr oft unterbrechbar und auch im Kern können Threads verdrängt werden. Die dadurch notwendige Synchronisation erfolgt durch Locks und “Compare and Swap” Maschineninstruktionen.

2.3 Adressräume

Adressräume bilden die Schutzbereiche und mit Inter-Adressraum-IPC können die Threads zwischen Adressräumen untereinander kommunizieren. Innerhalb eines Adressraumes

befindet sich der eigentliche Nutzeradressbereich, welcher alle vom Nutzer zugreifbaren Adressen umfasst. Je nach Kernimplementation befindet sich der Kern in einem eigenen Adressraum, wie bei Fiasco-UX, oder der Kernadressbereich wird gemeinsam in alle Adressräume, für den Nutzer nicht zugreifbar, eingeblendet.

2.4 Threads

Ausführungskontexte, kurz Threads genannt, sind die Aktivitätsträger. Es werden zwei grundlegende Konzepte von Threads unterschieden, Nutzer- und Kernthreads.

Nutzerthread Threads auf Nutzerebene sind dem Kern unbekannt, für ihn ist dies nur ein einziger Prozess. Die Threadverwaltung wie Erzeugung, Scheduling und Löschen erfolgt komplett durch den Nutzer. Weil keine Kerneintritte notwendig sind, ist die Kommunikation und Kontextumschaltung zwischen diesen Threads ist sehr schnell. Ebenfalls ist die Thread-Erzeugung und Löschung sehr effizient, welches dem Verteilerthread/Arbeitsthread-Modell entgegen kommt. Nachteile von Nutzerthreads sind, dass der Kern diese Threads nicht einplanen kann. Ferner, wenn ein Nutzerthread blockiert, blockieren auch alle anderen Nutzerthreads innerhalb dieses Adressraumes.

Kernthreads Threads auf Kernebene sind dem Kern bekannt und werden auch durch den Kern mit eigenen Datenstrukturen verwaltet. Sie sind langsamer als Nutzerthreads, weil für jede Threadoperation ein Kern Ein- und Austritt notwendig ist. Die Vorteile von Kernthreads sind, dass sie vom Kern problemlos eingeplant werden können. Ist ein Thread blockiert, können noch alle anderen rechenbereiten Threads dieses Adressraumes ausgeführt werden. Um die Vorteile von beiden Konzepten zu nutzen, werden oft m Nutzerthreads auf n Kernthreads abgebildet.

L4 bietet Kernthreads an, und ein Adressraum in V2 enthält bis zu 128 Threads. Die zugehörige Kontrollstruktur zu einem Thread, der Thread-Kontrollblock, kurz TCB, enthält einen Kern Stack, die Nutzer Register, den Nutzerprogrammzähler, engl. User Programm Counter, kurz UPC, und Nutzerstackzeiger, engl. User Stack Pointer, kurz USP, sowie andere Status und Kontextinformationen.

2.5 Inter Prozess Kommunikation

Mit IPC können Threads innerhalb und zwischen Adressräumen untereinander kommunizieren. L4 bietet nur synchrone IPC an, dies heißt, der Sender und der Empfänger müssen das Rendezvous zur gleichen Zeit vollziehen. Falls ein Partner nicht bereit ist, blockiert der andere IPC Teilnehmer, solange bis die IPC vollzogen wird oder eine vorher festgelegte Zeit, der Timeout überschritten wird. L4 bietet 3 grundlegende IPC Varianten an, das Senden, das Empfangen, sowie das kombinierte Senden und Empfangen.

Das kombinierte Senden und Empfangen, kurz “call” oder “reply_and_wait” genannt, schaltet atomar von dem Sende- in den Empfangsteil um, und spart einen zusätzlichen Kerneintritt ein. Erstens dient diese Kombination dazu Kerneintritte einzusparen. Ferner ist das atomare Umschalten notwendig, damit Server immer mit Timeout Null antworten können. Dadurch können “Denial of Service” Angriffe verhindert werden.

Die L4 IPC Varianten umfassen den einfachen Registerwerte-Transfer, kurz Short-IPC genannt, komplexere Operationen, Long-IPC, welche das Kopieren von Speicherbereichen umfassen, und das Versenden von Speicherseiten, Flexpages genannt. Man kann bei einer IPC unterschiedliche Arten von Timeouts angeben, den Sende- und Empfangs-Timeout, sowieo Timeouts bei Seitenfehlern.

2.6 IPC Optimierungen

Optimierung von IPC ist schon ein sehr altes Thema. In [Liedtke 1993], “Improving IPC by Kernel design”, wurden die Prinzipien ausgearbeitet, durch die heutige L4 Kerne sehr schnelle IPC zulassen.

Registerwerte-Transfer Hier werden zur Nachrichtenübermittlung auch die Register der CPU genutzt. Wenn eine Nachricht nur aus Registerwerten besteht, spricht man von einer Short-IPC, sonst ist es eine Long-IPC. Der Vorteil von Short-IPC ist, es sind keine Speicherzugriffe in den Nutzerspeicher notwendig. Es können keine Seitenfehler im Nutzeradressraum auftreten, und der Overhead zur Fehlerbehandlung kann eingespart werden.

Lazy Scheduling Hier werden die Warteschlangen im System erst nachträglich vom Scheduler aktualisiert. Insbesondere betrifft dies die Bereitwarteschlange. Dadurch kann bei einfachen IPCs, welche vom Sender zum Empfänger umschalten, das Einketten in diese Warteschlange entfallen.

Thread ID Durch spezielle Thread IDs, kann man aus einer gegebenen Thread ID schnell durch einfache Rechenoperationen den zugehörigen Threadkontrollblock, TCB, errechnen. Es wird eine Indirektion eingespart, welches besonders bei Inter-Adressraum-IPC zusätzliche TLB-Misses [1] vermeiden kann.

Direkte Prozessumschaltung Hier wird gleich zu dem Empfänger der Nachricht umgeschaltet und der Sender schenkt seine restliche Ausführungszeit dem Empfänger. Dies erfordert einfach ein Umschalten des Stackzeigers und des Adressraumes. Besonders bei hohem IPC Aufkommen, wird der Scheduler nicht zum Flaschenhals.

Direkter Nachrichtentransfer Hier wird einfach ein Teil des Empfängeradressraumes in den Senderadressraum temporär eingeblendet, und die Nachricht wird durch einmaliges Kopieren übertragen. Das doppelte Kopieren, vom Sender in einen Kernpuffer und vom Kernpuffer zurück zum Empfänger, wird dadurch eingespart.

Wenn man die Kosten einer IPC ohne die Nachrichtenkosten analysiert, ergibt sich folgendes Bild, Tabelle 2.1. Auffällig sind die sehr hohen Kosten für den Kerneintritt und den Kernaustritt. Auch ohne eine Adressraumumschaltung sind die IPC-Kosten sehr hoch.

[1] TLB: Ein Cache für die Umsetzung von virtuellen zu physischen Adressen. Bei einem TLB-Miss ist kein Eintrag zu dieser virtuellen Adresse in dem Cache vorhanden und es wird auf die Seitentabelle zugegriffen.

	Aktion	Takte
Nutzer	Lade Ziel-ID	2
	Setze Nachricht-Deskriptor	1
	lade 2 Message Register	2
	Kerneintritt	71
Kern	Zugriff auf Ziel-TCB	4
	lade Quelle-ID	2
	setze Status der Quelle auf Warten	1
	schalte Stack um	4
	Setze Status des Ziels auf lauffähig	1
	Kernaustritt und Rückkehr zum Ziel	36
Nutzer	Analyse der Nachricht	2

Tab. 2.1: CPU Takt Kosten für eine einfache IPC auf i486, Quelle: [Liedtke 1993]

2.7 LIPC Grundidee

Local-IPC, kurz LIPC wird in [Liedtke und Wenske 2001] vorgestellt. Local-IPC beschreibt sehr schnelle und kurze IPC Operationen zwischen Kernthreads innerhalb eines gleichen Adressraumes. So können die Vorteile von Kernthreads, Einplanbarkeit und problemlos blockierende Systemaufrufe, mit dem Vorteil der Nutzerthreads, sehr schnelle synchrone IPC, verbunden werden.

Der unter Linux entwickelte Prototyp für eine einfache LIPC ohne Marshalling-Kosten [2], erreichte mit gefüllten Caches und gefüllten TLBs auf einem Pentium III System, 12 Takte. Die Nachrichtenübermittlung und die Umschaltung der Threads wird komplett vom Nutzer ausgeführt und es sind keine teuren Kerneintritte notwendig. Der Kern vollzieht die Umschaltung der Kernthreads nachträglich beim nächsten Kerneintritt.

Der LIPC-Code 2.1 muss nichts weiter, als seinen UPC und USP sichern und nachprüfen ob der IPC-Partner empfangsbereit ist. Wenn dies der Fall ist, erfolgen die Kontextumschaltung zum IPC-Partner und die entsprechenden Statusänderungen. Die Nachrichtenwerte verbleiben in ihren Registern, so dass der Empfänger sie einfach wieder auslesen kann.

Um Inkonsistenzen zu vermeiden, muss der Test des Empfängerthreads, die Datenübertragung und Statusänderung atomar erfolgen. Da LIPC im Nutzermodus stattfindet, ist die Atomarität aber nicht gewährleistet. Wenn der Kern eine laufende LIPC Operation unterbricht, muss dieser entsprechend handeln, um die Atomarität sicherzustellen.

Eine Analyse der LIPC Operation ergibt folgende verschiedene Probleme: Der Nutzer muss auf Kerndaten zugreifen, die LIPC Operation muss atomar sein und Threadumschaltungen durch LIPC müssen mit dem Kern synchronisiert werden.

2.7.1 Kerndatenzugriff

Der UPC, USP, der Thread-Status und der aktuelle Threadzeiger werden von der LIPC Operation benötigt und auch modifiziert. Demzufolge müssen diese Daten dem LIPC-Code zur Verfügung gestellt werden. Um dies zu ermöglichen, wird der bisherige TCB

[2] Marshalling: Ein- und Auspacken der Nachricht

```
Thread A -> Thread B:

call IPC function, i.e. push A's instruction pointer;
save A's stack pointer
If B is a valid thread id AND B waits for thread A THEN
        set A's status to "wait for B";
        set B's status to "run";
        load B's stack pointer;
        current thread :=B;
        return, i.e. pop B's instruction pointer;
ELSE
        more complicated IPC handling;
ENDIF
```

Abb. 2.1: Pseudo LIPC-Code

in einen Kern-TCB, KTCB, und in einen Nutzer-TCB, UTCB, aufgeteilt. In dem UTCB werden die benötigten Daten für den Nutzer les- und schreibbar zur Verfügung gestellt.

Der aktuelle Threadzeiger kennzeichnet den aktuellen Thread und wird für den Nutzer schreib- und lesbar dupliziert. Der Kern kann dann einfach Kontextumschaltungen durch LIPC feststellen, indem er die Nutzer Version des aktuellen Threadszeigers mit seiner Kern-Version vergleicht, siehe Abbildung 2.2. Wenn nötig, muss der Kern die darunterliegenden Kernthreads umschalten.

```
CurrentUTCB inconsistency:
    IF CurrentUTCB_u is in valid UTCB region THEN
        NewKTCB := CurrentUTCB_u->KTCB;
        IF NewKTCB is in valid KTCB region and aligned
            AND NewKTCB->UTCB=CurrentUTCB_u THEN
                switch from CurrentKTCB to NewKTCB;
                CurrentKTCB := NewKTCB;
                CurrentUTCB_k := CurrentUTCB_u;
        ENDIF
    ENDIF
```

Abb. 2.2: aktuelle Threadzeiger Inkonsistenz

Wenn der Kern Inkonsistenzen des Thread Status entdeckt, muss er Zustandsübergänge des Nutzerstatus der Threads von "Lauffähig zum IPC-Warten" und "IPC-Warten zu lauffähig" mit dem Kernstatus synchronisieren, siehe 2.3.

2.7.2 Atomarität

Der LIPC-Code kann an jeder beliebigen Stelle unterbrochen werden. Um Inkonsistenzen zu vermeiden, müssen LIPC Operationen atomar sein. Um dies zu gewährleisten, wird der LIPC-Code in zwei Teile aufgeteilt, siehe Abb. 2.4, welche dann bei einer Unterbrechung speziell behandelt werden.

```
thread status inconsistency:
    IF status_u = "run" AND status_k is "wait for" THEN
        insert thread into run queue;
        status_k := status_u;
    ELSE IF status_u is "wait for" AND status_k = "run" THEN
        delete thread from run queue;
        status_k := status_u;
    ELSE
        kill thread;
    ENDIF
```

Abb. 2.3: Threadstatus Inkonsistenz

```
Thread A -> Thread B:

call IPC function, i.e. push A's instruction pointer;
save A's stack pointer
--- RESTART POINT ---
If B is a valid thread id AND B waits for thread A THEN
        --- FORWARD POINT ---
        set A's status to "wait for B";
        set B's status to "run";
        load B's stack pointer;
        current thread :=B;
        -- COMPLETION POINT ---
        return, i.e. pop B's instruction pointer;
ELSE
        more complicated IPC handling;
ENDIF
```

Abb. 2.4: Aufteilung LIPC-Code

Der erste Teil, vom “Restart Point” zum “Forward Point”, ist der “Restart Block” und der zweite Teil ist der “Forward Block”, welcher den Bereich vom “Forward Point” zum “Completion Point” umfasst. Der LIPC-Code prüft zuerst im “Restart Block”, ob überhaupt LIPC möglich ist. Um die Atomarität zu gewährleisten, wird bei einer Unterbrechung in diesem Teil einfach der Programmzähler des Nutzers auf den Anfang zurück gesetzt, so dass der Nutzer diese Operation erneut ausführt. Da noch keine Statusänderungen durch die LIPC Operation erfolgt sind, ist dies problemlos möglich.

Im “Forward Block” erfolgt die eigentliche IPC und die Umschaltung zum Empfängerthread. Hier erfolgen auch die Statusänderungen der betroffenen Threads, der Senderthread geht in den IPC Wartezustand und der Empfänger wird lauffähig. Wenn jetzt eine Unterbrechung durch den Kern erfolgt, vervollständigt der Kern die begonnene LIPC Operation, und stellt so die Konsistenz sicher.

2.7.3 Kernsynchronisation

Die Synchronisation mit den Kerndatenstrukturen erfolgt nachträglich bei dem nächsten Kerneintritt. Dieser kann durch asynchrone Hardwareunterbrechungen und durch syn-

chrone Ausnahmen, wie Seitenfehler, Systemaufrufe, Debug-Ausnahmen und andere erfolgen. Bei einem asynchronen Kerneintritt muss der Kern gegebenenfalls eine unterbrochene LIPC behandeln, um die Atomarität sicherzustellen.

Der Kern erkennt durch Vergleich des aktuellen Threadzeigers vom Nutzer mit dem vom Kern, dass eine Threadumschaltung durch den Nutzer mittels LIPC erfolgt ist. Der Kern prüft ob diese Umschaltung korrekt ist und vollzieht dann gegebenenfalls diese Umschaltung im Kern. Wenn die Umschaltung ungültig ist, wird der aktuelle Thread deaktiviert.

Besonders problematisch sind aufeinanderfolgende LIPC Operationen, welche mehrere Threads umfassen. Im nächsten Kapitel wird auf die daraus resultierenden Probleme eingegangen und nach Lösungen gesucht. Es wird auch untersucht, in wieweit sich die Ideen aus [Wenske 2002] dafür eignen.

3. Entwurf

3.1 Entwurfsanforderungen

Das Hauptziel bestand darin, eine funktionierende LIPC Implementierung für den Fiasco Kern bereit zu stellen. Damit diese Funktionalität problemlos in schon vorhandene Software einfließen kann, wurden einige Entscheidungen hinsichtlich der LIPC Schnittstelle und Verhalten getroffen.

Anforderungen:

- ABI [1] Rückwärtskompatibel zum normalen IPC-Pfad
- Kommunikation untereinander mit LIPC und IPC
- Einfache Integration in bestehende Anwendungen.

Die ABI Kompatibilität bedeutet, dass Threads untereinander mittels LIPC und IPC kommunizieren können. Dies erfordert etwas zusätzliche Arbeit im LIPC-Pfad, damit alle Threads die gewohnten Registerwerte nach einer IPC erhalten.

3.2 Entwurfsentscheidungen

Die Aufruf-Konvention für LIPC wird aus Geschwindigkeitsgründen so gewählt, das möglichst wenig Overhead notwendig ist. Der Kompromiss aus Geschwindigkeit und einfacher Nutzbarkeit sieht vor, dass anstelle von normalen L4 IDs einfache Threadnummern verwendet werden, welche den Threads 0 bis 127 eines Adressraumes entsprechen.

Eine LIPC Operation besteht immer aus einem Sende- und Empfangsteil. Nach dem Abschluss der Sendeteils geht der betroffene Thread atomar in den Empfangsteil über. Der Nutzer kann vor dem Beginn der LIPC Operation festlegen, ob nach dem Senden ein offenes Warten, “lipc_reply_and_wait” vergleichbar zum L4 “reply_and_wait” Aufruf, oder ein geschlossenes Warten auf einen Thread, “lipc_call” vergleichbar zum L4 “call” Aufruf, erfolgen soll. Beim geschlossenem Warten wird auf einen speziellen Thread gewartet, während beim offenen Warten alle Threads akzeptiert werden.

[1] ABI: Application Binary Interface

```
IF CurrentUTCB_u != current_thread()->UTCB THEN
    IF CurrentUTCB_u is a valid UTCB
        NewKTCB := CurrentUTCB_u to KTCB;
        IF NewKTCB is a valid KTCB
           AND NewKTCB->UTCB=CurrentUTCB_u
           AND NewKTCB->status & "lipc ready"
           AND NewKTCB is unlocked
                switch from CurrentKTCB to NewKTCB;
                CurrentKTCB := NewKTCB;
        ENDIF
    ENDIF
```

Abb. 3.1: LIPC Erkennung von Threadumschaltungen

3.3 Synchronisation mit dem Kern

Die Synchronisation mit dem Kern, das Aktualisieren der Kerndaten, ist eines der Hauptprobleme von LIPC. Wenn der Kern aktiv wird, durchläuft er zuerst der LIPC Nachbereitungscode. Dieser muss durch LIPC verursachte Inkonsistenzen der Kerndaten entdecken, um sie dann zu beheben.

Durch den Vergleich des aktuellen Threadzeigers vom Kern mit dem vom Nutzer können Threadumschaltungen durch LIPC entdeckt werden. Der Kern stellt dann sicher, dass der neue Thread gültig ist und vollzieht die Threadumschaltung, siehe 3.1

3.3.1 LIPC-Ketten

Wenn mehrere Threads untereinander nur mit LIPC kommunizieren, ohne das dazwischen Kerneintritte passieren, entsteht eine LIPC-Kette. Bei LIPC-Ketten kennt der Kern den Anfangspunkt und den Endpunkt der Kette. Der Anfangspunkt ist der alte Kernthread und der Endpunkt ist durch den neuen aktuellen UTCB-Zeiger gegeben. Statusänderungen von Threads innerhalb der LIPC-Kette bleiben aber vom Kern unentdeckt.

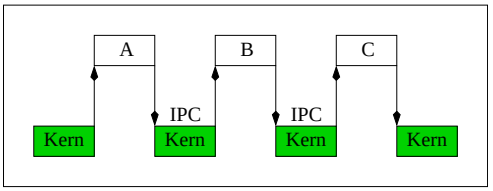

Abb. 3.2: Kern-IPC-Kette

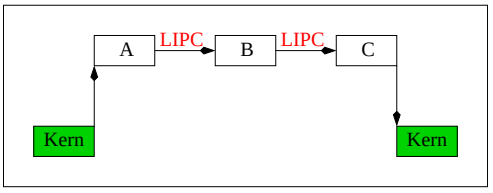

Abb. 3.3: LIPC-Kette

Modifikationen vom UPC und USP im UTCB eines Threads innerhalb einer LIPC-Kette sind problemlos möglich, da jedesmal beim Kernaustritt dieser Threads die aktuellen Werte aus dem UTCB genommen werden.

Kritisch hingegen sind dafür Änderungen des IPC-Status und der IPC-Partner. Bei einer Änderung des IPC-Partners sollten möglicherweise wartende Threads aufgeweckt werden. Wenn man auch das einfache Senden als LIPC Operation zulässt, müssen außerdem neue lauffähige Threads wieder vom Scheduler ausgewählt werden. Statusänderungen von Threads innerhalb einer Kette bleiben vom Kern unentdeckt. Der Kern kann zwar immer den eigentlichen IPC-Status im UTCB abfragen, aber um nach einem Kerneintritt alle betroffenen Threads zu finden, müsste er alle Threads dieses Adressraumes entsprechend untersuchen.

Folgende Probleme können dadurch auftreten.

3.3.2 Geschlossenes/offenes Warten Problem

Dieses Problem, in [Wenske 2002] beschrieben, ist eines der auftretenden Synchronisationsprobleme. Dadurch, dass dem Kern-IPC Partner Änderungen unbekannt bleiben, kann der Kern wartende Threads nicht aufwecken. Kritisch sind deshalb LIPC Operationen, die den “Empfangsbereich” erweitern, zum Beispiel der Übergang vom geschlossenen Warten zu einem offenen Warten, und somit IPC Operationen erlauben, welche vorher nicht möglich waren.

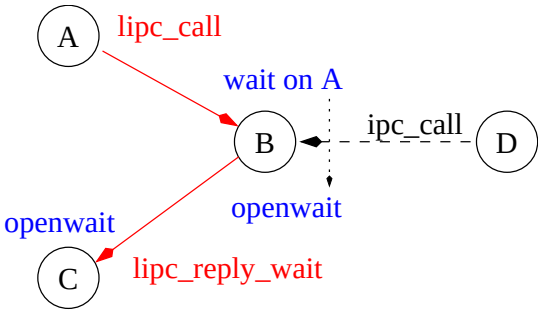

Abb. 3.4: Geschlossenes/Offenes Warten Problem

In diesem Szenario, Abb. 3.4, ist A lauffähig, B im geschlossenen Warten auf A und C im offenen Warten. Thread D will zu B eine IPC schicken, da aber B nicht für D empfangsbereit ist, blockiert D und trägt sich in die Warteschlange von B ein. Wenn A wieder vom Kern ausgewählt wird, kann er mittels LIPC zu B eine Nachricht übermitteln. Ohne dass jetzt ein Kerneintritt passiert, sendet B eine LIPC zu C und geht in offenes Warten über. Wenn jetzt bei dem aktiven Thread C der Kern wieder aktiv wird, bemerkt er nicht, dass er Thread D aufwecken kann.

3.3.3 Fairness

Eine ähnliches Problem wie das gerade besprochene, ist die Gewährleistung von Fairness der LIPC. Hier kann es zu dem “Verhungern” von Threads kommen, welche sich konkurrierend um einen Serverthread bewerben.

In dem folgenden Szenario 3.5 ist Thread A aktiv, B im offenen Warten und C ist lauffähig. Thread A schickt mittels LIPC eine Anforderung an den Server-Thread B, welche dieser anfängt zu bearbeiten. Der Kern bekommt die Kontrolle durch den Hardwarezeitgeber und schaltet zu C.

C wird aktiv und möchte eine LIPC an B senden. Da B nicht empfangsbereit ist, setzt der LIPC-Code eine Kern-IPC auf, welche dann C blockiert. B wird wieder vom Kern

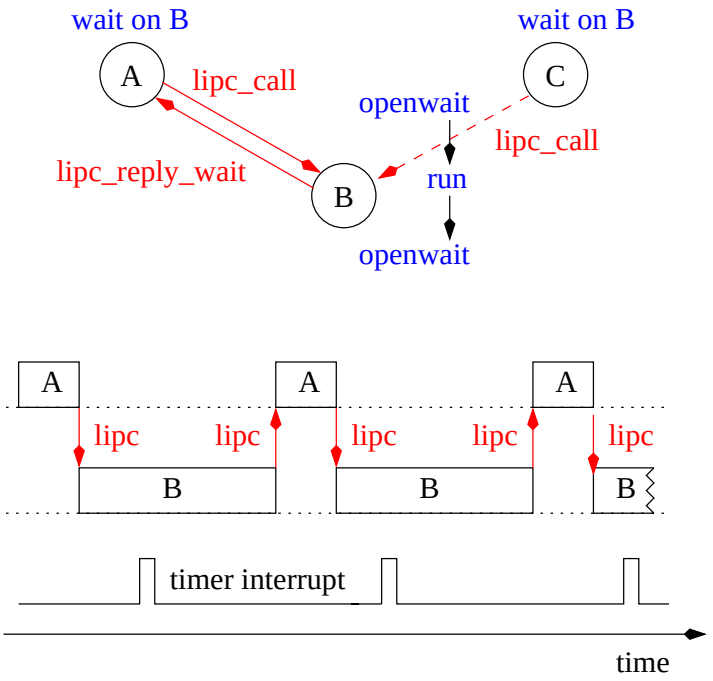

Abb. 3.5: Fairnessproblem

aktiviert und schickt die Antwort mittels LIPC “reply_wait” zurück an A und geht wieder ins offene Warten. Da B im einfachen LIPC Modell nicht sieht, dass C auf B wartet, nutzt er LIPC und nicht Kern-IPC.

Wenn A sofort wieder eine LIPC zu dem empfangsbereiten Thread B schickt und damit wieder B aktiviert, besteht nur ein sehr kleines Zeitfenster, in dem B empfangsbereit für A und C ist. Die Wahrscheinlichkeit, in den Kern zu fallen, wenn B im IPC Wartezustand ist, ist umso kleiner, je größer der Laufzeitunterschied von B zu A ist. Nur wenn der Kern aktiv wird und B im offenen Warten ist, kann der LIPC Nachbereitungscode C aufwecken. Obwohl keiner der beteiligten Threads den Empfangsbereich verändert, kann der Kern C nicht aufwecken.

3.3.4 LIPC Sende-Operationen

Ein drittes Problem tritt auf, wenn als LIPC Operation auch das einfache Senden ohne Empfangsteil erlaubt ist. Diese Sende-Operation kann für spezielle Anwendungen, wie zum Beispiel Aufwecken von Threads durch Semaphorthreads, sehr nützlich sein. Hier bleiben bei LIPC-Ketten Änderungen vom IPC Wartezustand zum Lauffähig-Zustand unentdeckt. Deshalb können diese Threads auch nicht vom Scheduler ausgewählt werden, da sie nicht in der Bereitwarteschlange enthalten sind, siehe Szenario 3.6.

3.3.5 Lösungsansätze für LIPC Kernsynchronisation

Um diese Probleme zu lösen, werden 3 mögliche Ansätze untersucht.

Liste von besuchten Threads

Es wird eine Liste von betroffenen UTCBs, Bild 3.7 in der LIPC-Kette vom LIPC-Code erstellt. Wenn der Kern wieder aktiv wird, durchläuft er diese Liste und aktualisiert den Status der betroffenen Threads. Ebenso werden blockierte Threads aufgeweckt, damit sie

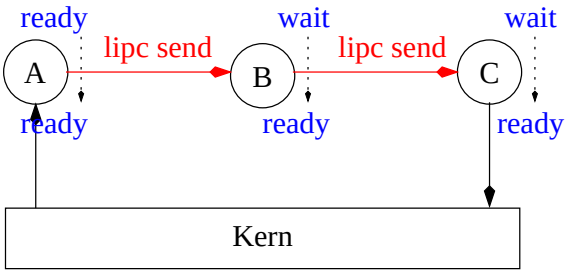

Abb. 3.6: LIPC Senden

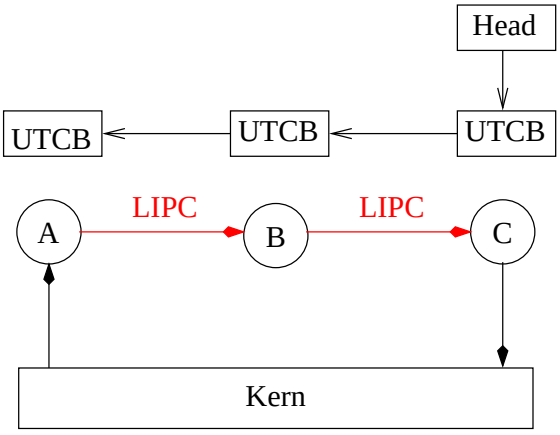

Abb. 3.7: verkettete Liste von UTCBs

ihre IPC durchführen können. Dadurch wird ein LIPC Send möglich, der Nachbereitungscode kann betroffene Threads einfach in die Kern Bereitwarteschlange eintragen.

Ebenso stehen für den Kern die aktuellen Werte wie IPC-Partner, IPC-Status, UPC und USP immer im Kern TCB. Daher muss der Kern-IPC-Code nie Werte aus dem UTCB lesen, nur beim Blockieren müssen der UPC, USP und IPC-Partner in den UTCB geschrieben werden. Die zusätzlichen Kosten für den Kern-IPC-Pfad erhöhen sich deshalb nur minimal. Ein LIPC-Send ist möglich, da der Kern die Zustandsänderungen aller Threads bemerkt und er sie auch in die Bereitwarteschlange einketten kann.

Wenn die Liste mit CPU-Lock-Schutz abgearbeitet wird, erhöht sich die Latenzzeit und ermöglicht “DoS” Angriffe. Eine Möglichkeit dies zu umgehen, ist eine maximale Länge der LIPC-Kette, welche den LIPC-Code um einen zusätzlichen Vergleich erweitert. Eine andere Möglichkeit, ist die Abarbeitung ohne CPU-Lock, dies würde eine aufwenige Synchronisation für Zugriffe auf die verkettete Liste der UTCBs notwendig machen. Außerdem tritt ohne weitere Vorkehrungen auch das Fairnessproblem auf.

Vermeidung von IPC-Partneränderungen

In der von [Wenske 2002] implementierten Lösung, wird bei einer Änderung des IPC-Partners die LIPC abgebrochen, und eine Kern-IPC wird aufgesetzt. Hier werden im UTCB der aktuelle und der alte IPC-Partner, d.h., der letzte dem Kern bekannte IPC-Partner, abgespeichert. Weil sich für den Kern der IPC-Partner von Threads innerhalb der IPC Kette nie ändert, wird das Geschlossene/Offene Warten-Problem vermieden.

```
LIPC status check:
        IF old_thread_status = new_thread_status THEN
```

```
            proceed with LIPC;
    ELSE
            abort LIPC and initiate normal IPC;
    ENDIF
```

Diese Lösung erfordert sehr wenige Anpassungen des Kern-IPC-Pfades. Der LIPC-Code ist sehr schnell, weil keine Liste aktualisiert werden muss. Der Nachbereitungscode muss nur die Endpunkte der LIPC Operation betrachten und wird deshalb erheblich einfacher und schneller. Ferner werden keine UTCBs innerhalb einer LIPC-Kette betrachtet, deshalb ist kein einfaches LIPC-Senden möglich. Da bei IPC-Partner Änderungen sofort die LIPC abgebrochen wird, ist diese Lösung sehr unflexibel. Ferner muss der Kern bei jeder Kern-IPC gegebenenfalls Werte aus dem UTCB, wie UPC und USP, nachladen, welches zusätzliche UTCB-Zugriffe erfordert.

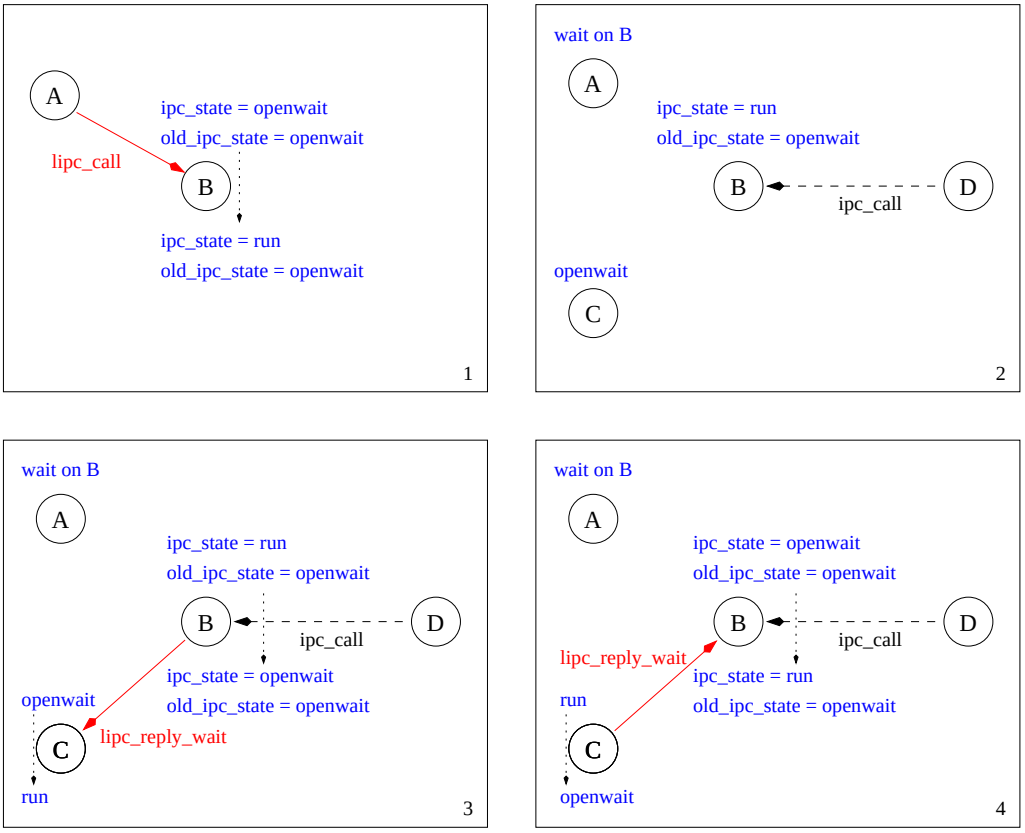

Abb. 3.8: Fairnessproblem (2)

Hier besteht auch das Problem der Fairness. Folgendes Szenario 3.8: A ist aktiv und B befindet sich, durch eine Kern-IPC, im offenen Warten. Der neue und der alte IPC-Status von B ist offenes Warten. Im ersten Bild führt A einen “lipc_call” zu B durch und geht dadurch ins geschlossene Warten auf B über. Der IPC-Status von B wird auf lauffähig gesetzt. Der alte IPC-Status von B bleibt aber unverändert. Im zweiten Bild wird der Kern aktiv, der LIPC Nachbereitungscode wird durchlaufen und der Kern schaltet zum Thread D. Dieser versucht eine IPC Nachricht an B zu senden und blockiert, weil B nicht empfangsbereit ist.

B wird wieder vom Scheduler ausgewählt. B führt im dritten Bild ein “lipc_reply_wait” Operation zu C durch und diese wird nicht abgebrochen, weil sich der alte und der neue

IPC-Status, “offenes Warten”, nicht unterscheiden. Demzufolge erfolgt kein Kerneintritt, welcher D aufwecken kann. B bildet somit den Anfangspunkt einer neuen LIPC-Kette.

Obwohl D schon in der Senderwarteschlange von B enthalten ist, kann C sofort eine Antwort an B schicken und die Fairness verletzen, siehe Bild vier. Der Kern kann zwar beim nächsten Kerneintritt Statusänderungen von B feststellen, weil B Anfang der LIPC-Kette war, und D aufwecken. Die Fairness wird hier aber nicht gewährleistet.

Nutzerlesbares Senderwarteschlangen-Flag

Es wird ein einfaches Flag genutzt, welches vom Kern im UTCB eines Threads gesetzt wird, wenn ein anderer Thread auf diesen Thread wartet.

```
LIPC status check:
        IF send_state = 0 THEN
                proceed with LIPC;
        ELSE
                abort LIPC and initiate normal IPC;
        ENDIF
```

Jedesmal wenn der Thread in seinem UTCB feststellt, dieses Flag ist gesetzt, bricht er die LIPC ab und setzt eine Kern-IPC auf. Da eine LIPC Operation aus einem Sende- und Empfangsteil besteht, kann bei dem Übergang in den Empfangsteil ein wartender Sender aufgeweckt werden.

Durch eine geschickte Codierung dieses Flags mit dem, mathematischen Koprozessor, kurz FPU, Test ist dazu kein zusätzlicher Aufwand notwendig. Ferner ist dieser Test bei einem “lipc_call” nicht notwendig, da auf den gleichen IPC Partner gewartet wird.

Das geschlossene/offene Warten Problem und das Fairnessproblem tritt hier nicht auf. Eine LIPC Operation, welche diesen Thread in den Wartezustand bringt, wird sofort abgebrochen, sobald auf diesen Thread ein Sender wartet. Es wird eine Kern-IPC aufgesetzt, welche diesen Sender aufweckt.

Diese Lösung hat nicht die IPC-Partner Einschränkungen wie die vorherige. Der Kern Nachbereitungscode ist unkompliziert, da nur die IPC Endpunkte betrachtet werden und auch keine Threads aufgeweckt werden müssen. In dieser Lösung treten bei Inter-Adressraum-IPC auch die zusätzlichen Kosten auf, wenn der IPC-Partner sowie UPC und USP aus dem UTCB gelesen werden. Ferner ist kein einfaches LIPC Senden möglich, da nur die Endpunkte betrachtet werden.

3.4 Kerndatenzugriff

Um überhaupt LIPC möglich zu machen, müssen bisherige Kerndaten dem Nutzer zur Verfügung gestellt werden. Dafür wird der bisherige TCB, in einen Kernteil, der KTCB, und in einen Nutzerteil, den UTCB, aufgespalten. In den UTCB werden die für LIPC benötigten Daten ausgelagert und in den Nutzeradressraum schreib- und lesbar eingeblendet. Obwohl der UTCB komplett für den Nutzer beschreibbar ist, wird doch eine Unterscheidung zwischen logisch nur lesbaren und logisch les- und schreibbaren Daten getroffen.

Logisch lesbare Daten werden von dem Kern nur in den UTCB geschrieben aber nie wieder vom Kern ausgewertet. Die Werte dienen dem Nutzer nur zur Information, der Kern verwendet stets die gültige Kopie aus dem KTCB.

Logisch beschreibbare Werte werden von dem Kern nach Bedarf geschrieben und wieder ausgewertet. Wenn falsche Daten den Kern oder andere Prozesse unzulässig beeinflussen können, muss der Kern die Integrität dieser Daten durch eine entsprechende Verifizierung sicherstellen.

In den UTCB werden folgende Daten ausgelagert.

3.4.1 Nutzerprogrammzähler und Nutzerstackzeiger

Der UPC und USP wird bei dem Übergang in den IPC Wartezustand aus dem Kernstack in den UTCB kopiert. Kurz vor dem Kernaustritt des Threads werden diese Werte aus dem UTCB genommen und damit die Werte in dem Kernstack aktualisiert. Dies ist notwendig, falls aufgrund von LIPC der entsprechende Thread im Nutzerbereich bereits aktiv war und jetzt an einer anderen Stelle des Programmtextes steht.

Threads innerhalb des gleichen Adressraumes können problemlos ihren Code sowie Daten modifizieren. Deshalb ist es sicher, dem Nutzer seinem UPC und USP zu exportieren.

3.4.2 IPC-Partner und Teile des IPC-Status

Der IPC-Partner und Teile des IPC-Status wird dem Nutzer logisch schreib- und lesbar zur Verfügung gestellt. Damit der LIPC-Code entscheiden kann, ob LIPC möglich ist, muss ein Teil des Thread IPC-Status zum Nutzer exportiert werden. Ferner muss der LIPC-Code den neuen aktuellen Thread lauffähig setzen und den alten Thread für LIPC freischalten. Es wird dafür ein Flag “LIPC möglich” eingeführt und eine Kopie zum Nutzer exportiert. Wenn LIPC zu einem Thread möglich ist, wird der IPC-Partner dem Nutzer im UTCB schreib- und lesbar exportiert.

Die Auslagerung dieser Informationen ist möglich, da Threads innerhalb eines Adressraumes ihren Code und Daten, zum Beispiel Variablen mit IPC-Partner Informationen, modifizieren können. Ebenso kann der Nutzer frei entscheiden, auf welchen Thread er warten möchte.

3.4.3 Senderwarteschlange-Flag

Eine LIPC Operation besteht stets aus einem Sende- und einem Empfangsteil. Bei dem Übergang des aktuellen Threads in den Empfangsteil müssen gegebenfalls wartende Sender aufgeweckt werden. Dazu dient das Senderwarteschlangen-Flag, um festzustellen, ob andere Sender auf den aktuellen Thread warten. Der LIPC-Code prüft nach, ob andere Threads auf aktuellen Thread warten, indem einfach im aktuellen UTCB dieses Flag ausgewertet wird. Wenn es gesetzt ist, wird die aktuelle LIPC Operation abgebrochen und eine Kern IPC aufgesetzt. Sobald diese Kern-IPC nach dem Senden in den nachfolgenden Empfangsteil übergeht, kann ein wartender Sender aufgeweckt werden.

Dieses Flag ist logisch nur lesbar. Da der betroffene Thread selber entscheiden kann, wann und auf wen er warten möchte, ist dies nur als Hinweis für den LIPC-Code zu betrachten.

3.4.4 Thread-Locks

Fiasco ist ein Echtzeitkern, welcher sehr oft unterbrechbar ist. Zugriffe auf gemeinsam genutzte Betriebsmittel werden durch Locks synchronisiert. Dies ist auch bei IPC der Fall. Der Fiasco IPC-Pfad sperrt den Empfänger-Kern-TCB mittels eines Thread-Locks, um die IPC durchzuführen. Ebenso wird dieses Lock von anderen Systemaufrufen, wie zum Beispiel “thread_ex_regs” ergriffen. Weil der Nutzer durch LIPC im Nutzermodus auch die geteilte Ressource Thread modifiziert, muss er in dieses Schema mit einbezogen werden.

Aus Sicherheitsgründen kann dem Nutzer nicht erlaubt werden, ohne Kerneintritt Threads zu sperren. Um die Synchronisation trotzdem zu gewährleisten, muss der LIPC-Pfad wissen, ob der entsprechende Zielthread gesperrt ist. Nur wenn der Thread nicht gesperrt ist, kann die LIPC Operation durchführt werden, sonst wird sie abgebrochen. Um dies zu ermöglichen, wird dem Nutzer, mittels eines Flags im UTCB logisch lesbar mitgeteilt, ob der Zielthread gesperrt ist.

Diese Information, neben dem Senderwarteschlangen-Flags, kann von anderen Threads in anderen Adressräumen modifiziert werden. Es entsteht möglicherweise ein verdeckter Kanal. Wenn eine spätere L4 Version einen IPC-Kontrollmechanismus implementiert, spielt dies keine Rolle mehr, da diese Flags nur modifiziert werden, wenn IPC erlaubt ist.

3.4.5 Globale-Thread-ID

Die Globale-Thread-ID im UTCB ist logisch nur lesbar und dient dazu, um beim Rückfall zur normalen Kern-IPC einfach die entsprechende Thread-ID des Empfängers zu finden. Ferner dient dieser Wert dazu, bei LIPC die ABI Kompatibilität der Rückgabe-Register mit dem normalen Kern-IPC Aufruf sicherzustellen. Es werden die Register, welche die globale Thread-ID des Absenders enthalten, einfach mit den Werten aus dem Absender-UTCB gefüllt. Dadurch kann zwar ein Thread einem anderen Thread im gleichen Adressraum eine falsche Absenderadresse vortäuschen. Dies beeinflusst aber nicht den Kern und andere Adressräume.

3.4.6 Aktueller Threadzeiger

Der aktuelle Threadzeiger kennzeichnet den gerade aktuellen Thread. Im Kern ist es der aktuelle Stackzeiger, welcher bitweise verknüpft mit einer speziellen Maske, den Zeiger auf den aktuellen Kern-TCB enthält. Da LIPC Threads umschaltet, muss der aktuelle Threadzeiger auch für den Nutzer schreib- und lesbar sein. Um dies zu erreichen, wird er für den Nutzer als Zeiger auf den aktuellen UTCB dupliziert.

Der LIPC-Code setzt diesen Zeiger bei erfolgreicher LIPC auf den neuen Thread. Bei dem nächsten Kerneintritt kann der Kern die Umschaltung durch den Nutzer durch einen Vergleich feststellen und entsprechend handeln.

Dieser Wert ist logisch schreib- und lesbar und absolut kritisch bezüglich Sicherheitsaspekten. Der Kern muss durch Validierung die Gültigkeit des neuen Threads sicherstellen.

3.5 LIPC und Atomarität

Die Grundidee ist, den Nutzer bekannten unmodifizierten Code ausführen zu lassen. Wenn dieser unterbrochen wird, reagiert der Kern entsprechend und stellt einen konsistenten Zustand her. Bei dieser Implementation liegt der LIPC-Code auf der Kern-Informationsseite,

engl. Kernel Info Page, kurz KIP. Der Nutzer kann sich die KIP und damit den LIPC-Code an jede gewünschte Adresse des Nutzeradressraumes einblenden. Dadurch wird eine Erweiterung der Kernschnittstelle notwendig, um dem Kern mitzuteilen, wo der LIPC-Code im Nutzeradressraum liegt.

Da die UTCBs schon bei der Erzeugung des Adressraumes eingeblendet werden, ist keine spezielle Behandlung von Seitenfehlern notwendig. Andere synchrone Unterbrechungen werden wie gewohnt von dem Kern behandelt. Eine Nachbereitung, um die Atomarität sicherzustellen, ist nur bei asynchronen Hardwareunterbrechungen notwendig.

Der LIPC-Code, siehe 2.4, wird wie schon erwähnt, in zwei unterschiedliche Bereiche eingeteilt, dem Roll-Back, welcher den Bereich vom “Restart Point” zum “Forward Point” umfasst, und dem Roll-Forward-Teil, vom “Forward Point” zum “Completion Point”. Unterbrechungen in einem dieser Bereiche getrennt behandelt.

3.5.1 Roll-Back

Im Roll-Back-Teil wird durch den LIPC-Code getestet, ob überhaupt LIPC möglich ist. Wenn in diesem Bereich eine Unterbrechung durch einen asynchrones Ereignis erfolgt, kann sich möglicherweise der IPC-Status der beteiligten Threads geändert haben.

Da in diesem Teil noch keine Zustandsänderung erfolgt ist, können bei einer Unterbrechung diese Tests einfach wiederholt werden. Der Nachbereitungscode setzt bei einer Unterbrechung in diesem Bereich den Instruktion Zeiger einfach wieder auf den Anfang des Roll-Back-Blocks, so dass der Nutzer diese LIPC Operation einfach noch einmal ausführt.

3.5.2 Roll-Forward

Im Roll-Forward-Teil wird die eigentliche IPC abgehandelt. Der Sender sichert seinen Kontext, den UPC und USP, und geht in den IPC-Wartezustand über. Der Zustand des Empfänger wird lauffähig gesetzt und sein gesicherter Kontext durch Laden des neuen UPC und USP wieder hergestellt.

Bei einer Unterbrechung in diesem Bereich muss der Kern die unterbrochene LIPC fertigstellen und den Instruktion Zeiger auf das Ende des Roll-Forward-Blocks, den “Completion Point”, setzen.

3.6 FPU

Fiasco sichert und stellt den Zustand der FPU “faul” wieder her. Bei einem Umschalten des Threadkontext sperrt Fiasco die FPU, wenn der neue Thread nicht der FPU Besitzer ist. Wenn der neue Thread auf die FPU zugreift, wird eine Ausnahme ausgelöst. In der entsprechenden Ausnahmeroutine des Kerns wird dann der FPU Zustand gesichert und der FPU Zustand des neuen Threads geladen.

Das Sperren der FPU kann im Kern erfolgen. Da bei LIPC Threads ohne Wissen des Kerns umgeschaltet werden, müssen entsprechende Vorkehrungen getroffen werden, um ein Überschreiben des FPU Stacks zu vermeiden. Denn, ist die FPU freigeschaltet und durch LIPC wird ein neuer Thread aktiv, kann dieser den Zustand der FPU ändern, ohne dass der alte Zustand gesichert und der neue geladen wird.

Um ungewolltes Überschreiben des FPU Stacks zu verhindern, wird dem Nutzer mitgeteilt, ob die FPU freigeschaltet ist. Der Thread muss in diesem Fall die LIPC abbrechen und eine normale IPC durchführen.

4. Implementierung

4.1 Kerndatenzugriff

Der UTCB Bereich wird bei der Adressraumerzeugung an eine feste Adresse eingeblendet. Bei Fiasco-UX liegt dieser Bereich unterhalb von drei GB, weil der Linux Kern das letzte GB für sich beansprucht. Auf x86 Hardware liegt der UTCB Bereich im letztem GB, um den Applikationen drei Gigabyte Nutzeradressraum zur Verfügung zu stellen. Der Speicher für die UTCBs wird vom Kern bereitgestellt.

Das Einblenden an eine feste Adresse in jedem Adressraum ermöglicht ein einfaches Umrechnen von UTCB-Zeigern zu den entsprechenden KTCB Zeigern. Der Kern greift auf den UTCB Speicher über Adressen im Adressbereich des eins zu eins Mapping des Kernspeichers zu. Dies ist besonders durch Fiasco-UX bedingt, da dort der Kern im eigenen Adressraum läuft. Dieses eins zu eins Mapping des Kernspeichers ist global markiert, es werden dadurch zusätzliche TLB Misses auf nativer Hardware eingespart, wenn die CPU global markierte Seiten zulässt.

Damit der Nutzer seinen aktuellen UTCB findet, gibt es einen Zeiger auf den aktuellen UTCB, welcher auf der UTCB Zeigerseite zu finden ist. Diese Seite wird schreib- und lesbar für den Nutzer in alle Adressräume gemeinsam eingeblendet. Der Zeiger wird bei jedem Kontextumschalten, durch den Kern oder durch den LIPC-Code, neu gesetzt

Der Nachteil dieser Lösung ist, das ein versteckter Kanal in den restlichen Bereich der Seite entsteht. Mögliche Lösungen zur Vermeidung wären, der Zugriff über ein speziell limitiertes Speichersegment. Ferner ist es möglich, diese Seite ungeteilt für jeden Nutzeradressraum zur Verfügung zu stellen oder diesen Zeiger in das UTCB Feld zu integrieren.

Der LIPC-Code findet den aktuellen UTCB mittels dem Segment Register gs:0. Linux erlaubt es eine eigene LDT [1] zu laden, und somit kann die Adressierung über gs:0 auch unter Fiasco-UX erfolgen.

Der Nutzer selbst verwendet keine UTCB Zeiger, sondern einfache Nummern, um die Threads 0 bis 127 eines Adressraumes mittels LIPC yzu adressieren. Die Umrechnung in UTCB Zeiger erfolgt im LIPC-Code durch einfache arithmetische Operationen. Dies

[1] LDT: Local Descriptor Table

ist ein Kompromiss zwischen Schnelligkeit und einfacher Benutzbarkeit. Offenes Warten wird durch eine spezielle Threadnummer kodiert.

Um mit möglichst wenig Vergleichsoperationen im LIPC-Code auszukommen werden einige Flags speziell kodiert und gruppiert. Der IPC Status, Abb. 4.2, enthält den IPC Partner, das Thread Lock und das LIPC Bereit Flag. Der IPC-Sender-Status, Abb. 4.3, enthält das FPU-Flag für eine freigeschaltete FPU und das Senderwarteschlangen-Flag.

UTCB Einträge:

- id: Die komplette V2 L4 UID.
- sp, ip: Der Stack- und Instruktionszeiger des Threads.
- state: Der IPC Partner sowie das Thread Lock und das LIPC möglich Flag.
- send_state: Enthält das FPU und das Sender Warteschlage Flag, wenn 0, dann ist LIPC erlaubt.

ro	0	id	L4 V2 UID
rw	8	pc	Programmzähler
rw	12	sp	Stackzeiger
rw	16	state	IPC Status
ro	20	send_state	IPC Sende Status
	24	pad	

Abb. 4.1: UTCB Format

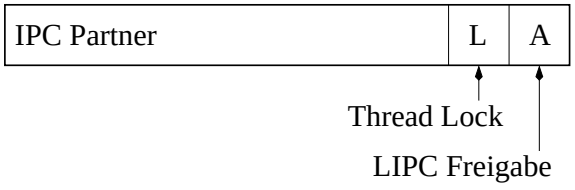

Abb. 4.2: IPC-Status Feld

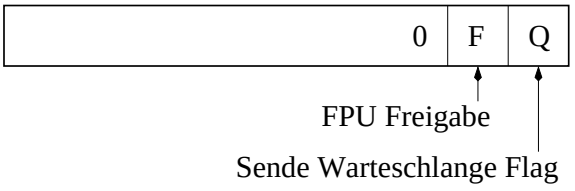

Abb. 4.3: Sender-Status Feld

4.2 LIPC-Code im Detail

Wie schon gesagt, eine LIPC Operation umfasst immer einen Sende- und einen Empfangsteil. Es wird nur Short-IPC, zwei einfache Werte in Registern, unterstützt. Es werden keine Map/Grant Operationen sowie keine Long-IPC Nachrichten ermöglicht. Außerdem werden Sende- und Empfangs-Timeout mit unendlich angenommen.

```
Thread A -> Thread B:

load current_UTCB_Pointer;
calculate UTCB pointer from the given target and from_specifier id’s;
save ip and sp;
-- RESTART POINT --
IF send_state != 0 THEN
        GOTO kernel ipc;
ENDIF

IF ipc_state != current_UTCB_pointer
   OR ipc_state != open wait THEN
        GOTO kernel ipc;
ENDIF

Save user instruction and stack pointer in the UTCB
-- FORWARD POINT --
        set A’s status to the given from_specifier;
        set B’s status to "run";
        set the stack pointer to B’s stack pointer in the UTCB;
        preload the instruction pointer;
        current_UTCB_pointer :=B;
        set source registers for ABI compatibility;
-- COMPLETION POINT ---
jump to B’s ip;

kernel ipc:
        push the return address on the stack;
        set up a kernel IPC with timeout = infinity;
        return;
```

Die Kodierung des IPC-Status und des IPC-Sender-Status erlaubt die Vergleiche zusammen zufassen. Der IPC-Status des UTCB enthält den IPC-Partner in den obersten Bits. In den letzten zwei Bits ist das Thread-Lock-Bit sowie das IPC-Bereit-Bit enthalten. Diese Bits sind gelöscht, wenn LIPC möglich ist. Da die UTCBs auf 32 Bytes ausgerichtet sind und deshalb die letzten fünf Bits immer Null sind, erlaubt dies einen einfachen Vergleich mit den aktuellen UTCB Zeiger, ob ein Thread empfangsbereit ist. Ähnlich ist der Test auf offenes Warten.

Der IPC-Sende-Status enthält das FPU-Flag und das Senderwarteschlangen-Flag. Der IPC-Sender-Status ist ungleich Null, wenn eines dieser Flags gesetzt ist. Demzufolge reicht auch ein einfacher Test auf Null aus, um zu prüfen, ob LIPC möglich ist. Wenn

dieser Test fehlschlägt, wird eine Kern-IPC aufgesetzt, welche gegebenenfalls den wartenden Sender aufweckt.

Bei Ein-Prozessor Systemen kann dass lauffähig Schalten des Zielthreads entfallen. Dies kann faul im nächsten Kerneintritt erfolgen. Dies ist möglich, weil nur der letzte Thread der aktuellen LIPC-Kette stets lauffähig ist. Von den anderen Threads wird nur der IPC-Partner geändert. Wenn dann der Kern aktiv wird, kann er den Endpunkt der LIPC-Kette nachträglich auf lauffähig setzen.

4.3 LIPC KIP

In Fiasco-UX benutzt das als Host-OS verwendete Linux das letzte GB für sich. Um dort die UTCBs, die aktuelle UTCB Zeigerseite und den eigentlichen LIPC-Code unterzubringen, müssen Adressen kleiner als drei GB genutzt werden. Auf reiner x86 Hardware, die diese Begrenzungen nicht hat, sollen dem Nutzer die gesamten virtuellen drei GB Speicher zur Verfügung stehen. Demzufolge liegen bei Fiasco-UX und bei Fiasco/x86 diese Bereiche an unterschiedlichen Adressen.

Für die UTCBs und die aktuelle UTCB-Zeigerseite ist dies kein Problem, da der Kern immer die Adressen kennt, wo diese eingeblendet sind. Aber um LIPC Anwendungen ohne neu zu übersetzen auf Fiasco-UX und Fiasco/x86 zu ermöglichen, müssen für den LIPC-Code geeignete Maßnahmen ergriffen werden.

Ferner sollte der Sprung zu dem LIPC-Code so schnell wie möglich sein. Die x86 Architektur kann für effiziente Sprünge und Prozeduraufrufe nur relative Adressen verwenden. Im Normallfall werden diese von dem Linker errechnet und eingesetzt, und dies bedeutet, dass die Adressen zur Übersetzungszeit bekannt sein müssen.

Um dies zu ermöglichen, wird der LIPC-Code in einen Bereich der KIP abgelegt. Der Nutzer überblendet dann einen eigenen Bereich, dessen Adresse zur Übersetzungszeit bekannt ist, mit dieser KIP. Dadurch kann der schnellste Sprung verwendet werden, ohne die Anwendungen für Fiasco und Fiasco-UX neu übersetzen zu müssen. Das Überblenden durch die KIP, Abb 4.4, erfolgt am Programmstart durch die LIPC Bibliothek.

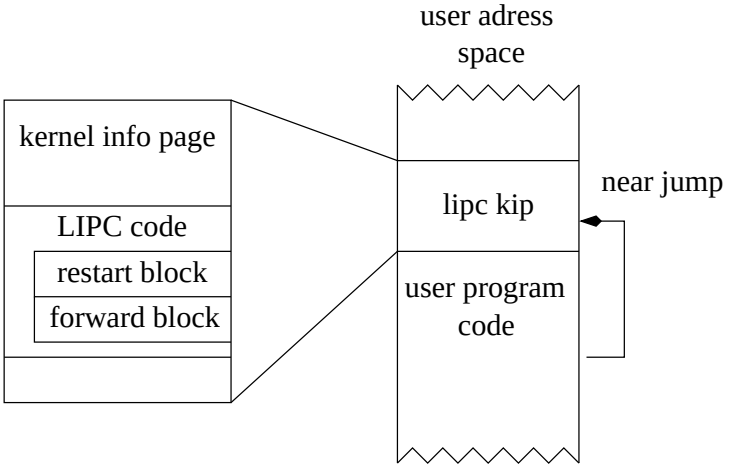

Abb. 4.4: LIPC KIP

Damit der LIPC Nachbereitungscode weiß, wo der LIPC-Code liegt, wird als Konvention festgelegt, dass die erste eingeblendete KIP die LIPC KIP ist. Dieser Wert wird in einem unbenutzten Eintrag im Seitenverzeichnis des betroffenen Adressraumes eingetragen

4.4 Kern LIPC Nachbereitungscode

Wenn ein Thread mittels LIPC zu einem anderen Thread umschaltet und der Kern durch eine asynchrone Unterbrechung, zum Beispiel durch den Zeitgeber, oder durch synchrone Unterbrechungen, wie Systemaufrufe, Seitenfehler und andere Ausnahmen, aktiv wird, muss er die Threadumschaltung im Kern nachvollziehen.

Bei asynchronen Unterbrechungen muss der Kern auch unterbrochene LIPC Operationen beachten, um die Atomarität sicherzustellen. Der Nachbereitungscode unterteilt sich in zwei Teile. Der erste Teil stellt die Atomarität der LIPC Operationen sicher und wird nur bei asynchronen Unterbrechungen durchlaufen. Der zweite Teil, welcher bei allen Unterbrechungen durchlaufen wird, zieht die eigentliche Threadumschaltung im Kern nach.

Bei einer LIPC vom Thread A zu B wird im Nutzermodus zu Thread B mit seinem Nutzerstack umgeschaltet. Wenn jetzt der Kern aktiv wird, sichert die CPU den UPC, den USP und die Flags von B, sowie das Stack- und Codesegment und gegebenfalls einen Fehlercode auf den Kernstack von A. Um arbeiten zu können, muss der Kern noch verwendete Register auf dem Kernstack von A sichern, Abb. 4.5.

Dann erfolgt bei einer asynchronen Unterbrechung der Test, ob eine laufende LIPC Operation unterbrochen wurde. Falls der UPC im “Restart-Block” liegt, wird der UPC im Kernstack auf den “Restart Punkt” gesetzt. Wenn der UPC im “Forward-Block” liegt, wird die LIPC anhand der gesicherten Register vervollständigt.

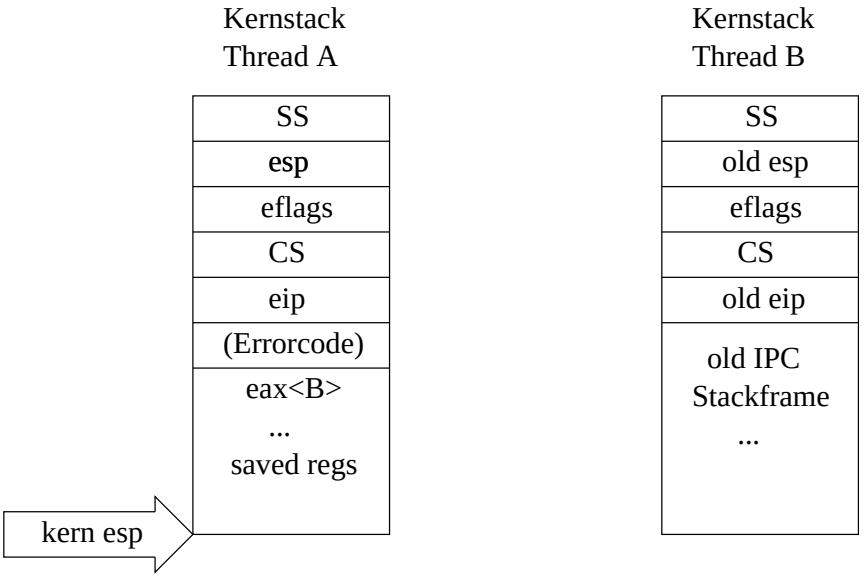

Abb. 4.5: Stacklayout nach Kerneintritt

Der zweite Teil kann anhand von Inkonsistenzen des aktuellen UTCB Zeigers mit dem Kernstackzeiger feststellen, ob mittels LIPC eine Threadumschaltung zu dem Thread B stattfand. Er prüft auch, ob diese Umschaltung auch gültig ist.

Da Fiasco festzugeordnete Kernstacks zu den einzelnen Kernthreads hat, müssen die kompletten Daten des Kernstacks vom Thread A zu dem Stack von B kopiert werden, Abb. 4.6.

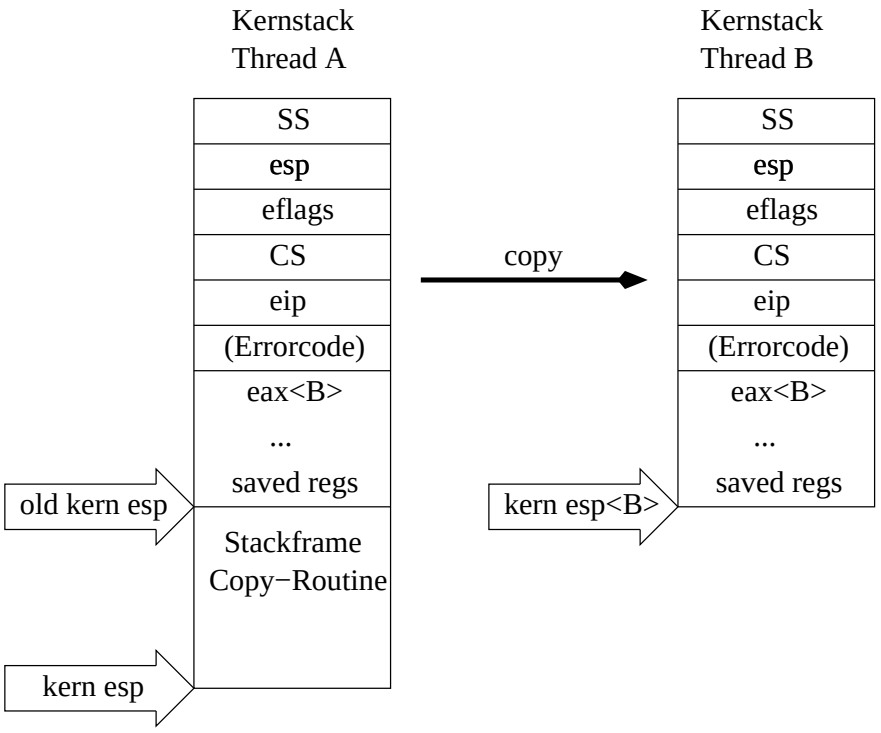

Abb. 4.6: Stacklayout nach Kopieren

Dann wird der Kernstackzeiger auf den neuen Stack umgesetzt und im TSS [2] wird der neue Kernstack eingetragen. Der Threadzustand von B wird auf lauffähig gesetzt.

Der alte Thread A befindet sich im IPC-Wartezustand. Damit zu diesem Thread Kern-IPC möglich ist und um eine Aktivierung dieses Threads durch den Kern mittels **switch_to()** zu erlauben, wird ein spezieller IPC-Stackrahmen aufgesetzt. Der Threadstatus von A wird auf den IPC Wartezustand gesetzt. Als Rückkehr Adresse wird eine spezielle Assemblerfunktion, **_asm_user_invoke_from_localipc** eingetragen, welche nach einem **switch_to()** zu diesem Thread aktiv wird, Abb. 4.7.

Wenn zu Thread A nach einer Kern-IPC im Kern mittels **switch_to()** geschaltet wird, wird er an **_asm_user_invoke_from_localipc** fortgesetzt. Diese Funktion aktualisiert den UPC und USP mit den Werten aus dem UTCB, stellt die IPC Register wieder her, und kehrt in den Nutzermodus zurück.

Es muss kein auf A wartender Sender aufgeweckt werden. Wenn ein Sender sich in die Warteschlange von einem Thread einkettet wird auch das entsprechende Flag im UTCB gesetzt. Der LIPC-Code bricht die LIPC Operation aufgrund des gesetzten Warteschlangenflag ab. Dadurch wird sichergestellt, wenn auf einen Thread Sender warten, kann dieser niemals Anfang einer LIPC-Kette sein. Deshalb muss der LIPC Nachbereitungscode keine wartenden Sender aufwecken.

Der LIPC Nachbereitungscode muss sicherstellen, dass die Umschaltung durch den Nutzer auch gültig ist. Um dies zu vereinfachen, wird ein neues Threadstatus-Flag definiert, welches anzeigt, ob zu diesem Thread LIPC möglich ist. Dies ist der Zwilling zu dem LIPC Bereit Bit im UTCB dieses Threads. Diese beiden Flags werden nur gesetzt, wenn ein Thread im IPC-Wartezustand ist und LIPC möglich ist. Der Test ob die Umschaltung

[2] TSS: Taskstatesegment, eine Datenstruktur im Kern, welche den Kernstackpointer für einen Kerneintritt enthält.

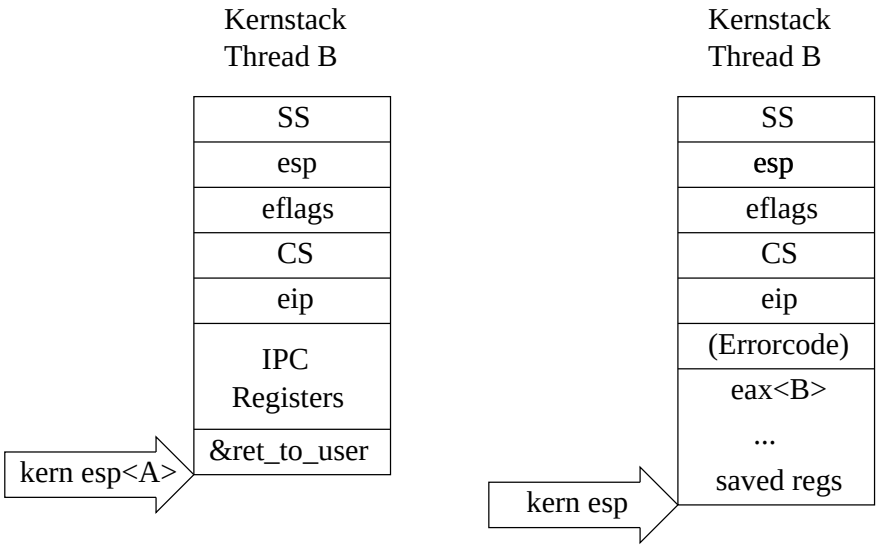

Abb. 4.7: Stacklayout nach LIPC Nachbereitungscode

zu B gültig ist, muss dann einfach nur noch dieses Flag von B testen. Ferner wird geprüft, dass B auch entsperrt ist, sowie die FPU deaktiviert ist.

```
LIPC: Thread A -> ... -> Thread B

IF exception in kernel mode THEN
        GOTO end;
ENDIF

IF synchronus exception THEN
        GOTO copy_kernel_stack;
ENDIF

If exception in RESTART_BLOCK
        set user eip to RESTART_POINT;
ELSE IF exception in FORWARD_BLOCK
        complete LIPC operation;
        set user eip to FORWARD_POINT;
ENDIF

copy_kernel_stack:

IF CURRENT_UTCB = CURRENT_UTCB_KERNEL THEN
        GOTO end;
ENDIF

IF NOT (THREAD_STATE(B) & LIPC_READY)
   OR B is locked
   OR FPU is not locked
        GOTO end;
ENDIF
```

```
copy kernel stack frame from A to B;

switch kernel stack to B stack;
set correct kernel stackpointer in the TSS;
set state of B to READY;

create artifical IPC stack frame on A's stack;
push adress of _asm_user_invoke_from_localipc to A' stack;
update the stack pointer in A's kernel TCB;
set A's kernel TCB status to WAITING and LIPC_READY;

end:
```

4.5 Anpassungen an den Kern-IPC-Pfad

Wenn ein Thread in den IPC-Wartezustand übergeht, wird sein UPC, USP und der IPC-Partner in den UTCB geschrieben, um damit LIPC zu ermöglichen.

Da Fiasco ein Echtzeit-Kern ist, ist der normale Kern-IPC-Pfad unterbrechbar. Außerdem ist ein Thread schon empfangsbereit für IPC, obwohl dieser Thread im Kern noch lauffähig ist. Um LIPC so zeitig wie möglich zu erlauben, wird dennoch LIPC freigegeben.

Fiasco nutzt Locks für die Synchronisation. Ferner ist Fiasco in C++ geschrieben und C++ unterstützt auch die automatische Destruktion von Objekten. Wenn zu einem Thread LIPC möglich ist, wird der Nachbereitungscode den Kernstack dieses Threads möglicherweise mit einem neuen Stackinhalt überschreiben und ihn somit hart abbrechen. Um Ressourcenlecks durch das harte Abbrechen zu vermeiden, muss für diese kritische Bereiche LIPC wieder deaktiviert werden, damit der Thread nicht hart abgebrochen werden kann. Dadurch wird sichergestellt, dass stets alle Objekte wieder zerstört werden, sowie das Freigeben von Locks nicht vergessen wird.

```
write the ip and sp in the utcb;
set RELOAD_IP_SP in the thread state;
sys_ipc()
    prepare_receive()
        set the IPC partner in the sender's UTCB;
    do_send()
        ipc_send_regs()
            ipc_trylock()
                set the "lock bit" in the receier's UTCB;
                sender_ok()
                    test ipc partner in the receiver's UTCB;

            IF ipc allowed THEN
                do ipc;
            ELSE
                unlock and  sleep;
```

```
            ENDIF

            set sender's state to ipc waiting;
            set receiver's state to ready;

            get cpu lock;
            "unlock" the receiver's UTCB;
            IF LIPC possible THEN
                allow LIPC in the sender's UTCB;
            ENDIF
            switch_to(receiver);
            release cpu lock;

    do_receive()
        deny lipc;
        IF a thread is waiting THEN
           lock this thread;
           wakeup and unlock this thread;
        ENDIF

        clear ready flag;
        IF thread state is in ipc waiting THEN
                allow_lipc in the utcb again;
        ENDIF
        schedule();

reload user ip and sp from the UTCB if necessary;
return to user;
```

Wenn der UPC und USP am Anfang der Kern-IPC in den UTCB geschrieben werden, wird zusätzlich ein spezielles Flag, das **Thread_reload_ip_sp** Flag, gesetzt.

Kurz vor der Rückkehr zum Nutzer wird geprüft, ob dieses Flag noch gesetzt ist und wenn ja, werden die Werte aus dem UTCB gelesen und die Werte im Kernstack damit aktualisiert. Dies dient dazu, dass auch der Systemaufruf **thread_ex_regs** stets die aktuellen Registerwerte auslesen kann. Wenn mittels **thread_ex_regs** eine IPC eines Threads abgebrochen wird und falls dieses Flag gesetzt ist, wird der UPC und USP und USP mit den Werten aus den UTCB aktualisiert. Danach wird dieses Flag gelöscht. Ferner wird LIPC für diesen Thread verboten.

5. Auswertung

Zum Ausmessen der verschieden IPC Operationen auf einem unmodifizierten und auf einem LIPC fähigen Fiasco Kern, wird das unveränderte Dresdener L4 Pingpong Programm genutzt. Zum Ausmessen von LIPC Operationen und zum Vergleich mit Kern-IPC wird ein eigenes Testprogramm eingesetzt.

Es werden die Kosten für ein einfaches Pingpong mit LIPC ausgemessen und mit den Kosten für Kern-IPC verglichen. Ferner werden mittels dem L4 Pingpong die zusätzlichen Kosten eines LIPC fähigen Kerns gegenüber dem originalen Kern ermittelt. Um Aussagen über die Latenz bei Unterbrechungen zu geben, werden die Kosten des LIPC Nachbereitungscodes analysiert.

Für die Vergleichsmessungen wird in allen Fällen der IPC Assemblershortcut genutzt. Bei IPC Messungen wird stets eine kompletter IPC Zyklus ausgemessen. Ein Zyklus umfasst die zwei IPCs hin und zurück und die Verpackungskosten.

Als Testplattformen standen ein Pentium mit 100Mhz mit 256Kbyte L2 Cache, sowie ein K6-2 mit 350Mhz mit 512Kbyte L2 Cache zur Verfügung.

5.1 Pingpong Performanz

Zuerst wird der Overhead eines LIPC Kernes gegenüber einem unmodifiziertem Kern ausgemessen.

Pentium	normal	LIPC	Overhead
intra(warm)	430	602	172
intra(kalt)	692	1102	410
inter(warm)	873	1192	319
inter(kalt)	1321	1851	530

Tab. 5.1: Kosten für das normale Pingpong Pentium 100Mhz

Der zusätzliche Mehraufwand für LIPC ist nicht unerheblich. Zum großen Teil ist dies durch eine komplexere Logik im Assemblershortcut und durch dem UTCB Zugriff bedingt. Der Shortcut muss bei einem kombinierten Senden und Empfangen prüfen ob er

K6-2	normal	LIPC	Overhead
intra(warm)	462	564	102
intra(kalt)	548	944	396
inter(warm)	930	976	46
inter(kalt)	1355	1635	280

Tab. 5.2: Kosten für das normale Pingpong K6-2 350Mhz

LIPC erlauben kann und wenn ja, muss er in den UTCB den IPC Status, den Stackzeiger und den Programmzähler eintragen.

Dadurch kann man auch den geringeren Overhead bei Inter-Adressraum-IPC beim K6 gegenüber der Intra-Adressraum-IPC des K6 erklären. Ein Pingpong Partner, der Klient, ist immer im geschlossenes Warten auf den Server. Bei Inter-Adressraum-IPC ist demzufolge LIPC nicht möglich und es werden die Kosten für diesen UTCB Zugriff eingespart.

Der Pentium kennt keine global markierten Seiten, so das bei einem Umschalten des Adressraumes alle TLB Einträge entfernt werden. Deshalb fallen für die UTCB Zugriffe auch zusätzliche TLB Misses an.

5.2 LIPC Performanz

Zum Ausmessen der LIPC Geschwindigkeit wird ein eigenes Pingpong verwendet. Zum Vergleich werden außerdem die Kern-IPC Zahlen von dem LIPC Kern und dem originalen Fiasco Kern für Intra Adressraum IPC ausgemessen. Es wird auch ein kombiniertes LIPC/Kern-IPC Pingpong analysiert, wo ein Partner LIPC und der andere Partner Kern IPC verwendet.

In beiden Fällen werden C-Bindings der Operationen verwendet. Es wird wieder ein kompletter Zyklus ausgemessen, also zwei (L)IPC inklusive den Marshalling Kosten. Es erfolgen 1000 Schleifendurchläufe und die komplette Laufzeit geteilt durch die Anzahl der Schleifendurchläufe ausgeben. Die Marshalling-Kosten betragen auf dem Pentium und K6 rund 10 Takte pro LIPC. In allen Messungen wird mit warmen Caches gearbeitet.

	Pentium 100Mhz	K6-2 350Mhz
LIPC Pingpong	113	62
Kern-IPC/LIPC gemischt	399	344
Kern-IPC, LIPC Kern	647	563
Kern-IPC, org. Kern	457	493

Tab. 5.3: LIPC im Vergleich

Auf dem Pentium 100 spart LIPC nur den Kerneintritt ein. Eine einfache LIPC, abzüglich der zusätzlichen Marshalling Kosten, benötigt 50 Takte. Dagegen ist auf dem K6 LIPC sehr schnell, die Kosten für eine LIPC Operation betragen 31 Taktzyklen. Wenn man davon noch die zusätzlichen Kosten wie Marshalling abzieht, bleiben weniger als 25 Takte für eine LIPC übrig. Auf einem aktuellem System, ein Pentium 4 mit 1.6Ghz, erreicht das

Pingpong für einen kompletten Zyklus 49 Takte, also weniger als 20 Takte pro IPC nach Abzug aller zusätzlichen Kosten.

Die Kosten für ein kombinierten LIPC/Kern-IPC Zyklus entsprechen den Erwartungen. Erstens ist die Kern-IPC auf dem LIPC Fiasco Kern langsamer als auf den unmodifiziertem Kern. Zweitens wird jedesmal vor der Kern-IPC der LIPC Nachbereitungscode durchlaufen, um die Threadumschaltung durch LIPC im Kern nachzuvollziehen, was extra Kosten verursacht.

Die zusätzlichen Kosten für eine Kern-IPC sind nicht unerheblich. Dies ist besonders durch extra Prüfungen im Assemblershortcut bedingt. Der Kern muss aufwendig prüfen, ob er beim Blockieren LIPC erlauben kann oder nicht. LIPC kann nur erlaubt werden, wenn der Thread ein offenes Warten, oder ein geschlossenes Warten auf einen Thread innerhalb des Adressraumes, beides ohne Timeout, durchführen will.

TLB Misses durch Kern UTCB Zugriffe spielen zwar auf neueren CPUs keine große Rolle, da der Kern auf die UTCB über eine global markierte Kernspeicher Abbildung im Kernadressraum zugreift. Dies erfordert aber eine zusätzliche Indirektion, um den UTCB Zeiger zu laden. Diese Zugriffsweise ist durch Fiasco-UX bedingt, wo der Kern in einem extra Adressraum läuft. Der Nutzer greift auf die UTCBs über ein eigenes Mapping zu.

5.3 Analyse des Nachbereitungscodes

Der LIPC Nachbereitungscode gliedert sich in zwei Teile. Der erste Teil stellt die Atomarität der LIPC Operation sicher, siehe 2.4. Wenn durch LIPC im Nutzermodus eine Threadumschaltung erfolgte, vollzieht der zweite Teil des Nachbereitungscodes die Umschaltung der Threads im Kern nach. Es gibt zwei Versionen dieses Teils, eine allgemeine Version, welche mit verschiedensten Stacklayouts zurechtkommt, und eine optimierte Version für den Assemblershortcut, wo das Stacklayout fest ist.

Um den Einfluss des Nachbereitungscodes auf die Unterbrechungslatenzzeit zu ermitteln wird ein möglichst schlechter Fall ausgemessen. Abb. 5.1 zeigt das Szenario. Thread A und B befinden sich im ersten Adressraum, sie bilden die Servertask. A nimmt die Anforderungen entgegen, reicht diese mittels LIPC an den Arbeitsthread B weiter, welcher dann die Antwort mittels Inter-Adressraum-IPC an den Klient C in einem anderen Adressraum weiterreicht. Es treten pro Zyklus zwei Inter-Adressraum-IPC und eine LIPC mit folgendem Nachbereitungscode auf.

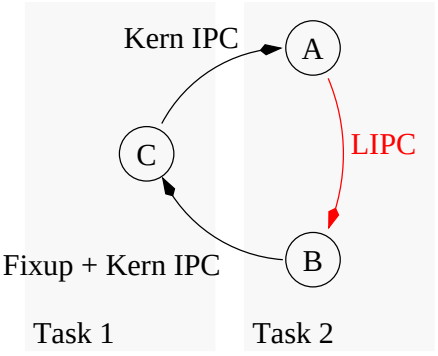

Abb. 5.1: Szenario LIPC Nachbereitungscode

	Pentium 100Mhz	K6-2 350Mhz
Fixupcode Restart-Block	28	31
Fixupcode Forward-Block	141	178
Copy-Stack, allg. Version	278	216
Copy-Stack, Shortcut-Version	229	163

“Fixup” bezeichnet den ersten Teil des Nachbereitungscodes, “Copy-Stack” den zweiten Teil.

Im schlechtesten Fall können bei einem asynchronen Hardwareinterrupt die Kosten für den ersten und zweiten Teil addieren. Dadurch entstehen nicht unerhebliche Latenzzeiten. Wenn man von 450 Takten für den schlechtesten Fall ausgeht, entstehen bei dem Pentium 4,5us und bei dem K6 1.3us Verzögerung. Ferner wird der zweite Teil vor jeder synchronen Unterbrechung wie Seitenfehler und besonders Systemaufrufen durchlaufen.

Eine Optimierung des Nachbereitungscodes wäre, sich von der festen Bindung KTCB zu seinem Kernstack zu trennen. Der Nachbereitungscode muss dann praktisch nur den Stackzeiger im KTCB verändern. Das kopieren des Kernstacks würde entfallen. Ferner muss auch kein IPC Stackrahmen auf dem “alten” Kernstack angelegt werden, da dort schon einer vorhanden ist.

5.4 Allgemeine Leistungsbewertung

LIPC ist sehr schnell, verteuert aber auch die normalen Kern-IPC Kosten. Um Abzuschätzen, ab wann sich LIPC lohnt, wird ein einfaches Anwendungsszenario, Abb. 5.2 ausgemessen. Der Server besteht aus einem Verteilerthread A, einen Arbeitsthread B sowie einen extra Thread C zur Synchronisation. Der Verteilerthread nimmt die Anforderung entgegen, reicht sie mittels LIPC an den Arbeitsthread weiter. Um nun die Performanz von LIPC auszumessen, kommuniziert der Arbeitsthread B noch mit dem Thread C. Dieser schickt die Antwort an den Arbeitsthread wieder mittels LIPC zurück. Die Anzahl dieser Anforderungen an Thread C wird von Null schrittweise auf Neun erhöht. Wenn der Arbeitsthread fertig ist, schickt er mittels Kern-IPC die Antwort zurück an den Klient D. Zusammengefasst erfolgen so pro Zyklus zwei Inter-Adressraum-Kern-IPCs, ein LIPC “Forward”, Null bis 2*9 LIPC Operationen mit Tread C, und mindestens einmal wird Nachbereitungscode durchlaufen. Zum Vergleich wird dieses Szenario ebenfalls komplett mit Kern-IPC ausgemessen, auf dem LIPC und dem original Kern.

Die Ergebnisse, siehe Abbildung 5.3 und 5.4, zeigen deutlich, dass in diesem Szenario sich ein LIPC Kern nur bei mehr als fünf LIPC pro Zyklus lohnt. Wenn es weniger sind, lohnt sich der Overhead eines LIPC Kerns nicht.

Überraschend ist, dass selbst bei zwei Inter-Adressraum-IPC und einer LIPC, die Kosten sehr hoch sind. Wenn man sich die reinen Inter-Adressraum-IPC Pingpong Kosten anschaut, sind die extra Kosten durch diese eine zusätzliche LIPC gewaltig. Dies ist dadurch bedingt, dass zusätzliche TLB Misses entstehen, wenn der LIPC-Code im Nutzermodus auf die UTCB und den aktuellen UTCB Zeiger zugreift. Außerdem wird jedesmal pro Zyklus der LIPC-Nachbereitungscode des Kerns durchlaufen.

Eine Möglichkeit die zusätzlichen Kosten für Kern-IPC zu vermeiden, ist normale Kern-IPC und LIPC als disjunkte IPC Operationen zur Verfügung zu stellen. Dies heißt, ein

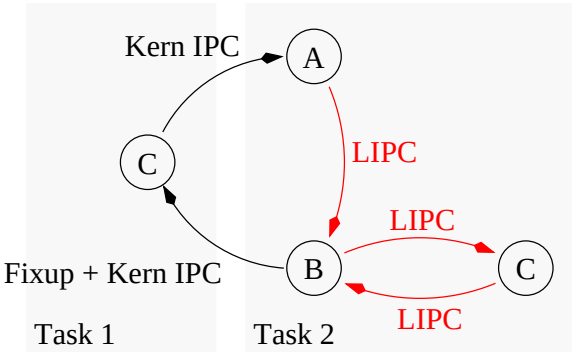

Abb. 5.2: Anwendungszenario

Thread kann nicht mit LIPC zu einem in Kern-IPC stehenden Thread Nachrichten schicken und umgekehrt. Dadurch fallen im Assemblershortcut für Kern-IPC extra keine Kosten an, der Shortcut wird automatisch abgebrochen, sobald der IPC Empfänger LIPC bereit ist. Ferner fallen auch keine zusätzlichen Kosten für den Empfangsteil des Kern-IPC Shortcuts an, da der blockierende Thread nicht für LIPC freigeschaltet werden muss.

Der Nachteil ist, dass dann kein atomares LIPC “Forward” möglich ist, welches ein Verteilerthread zur Kommunikation mit einem Arbeitsthread nutzen kann. Dies ist dadurch bedingt, dass der Verteilerthread mittels LIPC in den LIPC Wartezustand übergeht aber nichtmehr von Klienten außerhalb des Adressraumes mit Kern-IPC erreichbar ist.

Der Kern muss spezielle IPC Operationen für LIPC bereitstellen, welche die notwendigen UTCB Zugriffe durchführen. Durch dies entsteht kein zusätzlicher Aufwand bei normalen Kern-IPCs. Die Kosten für LIPC Nachbereitungscode bleiben bestehen und ein Server muss diese Kosten akzeptieren, wenn er LIPC verwendet.

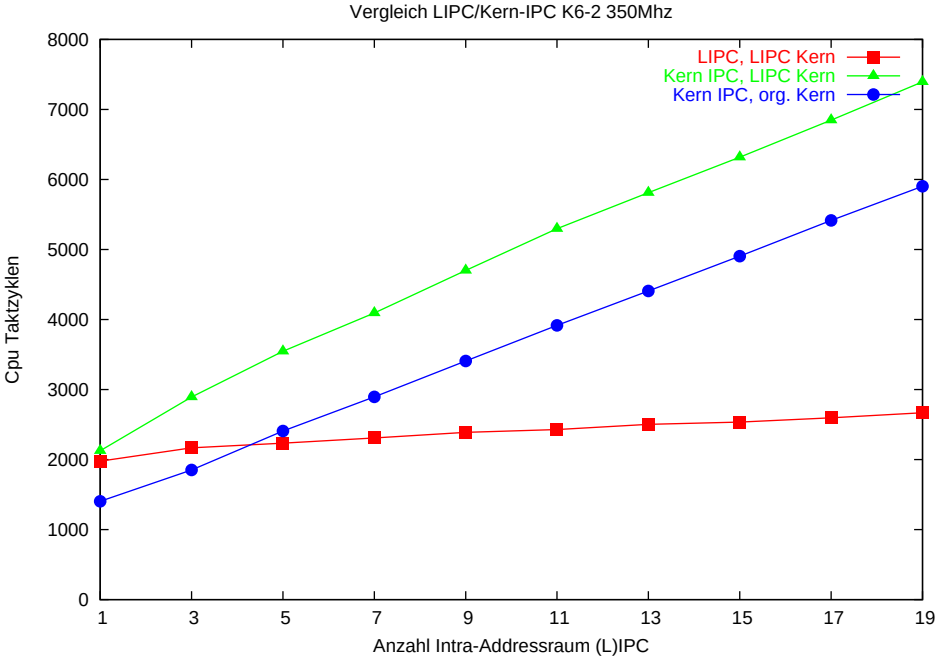

Abb. 5.3: Vergleich LIPC/Kern-IPC K6-2 350Mhz

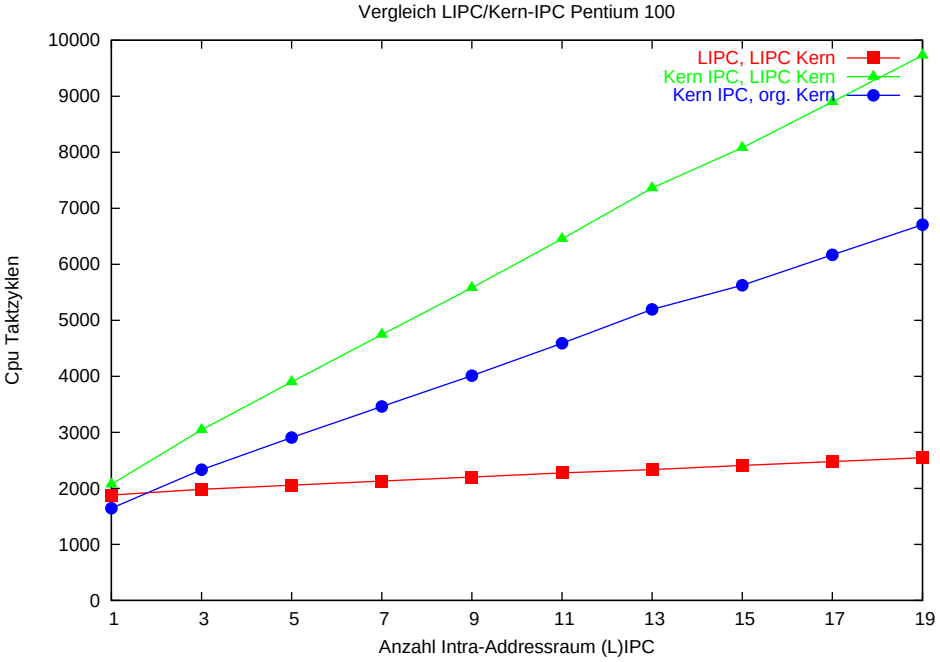

Abb. 5.4: Vergleich LIPC/Kern-IPC Pentium 100Mhz

6. Zusammenfassung und Ausblick

LIPC ist eine sehr schnelle Möglichkeit für Threads eines Adressraumes, untereinander zu kommunizieren, und kann sogar mit den IPC-Kosten von Nutzerthreads konkurrieren. Dadurch können Applikationen mit vielen Threads erheblich profitieren und es erlaubt schnelle Synchronisationsprimitive.

Die Nachteile eines LIPC Kerns aber sind erheblich. Der LIPC Kern Assemblershortcut ist zwar wenig optimiert, aber es zeigt sich, dass die Kosten für Kern-IPC erheblich steigen. Ferner vergrößern sich die Latenzzeiten bei Hardwareunterbrechungen, was besonders für einen Echtzeit-Kern kritisch ist.

Die Messungen zeigen auch deutlich, dass sich ein LIPC Kern erst bei mehr als vier bis fünf LIPCs zwischen jeder Inter-Adressraum-IPC lohnt. Sonst sind die zusätzlichen Kosten zu hoch. Wenn nur sehr wenige Prozesse LIPC nutzen, wird das Gesamtsystem sogar signifikant langsamer.

Eine Untersuchung eines realen Anwendungsfalles, wie zum Beispiel die Sempahorbibliothek von DROPS [1], steht noch aus. Der entstandene LIPC Kern unterstützt problemlos L4Linux, und eine Untersuchung, ob L4Linux von LIPC profitieren kann, ist auch eine interessante Frage. Unbetrachtet blieb auch der Aspekt von LIPC auf SMP Maschinen. Auch spielen auf auf neuen CPUs schnelle Maschinenbefehle um den Kern zu betreten, wie “sysenter” auf Intel CPUs und “syscall” auf AMD CPUs, eine große Rolle und der LIPC Kern sollte auch diese unterstützen.

[1] Dresden Realtime Operating System

Glossar

ABI Application Binary Interface

CPU Central Processing Unit, der Hauptprozessor

DROPS Dresden Realtime Operating System

IA-32 Intel 32-Bit Architektur, auch x86

ID Bezeichner, engl. Identifier

IPC Inter Process Communication

KIP Kernel Information Page

KTCB Kernel Thread Control Block

LDT Local Descriptor Table

LIPC Local Inter Process Communication

FPU Floating Point Unit

TCB Thread-Control-Block

TSS Task State Segment

UPC User Programm Counter

USP User Stack Pointer

UTCB User Thread Control Block

Literatur

[Dannowski u. a. 2004] DANNOWSKI, U. ; LEVASSEUR, J. ; SKOGLUND, E. ; UHLIG, V.: L4 eXperimental Kernel Reference Manual, Version X.2. 2004. – Forschungsbericht. Latest version available from: `http://l4hq.org/docs/manuals/`

[Hohmuth 1998] HOHMUTH, Michael: The Fiasco Kernel: Requirements Definition / TU Dresden. Dezember 1998 (TUD–FI–12). – Forschungsbericht. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`

[Hohmuth 2002] HOHMUTH, Michael: The Fiasco Kernel: System Architecture / TU Dresden. 2002 (TUD-FI02-06-Juli-2002). – Forschungsbericht

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

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

[Liedtke 1999] LIEDTKE, J.: L4 Nucleus Version X Reference Manual (x86). September 1999. – Forschungsbericht

[Liedtke und Wenske 2001] LIEDTKE, J. ; WENSKE, H.: Lazy Process Switching. In: *8th Workshop on Hot Topics in Operating Systems (HotOS)*, 2001, S. 15–20

[Peter 2002] PETER, M.: *Leistungs-Analyse und -Optimierung des L4Linux-Systems*. 2002

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

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

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

[Wenske 2002] WENSKE, H.: *Design and Implementation of Fast Local IPC for the L4 Microkernel*. 2002