# Design and Implementation of Fast Local IPC for the L4 Microkernel

Study Thesis/Studienarbeit

Horst Wenske
System Architecture Group
University of Karlsruhe, Germany
Horst.Wenske@h-wenske.de

Supervisors: Dr. Kevin Elphinstone and Dipl.Inf. Uwe Dannowski

July, 2002

**Abstract**

This work extends the paper "Lazy Process Switching" [1]. It describes a concrete prototype implementation of fast Local Inter Process Communication for the L4 version X.0 microkernel. The main idea behind LIPC is to extend kernel-level threads with user-level characteristics, which means user-level IPC with kernel-level threads is in special cases possible. The necessary synchronisation between the different user and kernel-thread context can be done lazily when the kernel entry is inevitable (e.g. timer interrupt). This L4 LIPC prototype implementation needs 23 cycles for a short intra-address-space IPC between two threads, what is about 8-13 times faster than a normal L4 short IPC. LIPC is fast enough to reconsider the usage of IPC for fine-grained synchronisation and multi-threaded servers in the same address space.

# Acknowledgements

My thanks go to Dr. Kevin Elphinstone and Uwe Dannowski for supporting and supervising my thesis as well for their insights and discussions.
Most of all, my gratitude go to Uwe Dannowski and Andreas Haeberlen who helped me several times to debug my source code.
Finally, I thank all who supported my work in one way or another[1].

[1] Even a Starcraft game can sometimes help to find a bug ...

# Contents

# 1 Introduction

## 1.1 Motivation

Although Inter Process Communication (IPC) performance has improved in the last five years, it is still too slow on certain processors, for instance Intel IA-32 based processors. Critical sections in real-time applications and multi-threaded servers, two examples motivating even faster IPC, are briefly discussed below.

*Critical sections* in real-time applications suffer from the well-known priority-inversion problem [10]. Multiple solutions have been proposed, e.g. priority inheritance (which is generally not sufficient), priority ceiling [10] and stack-based priority-ceiling [9]. All mentioned methods need to modify a thread's priority while the thread executes the critical section. In the stack-based priority-ceiling protocol, for example, a thread has to execute the critical section always with the maximum priority of all threads that might eventually execute the critical section, regardless of its original priority.

A very "natural" solution for a stack-based priority ceiling in a thread/IPC-based system is to have a dedicated thread per critical section. This thread's priority is set to the (static) ceiling priority. Any "client" thread calls the critical section through a Remote Procedure Call (two IPCs). Priorities are automatically updated through the underlying thread switch. The synchronous IPC mechanism also serialises threads automatically that compete for the critical section. Provided that simultaneously pending request IPCs are delivered in prioritised order, we have a simple and elegant implementation of stack-based priority ceiling.

For achieving highest performance, *multi-threaded servers* often need customised policies to distribute incoming requests to worker threads. For instance, a server might want to handle up to three requests in parallel, but queue further requests. The "natural" solution is to use one distributor thread which also implements a request queue, and three worker threads which communicate via IPC with the distributor thread.

All these methods require a fast IPC mechanism which does not degrade system performance. The current L4 microkernel intra-address-space IPC time on a Pentium III is about 100-200 cycles, which can be too expensive for the

mentioned examples. Such costs are only acceptable when real synchronisation actions are necessary. For example, entering the invoker thread into a wait queue if the critical-section thread is blocked on a page fault. These IPC costs (100-200 cycles) are relatively low in comparison to the overall consumed time.

This works aims at providing the first prove of concept LIPC prototype implementation. The LIPC prototype should give new insights implementing LIPC and provide a foundation for further LIPC implementations

## 1.2 Synopsis

This thesis is organised in eight main chapters. Following this introduction, Chapter 2 briefly describes the terms being required for understanding this work. In Chapter 3 related approaches are introduced and Chapter 4 presents the necessary theory and algorithms behind LIPC. Chapter 5 points out certain aspects of the implementation and Chapter 6 discusses performance. Finally, Chapter 7 summarises this work and Chapter 8 discusses the scope for further work.

# 2 Background

An extremely short and surely incomplete overview of the microkernel approach, threads, tasks and IPC is given in this chapter. A special focus is set on the IPC section which describes the most important L4 IPC specifics.

## 2.1 The Microkernel Approach

The following microkernel introduction is (mostly) taken from J. Liedtke [7] and inspired by S. Wagner study thesis [16].

The key idea of microkernels is to keep the kernel minimal. Ideally, all Operating System (OS) services which can safely and securely be implemented outside the kernel are implemented as user-mode servers in their own address space. The microkernel approach reduces the kernel's size to only mandatory kernel parts and protects the OS services from each other and the users.

It is possible to introduce OS services that are not necessarily fully trusted by every user and by any other OS service (which is not possible with the monolithic kernel approach).

The central idea of this approach is the *address-space paradigm*. An address space on the hardware level is a mapping between virtual pages and physical frames. Any server and any user-mode program has its own protected address space. The kernel, preventing corruption of address spaces, controls all address space changes.

For example, a file system service and the network service (like TCP/IP) reside in different address spaces so that they are isolated and protected from each other. As a consequence, the microkernel has to supply user and system services (there is no difference from the kernel point of view) with a cross-address-space communication facility. This key mechanism is usually called *Inter Process Communication (IPC)*.

But the microkernel approach has also some other advantages. Different Application Program Interfaces (API), file systems and OS personalities can coexist in the system. Furthermore, server malfunctions are isolated like normal application malfunctions, the system structure is modular and easy to maintain.

The L4 [17] is a second generation microkernel developed by Jochen Liedtke at the GMD, IBM and the University of Karlsruhe. Currently there are three implementations of L4 for x86 available [17, 18] which differ in their implementation and API. Additionally, there are implementations available for DEC-Alpha from the Dresden University of Technology and for MIPS from the University of New South Wales. L4 is one of the smallest and fastest microkernels developed so far.

## 2.2 Threads and Tasks

A thread is an activity, being characterised by some kind of state information and an associated address space. Typically, a thread is represented by a sequence of code that is operating as a unit on behalf of a single user or transaction. Each thread has an execution state (running, ready, etc.), saves its context (e.g. instruction pointer) when not running, and has its own storage for local variables and execution stack. Threads have shared access to the address space and resources (files etc.) of their task.

*A task* is the entirety of exactly one address space and all threads executing within this address space. Moreover a task is also a protection domain for IPC.

*User-Level threads* are handled by a *thread library* in user-space (Many-to-One model). The kernel is not aware of the existence of user-level threads and is not involved in a user-level thread switch. The thread library contains the code for creating and destroying user-level threads, passing messages and data between them, saving and restoring thread contexts and scheduling the thread execution.

For *kernel-level threads* (One-to-One model) all thread management is done by the kernel. An API to the kernel thread facility is required. The kernel maintains context information for the task and its threads so that switching between kernel-level threads requires the kernel.

Threads are sometimes described in terms of their weight, meaning how much contextual information has to be saved for a given thread so that it can be referred to by the system during the life of the thread. The context of a traditional Unix process, which is one task with exactly one thread, includes the hardware registers, the kernel and user stack of the process, etc. The time required to switch that much context is considered large so that an Unix process is said to be a *heavyweight thread*. In some modern operating system kernels, such as L4, multiple threads can exist in a single address space, which decreases the amount of context information that has to be saved with each, and reduces the switching time. These kernel-level threads are considered to be *middleweight threads*. When all necessary context and thread operations are exposed to the user-level, each application needs only the minimal amount of context saved with it so that context switching can be reduced to a minimal amount of cycles. Therefore, user-level threads are considered *lightweight threads*.

## 2.3 Inter Process Communication

This IPC introduction is restricted to L4 specifics since IPC in general is a wide field. IPC is used for data transfer, event notification and synchronisation. Thread communication via IPC is the fundamental feature of the L4 kernel. During the IPC, there is an agreement between the sender and the receiver. The sender decides which information will be sent, and the receiver

can decide whether to interpret and accept the received information. An IPC will only take place if the receiver has agreed, i.e. the receiver is in the state "receiving from sender thread".

IPC delivers a message from one thread to another. L4 supports several types of IPC:

- Short IPC
- String IPC
- Map and grant

Rich types help to improve end-to-end IPC performance.
A *short IPC* transfers data in general-purpose registers. On the IA-32 platform, up to three 32 bit words can be transferred as a register message. As no extra copy operations between address spaces are necessary (all data is being held in registers), short IPC has the lowest IPC costs, totalling 100-200 cycles on a Pentium III 450 MHz.
An IPC memory message consists of a string (direct string IPC) up to two megabytes, which can be used to copy longer messages from the sender's to the receiver's address space. This message transfer needs copy operations, and the kernel might establish a temporary mapping (map and copy) to reduce the copying costs [4].
A *direct string IPC* can transfer up to 31 indirect strings in a memory message. An indirect string consists of a base address and a size and specifies a buffer in the sender's address space. It avoids unnecessary copy operations to/from the message buffer. On the receiver side, buffers for such strings can be specified so that the IPC can transfer directly from the sender to the receiver. *Scatter/Gather flags* permit strings to be gathered on the sender side and/or scattered on the receiver side. Thus multiple blocks can be directly transferred to a single receiver buffer; a single send buffer can be split into multiple blocks.
Map messages map pages or larger parts of the sender's address space, called a flexpage, into the receiver's address space; a flexpage consists of a base and a size of a memory region. By invoking a *map operation,* a thread can make a memory region in its address space accessible to another address space, provided the recipient thread agrees. The *grant operation* is a specialisation

of the map operation. By granting a flexpage, the pages are mapped to the grantee's address space, but in contrast to the map operation, the mapped pages are removed from the granter's address space. The granter has no longer access to the granted pages. Map and grant enable user-level pager and main memory management on top of the kernel. Special communication mechanisms based on shared memory regions can also be constructed. Further explanations can be found in the microkernel lecture notes [7].
LIPC is a special short IPC optimisation in the case two threads in the same address space communicate via IPC. Since both threads are in the same address space, there is no need of real data transfer via IPC. Both threads can access the same data and destroy each other if they like.

The atomic IPC system call can be parameterised as *send*, *call*, *receive* and *atomic send and receive*. Atomic send and receive means in this context that both operations are done atomically and cannot be interrupted after the send phase. An *open* (receive from any thread) or *closed wait* (receive only from a specified thread) can be specified (see [6]).

L4 supports only *synchronous IPC*; after invoking an IPC operation, the sender thread blocks until the message has been transferred, the IPC has been aborted or the given timeout has expired.

The IPC in L4 version X.0 allows various timeouts. A send, a receive, a sender page fault and a receiver-page-fault timeout. The *send timeout* determines how long IPC should try to wait for the receiver to become ready. The *receive timeout* specifies how long IPC should wait for an incoming message. Both timeouts specify the maximum period of time before message transfer starts. Once started, message transfer is no longer influenced by send or receive timeouts.

Page faults (e.g. when a page is not present in memory) that occur during IPC are controlled by send and receive page fault timeouts. A page fault is translated to an RPC (two IPCs) by the kernel. In the case of a page fault in the receiver's address space, the corresponding RPC to the pager uses the send-page-fault timeout (specified by the sender) for both send and receive timeout. In the case of a page fault in the sender's address space, receive-page-fault timeout specified by the receiver is taken. A timeout can be specified as 0 (do not wait), $\infty$ (wait forever) or time periods from $1\mu s$ to approximately 19 hours. The timeouts are necessary to avoid denial of service attacks, e.g. a malicious pager or a non responding sender/receiver.

# 3 Related Work

Speeding up the IPC is not a new topic. There are several papers available which discuss new techniques to speed up IPC [4, 13, 11, 12]. IPC is a valuable mechanism for structuring complex systems, as it allows systems to be decomposed. In addition to being a good structuring mechanism, IPC is a fundamental primitive in distributed and parallel computing; the performance of distributed systems and parallel programs is often determined by the performance of IPC primitives. To develop efficient programs, the IPC performance is the determining factor how far it is possible to decompose a program. Usually, there are two different approaches to implement threads, which has direct consequences on the IPC performance. Threads can be supported either at user-level or in the kernel. Both approaches have several advantages and disadvantages.

## 3.1 Middleweight-Kernel-Level Threads

The kernel-level thread approach is a nice concept, but suffers normally from poor performance. In the last years new techniques have been presented to overcome the poor performance [4, 15, 12]. Some of the new approaches try to reduce the complexity and message transporting costs of IPC. With optimised assembler code, the L4Ka achieves an IPC performance of about 100-200 cycles, which is a real milestone compared to old IPC performance values.

## 3.2 User-Level Threads

*User-level threads* might achieve the required speed, but don't have the nice semantics of a kernel-level thread. Furthermore, making user-level threads able to run in the kernel schedule, more than compensates their speed gain [2]. Having two concepts, kernel and user-level threads, is conceptually inelegant and contradicts the idea of conceptual minimality. There are several approaches [13, 15, 14] to overcome the disadvantages of user-level threads. For example, the Solaris user-level threads in the "green thread library" where several kernel-level threads are mapped to a group of user-level threads to reduce IO-blocking times.

# 4 Design

This chapter points out the design goals. After inspecting an intra-address-space thread switch, the first subsection identifies the problems of a local IPC implementation. The following subsections present the proposed solutions and mechanisms to overcome the identified problems.

## 4.1 Concepts and Problems of LIPC

In all the above mentioned examples, we need a very fast IPC, particularly for intra-task communication which does not include an address-space switch. Therefore, the goal is to find an implementation of kernel-level threads that offers all speed advantages of user-level threads for intra-task communication. The main idea is to extend kernel-level threads with user-level characteristics, which means user-level IPC with kernel-level threads is in special cases possible. The necessary synchronisation between the different user and kernel-thread context can be done lazily when the kernel entry is inevitable (e.g. timer interrupt).

To find a way to extend kernel-level threads with user-level IPC characteristics, let us revisit an intra-address-space thread switch for an atomic SendAndWaitForReply IPC, which is typically used for RPC. Client and server variant of this call differ only marginally. The client thread sends a request to a server thread and waits for a reply from the server. Correspondingly, the server thread replies to the client thread and waits for the next request which may arrive from any client.
The client variant:

```
Thread A → Thread B
call IPC function, i.e. push A's instruction pointer;
IF B is a valid thread id AND thread B waits for thread A THEN
   save A's stack pointer;
   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
```

Analysing the system call above, you can recognise three main problems.

#### Kernel Data Access

Stack pointer, thread status and "current thread" are protected data that can only be accessed by the kernel to prevent user-level code from compromising the system. The kernel data has to be made somehow controlled accessible, without compromising the system.

#### Atomicity

Checking B's state and the following thread switch have to be executed atomically to avoid inconsistencies. Normally, any user-level code can be interrupted, e.g. at an end of a time slice or a hardware interrupt, which can cause inconsistencies. The LIPC system call must prevent any exception or the exception must be properly handled.

#### Inconsistent User/Kernel Thread Status

Using LIPC, the thread status, user instruction and user stack pointer in user-space can differ from saved values in the kernel thread control block. The LIPC implementation has to provide user/kernel status synchronisation to ensure consistency.

### 4.2 User/Kernel Thread Control Block

Normally, each thread has its own *Thread Control Block* (TCB) in kernel space, which contains the whole context of the thread. A general applicable technique to make kernel protected variables of the TCB user-accessible is based on the idea to have a kernel twin of the *Kernel Thread Control Block* (KTCB) in users-pace, the so called *User Thread Control Block* (UTCB), for each unprotected variable. In this vein the LIPC code can access and modify

the stack pointer or the status of a thread or others, without accessing the kernel space. Before the unprotected variable is used by the kernel, the kernel always checks for consistency. If the unprotected variable and kernel twin do not match, the kernel takes the appropriate actions to reestablish consistency.

## 4.3 General Structure of LIPC

Ensuring atomicity in user mode is relatively simple as long as the kernel knows the executed code. The method goes back to an idea that Brian Ford [3] proposed in 1995: "Let some *unmodifiable 'kernel code' execute in user space* so that the kernel can act specifically to this code if an interruption occurs within this 'kernel code' ".

In the LIPC system call, the kernel resets the thread's instruction pointer to the beginning of the IPC routine *( restart point)* if an interrupt occurs before a real status modification has become effective. After the system state has been partially modified, the kernel would have to either undo those modifications or complete the LIPC operation before handling the interruption. Such a method cannot ensure atomicity in general; e.g., it fails if the code experiences a page fault. That's why the LIPC system call code is always mapped (mapped read-only kernel memory in user-space).

```
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 thread 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 ...
```

Interruption including page faults between *restart point* and *forwarded point* occur before the system's state has really changed. Provided that no required

registers have been overwritten, there should be no problem. If a page fault occurs when (U)TCB B is accessed to check B's status, the IPC operation simply restarts at the restart point after page-fault handling. We assume that *after* the forward point, no legal page faults can occur since both UTCBs have been accessed in the check phase. However, illegal page faults might occur, e.g. if a user program jumps directly to the middle of the code or even to the middle of an instruction. Consequently, any page fault in this region is illegal and permits to kill the thread. If a "normal" interruption between forward point and *completion point* occurs, the missing instructions are executed, and the instruction pointer is set to the completion point:

```
interruption between FORWARD POINT and COMPLETION POINT:
   IF is page fault THEN
      kill thread A;
   ELSE
      A's status := "wait for B";
      B's status := "run";
      load B's stack pointer;
      current thread := B;
      set interrupted IP to completion point;
   ENDIF
```

The only critical part of the LIPC system call is the section between forward and completion point, where the status of the thread is being changed. The LIPC implementation must ensure that any interruption, e.g. timer interrupt in this section causes a "roll forward" to the completion point. If an interruption after the completion point occurs, the IPC operation simply restarts after interruption handling.

On a uniprocessor, we have thus guaranteed atomicity without using privileged instructions. For multiprocessor, the method can be extended to work for threads residing on the same processor. Anyhow, cross-processor communication is currently an order of magnitude more expensive than intra-processor communication. Restricting user-level IPC to intra-processor is thus acceptable.

## 4.4 LIPC Inconsistency Detection

The fundamental insight is that twin inconsistencies need only to be checked on kernel entry. This sounds obvious, however, its immediate consequence is that an IPC executing completely in user-level *does not need to synchronise with the kernel.*

In particular, this type of IPC can switch threads, without directly telling the kernel. The kernel will synchronise, i.e. execute the thread switch in retrospect upon the next kernel entry, e.g. timer tick, device interrupt, cross-address-space IPC or page fault.

In general, lazily-evaluated operations pay if more of them occur than have to be effectively evaluated, i.e. real work phases are short. Correspondingly, lazy switching can pay if only a small fraction of lazy-switching operations lead finally to real kernel-level process switches. Such behaviour can be expected whenever a second IPC, for example the reply or a forwarding IPC, happens before an interrupt occurs. The motivating examples "critical section" and "request distribution" fall into this category provided that their real work phase is short.

The LIPC prototype implementation has to detect a possible inconsistency caused by LIPC. Therefore, an unprotected kernel variable $CurrentUTCB_u$ which is intended to point to the current thread's UTCB and its protected kernel-space twin $CurrentUTCB_k$ are introduced. The only variables that trigger synchronisation is $\text{CurrentUTCB}_{(u/k)}$. Inconsistencies that include only *status* changes are ignored because they are always illegal. Due to lazy scheduling [4], status inconsistencies can be tolerated.

```
CurrentUTCB inconsistency:
  IF CurrentUTCB_u is in valid UTCB region THEN //predefined
     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 ;
           return;
     ENDIF
  ENDIF
```

If a CurrentUTCB inconsistency is detected (CurrentUTCB$_u$≠CurrentUTCB$_k$) during the kernel entry, the above algorithm will perform a CurrentUTCB cross check, which verifies whether the CurrentUTCB$_u$ is valid and sets the new CurrentKTCB and CurrentUTCB$_k$ values.

## 4.5 Lazy Update and Thread Status Fixing

Inconsistency should only be caused via a performed LIPC. When the LIPC prototype implementation detects a CurrentUTCB$_{(u/k)}$ inconsistency, it has lazily to update the thread states. The algorithm below describes how to update the kernel thread status:

```
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
```

In principle, the algorithm synchronise the kernel and the user state of both threads, which have an inconsistent user-level vs. kernel-level state, and updates the run queue.

The algorithm can be straightforwardly extended to handle more thread states than only “run” and “wait for X” whereby it is flexible for different kinds of thread states.

## 4.6 Status Inconsistency Problems

Figure 1 illustrates a normal IPC and an LIPC chain.

Both scenarios have the (L)IPC chain A→B→C, which is another term for an (L)IPC sequence. When you compare both IPC types, you can recognise

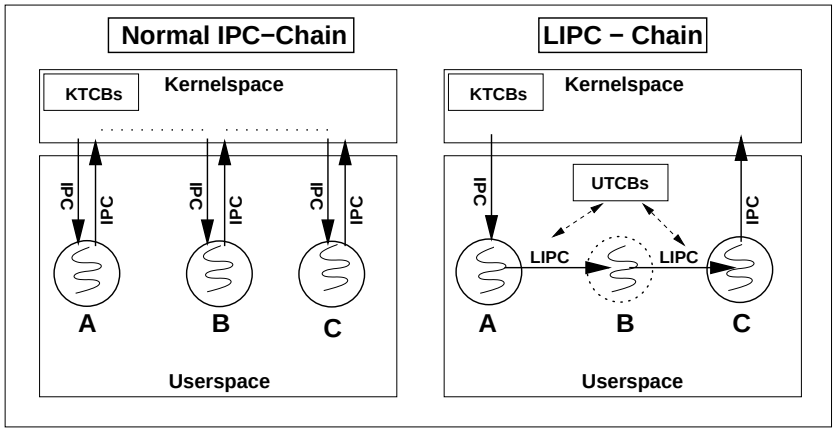


Figure 1: Normal IPC and LIPC Chain

several differences. Using the "normal" IPC you have three kernel entries and save all thread information only in the Kernel TCBs. All IPCs are visible from the kernel perspective and synchronised with the TCBs.

In the LIPC case, LIPC performs no kernel entry and is only synchronised with the UTCBs. From the kernel perspective thread A is running all the time, and when thread C does a normal IPC, the kernel detects an inconsistency and recognises that thread C is actually running. From the LIPC chain A→B→C the kernel can only detect the end points A and C, but that thread B was running in the middle and has changed its state, stack and instruction pointer is for the kernel not visible. We have to revisit all possible undetected changes in an LIPC chain and the consequences for the whole system. The changes are not really invisible for the kernel. It can access the UTCB of thread A and can read the current state, instruction pointer etc. The only problem is that the kernel doesn't detect that the state of B has changed and that the KTCBs and some queues (e.g. send queue) must be updated.

The modification of the stack and instruction pointer of thread B is not critical for LIPC since for further LIPCs the current values of the UTCBs are used. Normal IPC has now the problem that the values in the KTCBs don't necessarily contain the current values (modified via LIPC). That's why it must be ensured that the current values in the UTCBs are used directly. The KTCB access must be redirected to the current UTCB values. Only redirecting KTCB accesses has the problem that some kernel lists, queues,

etc. cannot be properly updated because no changes of thread B are detected. All these hidden updates concern only the state of the "hidden" thread B, that's why we must revisit all possible kernel-hidden state changes. The state changes of the UTCBs of thread A and C are not a problem since their kernel states can be updated during the kernel entry (see 4.5).

Let us focus on the possible "undetected" status changes of thread B. Figure 2 illustrates all possible state changes of thread B. Before and after an LIPC thread B must be in the receiving state. But the type of the receiving state can change. Thread B could have been in one of three possible receiving states (before it became active):

- Closed wait on thread A (only thread A can send an IPC to B)
- Open local wait[2] (any local thread in the same address space can send an IPC to B )
- Open global wait (any thread can send an IPC to B)

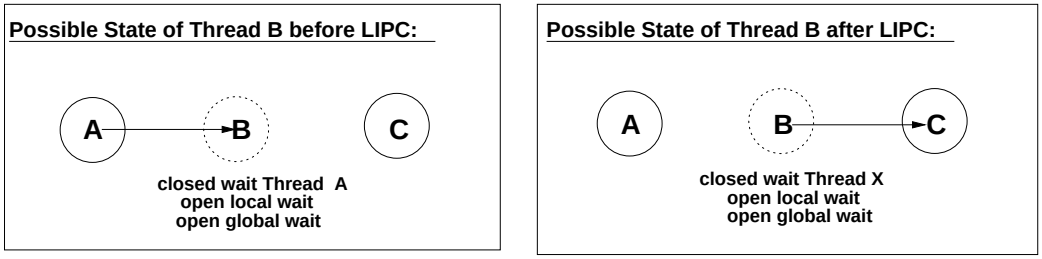


Figure 2: Invisible Status Changes of Thread B

After B has performed the LIPC, there are three possible receiving states:

- Closed wait on thread X (only a specified thread X can send an IPC to B)
- Open local wait (any local thread in the same address space can send an IPC to B)
- Open global wait (any thread can send an IPC to B)

Not detecting these receive status changes from thread B can cause some problems.

[2]Open local wait is not implemented in this LIPC prototype.

### 4.6.1 Closed/Open Send Queue Problem

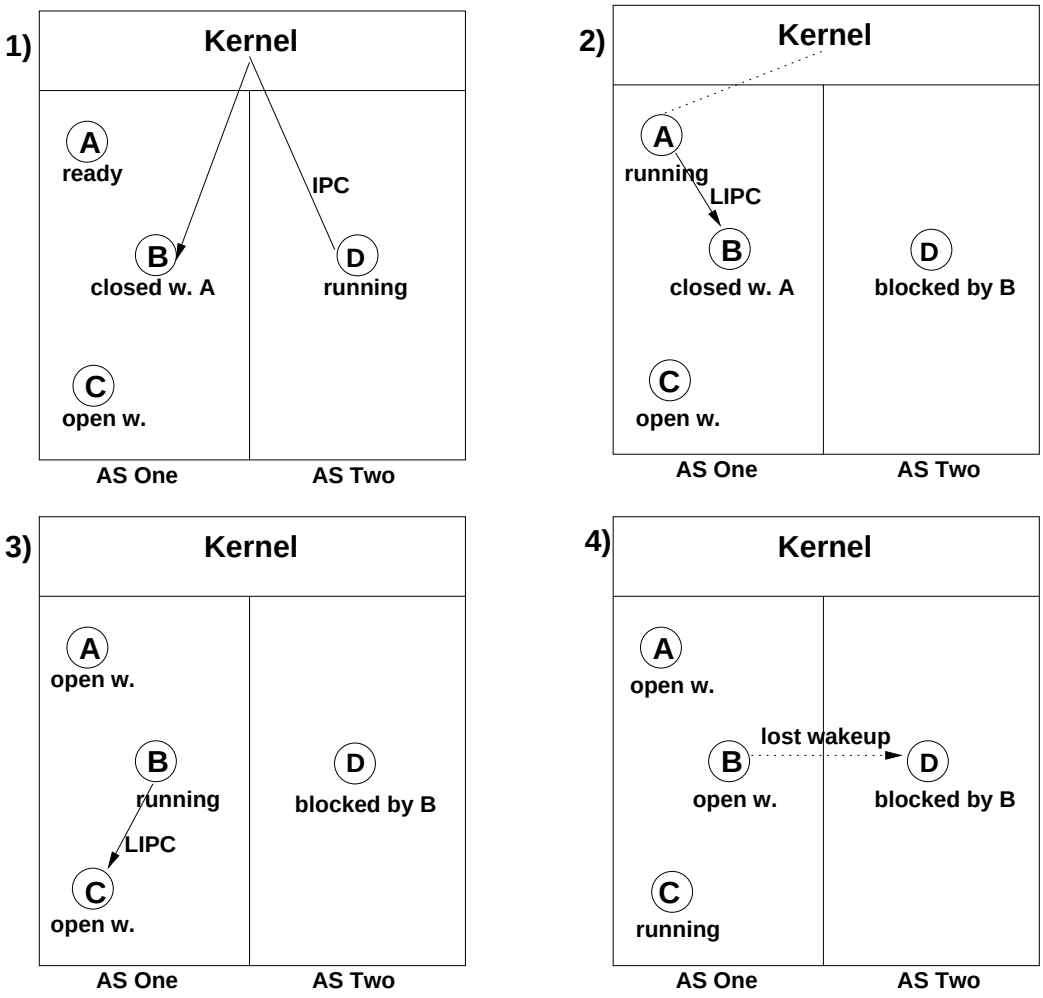


Figure 3: Closed/Open Send Queue Problem

Figure 3 illustrates a scenario which is called the "Closed/Open Send Queue Problem".

In the first picture, there are three threads (A,B,C) in address space one, and thread D in address space two. Thread D tries to send B a message, but this is blocked since thread B is on a closed-wait state for thread A. In the second picture, thread A starts running and sends an LIPC to thread B. In picture three, B sends an LIPC to thread C and changes its status to open receive. In picture four, thread C is running and thread B is in an open wait receiving state. The problem is that thread D has not detected the status

change of thread B, and B can not wake up thread D since it doesn't know that thread D is in "send pending" state on thread B. The message is not delivered although it would be possible and would be done in the normal IPC case.

In general, there are two cases where the receive state change is dangerous:

- Closed wait $\longmapsto$ open local/global wait
- Open local wait $\longmapsto$ open global wait/closed wait to the pending thread
- Open local wait $\longmapsto$ closed wait to a pending thread which has been waiting to send this thread a message.

In the for-mentioned cases, the receive state change can make new IPCs possible. There are several possible solutions to solve this problem.

#### Restrictive "Close/Open Send Queue Problem" Approach

For this approach the old receive state of the LIPC sender must be saved (e.g. in its UTCB). On each LIPC it must be checked if the sender thread will change to a different receive state compared to its old receive state. Since LIPC can only be used for call (see 5.6.1), there can only be transitions from one receive to another receive state. If the LIPC causes a different receive state, the sender thread has to abort the LIPC and perform a normal IPC, where it synchronises with the kernel.

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

This approach is relatively easy to implement and prevents the "Close/Open Send Queue Problem" since hidden-receive state changes are not possible. The second advantage of this approach is status consistency with the kernel. Before receiving an LIPC and after an issued LIPC, the thread has the same kernel-level and user-level state.

### Pending Send Approach

The latter approach is sometimes overly restrictive. For instance, there would be no problem to allow a state change, e.g. from closed to open wait if no thread is waiting to send to the LIPC thread. To implement the Pending Send Approach, we have to add a pending_send_counter to every UTCB. Its initial value is zero, and every time a sender is blocked by this thread, the pending_send_counter is incremented. When a pending IPC is executed, the pending_send_counter of the receiver is decremented. As soon as a thread tries to send an LIPC, its UTCB pending_send_counter is checked. If the value is greater than zero, the thread has to abort the LIPC and perform a normal IPC. In the other case the thread can deliver the message via LIPC. The gain is that the LIPC is only aborted if a thread exists which is in the state pending_send for the LIPC performing thread. The drawback of this approach is that the user-level state of a non running thread in can differ from its corresponding kernel-level state. That's why the current state of a thread can only be checked in its UTCB. While performing an inter-address-space IPC, the kernel has to access an UTCB in another address space to get the current state of the receiver thread, which usually causes costly cache and translation look-aside buffer misses.

## 4.7 Coprocessor Synchronisation

Several modern processors can save and restore floating-point registers and those of other coprocessors lazily. Those resources can be locked by the kernel. If another thread tries to access them, an exception is raised that permits the kernel to save the coprocessor registers in the TCB which used the coprocessor so far and reload the registers from the current TCB. Typically, coprocessors can only be locked by kernel-mode software.

LIPC can cause coprocessor confusion which would not occur using normal IPC. Let us assume thread A tries to access the coprocessor and performs an LIPC to thread B which also tries to access the coprocessor. The first time thread A accesses the coprocessor an exception arises (the coprocessor was locked), and the kernel saves the coprocessor registers in the TCB which has used the coprocessor so far. The coprocessor is now unlocked and thread A is working with the coprocessor (modifying floating-point registers). Thread

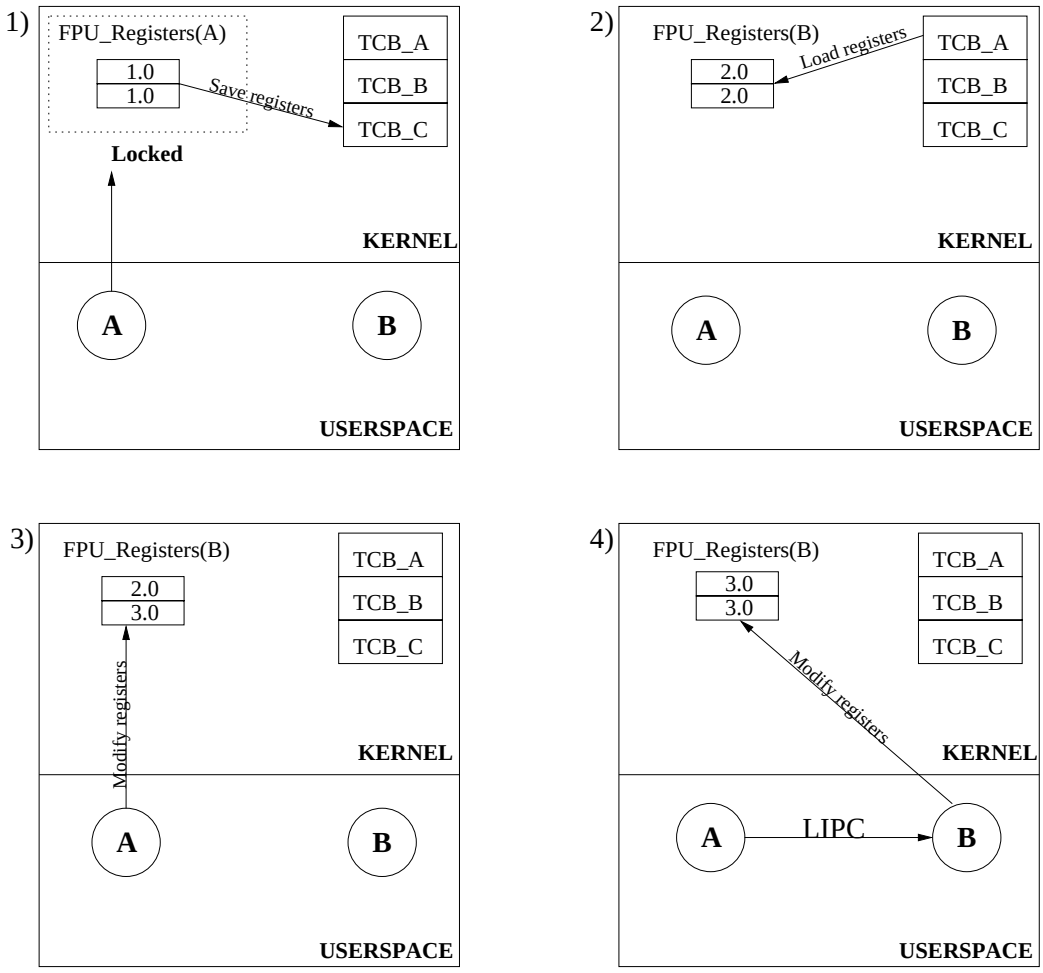


Figure 4: Coprocessor Synchronisation Problem

A performs an LIPC to thread B, which later also accesses the coprocessor. From the kernel point of view, thread A is still running, and the coprocessor is further unlocked. Thread B is working with the coprocessor and invalidates the floating-point register values of thread A.

LIPC coprocessor check:

```
IF coprocessor_used = false THEN
   proceed with LIPC;
ELSE
   abort LIPC and initiate normal IPC;
ENDIF
```

Therefore, we have to extend the above CurrentUTCB synchronisation algorithm to make it coprocessor safe. A new user-space-readable flag *CoprocessorUsed* is introduced. The flag is set by the kernel whenever the coprocessor is allocated. The user-level IPC code now checks whether CoprocessorUsed is not set. If it is set, user-level IPC is not possible and full kernel IPC is invoked. Doing a normal thread switch, the kernel resets the CoprocessorUsed flag so that further LIPC is possible.

## 4.8 Safety and Security of LIPC

The Kernel data involved are the A-TCB and B-TCB variables *stack pointer*, *status* and the system variable *current_ thread*. We have to analyse whether these variables must be really protected from unauthorised user access.

For the time being, let us assume that the above mentioned LIPC code runs read-only in user mode. The TCB variable *stack pointer* holds a thread's user stack pointer. Remember that A and B both run in the same address space so that they can arbitrarily modify each others' stack and perhaps even code. Protection would therefore not be significantly better if A's stack pointer would be protected against access from B. Consequently, the TCB variable stack pointer can be user accessible.

The state case is a little more complicated. Assume that a thread's status can only be "run" or "wait for X". We have to analyse three cases when thread A maliciously switches thread B's status: from "run"[3] to "wait for X", from "wait for X" to "wait for Y", and from "wait for X" to "run".

Whenever A modifies B's status illegally, we see user-level effects and system effects. User-level effects within A's address space can be ignored (see above). Effects in different address space that indirectly result (a transitive cause chain) from user-level effects within A's address space are also irrelevant, since A has full access to the threads of its address space and can even modify their code. As long as only thread states within the same task are accessible, user-level effects are thus uncritical.

System effects are more serious. Whenever the system state depends on a thread's state variable, we need provisions ensuring system integrity. Unau-

[3] On this level of abstraction, "run" is used to denote a ready-to-run thread as well as a thread that currently executes on a processor.

thorised modification of a state variable must in no case lead to system inconsistencies. For instance, the kernel can no longer assume that a thread with status "run" (ready-to-run) is always in the run queue. Similarly, a thread might be in the run queue although its status says "wait for X". The LIPC prototype has to use the presented LIPC algorithms and new structures to detect and respectively prevent such scenarios.

# 5 Implementation

This chapter discusses some aspects of the LIPC prototype implementation and its constrains. A few important data structures and implementation details are described as the concrete kernel fix up implementation and some LIPC important IA-32 specifics.

## 5.1 Implementation Constraints

To finish this work in a reasonable amount of time for a study thesis[4], the LIPC prototype must be submitted to some restrictions:

1. The LIPC prototype runs only on a single-processor machine
2. The LIPC prototype is only developed for the Intel IA-32 architecture
3. LIPC is an extension for the L4 - Hazelnut kernel version X.0
4. LIPC is only developed for intra-address-space communication
5. LIPC is only short IPC (means only register transfer)

Restrictions 1-3 specify the development environment and can be changed while restrictions 4-5 are consequences of the local IPC approach. Inter-address-space IPC involves automatically the kernel so that LIPC makes in this case no sense. Transferring larger messages via LIPC is not useful either, since threads in the same address space can access data from each other.

[4]Normally, you have only three months to complete a study thesis.

## 5.2 First Prototype in a Linux User-Space Environment

To develop and debug the first LIPC system-call version, an LIPC Linux user-space emulation was created. The emulation includes UTCBs, special thread stacks and the LIPC "system call"[5]. This simple prototype without any kernel-fix-up/synchronisation code (obvious in a pure Linux environment) takes 12 cycles for an "LIPC" on a Pentium III processor. These 12 cycles included the transport of four message registers (different LIPC API) from Thread A to B with hot[6] caches, hot *Translation Look-Aside Buffer* (TLB) and no extra marshalling costs.

## 5.3 UTCB and CurrentUTCB Implementation

Internally, threads are represented by *Thread Control Blocks* (TCBs). The LIPC prototype separates each thread's TCB into a *User-Level Thread Control Block* (UTCB) and a *Kernel-Level Thread Control Block* (KTCB). The UTCB is mapped user-level accessible and unprotected in the task's address space, while the KTCB can only be accessed by the kernel. A thread's UTCB holds mainly the thread's user stack pointer, its $\text{status}_u$ and its global id to find the KTCB address, which is of course untrustworthy. However, the KTCB holds a back pointer, respectively the local id, which is the same as the UTCB address to its corresponding UTCB so that the UTCB's KTCB pointer can be validated over the cross check algorithm 4.4.

UTCB Layout:

| +60-64 byte | dummy | not used |
|---|---|---|
| ... | ... | not used |
| +28-32 byte | from_specifier_old | old status of the thread |
| +24-28 byte | processor | processor id - not yet implemented |
| +20-24 byte | user_defined_handle | not yet implemented |
| +16-20 byte | myself | global thread id of the thread |
| +12-16 byte | succ_link | not yet implemented |
| +8-12 byte | ip | instruction pointer of the thread |
| +4-8 byte | from_specifier | saved state of the thread |
| +0-4 byte | stack | stack pointer of the thread |

[5] Under Linux it was a "normal" function call

[6] "Hot" caches is a common term for currently filled/updated caches.

The UTCB entries are shortly explained:

- stack: The stack entry contains the saved stack pointer of the thread.
- from_specifier: The from_specifier entry contains the current state of the thread.
- ip: The ip entry contains the saved instruction pointer of the thread.
- succ_link: The SuccLink entry points to the next UTCB. The pointer value is relative to the begin of the current UTCB. This is a appropriate way to find all valid local thread ids and the global ids of all threads which reside in the current address space (see myself field). All active UTCBs of an address space are in a circular linked list.
- myself: The myself entry contains the global id of the thread. A translation local id to global id is thus simple and fast.
- user_defined_handle: This handle can be freely set by user threads. It can e.g. be used for storing a thread number, etc.
- processor: The processor entry contains the processor number on which the thread currently executes.
- from_specifier_old: The from_specifier_old entry contains the old receive state of the thread.

UTCBs are 6 bit aligned and have in the prototype implementation a size of 64 bytes so that all UTCBs of an address space need exactly one 4 KB page of kernel memory (L4 version X.0 can have 64 threads in an address space [6]). For the ease of implementation, each address space has a fixed UTCB area (0xB0000000-0xB0000FFF), which contains the UTCBs of all threads residing in this address space.

The unprotected kernel variable $CurrentUTCB_u$ is saved in gs:[0] (IA-32 GS segment) and can be accessed from the user mode. Its protected twin, $CurrentUTCB_k$ lives in kernel space and can be calculated from the ESP0 pointer, which points to the TCB of the active thread (from the kernel point of view).

## 5.4 Local and Global Thread Identifier

Normally, the L4 version X.0 microkernel has only global thread identifiers (see [6]), but for LIPC it is sensible to introduce a local thread identifier for a fast UTCB access. Local thread ids identify threads within the same address space. They are identified by the sic lower-most bits being 0 (only 26 significant address bits). A local thread id is basically a direct pointer to the thread's UTCB.

| UTCB address/64 | 000000 |
|---|---|

## 5.5 Modified IPC Path

The normal IPC path in the kernel must be modified since the stack and instruction pointer of a thread can change without knowledge of the kernel (see Figure 2). The whole L4 IPC Path is one of the most complicated parts in the L4 microkernel; that's why only the modifications are outlined.

```
Enter Kernel IPC-Path
IF CLOSED_IPC THEN
   IF RECEIVE_FROM thread in the same address space THEN
      set FROM_SPECIFER to proper local thread_id (LIPC possible);
   ELSE
      set FromSpecifier to NIL_THREAD (no LIPC);
   ENDIF
ELSE IF OPEN_IPC THEN
   set FromSpecifier to ANYTHREAD (LIPC possible);
ENDIF
set current stack pointer in the UTCB;
set current instruction pointer in the UTCB;
...
--> set Waiting and switch to next thread;
<-- get active again (message received);
...
get and set UTCB user instruction pointer for return;
get and set UTCB user stack pointer for return;
```

```
set FromSpecifier to NIL_THREAD (no LIPC);
return to thread;
```

After doing the kernel fix up (see 5.8), the LIPC ensures a consistent "user-level" and kernel thread state, but the user stack/instruction pointer can be different from the kernel point of view. A thread in the middle of an LIPC chain has a changed user stack/instruction pointer, which the kernel cannot detect. That's why the kernel can only use the stack and instruction pointer values of the UTCB and must ensure that any stack/instruction pointer modification via kernel code is passed through to the UTCB (set/get stack and instruction pointer in the IPC path). The kernel fix up ensures that user $\rightarrow$ kernel status synchronisation is done, but it must be also ensured that any thread status change in the kernel is propagated to user space.

## 5.6 LIPC Implementation

The LIPC system-call code is mapped read-only to the fixed address 0xF00D0000 in user address space and can be executed in user mode. Atomicity is guaranteed as described in 4.3.

### 5.6.1 General LIPC Restrictions

1. The operation must contain a send and receive phase.
2. The recipient (to thread) must exist and reside in the sender's address-space.
3. The recipient (to thread) must be specified as a local thread id.
4. The send message may consist of up to 3 untyped words and must not contain typed elements.
5. The send timeout is implied to be infinity.
6. The receive timeout is implied to be infinity.
7. LIPC includes no map/grant operations.

A receive only operation is not sensible since kernel support would be needed. For a receive only operation, the kernel has to remove the thread from running queue and to insert it into the waiting queue. A send only operation is possible, but would complicate the LIPC path and would require further changes of the LIPC interface.

Restriction two and thre follow directly from the LIPC restriction that it is a local IPC in the same address-space. The local thread id is needed to find the proper UTCB. Restriction four is a design decision to have an easy data transfer without kernel support. Since you have no kernel-space access (no kernel entry, no wakeup queue), you cannot specify other timeouts (restriction five and six) and cannot access page tables for a map and grant operations (restriction seven).

### 5.6.2 LIPC Application Binary Interface

| Pre: | | $\Longrightarrow$ | Post: | |
|---|---|---|---|---|
| UTCB_ADDR | EAX | | UTCB_FROM | EAX |
| Return Address | ECX | | not defined | ECX |
| MSG.W0 | EDX | | RCV.W0 | EDX |
| MSG.W1 | EBX | | RCV.W1 | EBX |
| FROM_SPECIFIER | EBP | | not defined | EBP |
| UTCB_TO | ESI | | not defined | ESI |
| MSG.W2 | EDI | | RCV.W2 | EDI |
| USER SP | ESP | | USER SP | ESP |

LIPC is a subset of the general IPC system call. It may be used for intra-address-space communication and synchronisation.

All communication is synchronous and unbuffered: a message is transferred from the sender to the recipient if, and only if, the recipient has invoked a corresponding IPC operation (L4 policy [6]). The sender blocks until the destination becomes ready to receive from the sender, or the given timeout expires. For LIPC the send and receive timeout is implied to be infinity. You can't enforce other timeouts without kernel access. If the receiver thread is not ready to receive or a global thread id is given, a normal IPC is initiated (LIPC initiates IPC).

### 5.6.3 The LIPC Call

```
word l4_lipc[7](utcb_t to, dword_t FromSpecifier,
                dword_t sndw0, dword_t sndw1, dword_t sndw2,
                dword_t *rcvw0, dword_t *rcvw1, dword_t *rcvw2,
                utcb_t from)
```

**LIPC parameters:**

| | |
|---|---|
| **to** | Receiver local thread_id for the LIPC |
| **FromSpecifier** | Specifies the waiting status of the sender thread after a successful LIPC:<br>FromSpecifier=ANYTHREAD (0xFFFFFFFF)<br>Incoming LIPCs / IPCs are accepted from any thread.<br>FromSpecifier≠ANYTREAD<br>Incoming messages are accepted only from the specified thread (local thread id). |
| **sndw0 - sndw2** | Three send words which contain a message. |
| **rcvw0 - rcvw2** | Three receive words which contain the response message |
| **from** | Sender local thead id for the LIPC. |

## 5.7 LIPC Path

```
Thread A -> B
jump to LIPC function;
save return address in UTCB;
--- restart point ---
IF B is not Local_ID (last 6 Bits Zero and valid UTCB Area?) THEN
   initiate normal IPC1 (assume Global_ID);
ELSE IF B is not waiting for ANYTHREAD or sender thread THEN
   initiate normal IPC2;
ELSE IF A's Old Receiving Status != A's New Receiving Status THEN
   initiate normal IPC2;
ELSE IF coprocessor_used = TRUE THEN
```

[7] See sample LIPC C-Binding in the Appendix

```
      initiate normal IPC2;
   ELSE
      --- FORWARD POINT ---
      set A's status to FromSpecifier;
      save A's stack pointer;
      save B's old status;
      set B's status to "run";
      load B's stack pointer;
      set current_thread to B;
      --- COMPLETION POINT ---
      return, i.e. use return address in the UTCB;
   ENDIF
   initiate normal IPC1: (Assume a normal IPC call)
      push UTCB return address on the stack;
      trigger IPC;
      return, i.e. pop return address from the stack;
   initiate normal IPC2: (Assume an LIPC call)
      set snd_descriptor to short IPC (0_32);
      set rcv_descriptor to open/close IPC;
      set global id to the destination thread;
      push UTCB return address on the stack;
      set timeout (to infinity);
      trigger IPC;
      return, i.e. pop return address from the stack;
```

You can see from the LIPC pseudo code that LIPC can only be performed under special conditions:

1. A valid receiver local thread id is given
2. The receiver thread is ready to receive from the sender
3. The sender thread does not change its old/kernel waiting state
4. The sender thread has not used the coprocessor

All four conditions are reasonable (see theory section) but condition three can be troublesome. While creating a thread, a new UTCB is allocated and initialised. At the beginning FromSpecifierOld is initialised to ANYTHREAD.

When the newly created thread tries to perform a special LIPC sequence which changes the FromSpecifier from closed IPC to open IPC in alternating order, all LIPCs fail and a normal IPC is initiated. Normally, this shouldn't be a problem since a thread has a more or less fixed characteristic and does not change its receiving state all the time, but this depends on the application.

## 5.8 Kernel-Fix-Up Code

### 5.8.1 Exception Frame Fix Up

Let us assume that thread A performs an LIPC to thread B and thread B gets a timer interrupt. From the kernel point of view thread A is still running and the exception frame consisting of the context information of thread B is saved on A's kernel stack. For further information about Intel IA-32 exception frames look at the IA32 documentation [22].

It can not be prevented that the exception frame of thread B is saved on the kernel stack from thread A, but after the fix-up code has detected a user vs. kernel inconsistency the exception frame can be copied and modified. Figure 5 illustrates how to get the right exception frames on the right kernel stacks (very simplified).

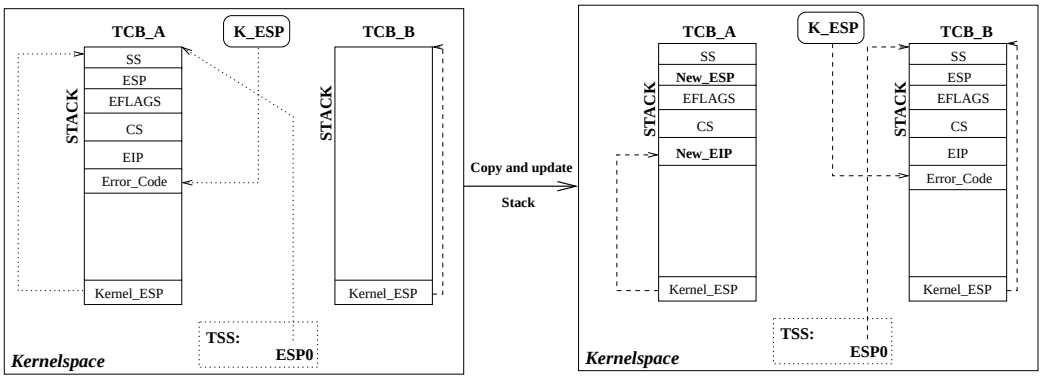


Figure 5: Exception Frame Fix Up

The basic strategy is to copy the exception frame from A's kernel stack to B's kernel stack and to generate a default exception frame with proper user stack and instruction pointer on A's kernel stack. Afterwards, the kernel

stack pointer and the ESP0 entry in the Tasks State Segment (TSS), a kernel variable which points to the TCB end of the active thread, is set to B's kernel stack to complete the lazy thread switch from thread A to thread B.

### 5.8.2 Kernel-Fix-Up Path

The pseudo code below outlines the kernel fix up path.

```
Thread A and B are the endpoints of the LIPC chain
and B gets an exception:
IF exception in kernel section THEN
   skip LIPC kernel fix up code;
ELSE IF EIP between RESTART_POINT and FORWARD_POINT of the LIPC THEN
   set (USER)EIP to RESTART_POINT and skip LIPC kernel fix up code;
ELSE IF EIP between FORWARD_POINT and COMPLETION_POINT of the LIPC THEN
   complete section and set(USER)EIP to COMPLETION_POINT;
ELSE
   skip LIPC kernel fix up code;
ENDIF
IF CURRENT_UTCB_U=CURRENT_UTCB_K THEN
   skip LIPC kernel fix up code;
ENDIF
IF CURRENT_UTCB_U is invalid THEN
   kill current thread;
   skip LIPC kernel fix up code;
ENDIF
--- LIPC fix up code ---
delete A from run queue;
set A's TCB status to WAITING;
IF A is in CLOSED_WAIT THEN
   set partner in A's TCB;
ENDIF
insert B in run queue;
set B's TCB status to RUNNING;
copy stack frame from A to B;
set proper (KERNEL)ESP for B;
set ESP0 in the TSS to B's stack end;
```

```
fix A's (USER)EIP in its exception frame (A's UTCB values);
fix A's (USER)EIP in its exception frame (A's UTCB values);
set (KERNEL)ESP of A to the end of its exception frame;
set (KERNEL)EIP of A to a special LIPC_WAIT function;
```

The first part of the LIPC fix-up code checks whether it was an exception in the kernel mode, then the fix up is skipped, or the exception occurs in an LIPC call. Depending on the position in the LIPC call, the code does a code roll forward or backward (see 4.3). Furthermore, it is checked whether a CurrentUTCB$_u \neq$ CurrentUTCB$_k$ inconsistency exists.

Is this the case, the status and exception frame fix up is done as described in the design chapter. After the fix up, thread A does not only have to be in a waiting state, it must also set in a proper "position" in the IPC path. That's why the LIPC prototype has a special sys_lipc_wait function which emulates the activation and receiving part of the normal IPC path (see pseudo code below).

```
<-- get active again (message received);
get and set UTCB user instruction pointer for return;
get and set UTCB user stack pointer for return;
set FromSpecifier to NIL_THREAD (no LIPC);
do special IPC argument marshalling;
return to thread;
```

When thread A becomes "active" again, it executes the sys_lipc_wait function, which sets the current user instruction and stack pointer of thread A. Furthermore, the sys_lipc_wait function marshalles the IPC arguments and returns back to user mode.

## 5.9 Implementation Restrictions

The LIPC implementation is a prototype implementation which is more a proof of concept than a complete implementation of all features. That's why some features of the L4 version X.0 microkernel can not yet be used.

1. Small Address Spaces

2. IPC via sysenter

*Small Address Spaces* is a technique to "simulate" tagged TLBs on the Intel IA-32 architecture by dividing one "big" address space into several smaller address spaces [8]. Since the LIPC prototype uses a fixed UTCB address scheme, Small Address Spaces are not yet applicable.
IPC via *sysenter* is a kernel entry optimisation which causes different exception frame layouts on the kernel stack, what would further complicate the kernel-fix-up code. That's why the LIPC prototype can only use IPC via int 0x30 which needs about 300 cycles for intra-address-space IPC, rather than about 180 cycles via sysenter.

# 6 Measurements

The pingpong test-suite from L4Ka is used to measure the overhead of normal IPC with the LIPC prototype. Further, kernel profiling identifies the code where the CPU spends the most time. The benchmark system is a normal Intel Pentium III 450 MHz PC with 64 MB RAM. The IPC measurements include one unidirectional IPC without marshalling costs. All IPCs are repeated 8000 times and the average is calculated.

**LIPC Prototype Development Environment:**

- Linux Debian (Woody) system
- Intel Pentium III 1 GHz
- gcc-3.04 and gcc-2.95.4 C++ Compiler
- binutils Version 2.12.90.0.1-4

## 6.1 Intra/Inter-Address-Space IPC

| | Intra-Address-Space IPC | Inter-Address-Space IPC |
|---|---|---|
| L4 microkernel | 306 | 488 cycles |
| LIPC prototype | 349 cycles | 634 cycles |
| Overhead | 43 cycles | 146 cycles |

The 43 cycles intra-address-space IPC overhead (about. 14%) are expected since the LIPC prototype kernel must do several checks and UTCB updates. The LIPC implementation tries to prevent jump failure predictions and unnecessary memory accesses, but some checks and extra updates are inevitable. The 146 cycles inter-address-space IPC overhead (about. 30%) are rather high. Code profiling has verified the idea that the IPC path has suffered TLB misses when it accessed the UTCB of another address space (not necessary for the unmodified IPC path). These extra UTCB accesses are inevitable, but the extra TLB miss overhead could perhaps be reduced by using Small Address Spaces[8] [8].

### 6.2 Intra-Address-Space LIPC

| | Intra-address-space LIPC | Inter-address-space LIPC |
|---|---|---|
| LIPC prototype | 23 cycles | not possible -> normal IPC |
| Speed gain | ~ 1330 % | - |

The 23 cycles for a complete LIPC satisfy the expectations. In comparison to the first Linux user-space prototype (12 cycles for an "LIPC"), the "real" LIPC prototype includes several modifications and new checks. LIPC can compete in speed with pure user-level IPC and is about 1330% faster than a normal intra-address-space kernel-level IPC.

## 7 Conclusions

It is possible to implement a fast local IPC with kernel-level threads which can compete in speed with pure user-level implementations. Together with a fast IDL compiler, for example IDL[4] [19, 21, 20], it should be possible to use LIPC for fine-grained synchronisation and fine-grained multi-threaded servers. The drawbacks of this approach are an overhead for normal kernel-level IPC (14-30%), and the general LIPC restrictions which prevent its usage in some cases. This work focuses to achieve a high LIPC performance and to handle inconsistencies caused by LIPC. Reducing the overhead for normal kernel-level IPC caused by LIPC extensions was not the main goal. A speed

[8]Small Address Spaces are not part of this study thesis

gain of over 1330% for intra-address-space IPC should more than compensate the "normal" IPC overhead, but this is subject of future research.

# 8 Future Work

Firstly, it has to be examined how much fine-grained synchronisation and fine-grained multi-threaded servers profit by LIPC. There are several open research and implementation aspects to this:

- LIPC Small Address Space implementation
- LIPC sysenter and IPC fast-path implementation
- LIPC on SMP machines
- Further reduction of the kernel-level IPC overhead caused by the LIPC extensions
- Further LIPC stability testing, e.g. L4 Linux on top of an LIPC capable L4 kernel
- Using the insights from this LIPC prototype implementation for the next L4 kernel version (L4 Pistachio [17])

Having a very fast IPC mechanism which prevents a kernel entry will be even more important in the future. You can recognise the trend that kernel entry costs for new CPU architectures are increasing (large amount of registers to save and restore); one example is the IA-64 [23] architecture (successor of the IA-32 architecture). Having a fast IPC mechanism which speed is independent from the kernel entry costs can help to increase the overall IPC performance of a system.

# 9 APPENDIX

## 9.1 Sample LIPC C Binding

```
word l4_lipc (utcb_t to, dword_t FromSpecifier,
              dword_t sndw0, dword_t sndw1, dword_t sndw2,
              dword_t *rcvw0, dword_t *rcvw1, dword_t *rcvw2,
              utcb_t from){
   dword_t dummy;
   __asm__ __volatile__ (
      /* save frame pointer */
      "pushl %%ebp \n\t"
      /* move from_specifier from ECX to EBP */
      "movl %%ecx, %%ebp \n\t"
      /* save return address in ECX register */
      "movl $1f,%%ecx \n\t"
      "jmp " LIPC_ENTRY " \n\t"          /* LIPC_ENTRY = 0xF00D000 */
      "1: \n\t"
      /* restore frame pointer */
      "popl %%ebp \n\t"
      : /* define Output */
      "=a" (utcb_from),   /* 0, Save UTCB_FROM -> EAX */
      "=d" (*rcv_dword0), /* 1, Save value of EDX into *rcv_dword0 */
      "=b" (*rcv_dword1), /* 2, Save value of EBX into *rcv_dword1 */
      "=D" (*rcv_dword2)  /* 3, Save value of EDI into *rcv_dword2 */

      : /* set registers for Input */
      "a" (utcb_from),      /* 4, UtcbAddr_from -> EAX */
      "d" (sndw0),          /* 5, sndw0 -> EDX */
      "b" (sndw1),          /* 6, sndw1 -> EBX */
      "c" (from_specifier), /* 7, from_specifier -> ECX -> EBP */
      "S" (utcb_to),        /* 8, to -> ESI    */
      "D" (sndw2)           /* 9, sndw2 -> EDI */

      : /* scratch memory */
      "memory");
```

```
        /* clobber ESI and EAX */
        __asm__ __volatile__ (
            " \n\t "
            :
            "=S" (dummy),
            "=c" (dummy));
}
```

# References


[1] J.Liedtke, H.Wenske. *Lazy Process Switching.* The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau/Oberbayern, Germany, May 2001.

[2] T.E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. *Scheduler activations: Effective kernel support for the user-level management of parallelism.* In 13th ACM *Symposium on Operating Systems Principles* (SOSP), Pacific Grove, CA, October 1991.

[3] B.A. Ford and J. Liedtke. private communication, December 1995.

[4] J.Liedtke. *Improving IPC by kernel design.* In $14^{th}$ ACM *S*ymposium on Operating System Principles *(*SOSP*)*, pages 175-188, Asheville, NC, December 1993.

[5] J.Liedtke. *Lazy Context Switching Algorithms for Sparc-like Processors.* GMD Technical Report No. 776. September 1993.

[6] J.Liedtke. *L4 Nucleus Version X Reference Manual X86*, September 1999.

[7] J.Liedtke. Lecture *Construction of Microkernels.* University of Karlsruhe (2001).

[8] J.Liedtke. *Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces.* GMD Technical Report No. 933. November 1995

[9] T.P.Baker. *A stack-based resource allocation policy for realtime processes.* $11^{th}$ Real-Time Systems Symposium (RTSS). IEEE, December 1990.

[10] L.Sha, R. Rajkumar, and J. P. Lehoczky. *Priority inheritance protocols: An approach to real-time synchronization.* IEEE Transactions on Computers, 39(9), September 1990.

[11] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl.*The Persistent Relevance of IPC Performance: New Techniques for Reducing the IPC Penalty.* Workshop on Workstation Operating Systems (1993)

[12] J.S. Shapiro, D. J. Farber, J. M. Smith. *The Measured Performance of a Fast Local IPC* (1996)

[13] B.D. Marsh, M. L. Scott, T. J. LeBlanc, E. P. Markatos. *First-Class User-Level Threads.* Proceedings of the $13^{th}$ ACM Symposium on Operating Systems Principle (SOSP). 1991

[14] D.S. Ritchie, G.W. Neufeld. *User Level IPC and Device Management in the Raven Kernel* (1993)

[15] B. Mukherjee, G. Eisenhauer, K. Ghosh. *A Machine Independent Interface for Lightweight Threads.* ACM Operating Systems Review 1994.

[16] S.Wagner. *An Architecture Independent Kernel Debugger for Hazelnut - Study Thesis.* University of Karlsruhe (September, 2001)

[17] $L^4$Ka Homepage: http://www.l4ka.org

[18] Fiasco Homepage: http://os.inf.tu-dresden.de/fiasco/

[19] A. Haeberlen. *Using Platform-Specific Optimizations in Stub-Code Generation - Study Thesis.* University of Karlsruhe (July, 2002)

[20] $IDL^4$ Homepage: http://www.uni-karlsruhe.de/~uhns/idl4/

[21] A. Haeberlen, J.Liedtke, Y.Park, L. Reuther, V. Uhlig. *Stub-Code Performance is Becoming Important.* In the First Workshop on Industrial Experiences with System Software (WIESS), San Diego, CA.

[22] Intel Architecture Developer Manual IA32 Vol. 1-3

[23] Intel Itanium Processor Family: http://www.intel.com/design/itanium/