Investigation of Mechanisms to Support User-Level Thread Packages on Top of the L4-Fiasco Microkernel

Dietrich Clauß
dc2@inf.tu-dresden.de

Technische Universität Dresden
Institute for System Architecture
Operating Systems Group

February 28, 2005

Abstract

Many approaches of parallel programming use threads as an abstraction for multiple concurrent activities. Threads can either be supported by the operating system kernel or by a user-level library in the application's address space. The attempt to combine the advantages of both kernel-level threading and user-level threading has already been made on various operating systems with positive results. This work discusses this problem with respect to L4 microkernels. We conclude that many-to-one threading clashes with the basic concepts of L4, but many-to-many threading can be built. It needs a kernel mechanism that notifies the user-level application about blocking and unblocking of threads. We implemented this mechanism in the L4-Fiasco microkernel and propose a design of a user-level thread library. The proposed design enables user-level threads to use L4 functionality.

Acknowledgements

I would like to thank Prof. Dr. Hermann Härtig for the opportunity to work on this interesting project in the Operating Systems Group at TU Dresden. My special thanks go to my supervisor Marcus Völp who fed me with a lot of interesting design ideas. I would also like to thank Michael Peter, Dr. Michael Hohmuth, and René Reussner, who always found time for a discussion, and Adam Lackorzynski who never disabled my account.

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

Declaration

I declare to have written this work independently and without using unmentioned sources.

Dresden, February 28, 2005

____

Dietrich Clauß

Diese Seite durch die Aufgabenstellung ersetzen!

Diese Rückseite durch die Rückseite der Aufgabenstellung ersetzen!

Contents

1 Introduction **1**

1.1 Multithreaded Programming . . . 1

1.2 Kernel-Level Threads and User-Level Threads . . . 1

1.3 Document Structure . . . 3

1.4 Terminology . . . 3

2 Background **5**

2.1 User-Level Threading . . . 5

2.2 L4 Microkernels . . . 5

2.2.1 Address Spaces . . . 7

2.2.2 Threads . . . 7

2.2.3 Communication . . . 7

2.3 L4 System Calls . . . 7

2.4 L4 IPC . . . 7

2.4.1 Timeouts . . . 9

2.4.2 Send and Receive Destinations . . . 9

2.4.3 Blocking Send . . . 9

2.4.4 Map and String IPC . . . 10

2.4.5 IPC Call and Atomicity . . . 10

2.4.6 Canceling and Aborting of IPC Operations . . . 10

2.4.7 Page Faults . . . 10

2.5 L4 Kernel Resources . . . 12

2.5.1 UTCBs . . . 12

2.6 L4 Trust Relationships . . . 12

2.6.1 Client-Server Trust Relations . . . 12

3 Related Work **15**

3.1 Event-Based Systems . . . 15

3.2 Scheduler Activations . . . 15

3.2.1 K42 . . . 16

3.2.2 Summary . . . 16

3.3 User-Level Management of Kernel Resources on L4 . . . 16

3.4 Local IPC on L4 . . . 16

3.5 Summary . . . 16

4 Design **18**
4.1 Goals . . . 18
4.1.1 Characteristics of User-Level Threads . . . 18
4.1.2 Boundary Conditions . . . 19
4.2 Investigation of Per-Thread Data . . . 19
4.3 Dispatching User-Level Threads on Kernel Threads . . . 20
4.3.1 Execution Contexts at User-Level . . . 20
4.3.2 In-Kernel Dispatcher Code . . . 20
4.3.3 User-Level Dispatcher Code . . . 21
4.3.4 Dedicated Dispatcher Thread . . . 21
4.3.5 Dispatching From Within the Virtual Processor . . . 23
4.3.6 Summary . . . 24
4.4 Long-Running System Calls . . . 25
4.4.1 Tolerate Blocking . . . 25
4.4.2 Deschedule Invokers of Long-Running System Calls . . . 25
4.4.3 Use Multiple Kernel Threads . . . 26
4.4.4 Summary . . . 26
4.5 Dispatching IPC Messages . . . 27
4.6 Multiple Virtual Processors . . . 27
4.6.1 Many-to-Many Threading . . . 27
4.6.2 Activation Messages . . . 29
4.6.3 Local Thread IDs and Local Communication . . . 30
4.6.4 Analysis . . . 30
4.7 Deschedule Blocking User-Level Threads . . . 31
4.7.1 Dispatcher Type . . . 31
4.7.2 IPC Scenarios . . . 31
4.7.3 User-Level Thread Blocks on Receive . . . 33
4.7.4 User-Level Thread Blocks on Send . . . 34
4.7.5 Kernel-Level Thread Blocks on Receive . . . 34
4.7.6 Kernel-Level Thread Blocks on Send . . . 34
4.7.7 Summary . . . 35
4.8 Split TCB into Execution Context and Message Context . . . 35
4.8.1 TCB Data in the Fiasco Microkernel . . . 35
4.8.2 Amount of Saved Resources . . . 35
4.8.3 Summary . . . 36
4.9 Summary . . . 36

5 Implementation **37**
5.1 Kernel Implementation . . . 37
5.1.1 Set the Dispatcher . . . 37
5.1.2 How to Send Activation Messages . . . 37
5.1.3 When to Send Activation IPC . . . 38
5.1.4 Conclusion . . . 39
5.2 Thread Package Layout Proposal . . . 39
5.2.1 Cooperative Scheduling . . . 39
5.2.2 Preemptive Scheduling and Timeouts . . . 39
5.2.3 Ready Queues . . . 39
5.2.4 Invoking Kernel IPC Calls . . . 40
5.2.5 Receiving Kernel IPC Calls . . . 40

5.2.6 The Exchange-Registers System Call . . . 42
5.2.7 Local Communication . . . 42
5.2.8 Locking of User-Level Resources . . . 42
5.2.9 Summary . . . 42

6 Evaluation **43**
6.1 Performance Overhead . . . 43
6.2 Resource Overhead . . . 43

7 Conclusion **44**
7.1 Future Work . . . 46

8 Summary **47**

A Fiasco Implementation Details **48**
A.1 Memory Layout . . . 48
A.2 Page Faults in Kernel Mode . . . 48
A.3 Access to User Memory . . . 49
A.3.1 Active Task . . . 49
A.3.2 Inactive Task . . . 49
A.4 Long IPC . . . 49
A.5 Summary . . . 49

Bibliography **50**

List of Figures

1.1 Multithreaded File-System Server. 2

2.1 Many-to-One User-Level Threading. 6
2.2 Many-to-Many User-Level Threading. 6
2.3 IPC with a Blocking Send Phase. 8
2.4 IPC with a Blocking Receive Phase. 8
2.5 Flow Diagram of an IPC Call. 11
2.6 Failing IPC Call. 11
2.7 Example of the L4 Trust Relations. 13

4.1 User-Level Threads with Dedicated Dispatcher Thread. 22
4.2 User-Level Threads Without Dedicated Dispatcher. 23
4.3 Many-to-Many Threading. User-Level Ready Queues at a Blocking System Call. 28
4.4 Activation Messages. 29
4.5 User-Level Thread IPC with Dedicated Dispatcher Thread. 32
4.6 User-Level Thread IPC Without Dedicated Dispatcher Thread. 32

5.1 Forwarding IPC Requests. 40
5.2 Delayed IPC Forward. 41
5.3 Dispatcher Activity on Delayed IPC Forward. 41

7.1 Contradicting Design Goals. 45

A.1 Fiasco Memory Layout. 48

Chapter 1

Introduction

1.1 Multithreaded Programming

When developing applications that are supposed to proceed multiple requests at the same time, a programming model that supports multiple concurrent activities is desired. *Multithreaded programming* is such a model. It allows an application to spawn several *threads*, which then execute independently from each other. Figure 1.1 shows an example layout of a file server that spawns a thread for each opened file.

1.2 Kernel-Level Threads and User-Level Threads

Threads can be provided either by the operating system kernel or by a user-level library. Each of these concepts has its specific benefits:

- **Kernel-Level Threads:**

  Preemptive Scheduling. The kernel can enforce different priorities, quality-of-service and real-time scheduling policies.

  Blocking Operating System Services. Kernel services provided to user-level applications (e.g., disk input–output or communication primitives) are usually provided on a per-thread basis. There are services that block the invoking thread until the requested action completes. If an application spawns multiple kernel threads, it can also use these services in parallel.

  Multiprocessing. On a multiprocessor system, kernel level threads can be assigned to different processors and thus they can actually run in parallel.

- **User-Level Threads:**

  User-Controlled Per-Thread Resources. The amount and type of resources needed for each thread is controlled by the user-level thread package. In particular, if the system supports swapping out memory content to disk, this will also apply to the memory used for thread management.

  Fast Thread Management. Because of the user-controlled per-thread resources, very light-weight threads can be built, whose creation and destruction will be accordingly efficient. In addition to that, thread management needs no kernel interaction. That

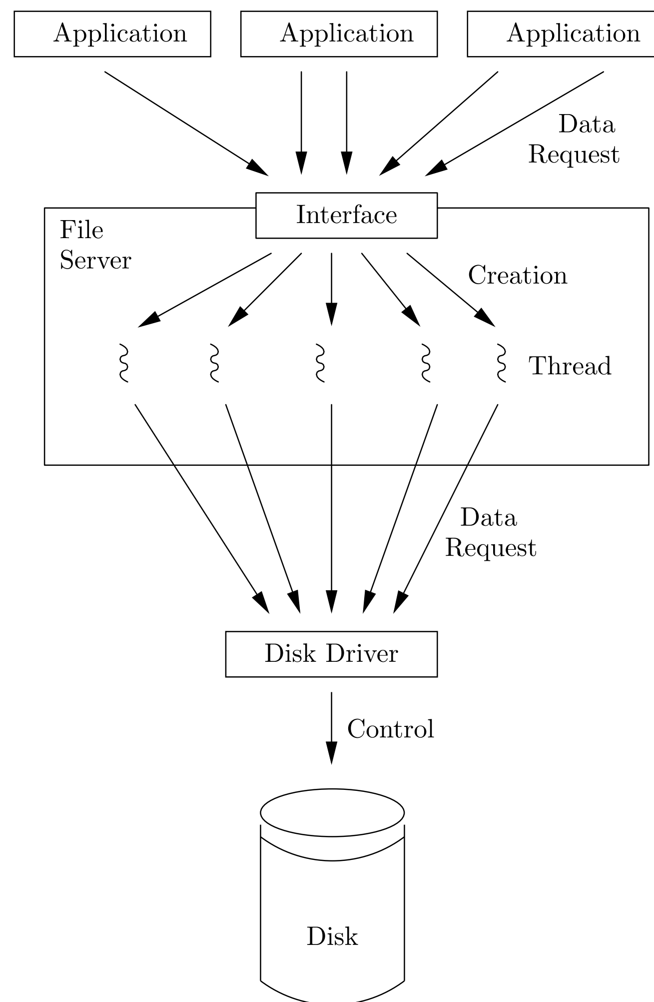

Figure 1.1: Multithreaded File-System Server.

causes a substantial performance gain on hardware architectures with high kernel enter/exit costs.

Fast Cooperative Scheduling. For the afore-said reasons, a user-level thread switch can be significantly faster than a kernel-level one.

In summary, user-level threads are cheap but not fully independent from each other, whilest kernel-level threads are independent but potentially expensive. When multithreading is used in high-concurrency servers, large amounts of threads are needed. As the available resources are usually limited, inexpensive threads are eligible. For applications that also need the positive characteristics of kernel-level threads stated above, a combination of the advantages of both threading concepts is desirable. For this to be accomplished, two approaches are imaginable: optimizing kernel-level threads, or enhancing user-level threads.

Optimize Kernel-Level Threads. Use kernel-level threads and optimize the kernel to make them as inexpensive as possible.

Enhance User-Level Threads. Use user-level threads and add kernel functionality to enable the user-level threading system to use also the features of kernel-level threads. This method is also known as *kernel-supported user-level threading* [ABLL92].

Both approaches have already been topic of various research projects. A listing and description of them is given in Chapter 3. This work focuses on enhancing user-level threads for microkernel-based operating systems.

1.3 Document Structure

In the next Chapter we introduce the relevant L4 internals and present possible approaches for user-level threading. Related work is presented in Chapter 3. In Chapter 4 we propose and discuss three design models for kernel-supported user-level threading on top of L4: one-to-many threading, many-to-many threading, and splitting of a thread's context into execution context and message context. In Chapter 5 we describe implementation highlights and problems that occured when implementing the mechanism for the many-to-many threading model in the Fiasco microkernel. In Chapter 6 we show an evaluation of the results. In Chapter 7 we conclude this work and a summary is given in Chapter 8.

1.4 Terminology

In the following we define the terminology used throughout this thesis:

Kernel-Level Thread. The term *kernel-level thread* always refers to a thread that is administered by the operating system kernel. The term *kernel thread* is used synonymously. It has not to be mistaken for kernel-internal threads such as the *idle thread* most kernels use to eat up unused processor time.

User-Level Thread. This term refers to a thread that exists at the user level and that is only known to user-level applications.

Thread. The sole word *thread* can mean both kernel-level or user-level thread, or even an activity in general.

Virtual Processor. A kernel-level thread that executes user-level threads is also referred to as a *virtual processor*.

Trusted Resources. If not explicitly stated by which system component(s) these resources are trusted, it means they are trusted by the L4 microkernel. The trustworthiness refers to the availability of the resource and the control of access rights other system components have for this resource.

Chapter 2

Background

In this Chapter we give the background information that is needed to understand the design and implementation. At first, an introduction to the fundamentals of user-level threading is given. We then describe the functionality an L4 kernel provides, and we give the reasons why this functionality is essential for the L4 operating system.

2.1 User-Level Threading

The possible assignments of user-level threads to virtual processors can be grouped into two classes: many-to-one threading and many-to-many threading.

Many-to-One Threading. A set of user-level threads executes on top of one virtual processor. The scheme of many-to-one user-level threading is depicted in Figure 2.1.

Many-to-Many Threading. A set of user-level threads executes on top of a virtual multiprocessor. This implies that user-level threads can be *moved* from one virtual processor to another.[1] The scheme of many-to-many threading has been depicted in Figure 2.2.

2.2 L4 Microkernels

L4 is a second generation microkernel interface. It has initially been designed [Lie95a] by Jochen Liedtke, who also wrote the first implementation. This first L4 kernel was written entirely in assembly language. The Operating Systems Group in Dresden developed [Hoh98, Hoh02] Fiasco, a kernel written in C++ for better maintainability and portability.

This Section is not intended to be a complete description of the L4 interface. Instead, we only introduce the mechanisms that are of interest when building kernel-aided user-level threading on top of L4. The complete description of the L4 interface can be found in [DLSU04].[2]

L4 provides address spaces, (kernel-level) threads, and a mechanism for communication between threads.

[1] A many-to-many threading model without moveable user-level threads is equivalent to multiple many-to-one models.

[2] At time of this writing, there were several L4 interface versions out there. This version, L4/x.2, was the most recent experimental version with working implementations available. If not explicitly stated, when referring to L4 in this work, we mean L4/x.2. Other versions are L4/v2 [Lie96] (latest stable), L4/x.0 [Lie99] (first experimental), and L4/Sec [PV05] (as yet unimplemented proposal for a next experimental version).

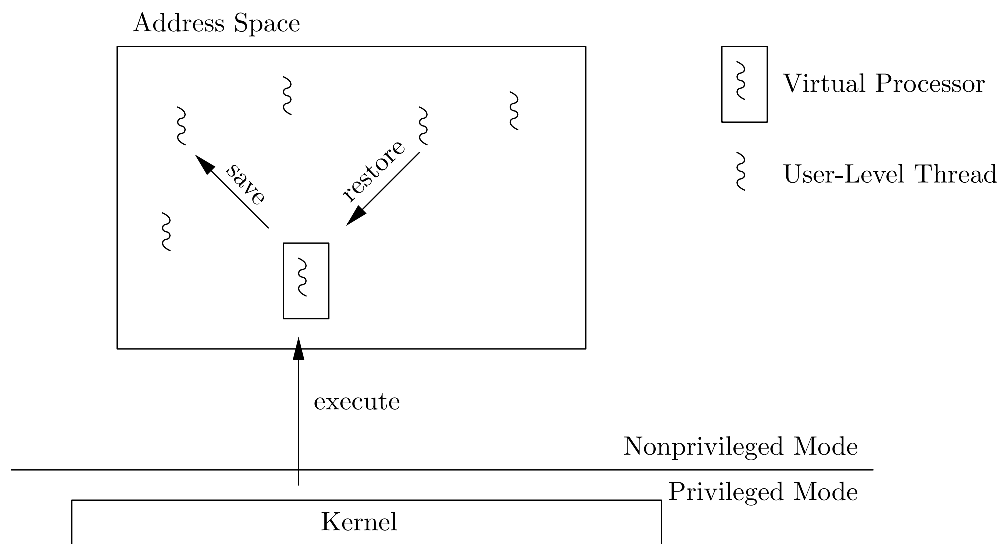

Figure 2.1: Many-to-One User-Level Threading.

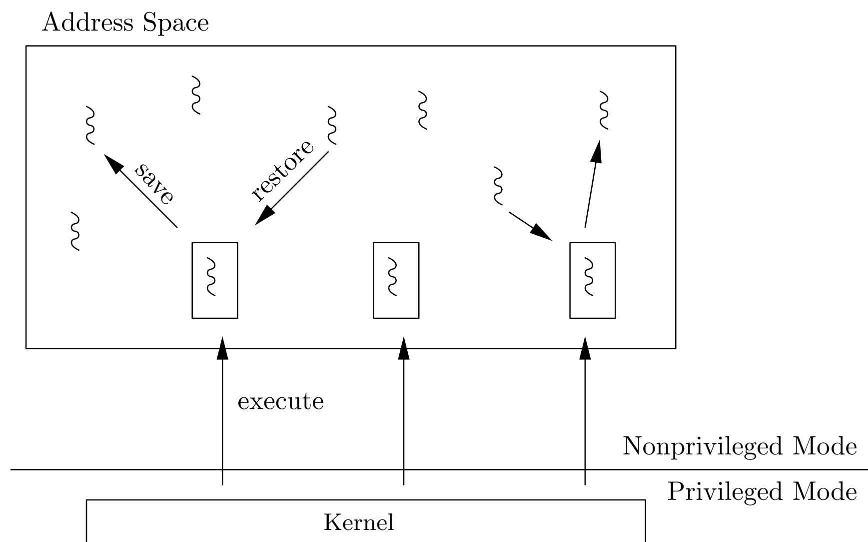

Figure 2.2: Many-to-Many User-Level Threading.

2.2.1 Address Spaces

Address spaces are the protection domains of an L4 system. There is one root address space, also referred to as *sigma0*, which contains all physical memory. To allow the user to construct address space layouts as needed, L4 provides the *map* operation. A page of memory can be mapped from one space to another, resulting in both spaces being able to access the page.

2.2.2 Threads

Threads are the schedulable activities that execute program code. A thread runs in only one of the address spaces, whereas an address space can contain multiple threads.

2.2.3 Communication

L4 provides a mechanism for communication between threads, called *inter-process communication* (IPC). L4 IPC is synchronous and unbuffered, which means that if one thread invokes a send operation to another thread, the communication only takes place if the other thread invokes a corresponding receive operation. The map operation (see Section 2.2.1) is a special form of IPC.

2.3 L4 System Calls

The L4 microkernel provides two types of system calls, nonblocking and blocking ones.

Nonblocking System Calls. A nonblocking system call enters the kernel, does its work and returns to user-level immediately. As the name says, these calls do never block the invoking kernel thread.

Blocking System Calls. A blocking system call may block in the kernel. For the time the call blocks, the invoking kernel thread cannot execute user-level code. The only blocking system call in L4 is the IPC system call.

The system calls can furthermore be classified into *short-running* system calls and *long-running* system calls.

Short-Running System Calls. A short-running system call needs a user-predictable amount of time.

Long-Running System Calls. A long-running system call always makes progress, that is, it does not block in the kernel. In contrast to a short-running system call, the time needed is not foreseeable by the caller.

2.4 L4 IPC

The IPC system call provides synchronous communication. Both sender and receiver have to invoke the appropriate function for the communication to take place. Depending on which partner comes first, the IPC has either a *blocking send phase* or a *blocking receive phase*.

Blocking Send. Figure 2.3 shows the flow diagram of a blocking send IPC. Thread A sends to B while B is still running and not yet ready to receive from A. Hence A blocks until B invokes the corresponding receiving IPC. As soon as B does receive, the kernel transfers the message and A becomes ready to run again.

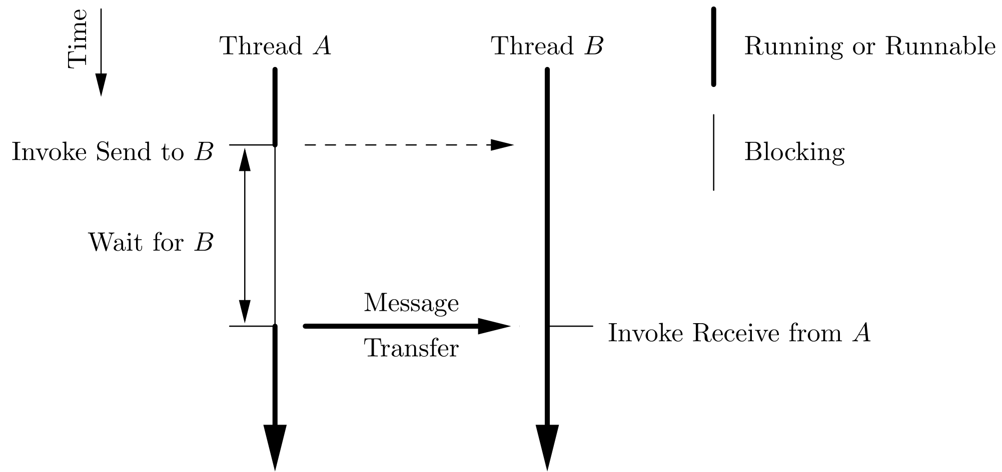

Figure 2.3: IPC with a Blocking Send Phase.

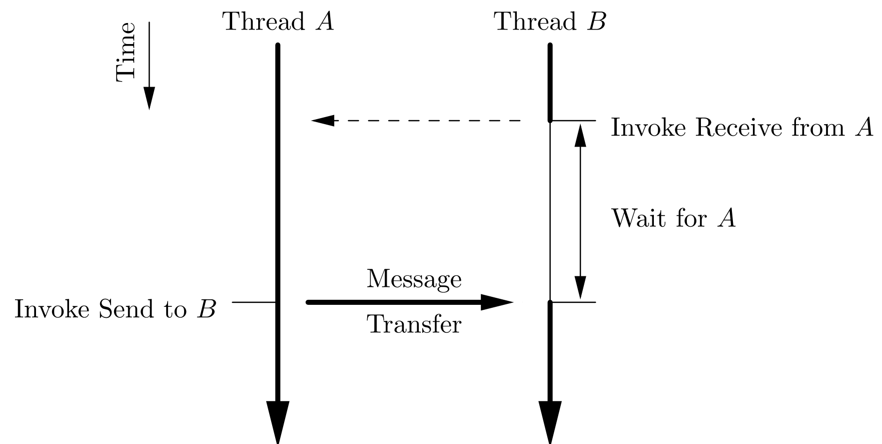

Figure 2.4: IPC with a Blocking Receive Phase.

Blocking Receive. A blocking receive IPC is shown in Figure 2.4. The receiver thread comes first, so it has to wait until the sender thread is ready to send.

The transferred data can consist of actual data (i.e., transferred message registers), map items, or strings, or any combination of the three. A map item denotes the transfer of a memory mapping from the sender address space to the receiver address space, which requires the agreement of the receiver. A string item causes the kernel to copy the contents of a sender-specified buffer into a similar one on the receiver side.

2.4.1 Timeouts

When a thread invokes an IPC system call, it has to specify a *timeout*. If the partner does not respond until the timeout expires, the kernel cancels the operation and the thread is set ready to run again. There are four types of timeouts:

Relative Timeout. The timeout is triggered after a specified amount of time expires.

Absolute Timeout. The timeout is triggered when a specified point in time is reached.

Never. No timeout is triggered. If the intended partner does not respond, the invoking thread will block until the invoker is killed or the IPC is cancelled by an `ex_regs` (see Section 2.4.6) system call.

Zero. The system call is ensured not to block. The operation succeeds if and only if the intended partner is already waiting.

2.4.2 Send and Receive Destinations

A send operation may be targeted only to a single thread. A receive operation, in contrast, can be a *closed wait* or an *open wait*.

Open Wait. The invoker accepts messages from every sender.

Closed Wait. The invoker accepts messages only from the specified sender. Other senders block until an open wait or a corresponding closed wait is invoked.

2.4.3 Blocking Send

If a send operation has a timeout greater than zero and the partner is not ready to receive, the sender blocks until the timeout triggers or the partner becomes ready. If a thread invokes an open wait IPC whilst multiple senders are waiting to send to it, the kernel has to pick one of them.

Fairness

The L4 specification [Lie96] does not demand a specific sender election strategy, but in practice it emerged that providing *fairness* to competing senders is essential for L4 applications. Therefore each kernel provides it, and for a user-level threading system to be useable, fairness will also be a requirement. The known implementations achieve a first-come-first-serve strategy by storing the waiting senders in a *sender queue*. A waiting send operation can be aborted by timeout or by an `ex_regs` system call from another thread (see Section 2.4.6). In this case the aborted sender dequeues itself from the sender queue.

Progress

When a thread blocks in an IPC system call, L4 ensures that other threads still make progress.[3] This is also significant for blocking receive operations.

2.4.4 Map and String IPC

To deliver an IPC message that contains memory mappings, the kernel has to check if the mapping fits into the *receive window* provided by the receiver. If this is not the case, the transfer fails. The kernel notifies both sender and receiver about this by setting the return value accordingly.

The transport of a string IPC is even more complicated. The kernel has to read the data from a sender-provided buffer and copy it to a buffer on the receiver side. On both sides, page faults may occur. On a page fault, the kernel sets up a second transparent message, requesting the faulting thread's pager to resolve the fault. As soon as it is resolved, the kernel continues the string transfer.

2.4.5 IPC Call and Atomicity

L4 allows to combine a send and a receive operation in a single IPC system call. This is also referred to as *IPC call*. On an IPC call, the send phase takes place before the receive phase. If the send fails, the receive phase is skipped. If not, the receive phase begins as soon as the send operation is finished. The kernel guarantees an *atomic send-to-receive switch*, which means that the switch from send to receive phase is unpreemptible. The atomic switch is needed because otherwise a zero-timeout reply could be missed if the partner replies too fast. Figure 2.5 shows a typical IPC call scenario. Figure 2.6 shows what could happen if there were no atomic send-to-receive switch.

2.4.6 Canceling and Aborting of IPC Operations

L4 provides the *exchange registers* system call (`ex_regs`), which is targeted at a specific thread and cancels an ongoing IPC system call.[4] The difference between cancel and abort is that cancel takes place *before* the actual IPC rendezvous, which means that no partner is involved yet. If the partner is already involved, the IPC can no longer be canceled but it is aborted.

If a timeout occurs, the IPC is also canceled or aborted, respectively.

2.4.7 Page Faults

Each L4 thread has assigned another thread as pager. When a page fault occurs, the kernel translates it into an IPC message to the faulting thread's pager. This mechanism makes it possible to implement paging policies at user level. For user-level threads we will have to provide this or an equivalent mechanism.

[3] Actually, it also provides per-thread time reservations and priorities, which must not be influenced if other threads block. This will be of interest when designing user-level schedulers. For this work it is sufficient to state that blocking of one thread must not block others.

[4] This is not the only functionality the `ex_regs` system call provides, but the others are not of interest here. A complete description can be found in [DLSU04].

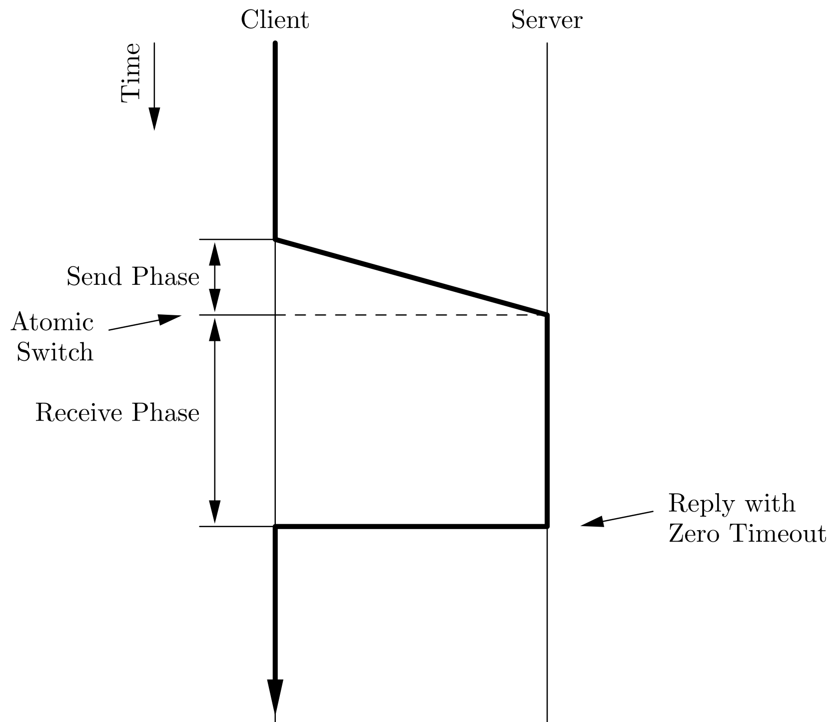

Figure 2.5: Flow Diagram of an IPC Call.

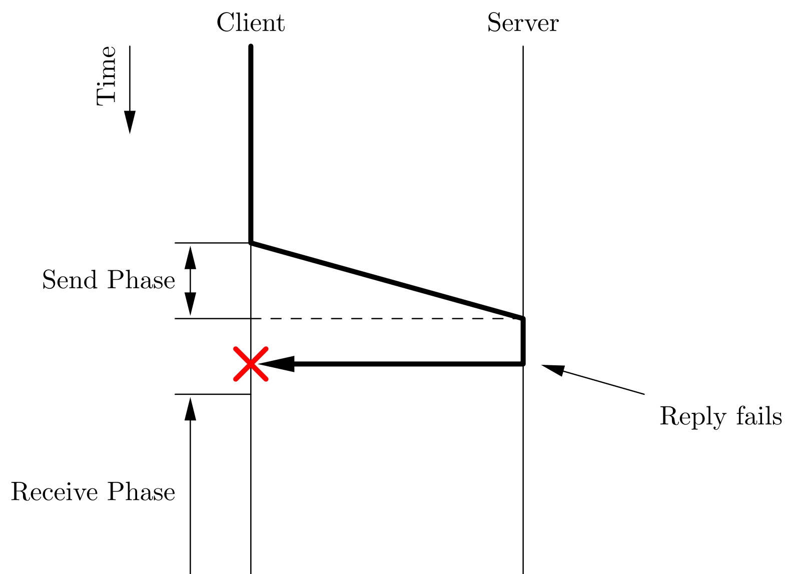

Figure 2.6: Failing IPC Call.

2.5 L4 Kernel Resources

In this Section we describes the resources the existing L4 implementations use to provide the functionality shown above.

For each thread the kernel reserves a *thread control block* (TCB). The TCBs contain information related to the corresponding thread as well as information that potentially affects the entire system. Therefore the TCBs reside in a special resource called *kernel memory*. To be able to provide correct kernel functionality, the kernel makes the following assumptions regarding kernel memory.

Accessible Everytime from Everywhere in the Kernel. In contrast to user memory, which has to be accessible only if the corresponding address space is active, each thread's kernel memory has to be accessible independent of which thread is currently active.

Access Needs a Predictable Amount of Time. Page faults may only be tolerated if they can be resolved by the kernel.

Accessible from Kernel Code Only. The kernel depends on the integrity of kernel-memory contents, and it stores sensitive information that must not be seen at user level.

2.5.1 UTCBs

A second type of kernel resources are *user-level thread control blocks* (UTCBs). A UTCB is a block of kernel memory for which the last of the afore-said assumptions is relaxed. It is read–write accessible within the task in which the corresponding thread resides. A UTCB is nevertheless called a kernel resource, because it is pinned memory and it cannot be revoked by a pager. UTCBs are used for exchanging information between user and kernel.

2.6 L4 Trust Relationships

To straighten out which resources can be used by which component, we provide an overview of the trust structure in a typical L4 system. Figure 2.7 shows an example. The entire system obviously depends on the hardware it runs on. The microkernel is the only component that runs in privileged mode. It needs nothing than correct hardware functionality to work correctly.[5] An application needs a set of servers, which on their part depend on correct kernel and hardware functionality.

2.6.1 Client-Server Trust Relations

In a client-server relationship usually the client trusts the server. The interaction is done by IPC calls from the client side, and the server replies via an IPC send. As the server does not trust the client, it is essential that the client cannot influence the server's functionality. This also covers blocking of server-side resources in the case that the server wants to send a reply and the client is not ready. Hence the server must be allowed to reply with zero timeout (see Section 2.4.1. On the other hand, the client relies on getting the answers from the server, so it needs a way to reliably receive the zero-timeout replies. Therefore L4 provides the IPC call facility, which first

[5] In fact, in the current FIASCO implementation the kernel also trusts the σ_0 server. This server has access to all physical memory, so it could compromise the kernel memory area. This is rather an implementation detail and does not matter, because any other component in the system has to trust σ_0 anyhow.

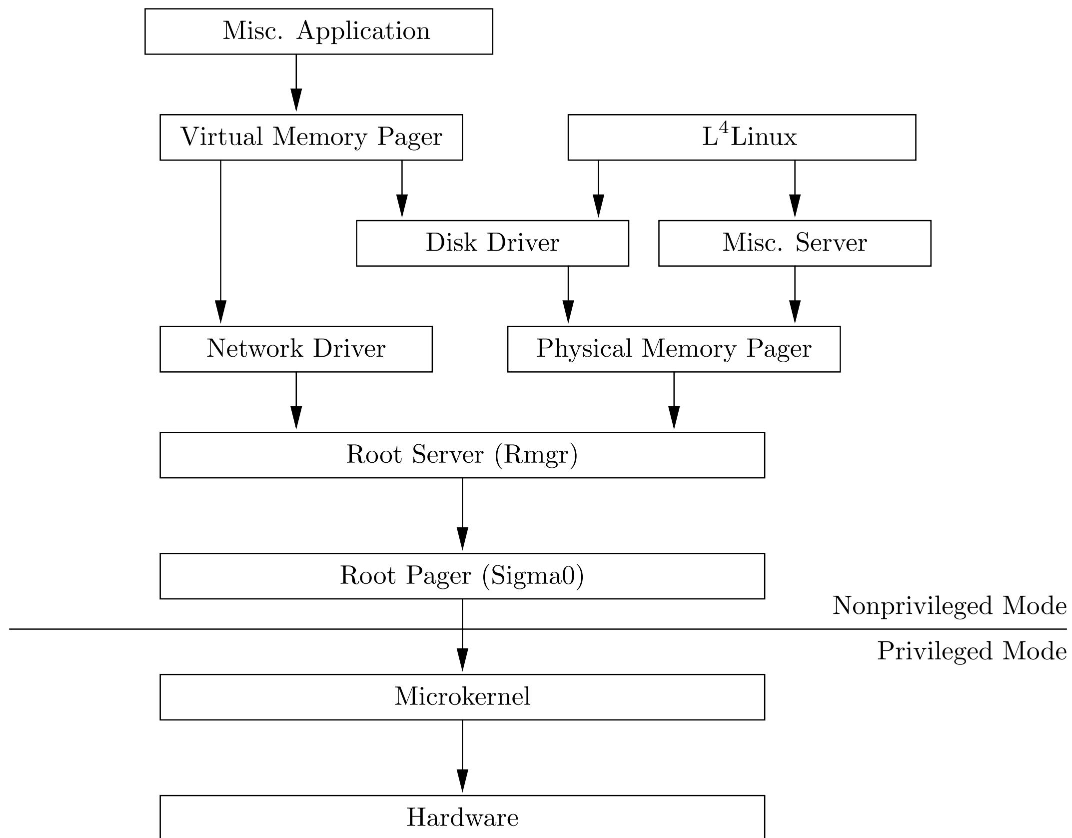

Figure 2.7: Example of the L4 Trust Relations.

sends the message and switches to receive mode as soon as the send has completed (see Section 2.4.5).

The trust structures in the user-level part can be weakened if the design of an application demands it. For example there could be a set of servers working together, thus not forming a strict hierarchy. Large circles in the graph, however, are an indicator of a bad design. Regarding the kernel, introducing dependencies to user-level servers is definitely not wanted. This would break the stability and trustworthiness of the L4 system as a whole, and it also indirectly violates the microkernel principle. The kernel would provide functionality that can also be implemented at user level.

Chapter 3

Related Work

In this Chapter we present related work with similar goals.

3.1 Event-Based Systems

Besides multithreading, another strategy to manage multiple activities is embarked by *event-based systems*. Instead of creating a thread for each activity, the activities are encoded in messages, called *events*. Events can be queued or processed by (kernel-level) threads.

More information about event-based systems can be found in [WCB01] and [BMD99]. My assumption about event-based systems and L4 is that L4 allows event-based systems to be built on top of it. It would use L4 servers to manage the event queues, and pools of L4 threads to handle the events themselves.

As both user-level threads and event-based systems address the same problems, there has been a long-time discussion [DZK+02, vBCB03] about which of them is the better approach. The task of this work is, however, building user-level threads on top of L4 and not building an event based system. Therefore, this discussion is out of scope here.

3.2 Scheduler Activations

Scheduler Activations [ABLL92] introduce various concepts of kernel support for user-level threading. The main goal is to turn a user-level thread into a *first-class object*. This means that user-level threads should be able to use the benefits of kernel-level threads, as shown in Section 1.2 on page 1. Scheduler Activations have been implemented in Solaris and [Wil02] NetBSD.

With Scheduler Activations, the kernel provides a virtual multiprocessor to the application. The kernel controls the amount of virtual processors allocated to an address space. It can allocate or deallocate processors at runtime. Such a virtual processor is called a Scheduler Activation. The kernel notifies the application about processor allocation and deallocation events by invoking an *upcall*. This is accomplished by creating a new activation that starts execution at a specific entry point, or by saving the state of an existing activation and making it jump to this entry point. The user-level threading system in each address space has complete control of which user-level threads run on its virtual processors. It notifies the kernel of the subset of user-level thread operations that can affect processor allocation decisions. User-level threading operations

that do not need to be reflected by the kernel take place entirely at user-level, preserving the efficiency of user-level threading.

3.2.1 K42

K42 [AAD+02a, AAD+02b] is a system designed to support Scheduler Activations right from the start. It is optimized for cache-coherent multiprocessors. The major design goals are to preserve spacial and temporal locality of code and data, and scalability to large-scale multiprocessor machines.

3.2.2 Summary

The goals of the Scheduler Activations concept are the same as the ones of this work. The difference is that the systems that currently implement Scheduler Activations are not microkernel-based. This work investigates the possibilities of applying the Scheduler Activations concept to a microkernel-based system. We aim at building Scheduler Activations using minimal kernel mechanisms.

The most serious borderline to existing implementations of Scheduler Activations is that a second-generation microkernel provides only synchronous and unbuffered communication, which is highly optimized for performance. There will be little scope left to perform upcalls or to do something similar each time a Scheduler Activation blocks. Therefore, we will have to look for more adequate solutions.

3.3 User-Level Management of Kernel Resources on L4

Making kernel resources pageable is a method to optimize kernel-level threads regarding resource usage (see Section 1.2 on page 1). This approach has been pursued in [HE03]. It takes advantage of the fact that not all kernel resources have to meet all criteria stated in Section 2.5 on page 12. Data that does not affect general kernel functionality, but a specific thread only, can therefore be handed out to user-level pagers. As there is also data that affects the whole system, this implies that this approach does not result in completely pageable thread objects.

Another approach is making kernel resources revokable by user-level pagers, but destroy the objects using these resources on revoke. This method is kept track in [PV05] L4/Sec. As the thread objects are destroyed if a pager revokes their resources, threads remain at kernel level and cannot be used to represent activities that stay alive when their resources are paged out (see Section 1.2). The goal is rather making kernel resource usage controllable by the user.

3.4 Local IPC on L4

Local IPC [LW01] is a method to optimize kernel threads regarding communication speed. It allows the threads to communicate without kernel entry if both communication partners share an address space. Local IPC has been implemented in both the Hazelnut [Wen02] kernel and the Fiasco [Reu04] kernel.

The performance of local IPC is actually comparable to the performance of user-level thread IPC. But local IPC cannot be applied in all cases of intra-task communication, and it introduces nonnegligible overhead to kernel-level IPC. The conclusion if local IPC can improve the overall performance of an L4 system has not yet been found out.

3.5 Summary

Kernel-aided user-level threading has already been implemented on various operating systems with positive results, but not on microkernel-based systems. Regarding microkernels, there have already been successful attempts to optimize the kernel-level threads regarding both resource usage and intra-task communication speed.

The focus of this work is to apply the concept of kernel-aided user-level threads on top of a microkernel-based operating system. As a reference, the L4 system and the Fiasco kernel are used, but we believe that the conclusions made in this work will also apply to microkernel-based systems in general.

Chapter 4

Design

4.1 Goals

The goal of this work is to extend the L4 API to improve support for user-level thread packages. For that to be accomplished, the potential design of a user-level thread package on top of L4 has to be discussed.

4.1.1 Characteristics of User-Level Threads

The benefits of user-level threads as stated in Section 1.2 on page 1 should be fully operative. That means in particular:

Fast Cooperative Thread Switch. The user-level thread library should be able to switch from one thread to another thread. The kernel should not restrict this.

Fast Thread Management. It should be possible to create and destroy user-level threads entirely at user level.

User-Controlled Per-Thread Resources. The layout of the user-level thread memory should be under control of the user-level thread library. In particular, the user-level threads should reside completely in user-level resources. This allows the user to omit unused data in the user-level thread contexts, thus building extra light-weight threads. In addition to that, this supersedes kernel interaction on both thread management and switch operations.

The kernel assistance for user-level threads should enable the user-level threads to also benefit from a fraction of the characteristics of kernel-level threads, as stated in Section 1.2. These are in particular:

Preemptive Scheduling. Priorities and Quality-of-Service are *not* covered in this thesis. But a mechanism to trigger user-level thread switches from within the kernel should be offered.

Operating System Services. User-level threads should be able to use the functionality provided by the L4 system calls. We want to provide an interface similar to L4 for user-level threads. Adapting L4 applications to use user-level threads should be possible with reasonable effort.

Multiprocessing. Multiprocessing is out of scope of this thesis. No attempt is made to enable user-level threads to run in parallel if multiple physical processors are available.

4.1.2 Boundary Conditions

Performance

The performance of applications that do not use user-level threads should not decrease significantly.

API Compatibility

The term "API extensions" usually means that the original interface stays as is, so that existing applications can still be used without modifications. For this work this assumption will be relaxed, because the extensions designed here are mainly thought to be included in the L4/Sec version of the L4 interface. At the time of this writing, there was no L4/Sec implementation available, and the specification was not completed. But the fact that L4/Sec will not be compatible to its predecessors was already known. Because of this, interface modifications that require adaption of existing applications are definitely allowed.

However, the L4 API has emerged to be an efficient and well-investigated microkernel interface, and there are lots of applications available. Thus, incompatible API modifications are to be handled with care, and we have to look at all consequences they might have.

Microkernel Principle

The main principle standing behind microkernels is to implement functionality in the kernel only if it would be impossible to implement it at user level [Lie95b]. The fact that L4 API extensions are required to match the microkernel principle is taken for granted.

4.2 Investigation of Per-Thread Data

As shown in Section 2.5 on page 12, the resources the microkernel uses to represent a kernel-level thread consist of both a TCB and a UTCB. The UTCB is only an optimization for transferring data between user-level applications and the kernel. Without loss of generality, we disregard the separation of per-thread data into TCB and UTCB in this Chapter.

The possible states of an L4 kernel-level thread can be grouped into two modes: It either can be running or runnable, or it can be blocking in an IPC.

Running or Runnable. The thread is currently executing, or it is enqueued in the ready queue and waits to be scheduled.

Blocking in an IPC. The thread invoked an IPC system call[1] and it blocks until the intended partner replies or a timeout occurs.

According to these states, the TCB data can be classified into two parts: an *execution context* and a *message context*.

Execution Context. The execution context of a thread is the data the kernel needs when switching from one thread to another. It consists of the thread's processor register values. Corruption of a thread's execution context affects only this thread. Therefore, the execution context needs not to be trusted.

[1] The kernel sets up a transparent IPC if execution causes a page-fault or an exception.

Message Context. The message context consists of the data needed to deliver an IPC message. An IPC does potentially involve other threads, and the kernel issues a set of guarantees regarding thread interaction, as described in Section 2.4. For these reasons, we cannot move the message contexts to user space without further consequences. These consequences are discussed in Section 4.7.

4.3 Dispatching User-Level Threads on Kernel Threads

In this Section we discuss the approach of running multiple user-level threads on top of L4 kernel-level threads. At first, we look at running or runnable threads only. Therefore, only the execution contexts are of interest.

4.3.1 Execution Contexts at User-Level

As shown in Section 4.2, the execution context of a thread does not need to be trusted by the kernel. The execution context of a user-level thread can therefore be safely moved to user level. The layout of the resulting threading model is shown in Figure 2.1 on page 6. A kernel thread functions as a virtual processor and the user-level threads are executed on top of it. The execution contexts are stored in user-level memory.

Dispatching kernel-level threads (i.e., electing a runnable thread and assigning it to the CPU) can only be done by kernel code, because it needs to access the thread states, which reside in kernel space. With user-level execution contexts this is no longer the case. Dispatching user-level threads needs access to the user-level execution contexts only. Therefore, the dispatcher code can be located:

In the Kernel. The code that dispatches the user-level threads is placed inside the microkernel,

On a Dedicated Thread. An extra kernel-level thread is used to dispatch the user-level threads on the virtual processor, or

On the Virtual Processor. The user-level dispatcher code runs on the same kernel thread as the user-level threads.

In the following Sections we discuss for each of these methods how the goals stated in Section 4.1.1 can be achieved. As the focus is still on runnable threads only, the following goals are of interest:

- User-controlled per-thread resources,
- thread management,
- cooperative thread switch, and
- preemptive thread switch.

4.3.2 In-Kernel Dispatcher Code

Using kernel code to dispatch user-level threads requires the kernel to know the layout of the execution contexts. This breaks the goal of user-controlled per-thread resources. Furthermore, the kernel then has to access user-level memory, which would require special cases in kernel page-fault handling (for details, see Appendix A on page 48). As the data the dispatcher code has to

deal with is untrusted, the trustworthiness of the code itself will be of no use. The functionality of switching from one user-level thread to another will be untrusted anyway. These reasons speak against using kernel code to dispatch user-level threads.

4.3.3 User-Level Dispatcher Code

When the dispatcher code is moved to user-level, the goal of user-controlled per-thread resources is fulfilled automatically, as only the dispatcher needs to access the execution contexts. The user-level thread library has complete control over the layout and amount of per-thread data.

Thread Management

Creation and deletion of user-level threads is done by the user-level dispatcher. To create a thread, the dispatcher allocates memory for a new execution context and initializes it. The exact sequence depends on the layout of the user-level thread library. The same applies to thread deletion.

Cooperative Thread Switch

To perform a cooperative user-level thread switch, the dispatcher has to perform the following actions:

1. Store the current state into the user-level execution context.
2. Load the state of the thread to switch to.
3. Resume execution of the user-level thread.

To perform these actions, the dispatcher only needs access to the execution contexts of the affected user-level threads. No kernel interaction is needed. The cooperative thread switch is therefore considered to be as fast as with pure user-level threads.

4.3.4 Dedicated Dispatcher Thread

When focussing on preemptive thread switches, we have to distinguish between two locations where the user-level dispatcher code may run: on a dedicated thread, or on the virtual processor itself. At first we focus on the dedicated-thread variant. Figure 4.1 shows the scheme of this design.

Preemptive Thread Switch

To perform a preemptive thread switch, the user-level thread library has to

1. get a preemption signal from the kernel, and
2. perform the actual switch on the virtual processor.

For the first step, the user-level thread library needs to interact with the kernel. The only way to fetch preemption signals from an L4 microkernel is to block in an IPC with the designated timeout set. As the virtual processor executes user-level code and therefore cannot wait for timeouts, this has to be done by the dispatcher. As soon as the dispatcher receives a timeout, it determines which user-level thread to run next, and performs the switch on the virtual processor. Therefore it has to interrupt the virtual processor and then to perform the same steps as described for the cooperative switch.

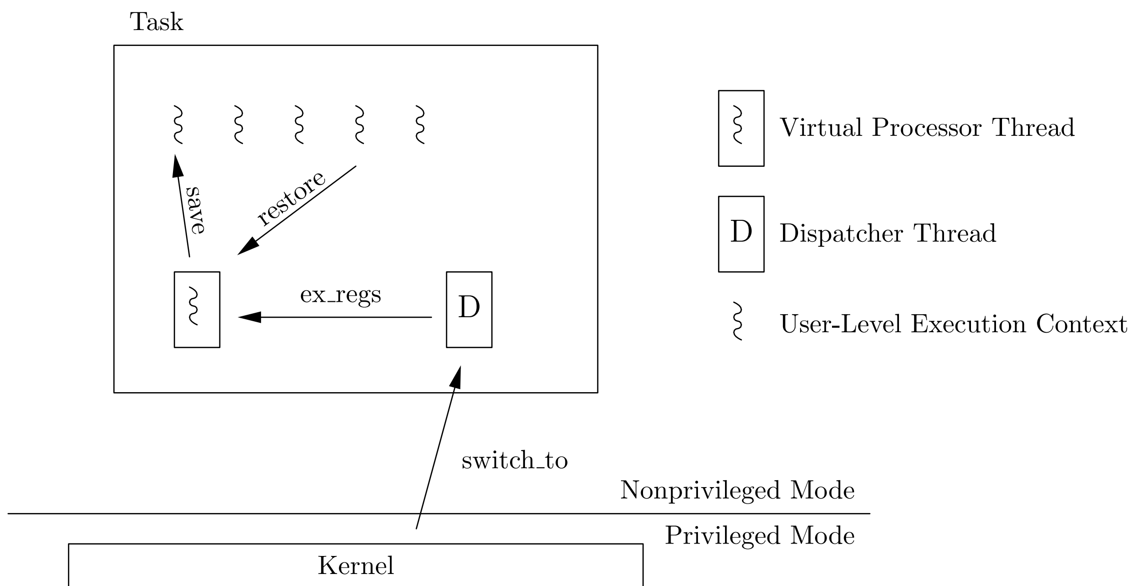

Figure 4.1: User-Level Threads with Dedicated Dispatcher Thread.

Dispatcher and Virtual Processor Priorities To assure the dispatcher to become running at all whilest the virtual processor is executing user-level thread code, the dispatcher has to run on a higher priority than the virtual processor. This has the positive side effect that the virtual processor is assured *not* to run as long as the dispatcher is active. The dispatcher thus does not need to bother about virtual processors running in parallel.

Interaction Between Dispatcher and Virtual Processor The dispatcher needs to change the virtual processor's register values. The only way to access the register values of another thread in L4 is the `ex_regs` system call. This system call allows to set the instruction pointer and stack pointer only, but not the other processor registers. Thus, the dispatcher will not be able to perform the entire switch operation. Instead, it can load the intended register values to a predefined memory location and make the virtual processor jump to the code that performs the thread switch.

Synchronizing Thread Switches A preemptive thread switch triggered by the dispatcher can collide with an ongoing cooperative thread switch on the virtual processor. This could lead to the situation that a half-saved user-level context is executed. To prevent this, preemptive thread switches have to be synchronized with cooperative thread switches. As the dispatcher runs on a higher priority, the synchronization can be enforced by the dispatcher.

Summary

The dedicated-dispatcher-thread model achieves all the goals. Changes to the L4 API are not required. A drawback is the performance of the preemptive thread switch. As it needs an `ex_regs` system call in addition to receiving the preemption signal, it is supposed to be more expensive than a switch between kernel threads.

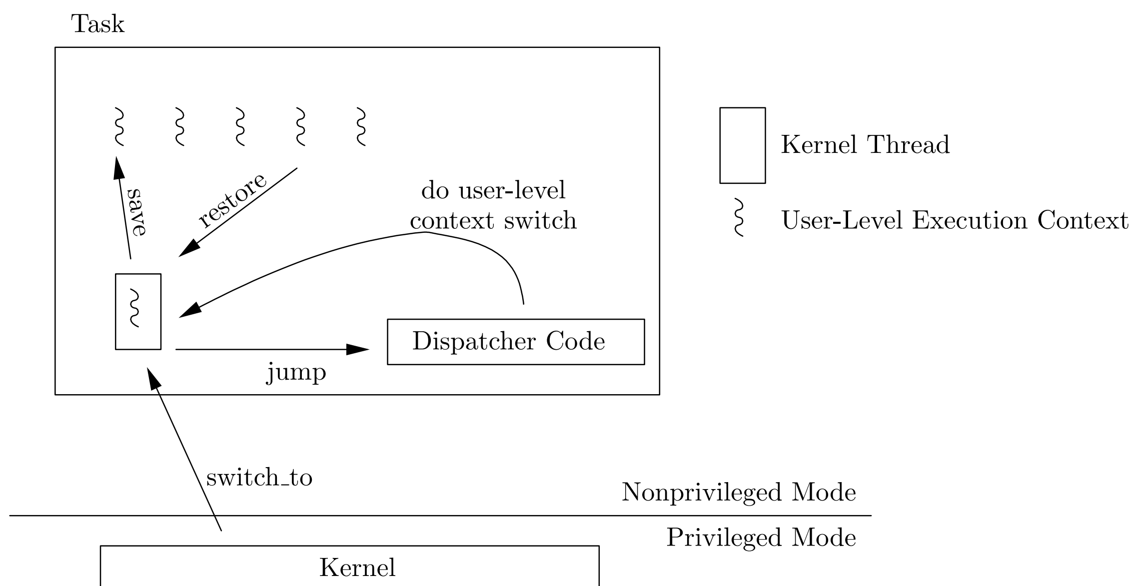

Figure 4.2: User-Level Threads Without Dedicated Dispatcher.

4.3.5 Dispatching From Within the Virtual Processor

As shown in the overview in Section 4.3.1, the dispatcher code can also run directly on the virtual processor. This design is depicted in Figure 4.2.

To accomplish preemptive scheduling, we need a new method to fetch preemption signals from the kernel. IPC cannot be used, because we have only the virtual processor that executes user-level thread code. It cannot execute and wait for a timeout IPC simultaneously. For this reason, a kernel extension is needed.

This extension could be a mechanism that enters a user-specified dispatcher entry point each time the kernel thread is scheduled. The kernel activates the kernel thread and jumps to the dispatcher entry. The dispatcher code decides which user-level thread to run next, saves the current user context and loads the new one. The synchronization issues described in Section 4.3.4 do not occur, because there is no dispatcher thread. Instead, there will be synchronization issues between the virtual processor and the kernel. When the kernel makes the virtual processor thread jump to an entry point, the operation must be ensured not to collide with potential ongoing cooperative thread switches. This synchronization has to be handled by the kernel, which by nature runs at a higher priority than the virtual processor. Therefore the kernel is able to handle it, but it requires another API extension.

As a summary, the API extensions needed by this model are the following:

Jump to Entry Points. The kernel has to make the virtual processor jump to a specific entry point if an asynchronous timeout (see below) occurs, or whenever scheduling this thread. Before triggering this jump, the kernel has to save the state of the virtual processor[2] in a memory location the virtual processor has access to.

Asynchronous Timeouts. If the entry-point jump should not be performed each time the

[2] As an optimization, saving only a part of the state may suffice. This depends on the layout of the dispatcher code.

virtual processor is scheduled, we need asynchronous preemption signals. But L4 provides timeouts only for threads that block in an IPC. For asynchronous preemption signals, the kernel has to be extended to administer timeouts for running threads.

Virtual Processor Synchronization. When jumping to an entry point of a virtual processor, the kernel has to check if the virtual processor is already in a critical section. If this is the case, it can either discard the timeout, or perform another asynchronous action at the virtual processor, for example, incrementing a lost-timeout-counter. Waiting for the virtual processor to finish the critical section is not an option, because the kernel then would have to trust the user-level task.

For this model, the following data has to be shared between kernel and dispatcher:

- Entry points,
- a save-state area,
- a lock for synchronization, and
- a lost-interrupt counter.

These structures cannot reside in ordinary user-level memory, because the kernel should not know anything about user-level thread structures (see Section 4.1), and kernel access to user-level memory is problematic. Therefore it has to be a kernel resource, but it also has to be accessible by the dispatcher code that runs at user level. For sharing information between user-level and kernel, L4 provides UTCBs (see Section 2.5.1 on page 12). Thus the dispatcher data structure can be a part of the virtual processor's UTCB.

Summary

The design of dispatching user-level threads without dedicated dispatcher achieves all the goals. A drawback is that it needs nontrivial kernel extensions.

4.3.6 Summary

The above discussion shows that moving execution contexts to user-level is considered to work and to achieve the goals stated. A precondition is that the code that dispatches the user-level threads has to be moved to user level, too. The individual benefits and drawbacks of the two dispatcher models are the following:

- **Dedicated Dispatcher Thread.**
  - Can be built on top of current L4, no kernel extensions are needed.
  - Costly preemptive thread switch.
  - Requires second kernel thread.
- **Dispatcher Code on Virtual Processor.**
  - Cheap preemptive thread switch.
  - One kernel thread only.
  - Requires API extensions.

4.4 Long-Running System Calls

When a user-level thread invokes a long-running system call (see Section 2.3 on page 7), this thread will occupy the virtual processor until the system call completes. As long as the virtual processor is occupied, other user-level threads waiting to be executed on this virtual processor cannot make progress. As shown in Section 2.4.3 on page 10, it is desired that each user-level thread makes progress until it blocks in an IPC. To cope with the situation, three approaches are imaginable:

Tolerate Blocking. Let the user-level task wait until the long-running system call is completed.

Deschedule the User-Level Thread. Make long-running system calls interruptible.

Use More Kernel Threads. Move the waiting user-level threads to a free kernel thread.

4.4.1 Tolerate Blocking

The trivial solution is letting the user-level task wait until the long-running system call completes. This weaks the goal of enabling all user-level threads to make progress. As user-level threads residing in the same task have to trust each other anyhow, and a long-running system call does not block but only needs a unforeseeable amount of time, allowing this constraint might be viable. This depends on the exact purpose the user-level thread package should serve.

4.4.2 Deschedule Invokers of Long-Running System Calls

Enabling the user-level threads to make progress when the virtual processor is occupied by a long-running system call could be achieved by descheduling the invoker and restarting the long-running system call as soon as the invoker is scheduled again. For this to be accomplished, long-running system calls have to be

1. interruptible and

2. restartable or continuable.

Interrupt Long-Running System Calls

Interruptability of a long-running system call can be achieved by making the `ex_regs` system call able to abort long-running system calls. This requires a kernel extension. The kernel has to ensure that kernel data structures are in a consistent state when interrupting the system call.

Restart or Continue Long-Running System Calls

Restarting a system call requires the starting position to be regenerated, to be still unmodified, or to be irrelevant. Continuing a system call requires the invoker to know the progress of the formerly incompleted call. The kernel therefore has to transfer the progress information to the user when a system call is aborted. Both methods can be applied and they cause that the user-level threads besides the invoker of the long-running system call make progress.

Progress of the Invoker

Of course the goal of making progress also applies to the invoker of a long-running system call itself. We thus have to investigate if it can still make progress when the thread package deschedules it and restarts or continues the long-running system call later. Progress is ensured if the system call is not interrupted until it has completed a part of its work. The amount of time that is needed until this is the case depends on the individual system call. There are three long-running operations in L4: string copy, map, and unmap.

String Copy. A string copy operation needs no specific preparation time and it can safely be interrupted. On restart, the string copy operation can be resumed at the position where it was preempted.

Map. A map operation inserts mapping nodes into the mapping database in the kernel. It can be interrupted as soon as a mapping node has been inserted. The time until this happens can be deterministic. This depends on the implementation of the mapping database.

Unmap. An unmap operation deletes mapping nodes or subtrees from a mapping tree in the mapping database. A detailed investigation of mapping database design is out of scope of this work, so we mention only the known problems. Before a node can be deleted, parts of the mapping tree have to be locked. This potentially takes long. The approach of introducing preemption points into a mapping database has been extensively discussed in [Voe02]. That thesis concludes that the time until an unmap operation makes progress cannot be foreseeable.

Summary

In case of the unmap operation, the invoker's ability to make progress cannot be assured. Besides this constraint, descheduling user-level threads on long-running system calls is considered to work. API extensions are required.

4.4.3 Use Multiple Kernel Threads

When an invoker of a long-running system call occupies the virtual processor, we could move the other user-level threads to another virtual processor. This increases the maximum number of concurrent running system calls up to the number of virtual processors available. The same approach is described in Section 4.6 regarding blocking IPC calls. This method will work, but it does not solve the problem of aborting long-running system calls if all virtual processors are occupied. For that to be accomplished, long-running calls have to be interruptible as described in the former Section.

4.4.4 Summary

The handling of long-running system calls depend on the exact goals the user-level thread package has. If blocking of the user-level task for the time a long-running call runs is tolerable, this issue can be ignored. Otherwise, possible solutions could be handling by many-to-many threading, or addition of appropriate API extensions.

4.5 Dispatching IPC Messages

When a user-level thread invokes an IPC system call that blocks, the whole user-level task will be blocked until the IPC returns. This is also the case if a user-level thread triggers a page fault. As shown in Section 2.4.3 on page 10, it is desirable that all user-level threads can make progress unless they block in an IPC. Thus we have to investigate methods how to prevent the whole task from blocking.

To enable the remaining user-level threads of a blocking task to run again, the following approaches are thinkable.

Use Multiple Virtual Processors. The blocking virtual processor remains blocking, but we use more than one virtual processor. The other user-level threads can continue to execute on another virtual processor.

Deschedule Blocking User-Level Threads. We break free the blocking user-level thread from its virtual processor, enabling the virtual processor to execute again. Regarding the blocking of user-level threads, we have to save the blocking state at user level.

In the following Sections, we discuss these both approaches individually.

4.6 Multiple Virtual Processors

When dispatching user-level threads on top of multiple virtual processors, there are basically two possibilities of assigning the user-level threads to the virtual processors:

Static Assignment. Each virtual processor owns a fixed set of user-level threads that run on top of it. Each user-level thread remains on its virtual processor. This results in having multiple many-to-one threading systems (see Section 2.1 on page 5). This model does not introduce new conflicts. It works with the very constraint that a blocking user-level thread blocks all others that are on the same virtual processor. However, this does not suffice to meet the requirement that all threads except those that block in an IPC make progress.

Dynamic Assignment. The user-level threads can be moved from one virtual processor to another. This results in a many-to-many threading model, as described in Section 2.1 on page 5. This design has been depicted in Figure 4.3

4.6.1 Many-to-Many Threading

When a user-level thread blocks on a virtual processor, the other user-level threads waiting to be executed on it are moved to another kernel thread. In the example at Figure 4.3 we have three kernel threads, each of them with its own user-level ready queue. When one of them blocks, the ready queues are rearranged to allow the user-level threads to make progress on another kernel thread. In analogy to the discussion regarding thread switches in Section 4.3, the ready-queue modifications have to be done by user-level code, but only the kernel has the knowledge about blocking and unblocking virtual processors. To transfer this knowledge to user level, we can:

Use a Dedicated Dispatcher. On a block or unblock event, the kernel sends a message to a dedicated dispatcher. The dispatcher updates the ready queues and then waits for further block and unblock notifications.

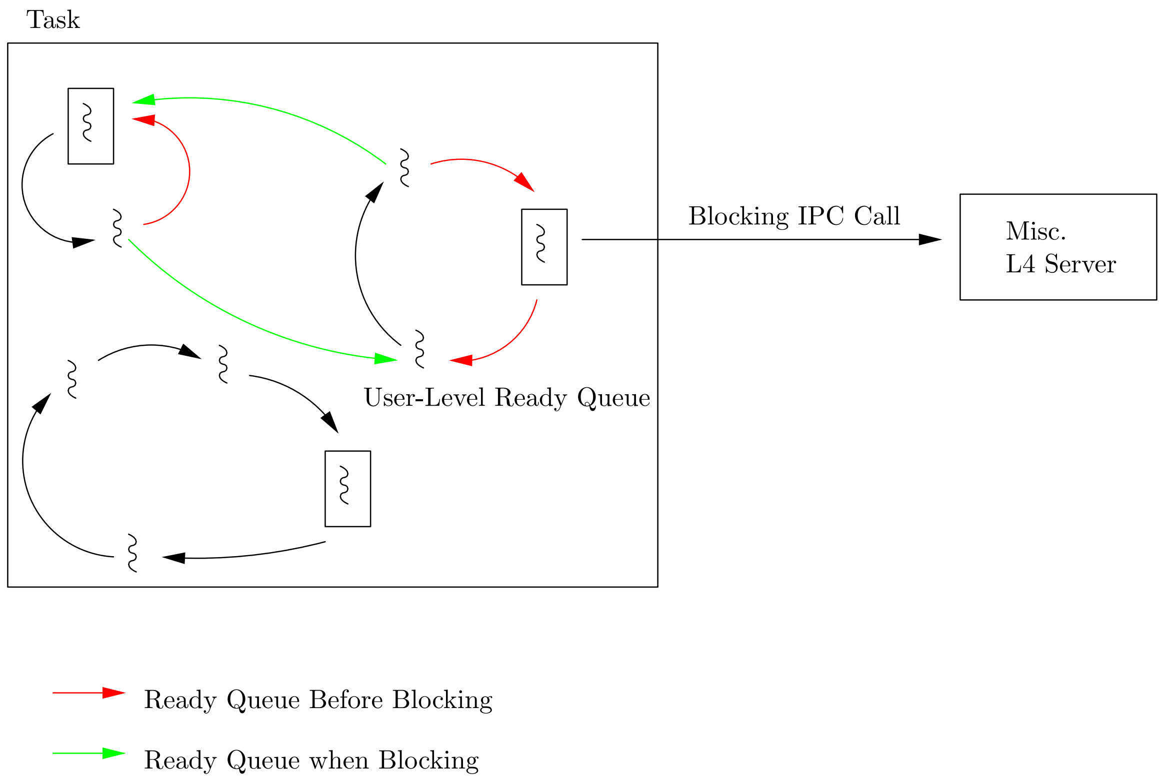

Figure 4.3: Many-to-Many Threading. User-Level Ready Queues at a Blocking System Call.

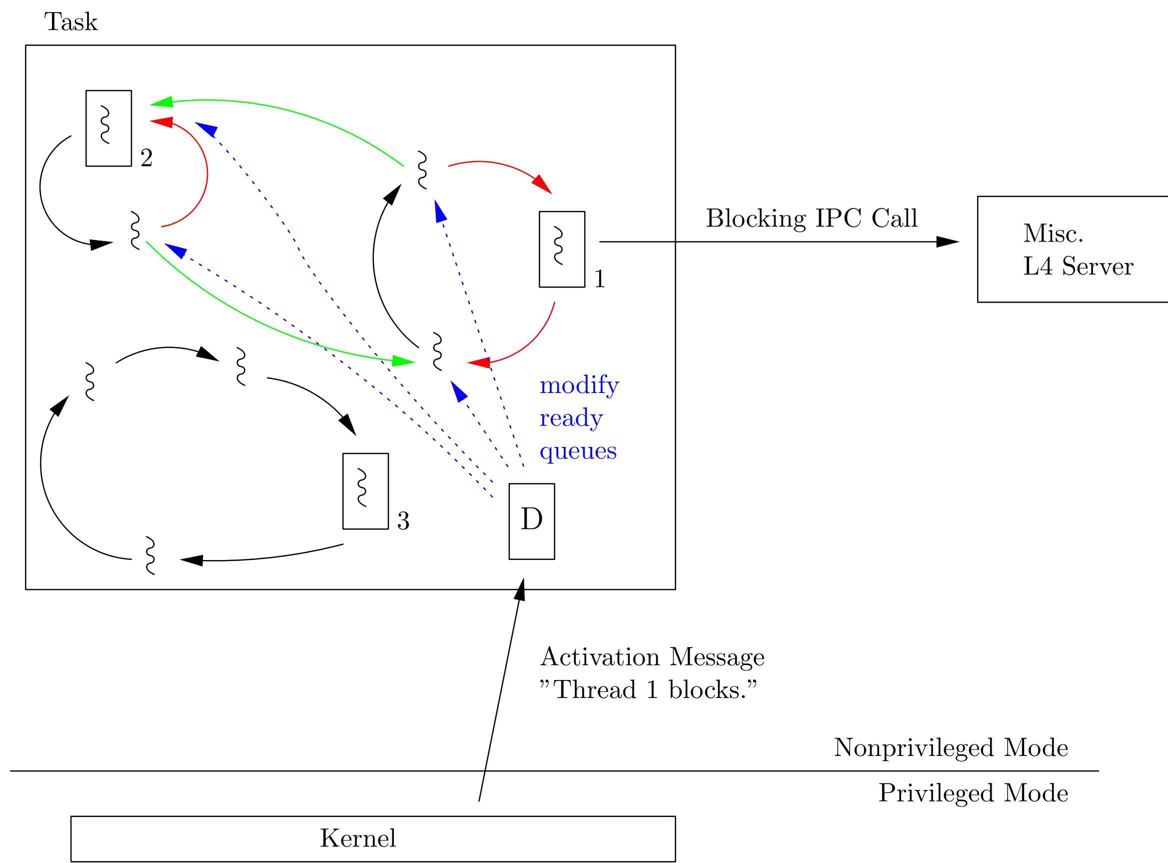

Figure 4.4: Activation Messages.

Dispatch From Within a Virtual Processor. Analog to the model described in Section 4.3.5, the kernel saves the state of the blocking virtual processor in a dispatcher data structure and makes it jump to a specific entry point. The virtual processor then performs the ready queue modifications, reloads the saved state, and continues to run. Unfortunately, this approach does not work for block events. At the time the kernel knows that a virtual processor invoked an IPC system call, this virtual processor already entered the kernel and thus is no longer available to run user-level code. Therefore block notifications do not work without a dedicated dispatcher. Unblock events might work as described in Section 4.3.5, but they would rise synchronization issues between the virtual processors among each other, because we have now more than one virtual processor. Introducing two concepts, a dedicated dispatcher for the block events and entry points for unblock events, is not minimal.

4.6.2 Activation Messages

The user has to provide a dispatcher thread, to which the kernel has to send the *activation messages*. The dispatcher is then able to perform the user-level ready queue modifications. This scheme is shown in Figure 4.4.

4.6.3 Local Thread IDs and Local Communication

Due to the fact that a user-level thread is no longer bound to a specific kernel thread, kernel thread IDs can no longer be used to identify user-level threads. Instead, the user-level threads need their own IDs. The communication between user-level threads cannot use kernel IPC for two reasons:

Local IDs are not Kernel-Known. A user-level thread cannot be addressed by kernel IPC, because the kernel does not know anything about user-level thread IDs.

Speed and Pageability. User-level threads should be pageable, and the communication should be cheap (see Section 4.1.1). When using kernel IPC, these desired benefits of user-level threads would be nonexistent.

Thus we need a separate communication mechanism for user-level threads. As user-level threads of the same task have to trust each other, this mechanism underlies more relaxed requirements than kernel IPC does. Especially, timeouts (see Section 2.4.1 on page 9) need not to be supported. To build such an intra-task communication mechanism, we probably need atomic sequences at user level. This could be achieved by using the *delayed preemptions* feature included in the x.2 version of L4.

4.6.4 Analysis

The many-to-many threading model with a dedicated dispatcher thread and activation messages provides a simple kernel mechanism that allows the user to implement a threading system similar to Scheduler Activations (see Section 3.2 on page 15), whereas the jobs of assigning user-level threads to kernel threads and allocating kernel threads have been delegated to the user level. The model meets the requirements stated in Section 4.1. No restrictions to the L4 IPC interface are made. Threads not involved in communication are not forced to hold out the communication-related resources.

One constraint is that the maximum number of user-level threads blocking in inter-task communication is limited to the number of kernel threads available in the thread pool. In this situation the dispatcher thread is still alive, so the user has the chance to react.

The situation that all kernel threads in a user-level task are blocking cannot be resolved by moving the remaining user-level threads to a still running virtual processor. Instead, the following other reactions are thinkable:

Enlarge the Thread Pool. The user-level task can create new kernel threads and move the blocking user-level threads to execute on top of them.

Abort IPC Operations. As the user has the knowledge, which of the blocking operations are essential and which can be safely aborted and retried later, it can abort the latter ones by an `ex_regs` system call and move other user-level threads to these kernel threads.

Do Nothing. Of course the user-level task can also wait until the IPC operations unblock, or wait a specific time and do one of the previously named reactions only if the situation did not self-adjust.

Communication with Stateful Servers

Another constraint is the reachability of user-level threads for inter-task communication. A user-level thread is system-wide visible only if it stays in a kernel IPC. The visible thread ID is the

ID of the kernel thread on top of which it currently executes, which means that the visible ID may change. This will lead into problems if the user-level thread interacts with a server that saves status information of its clients. To solve this problem, four methods are thinkable:

Make Stateful Servers use Other IDs. Servers could be modified to use client-provided IDs instead the L4 thread IDs to identify a client. This is probably a bad idea, because it changes the server protocol, so that all other clients of such servers have to be adapted. For servers built for serving user-level tasks only, however, this method might be viable.

Use a Wrapper. We could use a kernel thread as a wrapper. The user-level thread uses user-level IPC to communicate with the wrapper, which on its part talks to the stateful server. This would introduce the resource overhead of having one more kernel thread, and it introduces the performance hit of buffering the communication by the wrapper. The advantage is that neigher the server nor the user-level threading system need to be modified.

Pin User-Level Threads to a Kernel Thread. The user-level threading system could provide a mechanism to execute a sequences of code on one specific kernel thread. This would not require modifications of the server and does not introduce the wrapper overhead, but the user-level threading system will become more complicated. A kernel thread might become a bottleneck if too many user-level threads use it for external communication.

Use an Endpoint. L4/Sec introduces *IPC endpoints*, a facility for making multiple kernel-level threads visible under a single endpoint ID. User-level threads could then allocate a dedicated endpoint for the communication to an stateful external server. This is comparable to the wrapper method, but the endpoint IPC minimizes both resource and performance overhead by requiring only an endpoint and not an entire thread as wrapper.

4.7 Deschedule Blocking User-Level Threads

To enable the virtual processor to execute while a set of its user-level thread block in an IPC, we need to move the message contexts of the blocking threads to user space. The administration of the user-level message contexts behaves similar to execution contexts. There are once again two methods thinkable: use a dedicated kernel thread as dispatcher, or use entry points and execute the dispatcher code on top of the virtual processor itself. These issues have already been discussed in Section 4.3.4 and 4.3.5, respectively. Figure 4.5 and 4.6 show an inter-task IPC between two user-level threads with and without a dedicated dispatcher thread, respectively.

4.7.1 Dispatcher Type

The approach of descheduling blocking user-level threads implicates that the virtual processor itself does no longer block at all. Therefore, the no-dedicated-dispatcher method does not have the restrictions found out in Section 4.6.1. The two dispatcher types are rather equivalent regarding the functionality they can achieve. In the following Sections, we discuss the achievable functionality only, thus the dispatcher types can be disregarded.

4.7.2 IPC Scenarios

To be able to systematically explain the user-level thread IPC, we separate the scenario shown in Figure 4.6 into several parts. This scenario shows an inter-task IPC from one user-level thread to another one. Without loss of generality, we can also look at an IPC between a kernel thread and

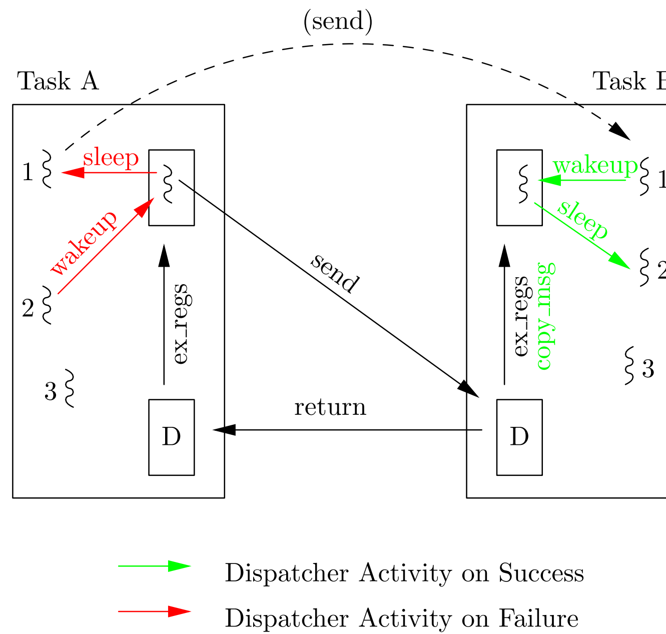

Figure 4.5: User-Level Thread IPC with Dedicated Dispatcher Thread.

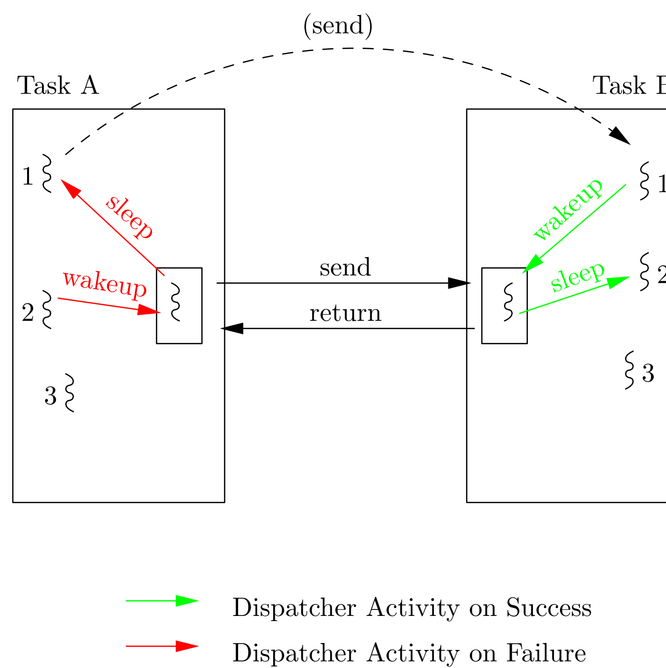

Figure 4.6: User-Level Thread IPC Without Dedicated Dispatcher Thread.

a user-level thread. As described in Section 2.4 on page 7, an IPC always involves two partners, one sender and one receiver, and it has either a blocking send phase, or it has a blocking receive phase. If one of the partners is a user-level thread, we also have to distinguish between who is the one that blocks, the user-level thread or the kernel thread. The IPC to and from user-level threads can thus be grouped into four categories:

User-Level Thread Blocks on Receive. A user-level thread invokes a receive operation from a specific kernel thread, or an open wait, and it blocks until a sender replies.

User-Level Thread Blocks on Send. A user-level thread invokes a send operation to a kernel thread and blocks until the kernel thread replies.

Kernel Thread Blocks on Receive. A kernel thread invokes a receive operation from a user-level thread, or an open wait, and blocks until a sender replies.

Kernel Thread Blocks on Send. A kernel thread invokes a send operation to a user-level thread and blocks until the user-level thread replies.

In the following Sections, these four scenarios are discussed separately. For each of them we have to discuss three phases: the blocking phase, the rendezvous, and the message transfer. We have to find out how the requirements of the L4 IPC, which are shown in Section 2.4 on page 7, could be achieved.

4.7.3 User-Level Thread Blocks on Receive

At the blocking phase, the user-level thread is ready to receive and it does not own the virtual processor. The rendezvous takes place as soon as a sender sends a message to it. The dispatcher receives the message, forwards it to the user-level thread, and changes its state to be ready-to-run again.

Zero Timeouts

If the kernel thread sends with a timeout of zero, the dispatcher has no time to react to the send operation. But the dispatcher needs time to find out if the send operation is valid. This is the case if the user-level thread:

Is Ready to Receive. The dispatcher has to look into the user-level thread's message context and determine if it is ready to receive a message,

Has a Receive Window. In case of a map or string IPC, the dispatcher has to check if the message sent fits into the receive window provided by the user-level thread.

The ready-to-receive check, in particular, has to be performed on every IPC. Therefore, zero timeouts will not work in any case.

A possible solution for this dilemma is to forbid zero timeout send operations to user-level threads. The sender then has to specify the maximum time in which the dispatcher has to deliver the message. As the user-level thread resources should be pageable, this must be a large timeout to make the model actually work. But, zero-timeout senders send with timeout zero because they do not trust their receivers. When these senders now block for long times until the receiver's dispatcher has paged in the thread state (or a malicious dispatcher lets simply wait them), the L4 trust relations (see Section 2.6 on page 12) are broken.

4.7.4 User-Level Thread Blocks on Send

The user-level thread blocks in a ready-to-send state. As soon as the kernel thread invokes a receive operation, the dispatcher delivers the message and wakes up the user-level thread.

Open Wait and Fairness

If the kernel thread invokes an open wait, we need a method to find the blocking sender(s). If there are more than one sender, we further have to select one of them and ensure fairness (see Section 2.4.3 on page 9). To ensure fairness between kernel threads, L4 uses a distributed sender queue. Thus the most obvious approach is to extend the sender queue mechanism to support user-level threads, too.

The sender queue of a receiver stores the senders that are waiting to send a message to it. The queue is distributed across the sender TCBs. Threads not wanting to send any longer dequeue themselves. This is essential, because a sender queue can potentially be long, and the kernel must not be required to iterate over lots of stale entries until it finds a real sender. In addition to that, a thread enqueued somewhere in the queue must not be able to compromise it. Otherwise the threads enqueued behind it would no longer be allowed to complete their send operation. Because of this, moving the sender queue to the sender's user-level thread contexts will not work.

An alternative is to store the sender list in the receiver's memory. This approach has already been pursued by K42 (see Section 3.2.1 on page 16). The problem is that the receiver does not have enough resources to store the thread IDs of each sender if a largish amount of senders want to send concurrently. In K42 this is solved by using a bit field where each bit matches to a set of sender thread IDs. When the receiver becomes ready, it sends a wakeup signal to every sender whose bit is set to 1. The senders then try to send once again if they are still ready to send. The sender responding first wins and starts the message transfer, whereas the others set their bits to 1 again. This method works, but it is not able to guarantee fairness, and it generates a lot of overhead if we have large amounts of senders. As stated in Section 2.4.3 on page 9, we need fairness and introducing this overhead is not desired for L4.

The conclusion is that trusted resources for the sender queues are required. Open wait and fairness will not work in this scenario.

Zero Timeouts

A zero-timeout receive by the kernel thread will not work for the same reasons as shown in the previous scenario.

4.7.5 Kernel-Level Thread Blocks on Receive

The kernel-level thread blocks in a ready-to-receive state. The rendezvous takes place as soon as the user-level thread invokes a send operation. The user-level thread then copies the message to the kernel thread. As an option, the message transfer could also be done by the dispatcher.

This scenario will work as expected. The sender has to trust its dispatcher anyway, so a zero-timeout send operation can give the dispatcher the time it needs. There are also no problems regarding fairness, because fairness is only relevant on blocking send operations.

4.7.6 Kernel-Level Thread Blocks on Send

The kernel-level thread blocks in a ready-to-send state. The rendezvous takes place as soon as the user-level thread invokes the receive operation. The dispatcher then receives the message.

This scenario will also work. Fairness can be achieved by the usual sender queue method, because the thread that blocks is a kernel thread and thus can be enqueued. Regarding zero timeouts, the situation is the same as in the last scenario. The user-level thread trusts the dispatcher and thus it can give the dispatcher the required amount of time.

4.7.7 Summary

The previous discussion shows that essential characteristics of the L4 IPC get lost as soon as blocking user-level threads are descheduled. In particular with regard to zero timeouts and fairness, the approach of descheduling blocking user-level threads in L4 is therefore not applicable.

4.8 Split TCB into Execution Context and Message Context

The previous discussion shows that moving the execution contexts to user-level will work, but moving the message contexts to userland will not. As a consequence, the kernel does not need to allocate an execution context for every thread. Instead, the execution context must only be present in the kernel as soon as the thread is running or ready to run. This will save kernel resources for threads that are blocking in an IPC.

When a thread invokes a blocking kernel IPC, the kernel has to transfer its execution context to user-level, and when the IPC finishes, the execution context has to be transferred back to the kernel again. Hence the kernel needs a mechanism to exchange information with user level, and this mechanism must not occupy kernel resources. As shown in the previous discussion, the only way to exchange information with user-level that does not break the goals stated in Section 4.1 is to communicate with a user-level dispatcher. That implies that dispatcher threads are needed, and context splitting can only be done on threads that have a dispatcher assigned. This model therefore can be treated as an optimization of the model discussed in Section 4.6.

4.8.1 TCB Data in the Fiasco Microkernel

Table 4.1 shows the members of a TCB in the current Fiasco implementation. When trying to classify these members whether they belong to the execution or the message context, it turns out that most of them will belong to the message context. To switch to a thread and execute it, we need only its user-level processor registers, which are saved on the kernel stack.

The kernel stack is the largest part of the TCB. The kernel needs this stack at the time a process enters kernel mode. This is the case when the thread invokes a system call, but also when it is preempted because a thread with a higher priority becomes ready to run. For these reasons the kernel stack must always be present as long as the thread exists. Separating the execution context from the other data saved on the kernel stack will work, but in case of the Fiasco kernel it will require a major redesign of the kernel code.

4.8.2 Amount of Saved Resources

As shown above, the execution context consists of the user-level processor registers. Hence the amount of resources saved by the context-split model depends on the processor architecture. On architectures with small register sets like IA-32, little memory will be saved, but on modern architectures with large register sets and/or 64-bit wide registers, the resource saving are considered to be relevant.

Sender	Receiver	Context	Thread
Global ID	IPC Partner	State Word	Deadline Timeout
Send Partner	Receive Registers	Kernel Stack Pointer	Space
Sender Next *	Pagein Address	Time Donatee	Lock
Sender Prev *	Pagein Error Code	Helper	Pager
	Sender First	Lock Count	Preempter
	Receive Timeout	Lock	Present Prev *
		Scheduling Context	Present Next *
		Time Period	IPC Window Addresses
		Scheduling Mode	
		Maximum Controlled Prio	
		FPU State	
		Consumed Time	
		Ready Prev *	
		Ready Next *	
		UTCB Kernel *	
		UTCB User *	Kernel Stack

Table 4.1: An L4 Thread Control Block.

4.8.3 Summary

The context split model is an optimization of the many-to-many threading model discussed in Section 4.6 regarding kernel resource usage. It is considered to work and meets the requirements stated in Section 4.1. The drawbacks are that it needs kernel extensions, these extensions are difficult to implement in Fiasco, and on architectures with small register sets like IA-32, the achievable resource savings are low.

4.9 Summary

The observations made in the previous discussion can be summarized as follows:

One-to-Many Threading. Providing L4-like semantics for one-to-many user-level threads is impossible.

Many-to-Many Threading. For many-to-many user-level threads we can provide an L4-like interface with passable constraints. It needs the kernel extension of *activation messages*.

Context Split. As an optimization to the many-to-many threading model, we could save further resources by splitting the kernel-thread context into execution and message context. This is difficult to implement in the Fiasco microkernel.

Starting from this, we are going to implement the activation mechanism in Fiasco. Due to temporal constraints, both the implementation of the entire user-level thread library and the context-split model cannot be done within the scope of this work. They are considered to be future work.

Chapter 5

Implementation

The goal of this work was to design kernel mechanisms to support user-level threading systems. These mechanisms are intended to be included in one of the development versions of the L4 interface. At time of this writing, the most recent development L4 version with working implementations available was L4/x.2. That's why I used Fiasco/x.2 to implement the prototype. Another reason for chosing Fiasco/x.2 was that I implemented [Cla04] the x.2 port, so I was already familiar with the Fiasco/x.2 code. However, most of the code evolved to be API-independent, so that backporting to v2 is considered not to be a large effort.

On the user-level side, I used the simple pingpong benchmark program coming with the [Gro03] Pistachio distribution. I extended it to spawn a dispatcher thread and report incoming activation messages. This did the job to test if the mechanism works. To determine if it is actually useful, an entire threading library would have to be implemented. Unfortunately this could not be achieved in the timeframe of this work.

This Chapter is divided into two parts. In the first part we describe the implementation of the kernel mechanism in Fiasco, and in the second part we propose a design for a user-level thread package that uses this mechanism.

5.1 Kernel Implementation

5.1.1 Set the Dispatcher

The user has to be able to set the dispatcher of a thread. There are two system calls thinkable to be extended for it, the `ex_regs` system call or the `thread_control` system call. For practical reasons, I choose the `thread_control` system call. It got just another parameter to set the dispatcher. Fortunately, the L4/x.2 API provides UTCBs, so the lack of processor registers available to transfer system call parameters does not matter any longer. On the IA-32 binding, the dispatcher parameter is transferred in the first message register in the UTCB.

5.1.2 How to Send Activation Messages

There were already two mechanisms implemented in the kernel to send transparent IPC messages: page fault messages and timeslice overrun messages. A page fault message is sent from within the faulting thread's context, targeted to its pager. They have an infinite timeout, and the faulting thread blocks until the pager replies. Therefore, the ordinary kernel stack of the faulting thread can be used to send the page fault messages.

Timeslice overruns are sent to the thread's preempter. In contrast to page faults, the overrunning thread does not wait for a reply, but it continues executing also if the preempter is not ready to receive the message. Furthermore, the overrunning thread can send normal IPCs independently from pending timeout messages. This is implemented by using the receiver's kernel stack to transfer the message. Therefore, each thread has a second sender role for timeslice overrun messages.

Activation messages are of a similar nature as timeslice overrun messages. They must not influence the thread's normal IPC. The consequence is to add another sender role. Sending messages without blocking rises the problem that messages can get lost if the receiver is not ready to receive. Regarding activation messages the solution for that is trivial: There are only two types of messages, block and unblock events. A blocked thread can only unblock and vice-versa, which means the two events always occur in turn. Furthermore, if an event occurs whilest another one is already pending, there is no need to tell the user these events at all. Two adjacent messages simply wipe out each other. Thus the kernel can even guarantee that the blocking status of the threads is always correctly fed to the dispatcher.

5.1.3 When to Send Activation IPC

A nontrivial question that arose on the implementation was, when does a thread actually block? Three possibilities are imaginable: on switch to the IPC partner, on timer interrupt, or when the invoker is dequeued from the ready list.

Activation Message on Switch to Receiver

One possibility would be to send a block notification whenever a thread switches to its IPC partner and waits for the result. This would result in sending a block message on *every* IPC system call except timeout-zero ones. This would introduce lots of overhead, as each blocking IPC system call would result in the actual message plus two additional IPCs to the respective dispatcher threads. This misses the design goal of preserving overall system performance, as stated in Section 4.1.2 on page 19. For this reason I discarded this variant.

Activation Message on Interrupt

An alternative when to send the block notification is the point in time where an IPC is preempted. Such a preemption occurs either when the kernel reschedules on a timer interrupt, or when a higher-priority thread becomes ready to run.

This method collapses because of IPC chains and time donation. When an interrupt occurs, the kernel knows only the thread that was active at this moment. This thread, however, is not necessarily the only one that starts to block at this interrupt. There can be an IPC chain containing potentially all threads availabe in the system. The kernel would have to traverse the list and send a block notification for each thread that has associated a dispatcher. Traversing potential long chains on a timer interrupt would introduce unpredictable overhead. This has already been shown in [Ste04] regarding priority inheritance.

Activation Message on Ready-List Enqueue

Fiasco implements a lazy ready list enqueueing and dequeueing mechanism as described in [Lie93]. A thread leaves the ready list only if the scheduler tries to activate it whilst it is not ready to run. This means that blocking IPCs that completely run on the sender's timeslice do not trigger a ready-list dequeue. This lends itself to couple the block notification to the

ready-list dequeueing. The unblock notification is sent when a thread becomes ready-enqueued, accordingly.

Summary

The only method that actually works and meets the design goals is sending activation messages on ready-list enqueue and dequeue events.

5.1.4 Conclusion

For the reasons stated above I decided to implement the activation send mechanism as an extra sender role. Activation messages are sent whenever a thread enters or leaves the ready queue.

5.2 Thread Package Layout Proposal

A user-level thread package using the activation mechanism consists of a pool of virtual processors and a dispatcher thread. To enable the dispatcher to control the virtual processors, the dispatcher has to run at a higher priority.

User-level threads can be in one of the following states:

Running. The user-level thread owns a virtual processor. It can either be actually executing or blocking in a kernel IPC.

Ready to Run. The user-level thread waits to be executed on top of a virtual processor.

Blocking. The user-level thread invoked a user-level IPC (see below), and it is waiting until the intended partner responds.

To administer ready-to-run threads, user-level ready queues (see Section 5.2.3) are needed.

5.2.1 Cooperative Scheduling

The speed gain of user-level threading can only be functional as long as most thread switches are cooperative ones. Therefore, the user-level threads have to be designed with preemption points where they yield the virtual processor. A thread invokes a cooperative switch by determining the next thread from the ready queue, saving its own context, loading the target's context and resuming execution.

5.2.2 Preemptive Scheduling and Timeouts

In normal operation, the dispatcher waits for activation IPCs from the virtual processors. To generate timeouts, it can set a timeout for this waiting IPC. When the timeout triggers, the dispatcher becomes running and can perform the thread switch as described in Section 4.3.4 on page 21.

5.2.3 Ready Queues

When a user-level thread yields the virtual processor, the thread package has to locate a ready-to-run thread it can switch to. Therefore, user-level ready queues are needed. Regarding the correlation between virtual processors and ready queues, two approaches are imaginable:

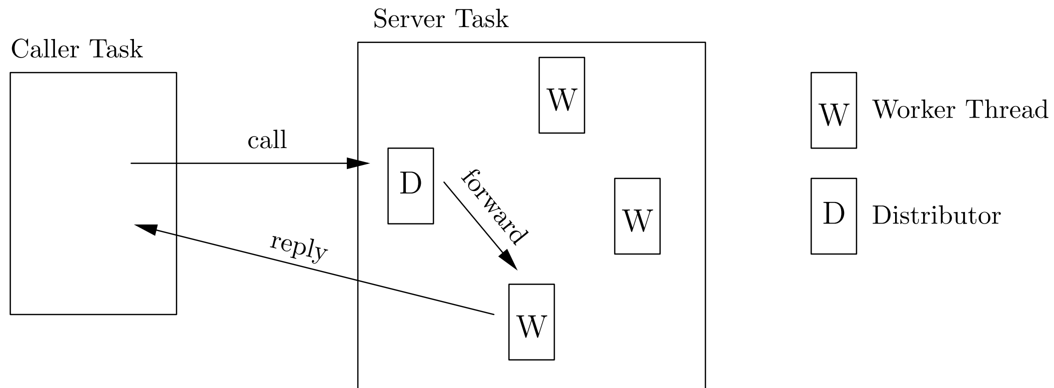

Figure 5.1: Forwarding IPC Requests.

Global Ready Queue. The user-level task uses one ready queue for all virtual processors. The virtual processors have to synchronize to each other when updating the queue. When a virtual processor is blocked, the dispatcher detaches that processor from the queue so that the other virtual processors can continue to execute the user-level threads.

One Ready Queue per Virtual Processor. Each virtual processor has its own ready queue. User-level threads are thus bound to a specific virtual processor as long as the queues are not rearranged. The dispatcher rearranges the queues if a virtual processor blocks or unblocks.

As the costs for synchronizing the accesses to a global ready queue are to avoid, we prefer the second approach.

5.2.4 Invoking Kernel IPC Calls

When a user-level thread performs an L4 kernel-level IPC call to an external server, the virtual processor blocks. If the blocking lasts on till the time slice exceeds, the dispatcher gets a block notification. It then rearranges the ready lists to allow the other user-level threads to make progress. This is shown in Figure 4.4 on page 29. If there is no running virtual processor left, the dispatcher has to perform one of the actions stated in Section 4.6.4 on page 30.

5.2.5 Receiving Kernel IPC Calls

To enable the user-level task to receive IPC calls from other tasks, at least one thread has to stay in a kernel IPC. We then need a mechanism to distribute incoming calls to the user-level worker threads. The usual design of such a distribution is shown in Figure 5.1.

For user-level threading this design is not optimal, because forwarding the request to another kernel thread cannot be achieved by user-level communication. Instead, we propose a different design we call *delayed forward*. This is shown in Figure 5.2.

The worker thread receives the request directly. If the request can be handled quickly, it also performs the reply. If it takes long, the time slice will exceed and the dispatcher will get an unblock notification for this kernel thread. The dispatcher then switches the worker thread away and puts an idle thread onto the receiving virtual processor. This is depicted in Figure 5.3. These actions of course take time, but they occur only if the caller's timeslice exceeds whilest

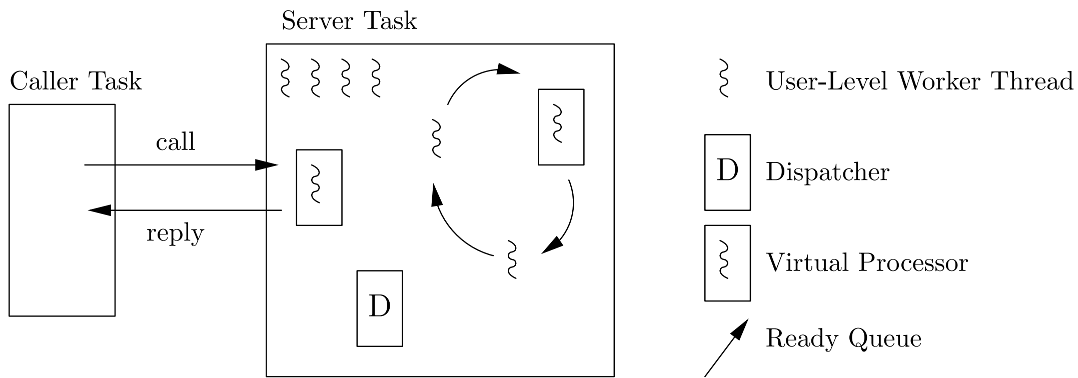

Figure 5.2: Delayed IPC Forward.

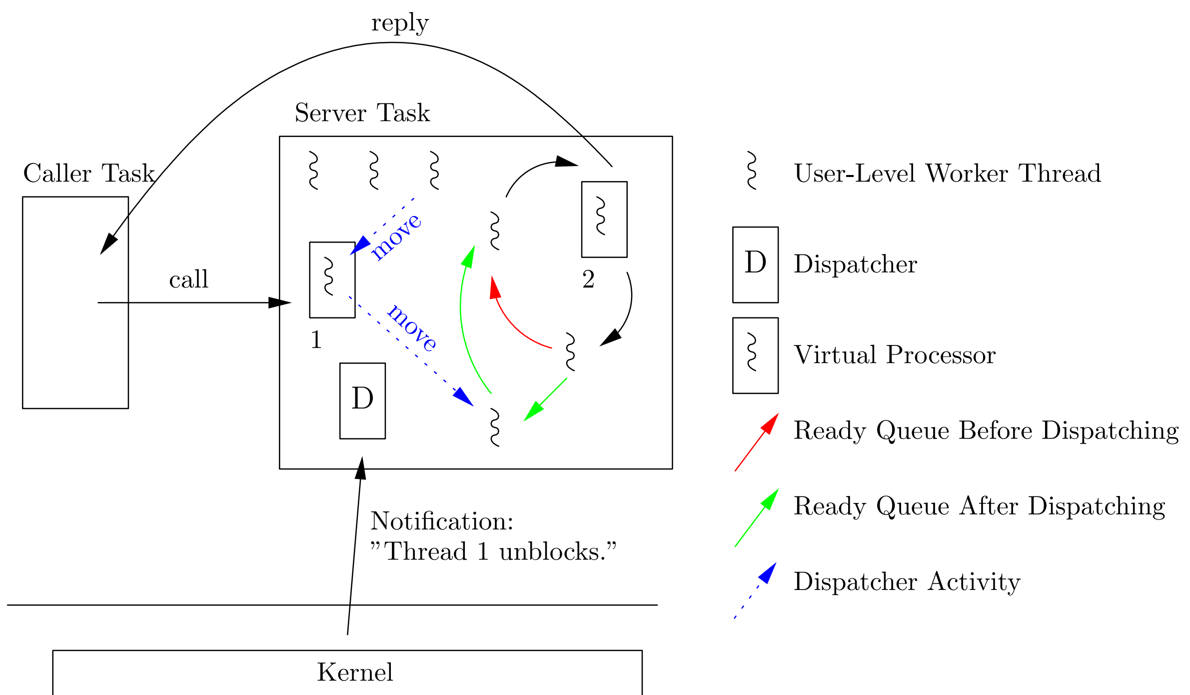

Figure 5.3: Dispatcher Activity on Delayed IPC Forward.

the server is processing the request. The probability that this occurs on fast-handled requests is considered to be low.

5.2.6 The Exchange-Registers System Call

The `ex_regs` system call can be used as is when targeted to kernel threads. A similar call for user-level threads can be implemented at user level. It reads the corresponding user-level execution context and fills it with the given new values.

5.2.7 Local Communication

A user-level thread can perform a local IPC call to another user-level thread by performing the following actions:

1. Lookup and lock (see below) the partner's context.
2. Check if the partner is ready to receive.
3. If not, mark myself ready-to-send and yield the virtual processor.
4. If yes, transfer the message and perform a cooperative switch to the receiver.

Therefore, user-level message contexts are needed. They are not to be mistaken for kernel message contexts. User-level message contexts are used only for local IPC, and their layout and size is under the complete control of the user-level thread library. The same applies for user-level sender queues.

Using this method, it should be possible to provide a local IPC facility with similar semantics as with kernel IPC, except that it works only intra-task. Depending on the purpose the user-level thread package should serve, a less powerful interface maybe sufficient. It will then need less user-level resources.

5.2.8 Locking of User-Level Resources

To implement local communication, thread switches, and the local `ex_regs` functionality as stated above, locking of both local ready lists and execution contexts is probably required. These local locks should not use kernel-level IPC, as this would negate the performance gain of user-level threading. Instead, we propose to use the *delayed preemption* mechanism described in L4/x.2 to build short atomic sequences at user level.

5.2.9 Summary

Using the proposed design it should be possible to create a user-level thread package that provides an interface similar to L4. It needs the activation mechanism and probably the delayed preemption feature of L4/x2. It should be possible to adapt multithreaded L4 servers to use user-level threads with reasonable effort. A precondition is that the servers do not need different priorities or realtime scheduling. The resource saving of user-level threads will always be operative, whereas for the speed gain to be effective, the servers need to be redesigned to use cooperative thread switching.

Chapter 6

Evaluation

6.1 Performance Overhead

As the actual user-level threading system could not yet be implemented, we cannot state anything about the performance gain of the user-level threading yet. But we can enumerate the overhead that the implemented mechanism introduces.

The implemented mechanism does not introduce performance overhead to legacy kernel IPC, because the corresponding program code remains unchanged. IPC calls originated by user-level threads also do not suffer from performance hits as long as the call uses time donation and it completes on the invoker's timeslice. If the call takes longer, the kernel sends two IPCs to the dispatcher threads, a block and an unblock notification. This turns one IPC into three, and the dispatcher itself probably needs time to react. This results in an overhead of at least 200 percent in this case.

We hope that this case does not occur frequently and the overhead can be compensated by the performance gain caused by user-level threading. This will only be known after the actual user-level thread library on top of this mechanism got implemented.

6.2 Resource Overhead

The thread control block got new member variables. The new TCB member variables are listed in Table 6.1. In the current FIASCO implementation on the IA-32 architecture, this results in a 24 bytes larger TCB at an overall TCB size of 2048 bytes. If it evolves that the user-level threading system proposed in this work is actually useful, this overhead is considered to be acceptable.

Member	Type	Size (bytes)
Dispatcher	Pointer	4
Pending Activation	Machine Word	4
Activation Sender Role	Sender ID and 3 Pointers	16

Table 6.1: New TCB Members.

Chapter 7

Conclusion

The original goal, which was using pageable resources for L4 threads, could not be reached. The reason is that three boundary conditions are in polar opposite to each other. These conditions are:

1. Pageable resources,
2. zero timeouts, and
3. synchronous communication.

Each combination of two out of the three works well, but it excludes the third, respectively.

Pageable Resources and Zero Timeouts. In the case that a thread sends with zero timeout to another one that is paged out, the message has to be buffered somewhere. The communication can therefore no longer be synchronous.

Pageable Resources and Synchronous Communication. If a sender sends to a paged-out thread, time is needed to page in the resources again and receive the message. As the communication is synchronous, the sender has to wait until the receiver's resources are available again. Zero timeouts or, even more generally, short timeouts intended to prevent denial-of-service attacks against senders, are not possible any longer.

Zero Timeouts and Synchronous Communication. Synchronous send operations with zero timeouts require the receiver's resources to be available immediately. The resources needed to receive a message therefore must not be paged out.

The achievements of the several goals, in a nutshell, are the following:

Pageable Resources. Allows creation of largish amounts of threads, suitable for high-concurrency programming models. Also enables threads to be used as potential long-living objects that do not occupy physical memory if they are not accessed.

Zero Timeouts. Allows reliable communication with untrusted partners. Needed to build a trust hierarchy.

Synchronous Communication. Frees the kernel from the job of allocating and administering message buffers. Needed to build small and efficient microkernels with a small cache footprint.

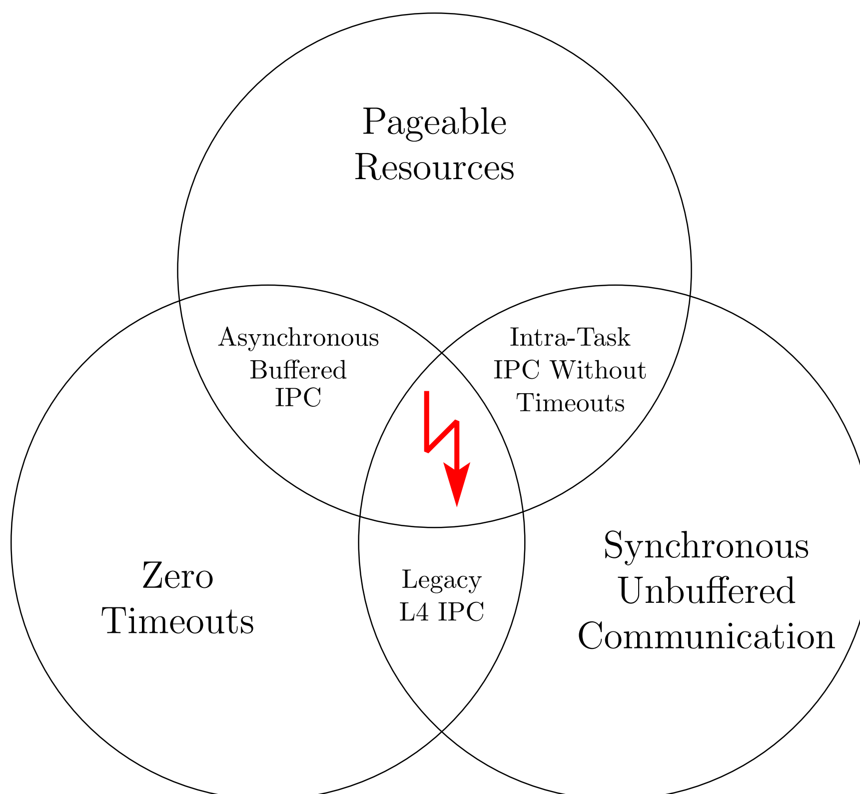

Figure 7.1: Contradicting Design Goals.

The contradicting design goals and the possible combinations have been depicted in Figure 7.1. Reconciling these three conditions by weakening some of them did not work, because when starting to weaken a condition, at first the achievements are buggered, and only if the condition is completely released, also the constraints, which were introduced by it, disappear. For example, allowing asynchronous communication "in some cases" requires to build message buffer management in the kernel, whereas user-level applications can profit from asynchronous communication only if it can be used everywhere. Allowing only "short timeouts" instead of zero timeouts circumvents the creation of a trust hierarchy, whereas the guarantee that a paged-out receiver is able to receive a message can only be given if there is no timeout at all.

The proposed solution is to use different communication mechanisms for inter-task and intra-task communication. Intra-task communication needs no timeouts, because the partners share an address space and thus have to trust each other anyway. It can therefore be targeted to pageable thread objects (i.e., user-level threads). Inter-task communication needs timeouts but it cannot be targeted to pageable threads. As a consequence, if a user-level thread should be reachable for inter-task IPC, it can either be pinned to a kernel thread (i.e., prevent it from being pageable), or another kernel thread residing in the receiver task has to be used as wrapper (i.e., building asynchronous communication at user level).

For the method of pinning user-level threads to a kernel thread, the activation IPC mechanism implemented in the kernel is needed to keep track of blocking threads.

7.1 Future Work

In this work, only the kernel mechanism needed to keep track of blocking threads got implemented, and a minimal user-level program that shows that the mechanism actually works. The next step will be to implement a user-level thread library for L4 using this mechanism and the design proposed in this thesis.

For that to be accomplished, atomic sequences at user level are required. Therefore the *delayed preemptions* feature described in [DLSU04] should be implemented in Fiasco.

When a working user-level thread package is available, the context-split model described in Section 4.8 on page 35 could be implemented.

When investigating the implementation of Long-IPC in Fiasco, it turned out that copying data between address spaces requires nasty hacks in the kernel that slow down every context switch and anyway enlarge the kernel's instruction cache footprint (see Section A.4 on page 49). The Long-IPC feature, however, could also be implemented entirely at user level, using map operations and a copy server. If this could be achieved with introducing passable overhead to Long IPC, the Long IPC code in the kernel could be abandoned. This is not really in the focus of this thesis, but it should be noted down here as an interesting secondary perception.

Chapter 8

Summary

It has been shown that complex kernel mechanisms to extend L4 threads to act like virtual processors are not adequate to be included in L4. Instead, a simple mechanism that allows user-level applications to keep track of blocking threads has been implemented in the kernel. The thesis also proposes a design of a user-level thread library using this mechanism.

The implementation of the activation mechanism has been merged to the main Fiasco tree and is available from remote CVS at: `http://os.inf.tu-dresden.de/drops/download.html`

Appendix A

Fiasco Implementation Details

In this appendix we describe the reasons why it is problematic to access user-level memory from kernel space. As example the FIASCO implementation on the IA-32 architecture is used, but most of the facts also match to other architectures and L4 implementations.

A.1 Memory Layout

The virtual address space is divided into two parts, a kernel and a user area. The address space layout is shown in Figure A.1. The user area contains the memory mappings of the current active task. The kernel area contains the mappings for the kernel memory. The kernel mappings are not changed on task switch, hence the kernel memory is always accessible.

A.2 Page Faults in Kernel Mode

As the kernel memory is always available and not revocable, page faults should normally not occur at all. However, there are some exceptions, for example if the kernel allocates a new TCB, switches to another task, and accesses that TCB. The mapping for this TCB is not yet available in the page directory of the now-active task, so a fault occurs, and the page fault handler has to copy the mapping from the master page directory. Which type of page fault occured is determined on the basis of the given page-fault address and instruction pointer. Thus for each

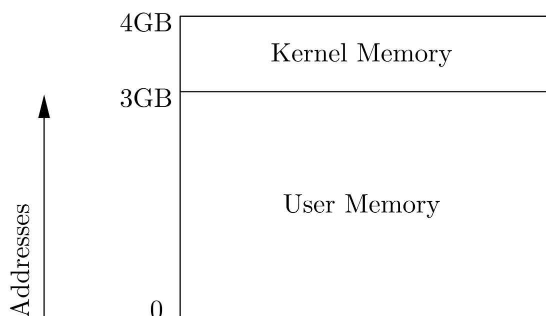

Figure A.1: FIASCO Memory Layout.

type of kernel page fault, a special case has to be implemented in the page fault handler. Each such special case introduces overhead to the page fault handling in general.

A.3 Access to User Memory

When accessing user memory from kernel mode, two cases have to be distinguished: if the accessed task is currently active or not.

A.3.1 Active Task

Access to memory of the active task is quite straight forward. The task's memory is directly visible in the user area of the address space. It only adds another special case to the page fault handler, introducing overhead as shown above.

A.3.2 Inactive Task

The kernel cannot access memory of an inactive task directly, because it is not visible. Instead, a temporary page table entry has to be built. This rises a set of problems:

Revoking and Preemptability. The kernel may be preempted after the temporary mapping has been set up. If the target task's pager unmaps the memory, the temporary entry becomes invalid. When the operation continues, it writes to physical memory that does no longer belong to the target task. Possible solutions are making the kernel operations atomic or removing the temporary mappings on each task switch. The latter introduces overhead to every context switch, whereas the former potentially rises issues regarding interrupt latency times.

Page Boundaries. The case that the accessed object resides on more than one memory page has to be handled manually. This can be done by setting up two adjacent mappings.

A.4 Long IPC

The Long-IPC feature copies data from one address space to another. The kernel therefore has to access user memory whilst the IPC is in progress. That raises all the afore said problems. They are solved as follows:

Page Faults. If a page fault in the user address space occurs, the kernel sends a nested IPC to the pager of the faulting thread. For this IPC the user has to specify an extra timeout, the *transfer timeout*.

Invalid Temporary Mappings. The temporary mappings are removed at every context switch.

Page Boundaries. The kernel always sets up temporary mappings for two adjacent pages of the target task.

A.5 Summary

As shown above, accessing user memory from kernel memory adds both overhead and code complexity to the kernel. Regarding user-level thread support it is therefore desirable to go without adding more cases of accessing user-level memory from within the kernel.

Bibliography

[AAD+02a] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42 overview. Technical report, IBM Research, 2002.

[AAD+02b] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Scheduling in k42. Technical report, IBM Research, 2002.

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions of Computer Systems*, 10(1):53–79, February 1992.

[BMD99] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999.

[Cla04] Dietrich Clauß. Implementation of the L4 version x.2 abi in the Fiasco microkernel. Technical report, TU Dresden, 2004.

[DLSU04] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. *L4 eXperimental Kernel Reference Manual, Version X.2*. University of Karlsruhe, 2004. Latest version available from: `http://l4hq.org/docs/manuals/`.

[DZK+02] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, September 2002.

[Gro03] System Achitecture Group. The L4Ka::pistachio microkernel. Technical report, University of Karlsruhe, 2003.

[HE03] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, September 2003.

[Hoh98] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`.

[Hoh02] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[Lie93] Jochen Liedtke. Improving ipc by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, December 1993.

[Lie95a] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[Lie95b] J. Liedtke. Towards real μ-kernels. Arbeitspapiere der GMD No. 958, GMD — German National Research Center for Information Technology, Sankt Augustin, December 1995.

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

[Lie99] J. Liedtke. *L4 Nucleus Version X Reference Manual (x86)*. University of Karlsruhe, September 1999.

[LW01] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems*, Schloß Elmau, Oberbayern, Germany, May 20–23, 2001.

[PV05] Michael Peter and Marcus Völp. *L4-Sec Kernel Reference Manual*. TU Dresden, 2005.

[Reu04] René Reussner. Implementierung von local-ipc auf L4/Fiasco. Technical report, TU Dresden, 2004.

[Ste04] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's thesis, TU Dresden, March 2004.

[vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*. University of California at Berkeley, 2003.

[Voe02] Marcus Voelp. Design and implementation of the recursive virtual address space model for small scale multiprocessor systems. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, September 2002.

[WCB01] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[Wen02] Horst Wenske. Design and implementation of fast local ipc for the l4 microkernel. Study thesis, System Architecture Group, University of Karlsruhe, Germany, July 2002.

[Wil02] Nathan J. Williams. An implementation of scheduler activations on the netbsd operating system. In *USENIX Technical Conference (Freenix Track)*, Monterey Conference Center, Monterey, CA, June 10–15, 2002.