L^{4}Linux Porting Optimizations

Adam Lackorzynski

Technische Universität Dresden
Faculty of Computer Science
Operating Systems Group

March 2004

Contents

List of Figures

Chapter 1

Introduction

For years L^{4}Linux has been available to the L4 community [27]. Starting from the initial port [21] it has gone through various Linux versions and has proven to be helpful for a number of projects or even made them possible. The first L4 microkernel [29] was designed with performance as the main goal. L^{4}Linux, a port of Linux to run on top of L4, showed that L4 is sufficiently versatile to run Linux as a user program on it, with only a negligible performance loss [17].

The initial L4 interface was the V2 interface. Over the time, it has gone through two revisions, one called X.0, with small changes to the V2 interface. With the X.2 specification the interface was completely revised, especially addressing the portability of the L4 interface. L^{4}Linux has been adapted to all L4 interfaces, L^{4}Linux-2.2, a port of Linux-2.2, also runs on X.0 kernels, L^{4}Linux-2.4 supports all three interfaces. The overall structure of L^{4}Linux has not changed with these adaptations. Still, the L4 interface lacks properties like communication control and a more general rights management scheme. This functionality is especially needed to build secure systems. Broad modifications are being discussed to come up with a new interface specification covering the addressed points. Nevertheless, L4 is fully functional for non-secure and robust systems.

L^{4}Linux will be affected by these modifications as well and benefit from the newly available possibilities in its design. L^{4}Linux, as a system that has to control other address spaces, can make use of the widened rights control and avoid the usage of helper constructs. Additionally, advanced means for communication control will improve L^{4}Linux's worthiness in secure systems. This work has the goal to port the current Linux revision, 2.6, to L4 and explore new, simplified ways in the design of L^{4}Linux. One methodology is to explicitly include modifying the microkernel by experimentally implementing some of the features planned for the next microkernel revisions.

1.1 Terminology

Discussing L^{4}Linux always includes at least two kernels, the microkernel and the Linux kernel. Both have tasks, threads, processes, inter-process communication and so on, generally they share terms in identical areas. To relief you of the confusion placed in this mixture, these terms shall be defined as follows.
The *kernel* refers to the microkernel and the *Linux server* to the Linux system kernel. A *thread* denotes an L4 thread, a *Linux thread* is a thread in a Linux context. A *task* is an

address space in the L4 context, which includes at least one thread. *Process* or *user process* refers to Linux user processes.

Differing meanings are stated explicitly in the text by either prefixed restrictions or further explanations.

1.2 Document Structure

This document continues with a chapter about the fundamentals needed for this work, as well as related work done on user level Linux implementations. Chapter 3 discusses changes to the microkernel as well as to L^{4}Linux itself to simplify its design. Chapter 4 explains the implementation of a few aspects from the previous chapter. Results will be presented in Chapter 5. Chapter 6 on page 47 gives an outlook to future areas of work that will, can, or should be done. The document closes with a summary.

1.3 Acknowledgments

I like to thank Prof. Hermann Härtig and the Operating Systems Group of the University of Dresden for giving me the opportunity to work with such a fantastic group. I especially like to thank Alexander Warg and Frank Mehnert for their invaluable help on the implementation of the Fiasco modifications. Udo Steinberg deserves many thanks for his incredible work on Fiasco-UX, which helped me a lot. I also like to thank everyone who has proofread this document for their useful feedback and recommendations. Finally, many thanks go to my family for giving me the possibility to study at the university and their constant support as well as to my friends for their incessant encouragement.

Chapter 2

Fundamentals and Related Work

This chapter will explain fundamentals this work is based on and related work with similar goals.

2.1 System Architecture

The work on L^4Linux is usually done in the context of DROPS, the Dresden Real-time OPerating System [16]. The DROPS project aims to build a system where real-time programs can run side by side with non-real-time applications without violating properties promised to the real-time programs. Examples of real-time components may be disk drivers or network servers. Other components are built on top of these services, examples include filesystems or video players. L^4Linux is located in the non-real-time part of DROPS because it provides the environment to run unmodified Linux applications in the system where no real-time guarantees can be given. The non-real-time part of the system receives the resources that are not utilized by the real-time part. Figure 2.1 on the next page shows the structure of a DROPS system.

It is important to note that the L^4Linux system uses underlying system services in the same way as other L4 applications. Additionally, L^4Linux may use the higher-level L4 services. For example if a network server will drive the network card of the system, L^4Linux must use this L4 server for network communication because only one program can drive one particular hardware device at a time. Real-time-capable L4 servers generally also offer their services to non-real-time applications and schedule their clients according to the available resources.

2.2 Environments

The L4 API is designed to be minimal and provides only simple functions to the user-level applications [31]. It is tedious to work with the pure L4 API for bigger applications. To increase application development productivity, supporting libraries, system services and tools are needed. This collection includes a C library, memory management, thread management, program loading, event handling, and an interface definition language along with a stub generator and others.

L4Env [26] provides the needed libraries and services by abstracting L4 concepts to higher levels. For example it provides a thread library to handle threads within an application

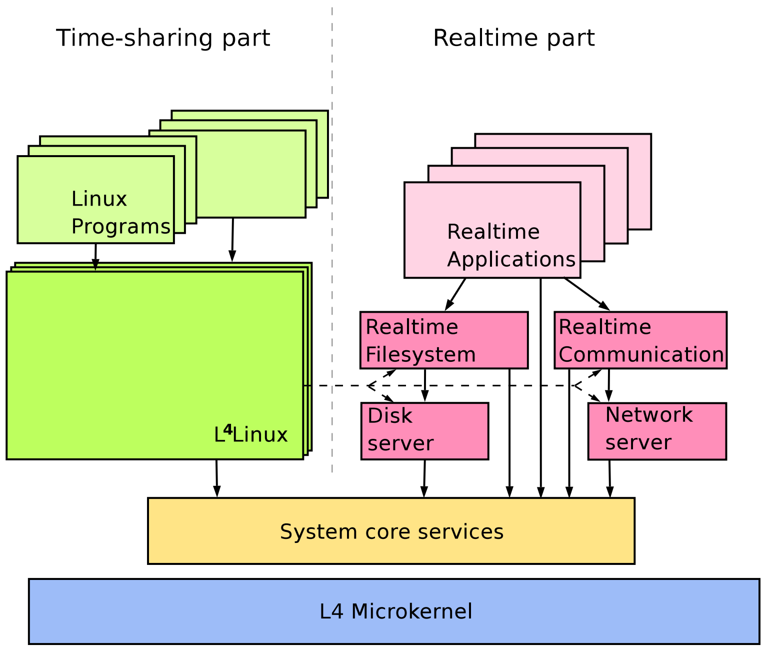

Figure 2.1: DROPS architecture

in a microkernel-independent way. Applications developed with this library are supposed to run on any L4 microkernel as long as the thread library is available for this particular kernel. Another example is the data-space manager that manages memory objects and provides them to clients as data-spaces, an abstraction of memory objects of arbitrary size, which can be mapped as a whole or in parts to L4 programs. The data-space concept is supposed to abstract the memory management in a way that it is portable between different memory architectures but retains the needed fine-grained control and flexibility.

2.3 L4 Microkernels

The whole system runs on an L4 microkernel. L4 microkernels have gone through several revisions starting with the initial version by Jochen Liedtke written entirely in assembly language [29] and implementing the *Version 2* interface [30]. The Operating Systems Group in Dresden developed Fiasco [11], a microkernel that is written in C++ for better maintainability and also implements the Version 2 interface. Fiasco is designed to be extremely preemptive by using special locking mechanisms making it well suited for real-time systems [22]. Fiasco is available under the terms of the GNU Public License [15] and is used as a basis for the DROPS project.

Over the time, platform portability came into the focus of the L4 community, an attribute that neither the original kernel nor the Version 2 API interface addresses. Jochen Liedtke designed the original V2 API in a way that it is extremely tailored to the x86 architecture to achieve the best possible performance. This customized design made it difficult for other architectures to exploit the full capabilities of the hardware. To address a row of insufficiencies the original kernel interface was slightly modified in a small step producing the X.0 interface [52]. Summarized, the changes reduced the size of a thread ID to have more space for message registers in IPCs. To achieve better portability the interface specification was completely revised and called *Version X.2* [53]. The *X* in the version name stands for *experimental.*

Currently, implementations for all L4 APIs exist as well as for different architectures. The main ones are Fiasco and L4Ka::Pistachio [36]. Fiasco currently implements the V2 and X.0 API and runs on the x86 as well as the ARM architecture. Support for the X.2 API is currently being added to Fiasco. L4Ka::Pistachio is written in C++, implements the X.2 API and was ported to the x86, IA64, ARM, MIPS, PowerPC and other architectures.

Besides the already mentioned implementations there are other ones as well. L4Ka::Hazelnut is the predecessor of L4Ka::Pistachio, implementing the X.0 specification and is also written in C++. Its development has been discontinued in favor of L4Ka::Pistachio. There are older experimental kernels for different single architectures, like for Alpha or MIPS.

All L4 microkernels have been developed and are mainly used in university and research environments but there are also commercially developed versions. One is P4, a mainly V2 compatible microkernel developed by the SYSGO AG [46]. A list of different implementation of the L4 microkernel interface can be found at [25].

2.4 Past L^{4}Linux Development

The first implementation of L^{4}Linux was done by the Operating Systems Group in Dresden in 1996 and based on Linux 2.0. Michael Hohmuth wrote his master's thesis about this port [21]. Results of the work have also been presented at the 16th SOSP conference 1997 [17].

Later L^{4}Linux has been brought up to date with Linux 2.2. Both ports support the original L4 V2 API, the 2.2 port additionally supports the X.0 API.

L^{4}Linux-2.4 marks a new milestone as it introduces support for the X.2 API as well as support for symmetric multiprocessors (SMP). Additionally it can run as an application in L4Env [28], which integrates it nicely into DROPS.

2.5 Linux in User Level

In the past there have been other approaches to port Linux to run in user level. There are different ways to accomplish this, one being full machine emulation and full virtualization to run unmodified operating systems on it. Other approaches modify the guest operating system in different grades and use varying host systems.

A few of the approaches that modify the guest operating system will be explained and a list of ways to run unmodified operating systems will be given.

2.5.1 MkLinux

MkLinux is a port of Linux to the Mach 3 microkernel and was presented in early 1996 [6]. Mach was developed at Carnegie Mellon University [1]. The design of MkLinux is similar to L^{4}Linux. Linux runs as a server and user processes in separate address spaces. Exceptions including system calls are redirected to the Linux server using Mach RPC. MkLinux does not map code into each user process address space. To manage the memory for each user process in an efficient way the server has to map the memory of every user process into its own address space, giving the server easy access to it. Signals are delivered through a "fake interrupt" mechanism, where the server gains control over the user processes and forces them to handle their signals. Overall, MkLinux is slower compared to other solutions like L^{4}Linux, mainly due to the design and performance insufficiencies of the underlying Mach kernel [17].

2.5.2 Xen

Xen [3] is an architecture that partitions resources of a host computer to run guest operating systems in the partitions. It is targeted for the x86 architecture and the port of Linux 2.4 is called XenoLinux.

The approach of Xen is called paravirtualization [54,55] because it does not provide a fully virtualized environment for its guest operating systems. Instead, guest operating systems need to be modified slightly to run on Xen. Full virtualization on the x86 architecture is hard to achieve as certain supervisor instructions do not trap in unprivileged modes. Full virtualization can be achieved but induces a higher complexity and lower performance. Another problem with full virtualization is device emulation where certain hardware devices like network cards or disks need to be emulated. The approach of Xen is to provide a nearly complete virtual machine and to adapt the guest OS to this paravirtualized system where full virtualization is too expensive and can be easily achieved by modifying the guest operating system. Communication with other systems is done with custom drivers, which use the best possible way like shared memory for data exchange. So by modifying the guest system it is possible to avoid the cost of full virtualization both in performance loss and code complexity in the monitor. Of course, the modifications in the guest OS have to be accounted for as well but they are smaller than the added complexity in the virtual machine monitor (VMM).

L^{4}Linux can also be seen as a paravirtualized system as it modifies Linux to run on an environment that differs from the x86 in certain aspects. The major difference of L4 and Xen is that Xen is targeted at ISPs or similar organizations, which want to use their hardware efficiently while giving customers their *own* machines. Xen is a VMM with no further abstractions; it can only be a platform for full operating systems like Linux but it cannot offer a system with its own applications supporting certain properties like microkernels.

2.5.3 User-Mode-Linux

User-Mode-Linux [48], or UML, is a port of Linux to the Linux user level, it runs Linux inside Linux. As the main techniques UML uses the `ptrace` system call and signals to intercept exceptions and system calls and to execute code in user processes.

Currently, UML supports two modes of operation, one can run on an unmodified host kernel (called '`tt`' mode), the other on a modified host kernel (called '`skas`' mode). Modifying the host kernel adds certain features to it to improve the performance of UML.

In the '`tt`' mode each UML process has its own process in the host system. UML processes are traced by the "tracing thread" that cancels their system calls and causes them to enter the UML kernel. The UML kernel is mapped in the upper part of each UML process.

This design has several drawbacks. By default, the UML kernel is mapped writable into the address space of the UML processes imposing a tremendous security risk like breaking out of the UML system. With the '*jail*' mode the UML kernel is mapped read-only, which fixes most of the security concerns but induces an even greater performance loss.

The '`skas`' mode puts the UML kernel in another address space solving the security problems described previously. Additionally, it currently comes with a patch for the Linux host kernel enhancing the ptrace system call with useful functions for UML and an interface for address space manipulation in other address spaces via `/proc/mm`.

Future versions of the patch will implement the functionality in other ways. Two new system calls will be introduced, one to create new address spaces and another to execute system calls in other contexts. It is likely that this patch will be merged with the main Linux kernel.

The '`skas`' approach of UML has similarities to L^{4}Linux. On both systems, the Linux kernel is located in its own address space. As the underlying systems are different, the implementations of both systems are done differently but conceptually they are similar.

2.5.4 FAUmachine

The FAUmachine project [9] works on an open source virtual machine. The project evolved from the UMLinux project [5], a port of Linux to the Linux user mode, similar to UML discussed in the previous section. To reflect the focus from the pure Linux port towards a virtual machine, the project name was changed. FAUmachine works similarly to recent UML versions. The system is primarily used for fault injection to test the behavior of network applications [42].

2.5.5 P4/Linux

The SYSGO AG company is active in the field of small real-time systems as well as in the corresponding Linux area [46]. They have developed a V2 compatible L4 microkernel called P4 that runs on the x86, ARM [13] and PowerPC [39] architectures. On top of this kernel, a port of Linux 2.4 based on User Mode Linux was done. It is called P4/Linux [56]. Its design is similar to the traditional L^{4}Linux design.

2.5.6 Other Approaches

Running Linux as a user-space application is also possible by introducing a machine emulation layer. The Linux system itself does not need to be modified, although modifying Linux is possible to improve the performance in the virtualized system.

VMware [51], Virtual PC [50], plex86 [37], z/VM [57] These systems provide virtual environments that can run various unmodified operating systems for the same hardware architecture as the host system, including Linux.

Bochs [4], QEMU [38], SID [41], Simics [43] These programs provide machine emulators that can emulate various computer systems including some hardware components. The emulators itself can run on different architectures.

Chapter 3

Design

The goal of this work is to simplify the design of L^{4}Linux by allowing modifications to the microkernel interface. This section will start with a short description of the design of past L^{4}Linux versions, then explain what can be changed to simplify the design and how the microkernel and its interface must be modified to achieve this.

3.1 Traditional L^{4}Linux design

L^{4}Linux-2.0 up to L^{4}Linux-2.4 almost had the same design. The initial design of L^{4}Linux is fully described in [21]. The system consists of a Linux task with several threads and a helper task as well as separate tasks for each user process.

The server task uses several threads. The main thread executes the actual Linux code, called the Linux server. To handle interrupts one or more separate threads are needed, depending on the L4 version used. Another thread, called root pager, handles page faults of the Linux server. The "Ping-Pong" task is used to remap memory from the Linux server to the address space of the Linux server again.

The tasks that handle user processes consist of two threads. The user thread executes the user program and the signal thread waits for commands from the Linux server to manipulate the user thread.

An overview of the traditional design is depicted in Figure 3.1 on the following page.

3.1.1 The Linux Server

The Linux server is the thread that executes the Linux kernel code. When the system is running, it is in the idle loop and waits to handle system calls, exceptions, and page faults of user processes.

3.1.2 Interrupts

L4 translates interrupts into synchronous IPC notifications sent to the thread attached to the interrupt. With V2 and X.0 kernels a thread can attach to only one interrupt, making interrupt threads in L^{4}Linux necessary. The X.2 interface supports multiple interrupts per thread.

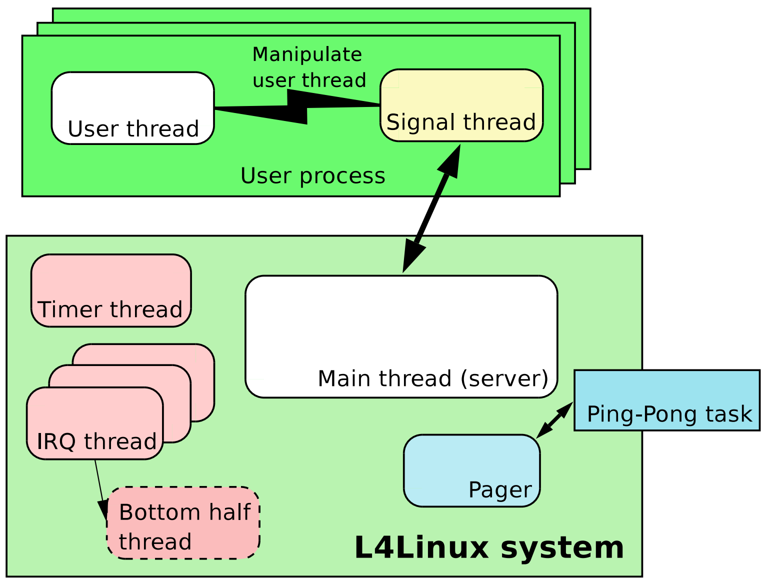

Figure 3.1: Overview of the traditional L^{4}Linux design

On native Linux, interrupt work is always executed in the context of the interrupted user program, which is not possible in L^{4}Linux as interrupt events are received in separate threads. Nevertheless, interrupt handlers that run on interrupt events execute Linux kernel code that expects a properly set up task structure on the stack. The stack of each interrupt thread is therefore initialized with the task structure of the idle process of Linux (also called `swapper` process).

Hardware interrupts are prioritized, L^{4}Linux emulates this behavior by giving interrupt threads different priorities. The priorities of all these threads are all different and higher than the priority of the Linux server. This way the behavior of hardware interrupts is emulated: an interrupt event interrupts the normal Linux server to handle top halves. L^{4}Linux-2.2 and L^{4}Linux-2.4 handle bottom halves slightly different. L^{4}Linux-2.2 uses an extra L4 thread that is prioritized between the interrupt threads and the Linux server. L^{4}Linux-2.4 uses the ksoftirqd Linux kernel thread for this, which is scheduled by the Linux scheduler and runs whenever deferred interrupt work needs to be done.

3.1.3 Signalling

In native Linux, signals to user processes are handled after a system call or any other exception just before the user process returns to user mode. Signals can be easily delivered when the concerning task is chosen by the scheduler to continue execution.

L^{4}Linux has a different behavior in this regard. Logically, user processes can also run when the Linux server runs, that is, they are not blocked on the Linux server. Consequently, a user process could prevent to handle signals by not entering the Linux server, for example through system calls. A simple endless loop will suffice for this. To avoid this situation, the Linux server must be able to force a user process to enter the server. Currently, for security reasons the L4 interface does not allow to modify threads in other address spaces. To circumvent this restriction, the aforementioned signal thread comes into play. Using this thread, the Linux server can interrupt the user process by sending a message to the signal thread. The signal thread then forces the user process to execute special code to enter the Linux server.

3.1.4 System-Call Emulation

On the x86 architecture Linux programs use the interrupt gate 0x80 to enter the kernel and execute a system call. On L4 systems the corresponding instruction will cause an exception and the causing program will be terminated unless it installs an exception handler. V2 and X.0 kernels allow thread local exception handling only, making it necessary that the Linux server maps emulation code into the address space of each user process. This code is called whenever the execution of an '`int $0x80`' instruction is detected. The emulation code needs to save the processor state of the process into a memory region shared with the Linux server and enter it by sending a blocking IPC of the type "system call." Upon returning to the user process the emulation code needs to properly set up the processor state with the possibly updated values again.

The X.2 specification of L4 offers the possibility to report exceptions to a designated thread, called exception handler, accompanied by the processor state of the faulting thread in an exception message. This mechanism allows to drop the system call related code from the emulation library and avoids one kernel entry.

Another possibility to execute system calls on L^{4}Linux is to modify the system-call code that the user programs use. For most programs this code is located in the C library to which the programs are dynamically linked. Instead of using the '`int $0x80`' instruction they directly use the IPC call in the emulation code avoiding the exception.

3.1.5 Scheduling

In an L^{4}Linux system two schedulers are active. One is in the L4 kernel scheduling all L4 threads including L^{4}Linux processes. The L4 scheduler uses static priorities and schedules threads with the same priority round-robin. The other scheduler is in the Linux server, deciding which Linux process to run next. L^{4}Linux is designed in a way that from the view of the Linux server multiple user processes can be running in the L4 system. Furthermore, the Linux scheduler can only consider those processes that are blocked in the server. Consequently, the Linux server needs to distinguish between user processes being served by the Linux server and the ones running.

3.1.6 Memory

Upon a page fault of a user process, L^{4}Linux has to map memory into the address space of the requesting process using the L4 map and unmap operations. The Linux server manages all the memory for the L^{4}Linux system and maps pages to user processes. All memory that user processes receive comes from the Linux server. L^{4}Linux also manages shadow page tables as it does not have access to the real hardware page tables but nevertheless needs to manage page attributes to be able to properly map and unmap pages of user processes.

L^{4}Linux-2.4 uses an optimization during task deletion. If an address space is destroyed by the kernel, all mappings are deallocated as well. This allows to avoid the explicit address space deletion done by Linux before destroying a process, which otherwise imposes an avoidable overhead. Furthermore it has the positive side effect of not unmapping shared pages in other address spaces.

3.1.7 Time Accounting

The Linux server needs to account the time of every user process in the system. Native Linux accomplishes this by updating the counters during each timer interrupt for the context in which the handler runs. In L^{4}Linux, interrupts including the timer interrupt run in separate threads and not in the (kernel) context of the currently executing user process. This can be circumvented by saving the last scheduled user process and adapting the corresponding parts of the generic timer handling code. Another problem is the inaccuracy of this scheme due to the design of L^{4}Linux. A process executing an endless loop without entering the Linux server will not be accounted anymore if another process is scheduled in-between which will likely happen. The fact that multiple user processes can be dispatched simultaneously is the main problem here.

3.1.8 Linux Threads

Linux can run several processes in the same address space, usually referred to as Linux threads. L^{4}Linux does not make use of this and puts every Linux process in a new address space even if they could share one. Implementing additional threads in an address space of a user process requires several modifications in the area of the code that is mapped in each user process as well as in the Linux server.

3.1.9 Summary

The previous sections gave an overview on the traditional L^{4}Linux design. With new mechanisms several parts of L^{4}Linux can be made easier or even abolished. One point is the signal thread and the code that is mapped into each user process. The code exists because the corresponding L4 primitive to modify a thread cannot be used across address space boundaries. This restriction can easily be relaxed. Another area for improvements is the scheduling. The current scheduling shows bad results when CPU bound processes are run, making the system sluggish.

3.2 Microkernel Modifications

When changing the microkernel and its interface to user programs we want to obey the rule that the behavior for existing programs is not changed, that is modifications must not change the semantics of existing functions. New mechanisms must be implemented by adding functionality. This approach assures that all existing programs keep working while programs aware of the new functionality can use it by either just accessing it or by switching the kernel to a new behavior for the context of the calling task.

The platform this work is done on is the DROPS system, which currently means Fiasco with the Version 2 interface and the x86 architecture.

3.2.1 Ex-Regs Across Tasks

L^{4}Linux needs to handle signals for user processes. As described in Section 3.1.3 on page 18, a signal thread in each user process' address space is used to force the process to enter the Linux server as this cannot be initiated from outside of this user address space. Consequently the first step is to allow the `l4_thread_ex_regs` system call to work across address space boundaries. Of course, it cannot be allowed that an arbitrary thread is able to modify any other thread or create new threads in other address spaces at its own will. Neither the current L4 interface nor any implementation offer a mechanism which would allow to control the use of this system call. An example mechanism would be thread capabilities, however, introducing capabilities to L4 induces major changes to both the API and the implementation. Fortunately there is a good compromise possible. Every thread has a pager that resolves page faults. The pager can map pages with arbitrary content into the address space of its clients or unmap pages in the clients' address space making it a trusted service for the thread. So it is suggesting itself to restrict the right to call the enhanced Ex-Regs system call from outside the target address space to the pager of the thread. For general use of this system call the restriction to the pager is too strict but sufficient for L^4Linux.

Besides modifying an existing thread, the new system call can be also used to create additional threads in other address spaces. This function is useful to implement Linux threads as L4 threads. The first thread is created while creating the address space. Consecutive threads in the address space need to be created with an Ex-Regs mechanism. Without an enhanced version of the system call a helper thread would be needed in the user address space to create the new thread on behalf of the Linux server.

3.2.2 Exceptions

Another point is the exception handling, as well as system-call emulation, which is just a special exception. With Version 2 and X.0 of the L4 interface exceptions are delivered through the installation of a thread local interrupt descriptor table. As the installation of descriptor tables is a privileged operation, the exception handling is emulated by the microkernel. This makes it necessary that every user process executes certain code that installs the tables. Additionally, when an exception occurs, it is trapped by the kernel, reflected to the thread that caused it and the thread is continued at the previously defined exception-handler address.

Version X.2 of the L4 specification implements the concept of "Exception IPC" [53]. Whenever a thread causes an exception, a specially tagged IPC is sent to the previously assigned exception handler, accompanied by the register state of the faulted thread. Exceptions are synchronous events, the thread will be stopped until the handler has replied to the exception IPC. The handler may change the state used to reply the exception to let the thread continue with a modified state, in particular with a new instruction pointer to continue execution behind the instruction that caused the exception.

The state of a thread that caused an exception must somehow be transfered from the microkernel to the exception handler. L4 X.2 uses User-level Thread Control Blocks, short UTCBs, for this. UTCBs are used to exchange thread related data with the kernel. In the case of exception IPC they are used to transfer the register state to the exception handler and transfer the new state back.

The state could also be transfered with the IPC itself. Short IPC is not usable for this as only two or three message words can be transfered. Long IPC must be used instead. Setting up a long IPC message in the microkernel is more complicated than for short IPC, memory areas must be assigned and properly copied. Additionally, the X.2 specification already uses UTCBs making it not worthwhile to introduce another mechanism to transfer the exception state.

Exception IPC makes the exception handling in L^4Linux a lot simpler. First, it avoids the code in the user process that handles the exception reflection. Furthermore it avoids the memory region that is shared between the Linux server and the user process. Additionally it saves one kernel entry while executing an exception as the exception is not reflected back to the faulting thread but sent directly to the exception handler.

Another way to enter the Linux server for a user process are page faults. To handle them properly, the Linux server needs the same full register state of the process as with exceptions. The Linux server may need to handle signals directly after handling a page fault, for instance to run a signal handler if a fault could not be resolved. To handle signals, the state of the user process is needed. Therefore, the microkernel needs to save the thread state when page faults happen. Then page faults are just a special form of exceptions that are treated differently from other exceptions by the microkernel. A negative side is that page faults can usually be resolved and thus the register state is not necessarily needed, imposing a penalty to the majority of page faults. Another way to get the register state of a thread is to force it to an exception so that it reenters the Linux server with its full state. This approach adds more complexity to the Linux server.

Besides the positive sides of exception IPC for the design of L^{4}Linux, it has negative sides as well. One problem is the size of the register set that needs to be transfered in the UTCB. The x86 architecture has a very small register set, which makes it affordable to copy the whole state of the thread into the UTCB even if the whole state may not be needed to resolve the exception. Compared to the x86 architecture others like IA64 have a quite large register set of about a kilobyte in size, making it expensive to copy the whole state into the UTCB. Most likely only very few registers are even needed to handle the exception. For example a Linux system call on IA64 only uses very few registers out of the available ones, the others can just be neglected. Exception IPC is not reasonable to use on these architectures. Other solutions are needed in this case. One could be to use the old L^{4}Linux design again that registers an exception handler function for each thread. This implies the need for additional code, which can transfer the needed state to the Linux server and receive a new state again.

Another solution could be a system call that allows to get a specific set of register values for a thread, provided it is available in the kernel. This approach has the consequence that the whole register set needs to be saved accessible upon kernel entry as it may be requested later. This imposes an additional overhead that will not be used in most cases. Additionally it requires another kernel entry, which may be expensive.

To avoid the need to save the complete processor state, exception handlers could register for the registers they need. The kernel could then only copy the requested register instead of the whole state if it is possible to implement such a mechanism efficiently.

Another problem is the time a thread needs to enter the kernel on different architectures. Compared with the exception handler design, the exception IPC mechanism allows us to save one kernel entry for exceptions. This is important on the x86 architecture as kernel entries and exits are relatively expensive and can reach up to a thousand cycles. On other architectures like IA64 or PowerPC, kernel entries and exits are quite cheap with about 40 cycles.

These facts lead to the decision that on the x86 architecture with a small register set and expensive kernel entries exception IPC is reasonable as it avoids one kernel entry for L^{4}Linux. On the other hand, architectures with big register sets need to handle exceptions in the address space where the exception happens and pass only the needed data to the exception handler, or use other techniques to reduce the overhead of a full register-set copy. Additionally, architectures with big register sets mostly also have quite low kernel entry times making additional kernel entries more affordable.

Looking from the L^{4}Linux perspective, exception IPC is easier to handle since it avoids to put code in each user process address space and additionally avoids the implementation of the needed communication protocol.

To summarize, for L^{4}Linux we need to implement exception IPC including UTCBs for Fiasco V2. Additionally, the page fault handling code in the kernel needs to be extended to save the complete state of the faulting thread and pass it in the UTCB to the page fault handler.

3.3 L^{4}Linux Modifications

3.3.1 Sequential Activity

One part in the design of L^{4}Linux worth discussing is the way how user processes are released to run and how the Linux scheduler is affected by this. In the traditional L^{4}Linux design multiple user processes can be running outside L^{4}Linux, that is they are not blocking in the Linux server. This fact has several consequences as the Linux design only expects one process per Linux server running at a time. On SMP systems separate Linux servers are started on each CPU. The scheduler in L^{4}Linux must therefore be modified to only consider user processes that are blocked in the server. The state of each process whether it is blocked in the server or runnable must be held in a data structure in the server. The L^{4}Linux system also suffers from decreased responsiveness when CPU bound processes are run. All user processes have the same static priority in the L4 system and CPU bound processes, compared to others, always take their full time slice, thus use relatively more CPU time and make the system less responsive. The Linux system adapts the priorities of the user processes dynamically to reflect their status. The L4 priorities of the processes would have to be adapted continuously imposing a sensible overhead. Additionally, static priorities have the benefit to be predictable within the L4 system, changing priorities may influence the whole L4 system with other programs, although the L^{4}Linux priorities may only change within a certain window.

To improve the responsiveness of the system, user processes must not run concurrently. Running only one process avoids the influence of the L4 scheduler. This requirement can be extended that user processes may not even run if the Linux server runs, giving the possibility to keep the Linux scheduler unmodified as all user processes can be chosen to run next. Additionally, the modification to the generic signal code of Linux to explicitly post signals as described in 3.1.3 on page 18 is not necessary any more. Through the timer interrupt user processes are now regularly interrupted and thus can consume their signals.

To accomplish this, the Linux server has to make sure that whenever it resumes activity no user process is running. This depends on the way the server is woken up. If it is woken up through an incoming exception or page fault, the causing thread now waits for the server to reply and thus is not running. Wakeup events may also be generated by interrupt activity. While an interrupt happens the Linux server can either be running or sleeping waiting for an event to wake it up. If it is running, the interrupt activity does not need to send any signal to the Linux server as it will pick up signals or reschedule before going to sleep again. If it is sleeping, the server needs to be woken up so that the server can take care of the new work. When such a wakeup happens, a potentially running user process needs to be interrupted as well. If a user process runs, the wakeup could happen by interrupting the user process, which then wakes up the Linux server by entering it. If no user process is running, a wakeup message must be sent to the Linux server.

Compared to the traditional design of L^{4}Linux, only the wakeup by interrupt activity differs. Besides the information whether the Linux server is sleeping, which is already available, we need to know whether a user process is running or not. If user process is running, it needs to be interrupted by the Linux server first. This procedure can be implemented using the already available enhanced Ex-Regs system call by setting the running user process to an instruction causing an exception and thus forcing it to enter the Linux server. In the server it can be put to sleep. The disadvantage of this approach is the overhead it causes, as this is not synchronous but involves two independent events, first the Ex-Regs system call and then the exception from the user processes. Handling these events needs additional code in the Linux server and also imposes an overhead to the L^{4}Linux system that will likely be noticeable.

3.3.2 Timeout-Driven Flow

The mechanism of sequential execution within an L^{4}Linux system can be brought further and is based on an approach similar to Preemption IPC [45]. Preemption IPC is used for periodic scheduling in Fiasco, threads can have a preemption handler receiving an IPC notification if a thread misses its previously defined deadline. Preemption IPC is designed to only work in periodic mode and in reservation scheduling contexts. L^{4}Linux runs in the conventional, nonperiodic scheduling mode. Furthermore preemption IPC is asynchronous, that is, the thread missing its deadline runs on if possible while the IPC notification is sent to the preemption handler.

L^{4}Linux could use a similar approach, which will be named *time fault* further on. A user process is given a certain amount of time it can consume while running. If it runs out of time, a *time fault* is generated and sent to its *time-fault* handler. In the case of L^{4}Linux this handler is the Linux server. The *time fault* needs to behave similar to exceptions as the same register-set handling is necessary. The user process is blocked until the time-fault handler responds with a new amount of time for the process to consume.

Using this scheme, the currently used timer-interrupt thread would not be needed anymore. The user process being dispatched could be given exactly the amount of time it needs to enter the Linux server again when the timer interrupt is due. If such a point is reached, the timer interrupt must be handled. Doing this in the Linux server has the benefit that it can run in the context of the thread that caused the *time fault* and letting time accounting for processes in Linux work unmodified. Currently, the time accounting must be altered since the timer interrupt runs in its own thread with its own stack and not in the context of the current user process. Depending on the frequency of the timer in the microkernel and the granularity that should be achieved it may need to be necessary to support timeouts with arbitrary times in the microkernel. One way to achieve this is to reprogram the APIC in every interrupt for the next timeout.

When using this mechanism for the timer interrupt in L^{4}Linux, specifying absolute points in time instead of relative amounts of time is beneficial. Calculating an amount of time and using it in an L4 system call are two separate steps. The system may be interrupted between this two steps, deferring the timeout as well. This scenario is not possible with absolute timeouts.

Using the Preemption IPC of Fiasco for the described mechanism would be possible if the L^{4}Linux user processes run in the reservation scheduling context. Stopping the user processes can be done by programming two time slices, the first is used to consume the admitted time and the second to stop the process by setting it to priority 0. Threads with priority 0 are not scheduled. With the X.2 interface a scheduler thread can define a time quantum, a period of time, absolute timeouts are not possible. Exhausting the quantum leads to a notification to the scheduler thread. Both systems do not treat these events as exceptions and thus do not provide the state needed for L^{4}Linux. Either this can be conditionally requested or another mechanism to access the state must be used.

This design allows further simplifications. The Linux server is now woken up with every timer interrupt as the server needs to handle it itself. Additionally no user process is runnable when the Linux server is running, fulfilling the requirement from the previous section. The explicit signal notification is also not needed anymore as a user process enters the Linux server at least every timer interrupt.

Integrating hardware interrupts in this concept looks complicated. In the current L4 model, interrupts are exported to user level using IPC, that is a program wanting to handle an interrupt needs to block in an IPC and wait for the interrupt to occur. For applications like L^{4}Linux this has the consequence that one or more interrupt threads are needed. To handle them in the same way as the timer interrupt, the microkernel would need to interrupt a possibly running user process. The problem is that the kernel does not know which process belongs to an L^{4}Linux system and which does not. Additionally, multiple L^{4}Linux systems may be running in the system making a process even harder to identify. This approach cannot be taken, the kernel needs to know a fixed endpoint for the delivery of interrupt events. A fixed point is the Linux server. The server could register itself to receive interrupt events side by side with page faults and exceptions. Now, if the Linux server receives an interrupt it needs to interrupt a potentially running user process first. The interrupt can then be handled in the context of the interrupted thread or the idle thread in case no process was active. The downside of this approach is that the delivery of an interrupt must wait until the receiver is ready to receive IPC messages again, increasing interrupt latency.

Another solution that does not increase interrupt latency is to propagate interrupt events with the Ex-Regs system call. When an interrupt occurs, the corresponding interrupt thread sets the Linux server to an interrupt handle function. The interrupt would then be handled by the server. Here again, a possibly running user process needs to be interrupted first, giving the possibility to execute the interrupt handler routine in the context of that process. When done, the Linux server needs to continue execution at the point it was interrupted. The original instruction pointer was saved by the interrupt thread, upon completion of the interrupt handling function the Linux server can instruct the interrupt thread to reset it to its previous location and the interrupt thread can wait for the next interrupt. This mechanism can also be used for the timer interrupt.

In the case multiple interrupt threads are needed, nesting of interrupt routines as well as interrupt priorities should be considered but as future designs of Fiasco will only need one interrupt thread this is not necessary.

3.3.3 Interrupting User Processes

Another way to interrupt a user process is to just stop it, for example by clearing its ready flag in the kernel and making sure that the process is not scheduled until the flag is enabled again. When stopping a user process, the Linux server also needs to get the register state of the process and when starting it again it also needs to set potentially new registers. For performance reasons this new functionality should go into one kernel system call, for instance an extended version of the Ex-Regs system call. With only one call for the two tasks, the stop and restart function can be accomplished with two kernel entries, whereas with exception IPC three kernel entries are needed, consisting of the Ex-Regs, the following exception IPC and the exception reply. One prerequisite for this to work is that the state is always known in the kernel and can be delivered. Figure 3.2 on the next page shows the two methods in comparison.

As discussed in previous sections, exception IPC has disadvantages on architectures with big register sets. Furthermore, on Linux on these architectures, the exception handler usually does not need all registers to resolve the fault, it mostly only needs very few of them. It is even not needed to preserve these other registers across system calls or exceptions.

This allows to copy only the needed registers and forget about the others but it now needs to be specified to the kernel which registers need to be copied. Specifying the registers in the call itself makes it necessary to store all registers when the thread itself enters the kernel as any registers may be requested later. Furthermore, checking for single registers to copy will nearly cost as much as copying the register immediately, making the method less attractive.

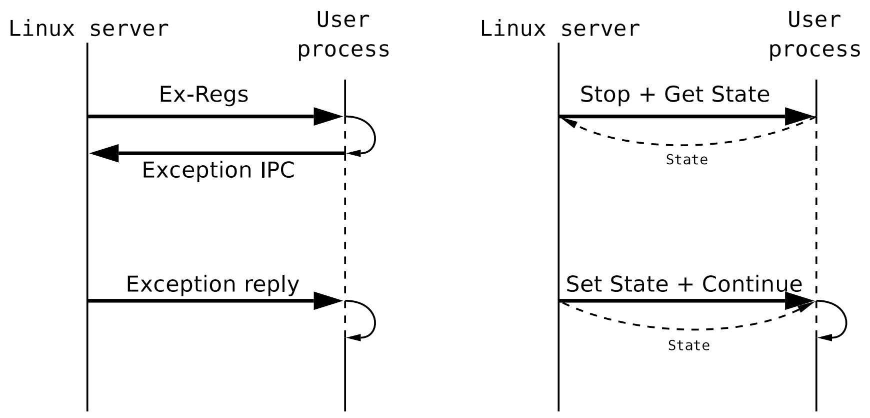

Figure 3.2: User process interruption

3.3.4 Mapping Helper Task

In previous L^{4}Linux versions a helper task called *Ping-Pong* was started to get vmalloc memory remapped into the address space of the Linux server again. The kernel should not allow intra task mappings, giving two threads in an address space the possibility to exhaust kernel resources by mapping many pages from one thread to another. Mappings can only be established between tasks, task creation is a privileged operation making it possible to restrict the creation to allowed tasks. Nevertheless, at least Fiasco does not implement this restriction. The helper for vmalloc memory can therefore be also located in address space of the Linux server. However, in the future, the kernel resources problems need to be solved.

L^{4}Linux-2.6 uses the L4Env environment, which already provides a pager for every L4Env application. A *Ping-Pong* task or thread is therefore not needed anymore. The Linux server just has to make sure to register vmalloc areas pinned at its region manager.

3.3.5 Mapping Linux Threads to L4 Threads

Previous versions of L^{4}Linux did not handle Linux threads differently from normal processes in any way, notably they did not share their address space with other threads in their process group. This unnecessarily wasted memory needed to store page tables and other address space related data in the microkernel. Mapping Linux threads to L4 threads has long been on the list of items to implement in L^{4}Linux.

With L^{4}Linux-2.6 this task has become considerably easier to implement, as no signal thread in the user process, no emulation library nor any process-local memory is needed anymore, which would need attention if multiple threads would run within one address space. As described in Section 3.1 on page 17 the traditional L^{4}Linux design only expects one user thread within an address space.

To create additional threads in an address space the enhanced Ex-Regs system call can be used making it possible to start threads without any help from a service in the address space itself.

If the L^{4}Linux design that multiple user processes can be running is kept, the Linux server needs to keep a data structure to lookup the task structure of each user process in the Linux system by thread IDs. This is needed as upon receiving a request from a user process with

IPC only the source thread ID is known. In previous L^{4}Linux versions this data structure is an array that can be directly accessed with the task number as the index. Using the task number as an index was possible as only one user thread with an address space could exist. Additionally, the L4 V2 API only uses 11 bits for task numbers implying that a maximum of 2048 tasks can exist in the whole L4 system. The amount of memory needed to store the data structure is therefore reasonably small and of fixed size. This scheme also works for the X.0 API as the corresponding fields in the thread IDs are smaller there.

Putting multiple threads into one address space makes it necessary to implement another way to store the relationship between L4 thread IDs and Linux process IDs as all threads within one address space have the same task number. Furthermore, it is now possible to execute more than 2048 Linux processes if some of them share their address space and run together in one task. As each task can contain up to 128 threads the theoretical maximum of Linux processes sums up to about 260,000 processes. This amount makes it necessary to dynamically allocate the data structure as otherwise a huge amount of memory would be wasted. A typical Linux system only contains about 100 processes.

The data structure used to store the relationship has to be optimized for fast lookups as this is the operation which will be most often used in the dispatch loop. Insertion and removal of entries is only done on process creation and deletion, which is a rare event compared to the amount of system calls done by user processes.

In the L4 V2 and X.0 APIs the number of possible address spaces is fixed, so that the structure could consist of two stages. The first is an array of task numbers that can be accessed directly, each element of the array points to either a list or array holding the threads of a particular task. Using two arrays has the benefit of fast lookups because both entries can be accessed directly. Address spaces that only consist of one thread can omit the seconds stage and encode the address of the task structure directly into the first array. On the other hand, a fixed sized array for the second stage wastes memory for most cases as there are system calls like vfork [49] that use a process that shares the its address space with the parent to immediately use the exec [8] system call to open a new user process in a new address space. In this case the second thread only exists for a very short time but would nevertheless allocate the second stage array. A linked list would use the memory more sparely but cannot be accessed directly, the list has to be searched linearly, which is slower than the direct access of the arrays. Using a tree structure instead of lists speeds up searches but unfortunately no generic tree data structure implementation is available in the Linux kernel, as opposed to a list implementation.

Considering the L4 X.2 API, address spaces and threads are separated, especially there is no limit how many threads can be in an address space. The thread ID only consists of a thread number and a version field, which are 18 and 14 bits long on a 32 bit architecture. The version field may be freely set by the thread that creates and deletes threads and can be used as an index into an array. Each element of the array would then be the head of a linked list, which holds the addresses of task structures with the same version number. The version number and therefore the index in the field must be determined when creating threads, a simple algorithm would be to apply a round-robin mechanism. This could lead to the situation that the array is unbalanced and lookups take longer as needed, especially if the array is small. To avoid this situation, a counter field in each array element denoting the number of list elements can be used, upon insertion of new elements the index in the array with the lowest number of list elements must be taken. Finding the number imposes a performance penalty for insertion but avoids potential delays for lookups.

Figure 3.3 on the next page depicts the two variants discussed previously. Both variants start with an array and either use lists or arrays to store the elements.

Summarized, for both L4 APIs a similar approach can be taken, they just differentiate

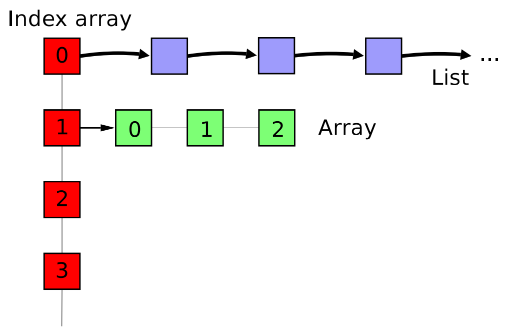

Figure 3.3: Data structures for thread to user process mappings

in small areas but the basic concept of an array indexing a linked list is the same. The implementation needs to offer the generic framework and API specific parts need to be implemented by L4 API specialized code.

3.4 Summary

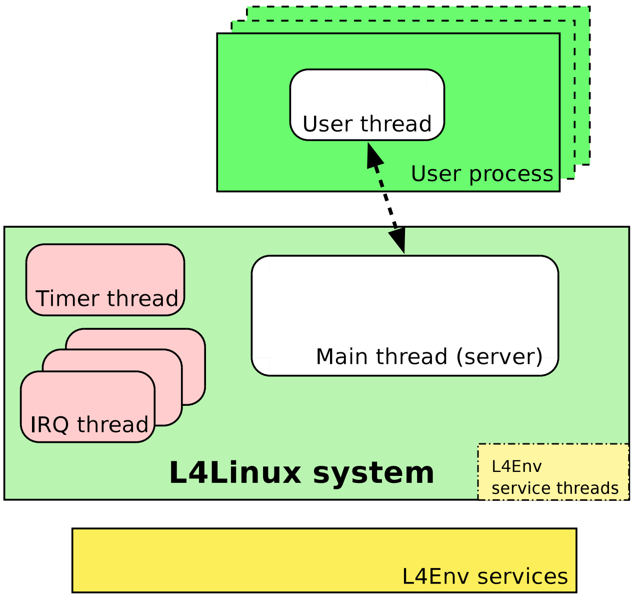

Figure 3.4: Design overview for L^{4}Linux-2.6

In the previous sections I presented several ways to change the design of L^{4}Linux. Not all ideas can be implemented in the context of this work, a compromise must be found. Figure 3.4 shows an overview of the design that can be achieved.

Notably, compared to the traditional design in Figure 3.1 on page 18, the signal thread and emulation code in the user processes are not needed anymore as the corresponding L4 primitive can easily be extended to support the needed functionality from the Linux server. By using the available L4 services the L^{4}Linux internal pagers are not needed anymore as well. No changes will be done to the interrupt subsystem with in L^{4}Linux, the timer and interrupt threads shall remain separate.

Chapter 4

Implementation

The main goal of this work is to create an L4 port of Linux 2.6. In the first step it has to be decided which environment to base the work on. As I worked on the `L4Env` version of L^{4}Linux before I also chose it as the basis for L^{4}Linux-2.6. Additionally, `L4Env` provides some services to L4 applications, which can be used by L^{4}Linux and do not need to be implemented by L^{4}Linux itself [28]. At the time of this writing, using `L4Env` implicitly means using the Version 2 or X.0 L4 API.

Another goal of this work is to find ways to simplify the design of L^{4}Linux by also allowing modifications of the microkernel. Some of the ideas discussed in Chapter 3 were also implemented. This chapter will first describe the implemented modifications to the Fiasco microkernel and then explain how this influences the L^{4}Linux design.

4.1 Fiasco Microkernel Modifications

This section will discuss the enhanced version of `l4_thread_ex_regs`, UTCBs and their influence on exceptions and page-faults.

4.1.1 `l4_inter_task_ex_regs`

The original `l4_thread_ex_regs` system call allows to create new threads and to manipulate some parameters of existing threads like the instruction and stack pointers. It has effect only in the address space it is called in.

In the context of L^{4}Linux it is valuable to be able to modify the instruction pointer of a thread in another address space, for example to force a user process to execute certain code and enter the Linux server.

The enhanced `l4_thread_ex_regs` system call was implemented by a new system call named `l4_inter_task_ex_regs`, denoting its possibility to have effect over address space boundaries. To implement this system call, the original `l4_thread_ex_regs` system call was split into two parts, one generic and one particular for `l4_thread_ex_regs`. The generic part can also be used by the new `l4_inter_task_ex_regs` system call and just needs a different initialization than the unmodified version.

Instead of implementing this feature with a new system call it would have been also possible to modify the existing `l4_thread_ex_regs` system call as the additional functionality should not influence existing programs. Nevertheless it may be possible that a programs relies on the fact that the call only uses the thread number out of the provided thread ID. With the enhanced version more field are used, which can lead to unintentional behavior. To avoid this situation, the possibility in the same system call was disregarded. The normal `l4_thread_ex_regs` system call still works as described in the specification.

The `l4_thread_ex_regs` system call encodes the thread number of the thread it wants to affect in the lower bits of a register. The remaining bits are unused. To address other tasks we also need to add at least the task number. For correctness the version field should also be passed to the kernel. In the V2 API a thread ID has a width of 64 bits whereas the task, thread and version bits are all located in the lower 32 bits of the ID, so that these lower bits can be passed to the kernel. The X.0 API defines thread IDs to be 32 bits wide so that the whole ID can be passed unmodified to the kernel.

Creating threads in remote address spaces also needed some changes in the code that enqueues the new thread into the present list. Thread 0 of a new address space must always be enqueued before its creator and additional threads need be enqueued after thread 0 of the same address space. The code in the Thread constructor handling the enqueue operation did always assume that a new address space was created when the creator of the thread was not in the same address space as the new thread. This assumption is not valid anymore with the enhanced Ex-Regs system call. The problem was solved by differing between thread 0 and additional threads when the creator of the thread is in another address space. Without this change Fiasco hangs when the corresponding threads are shut down.

Using the `l4_thread_ex_regs` system call over address-space boundaries implies certain security aspects. In general it should not be allowed that a thread in one address space can modify a thread in another address space. The introduced `l4_inter_task_ex_regs` system call allows this but is limited in a way that only the pager task of the thread to be modified can perform the system call successfully, other arrangements are prohibited. This may not be sufficient for general use but suits very well for L^{4}Linux. For general use it should be possible that a thread can register one or more Ex-Regs threads or that a master thread can impose the right to use this function or give capability to other threads without the help of the target thread.

4.1.2 UTCBs

UTCBs, or User-level Thread Control Blocks, are a concept introduced with the X.2 specification of L4. Generally, they are used to exchange data between a thread and the kernel, for instance to transfer message registers in IPCs. They are also being used for exceptions, where the register state of the faulting thread is stored in the UTCB of the exception handler.

In the Version 2 interface of Fiasco, UTCBs are not available and had to be implemented to be used for exception IPC. Additionally, at the time of this writing, Fiasco is being extended in various directions where UTCBs are also needed for an implementation of local IPC and the X.2 API. Only one UTCB implementation will finally be merged into the kernel.

The UTCB implementation for L^{4}Linux is quite simple and will not fulfill all the needs of the X.2 specification. A UTCB is a unique region of memory, which is shared between the kernel and a thread. This is implemented by allocating one page per task when the task is created and by mapping it to a fixed address just below the end of a user address space. With a fixed address no further protocol to find the UTCB is needed for a thread. Each thread of a task can find its UTCB area within the page with a simple offset calculation.

The size of a UTCB is fixed and is just big enough to hold the register state of a thread. Although one page is not enough to hold all UTCBs for all possible threads of a task it is sufficient for the Linux server since it only has very few threads. To cover all UTCBs only one additional page would be needed, as with a size of 48 Bytes per UTCB, 85 will fit into one page. The number of maximal threads per task is 128.

Field	**Description**
`status`	Enable exception IPC

`eip`
`esp`
`eflags`
	Thread state

`eax`, `ebx`
`ecx`, `edx`
`edi`, `esi`, `ebp`
	General purpose registers

`err`, `trapno`	Exception information
`snd_size`, `rcv_size`	Size of the UTCB to send or receive

Figure 4.1: UTCB layout on the x86 architecture

4.1.3 Exception IPC

Exceptions caused in the user level trap into the kernel and finally call the `handle_slow_-trap` function of the `Thread` class. This function deals with all the work needed to handle an exception including reflection to the user level, instruction emulation and the `int3` debugging extension. On the x86 architecture, the `lidt` instruction is used to register an interrupt descriptor table [23]. This instruction can only be called in privileged code like in the kernel and causes an exception for user level programs. The kernel uses this fact to emulate an IDT for user level programs, which can install an exception handler in their address space this way. This also means that exceptions are always reflected to their originating address space.

Exception IPC can now hook into the `handle_slow_trap` function and instead of calling the emulation code it generates an IPC message to the exception handler of the faulting thread. The generation of the exception IPC is similar to the generation of a page-fault IPC. The IPC operation consists of two phases, a send and a receive phase. First, the exception IPC needs to be sent to the exception handler, then the receive phase is entered waiting for the handler to reply. The thread continues when the reply was received.

The exception handler also needs to modify the state of the faulting thread so that the fault can be resolved. Mostly the instruction pointer will be modified, for example to jump over the instruction that caused the exception. The register state is saved during exception entry in the kernel and stored on the kernel stack of the thread causing the exception. A pointer to this memory region is given to the kernel function that handles exception IPC. The register state can be copied into the UTCB of the handler when the sender and the receiver have done their IPC rendezvous. When receiving the reply from the handler the contents of the UTCB are copied back to the memory region, overwriting it with possibly new values. The thread continues execution by reloading the values from its kernel stack before entering user mode again.

Another issue is the exception handler itself. For general use, it needs to be assigned for every thread when creating it. As L4 version 2 does not have the concept of exception handlers a way needs to be found to assign one. In the case of L^{4}Linux, this handler is the Linux server, which is also the pager of all user threads. The easiest way to determine the handler is to take the pager. To differentiate between page faults and exception messages, the exception message encodes special magic values in the IPC message words. The values are chosen in a way that they are unlikely to happen in page faults by setting the original page-fault address to a value greater than 3GB. Nevertheless, a type specifier for IPCs would solve the problem more nicely.

Changing the kernel behavior for exceptions does influence programs that register an exception handler in their address space or pagers that receive IPC page-fault messages they do not understand. To avoid this situation, a pager capable of handling exception IPC needs to enable this functionality by setting a bit in its UTCB. All other pagers will not receive exception IPCs and exceptions will be reflected in the traditional way.

4.1.4 Page Faults and UTCBs

Page faults are also exceptions, except that they are handled differently. Pagers of threads receive a page-fault IPC message on page faults and can reply with a mapping for this thread to resolve them. A page-fault IPC contains the instruction pointer and the page-fault address. No other data is available for the pager. For the Linux server this is not sufficient, as it might be necessary to run signal handlers, especially if a page fault could not be resolved. Running signal handlers needs the register state of the faulted thread.

As page-fault handling is a timing critical operation in the kernel, only the absolutely necessary state is saved upon entry in the kernel and the full state is not available. So to get the full state without modifying the page-fault path, the Linux server would need to force the thread into an exception. The thread would then immediately reenter the Linux server including its state. This approach needs some management efforts in the Linux server.

The most convenient way for the Linux server is if a page-fault IPC also delivers the state of the thread in the UTCB, which allows to handle exceptions and page faults nearly in the same way in the Linux server. Signals can be handled without further work as the needed register state is available. To implement this, the page-fault path in the kernel was modified. In the page-fault entry code, the full state of the thread must be saved and a pointer to the memory region with the state must be given to the handling function. In the handling function itself, the UTCB has to be copied appropriately before and after the IPC is sent to the pager and the reply is received.

4.2 The Linux Port

Linux has been ported to run on top of L4 several times in the past, as described in Section 3.1 on page 17.

The work on Linux 2.6 is based on L^{4}Linux-2.4. It will only take the parts over that are necessary to use L^{4}Linux as an `L4Env` application and leave the other parts behind. Additionally, in the beginning the port will only support one environment with one L4 API on one architecture to keep the amount of code and its complexity small. Other architectures, APIs or environments can be added later when the basic functionality has stabilized. The focus is to get a reasonably well working port in the limited time that is available for this work.

The port was started by adding an L4 architecture using the directory layout from L^{4}Linux-2.4 in an unmodified Linux 2.6 tree. Creating a new architecture in Linux means to create two directories, `include/asm-l4` for architectural header files and `arch/l4` for architectural code. L^{4}Linux is closely associated with the architecture it is ported on so that many code and files can be shared with this architecture. Header files need only slight modifications. The sharing is done by putting the modified header files in a subdirectory `arch-ARCH` under `asm-l4` and let the build process generate helper header files in `asm-l4`, which either include the file from `arch-ARCH` if available or the file from `asm-ARCH`. This way only the files that are modified need to be copied, the other ones are used from the corresponding architecture tree. The directory layout showed to be convenient to merge new Linux upstream versions into the L^{4}Linux source tree. The directory layout is depicted in Figure 4.2.

```
arch/
     i386/
          ...
     l4/
        kernel/
        lib/
        l4lxlib/
        ...
...
include/
        asm-i386/
        asm-l4/
               generic/
               arch-i386/
               api-l4env/
               l4lxapi/
```

Figure 4.2: Directory layout in the Linux tree

The actual porting work was started by adapting the header files for L^{4}Linux and getting every source file to compile. This work is finished when all the source files compile and just the final linking steps fail. The linking fails with lots of undefined symbols being referenced by the compiled files. Consequently the next step is to populate the `arch/l4` directory with the implementation of the architectural parts of the kernel. As with the header files, the source files in `arch/l4` share code with their architecture tree as well. Files that can be taken unmodified are linked from their architecture so that duplication is avoided. Other functionality must be implemented differently so that either modifications of existing files or new implementations are added. If modifications are needed, the corresponding file is taken and modified but in a way that the original file can still be recognized and merging of future Linux releases is not hampered too heavily. Despite code implementing the architectural interface of Linux, the code also needs L4 specific functionality. In L^{4}Linux-2.4 some of this code was put into an abstraction layer that defines a common interface to be implemented by all L4 APIs [28]. This layer was designed with the possibility in mind to use it across different L^{4}Linux versions. For L^{4}Linux-2.6 it could just be copied and worked instantly without modifications. This work is finished if all the symbols have been implemented and the kernel links to an executable.

With a kernel image available, the system can now be tested and all the missing functionality be added.

Compared to previous L^{4}Linux versions several things in the design of L^{4}Linux could be changed now that the microkernel offers more flexibility.

4.2.1 Exception handling

As described in Section 3.1.4 on page 19, previous L^{4}Linux versions installed local exception handlers in every address space to handle system calls and exceptions. With exception IPC, whenever a thread triggers an exception, the kernel will send an IPC to the exception handler of the thread along with the state in the UTCB so that the exception can be handled. The thread continues when the handler replies to the IPC with a new state.

When the Linux server receives an exception IPC for one of its user processes, it first needs to copy the state from the UTCB to the Linux internal `pt_regs` structure. The Linux server can then handle the user process. Upon returning to user level, the potentially new state needs to be copied back to the UTCB and a reply to the exception IPC needs to be sent.

The Linux server needs to distinguish among different exception types such as system calls and page faults. The `int $0x80` system call can be identified by inspecting the trap and the error number of the exception. Other methods could be code inspection by identifying the code at the instruction pointer or the instruction pointer itself if the it has a special value.

Exception IPC simplifies the exception handling in L^{4}Linux. No code or shared memory regions in the user processes as well as its handling in the Linux server are needed anymore.

4.2.2 User Mapped Page

The code, which was mapped into each user process in previous L^{4}Linux versions, was called emulib, short for emulation library. It included the code for exception handling as well as the signal thread. Additionally the Linux server needed to allocate a private memory region to store data for the message exchange between the Linux server and the user process. As described previously, neither the exception handling code nor the signal thread is needed anymore, so that the emulation library and the shared data region can be removed.

On the other hand, Linux 2.6 introduced the so called vsyscall page. The Linux kernel puts machine dependent optimized kernel entry code or system calls that can be executed in user mode on this page. This page can be used by user processes. So for the x86 architecture, instead of calling `int $0x80` directly, user processes can just call the kernel entry function in the vsyscall page. If the processor supports the faster `sysenter` instruction it will be used to enter the kernel, otherwise the always working `int $0x80` will be used. The location of the vsyscall page is poked into an ELF header when a program is started.

As L^{4}Linux is binary compatible with native Linux, user programs may use this mechanism so that L^{4}Linux needs to provide it as well. Linux maps the vsyscall page near to the end of the 4GB address space in the kernel memory so that the size of the address space available to user processes is not decreased. L^{4}Linux does not have this possibility as only the lower 3 GB of the address space are available for L4 user programs. The size of the address space for user processes must therefore be decreased.

In Linux, the vsyscall page is not solely used for kernel entry code but also for the return from signal contexts. L^{4}Linux needs to provide these entry points as well. Additionally the signal exception code as well as a task-startup exception code is located in this page.

As the page does not only contain vsyscall related code but also code unique to L^{4}Linux it is called upage, short for user page. The upage only contains code, no user-process-local memory is needed, the same page can be used across all user processes.

Using the upage, L^{4}Linux could also benefit from the increased flexibility in Linux 2.6. Instead of using an exception for system calls, the upage could provide for code to directly call the Linux server, avoiding the exception overhead. The register state, which must be transfered to the Linux server and back, can be exchanged on a memory region shared with the Linux server, similar to the mechanism in previous L^{4}Linux versions. With L^{4}Linux-2.6 it is not needed anymore to modify the C library as system call code is provided by the Linux server itself. Implementing this approach is an option for the future, it is supposed to be faster as it avoids the exception for system calls at all.

4.2.3 Signalling

Section 3.1.3 on page 18 describes how previous L^{4}Linux versions handle signals. Instead of utilizing a signal thread in the address spaces of the user processes, the Linux server can now use the enhanced Ex-Regs system call as it allows to modify a thread across address space barriers.

To force a user process to enter the Linux server, the server can now just use the Ex-Regs call directly on the process and force it to execute code to enter the server. One way is to set the instruction pointer to code that triggers an exception and enters the Linux server. The Linux server can identify this exception using the instruction pointer of the exception. The code that enforces exceptions must be mapped into the address space of each user process. Another way is to set the thread to a location like the microkernel memory area above 3GB that will cause a page fault when executed. This way no code for the forced exception needs to be mapped in the user processes. As the upage is available it is also used for a forced signalling exception.

4.2.4 Process Starting

When a new user process is started, its registers need to be filled with its initial state. L4 does not offer any way to set the registers directly when creating a thread, therefore the state needs to be set in a startup phase immediately after the thread has been started.

As the Linux server does not share any local memory with each user process anymore, the state cannot be set in the user process itself. Instead, the user process must enter the Linux kernel before executing any user code. The first time a user process enters the Linux server is on the first page fault. On this occasion the initial state can be set and the user process can start executing user code. This mechanism may not work if the address space of the user process is not completely empty, which can happen if the new process shares its address space with its creator. Therefore each user process is started on an undefined instruction in the upage. Either it will cause a page fault on the upage or an exception on the undefined instruction, in any case it will enter the Linux server. The server can then reply with the initial state of the user process.

Another possibility to set up the initial state of a user process is to put the code for this procedure on the stack of the user process and start the thread at this code. This method can be used if it is not possible to set the thread state with page faults and the initial exception should be avoided for performance reasons. A downside of this approach is that code execution on the stack may be prohibited.

Because of simplicity the first approach was implemented.

4.2.5 Timer Interrupt

Prior to L^{4}Linux-2.6 the timer interrupt handler was implemented with relative timeouts, that is a loop, which periodically calls the timer interrupt handler and sleeps some amount of time in-between. This introduces the problem that the timer interrupts get inaccurate as it is unknown how much time is spent within the interrupt handler. In the past the timeout was chosen a bit smaller than the wanted timeout to compensate for the additional time in the interrupt handler itself.

With the introduction of absolute timeouts in Fiasco [45] it is now possible to implement the timer loop in a way that it calls the interrupt handler on absolute points in time, that is the timer ticks cannot drift away and it is not relevant how much time the interrupt handler needs as long as it does not take longer than an interval.

4.2.6 Scheduling

The scheduling changes for user processes proposed in Chapter 3 were prototypically implemented in the Linux server. The Linux server responds to requests from user process with either replies to an exception or with memory mappings and then waits again for new request. If the next request comes from another user process, the previous one is suspended by setting it to a special instruction which forces it to enter the Linux server. When this process finally enters the Linux server, it is put to sleep with a schedule call.

The responsiveness of the system improves a lot by this change. The downside is that the system overhead increases making the system slower.

4.3 Fiasco-UX and Device Virtualization

Running Linux as a user-mode application in another system only gives real benefit if it can communicate with this system and its outside world. Given sufficient privileges, an L^{4}Linux system can use its own drivers to access hardware like the keyboard, graphics memory, network cards or disks, provided that no other component is also accessing the hardware.

In other scenarios, exclusive use of the hardware is not possible or the needed hardware is not even available. For example, certain setups require to start multiple instances of L^{4}Linux in the system. Presumed the system only has a single network card only one L^{4}Linux can use it, the other instances need other means for communication. The system environment may not even provide any hardware at all, Fiasco-UX [44, 12] is such an example.

4.3.1 Fiasco-UX

Fiasco-UX is a port of Fiasco to the Linux user level and runs nearly all L4 programs unmodified. Since its availability Fiasco-UX has become an invaluable tool for L4 developers as well as system administrators. Fiasco-UX runs on the development machine, no test machine with long reboot cycles is necessary, it can be stopped and restarted instantly. Fiasco-UX was used for the development of L^{4}Linux-2.6 from the beginning and still plays an important role. L^{4}Linux must therefore adhere the peculiarities of a Fiasco-UX environment.

A peculiarity of Fiasco-UX is that it is not a complete virtual machine monitor and does not provide virtualized hardware. It depends on the property that privileged instructions trap to the monitor when executed in an unprivileged mode. Unfortunately, not all instructions that may modify machine state trap on the x86 architecture. An example is the `popf` instruction that reads a words from the stack and loads it in the `eflags` register. A bit in the `eflags` register indicates whether interrupts are enabled or disabled. When the `popf` instruction is executed in user mode a new value for the bit may be loaded but it has no effect whether the interrupt state changed or not, it also does not generate a trap. Programs running on Fiasco-UX must not use these instructions and emulate them with other means.

Linux on the x86 architecture makes use of the `popf` instruction in the locking functions. When running L^{4}Linux on Fiasco-UX they must not be used and must be replaced with an implementation that achieves the same results with other means. The locking functions in Linux can be nested, the functions for entering a critical section return the previous state in a status variable. This variable is used when leaving the section to restore the previous state. On the x86 architecture the state contains the contents of the `eflags` register including the interrupts state. With the state value and the `popf` instruction the previous interrupt state is restored. This behavior can be easily emulated by saving the interrupt status into the variable before entering the critical section and enabling interrupts again after leaving the critical section if the status variable indicates enabled interrupts. A global variable must be kept to reflect the interrupt state.

A fully tamed L^{4}Linux (see Section 6 on page 48) will eventually solve all the hardware privilege problems, as its goal is to let L^{4}Linux run without any privileges at all.

Fiasco-UX supports no device emulation, L^{4}Linux must therefore be configured without any drivers that may use port-I/O instructions like the keyboard, PCI or ATA subsystems. As long as no other possibility is available, L^{4}Linux on Fiasco-UX is booted with a RAM disk that is loaded with the help of the file provider services of `L4Env`.

4.3.2 Input–Output

Nevertheless, at least input and output must be available so that L^{4}Linux can be usable. Fiasco-UX initially only supported output over the debugging extensions, another approach had to be found.

Within the DROPS project two graphical systems are available, `con` [19,20] and DOpE [10]. Both offer graphical input and output functionality. Additionally, a `con` driver stub for older L^{4}Linux versions was available that could easily be adapted for L^{4}Linux-2.6. The two systems both use the VESA framebuffer that is broadly available on nearly any x86 computer system. For the systems to work, I added a VESA emulation to Fiasco-UX. Upon startup, Fiasco-UX spawns a graphical console program and shares a frame buffer memory with it. The console program is a Linux program, runs besides Fiasco-UX and uses the SDL library [40] to display the contents of the frame buffer. On real hardware the availability and settings of the VESA subsystem are announced through the `multi_boot` structure that every program has available. Fiasco-UX uses the same way, no changes to the L4 applications are needed. Applications just map the frame buffer memory that is finally shared with the Linux console program.

The input functionality needs to get events from the console program to the L4 program. The data is exchanged over another shared memory region used for a ring buffer, the console program writes the input events into the buffer and the L4 application reads them out again. To avoid polling, notifications must be sent from the console program to the L4 application reading the input data. A possibility to send such notifications did not exist

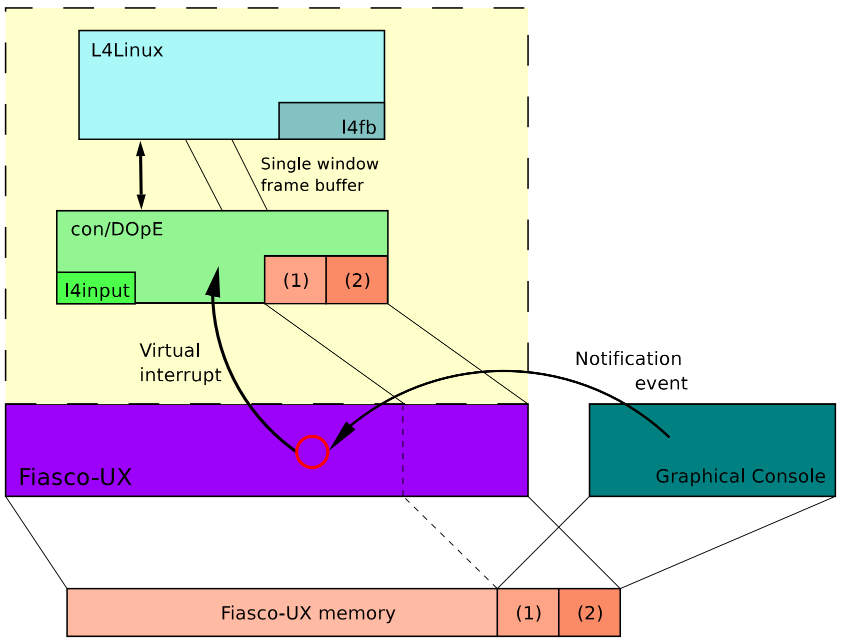

Figure 4.3: Input–output system for Fiasco-UX; (1) denotes the frame buffer memory; (2) denotes the shared memory for the exchange of input events

but could easily be added. Fiasco-UX uses an external Linux program to generate timer interrupts that regularly writes into a pipe. Whenever something can be read from the pipe, Fiasco-UX receives a SIGIO signal and consumes the data from the pipe. As the timer interrupt is the interrupt 0 in Fiasco, I extended this scheme to support more interrupts. Consequently, the console program sends notifications by writing into the pipe and the L4 application registers for the input interrupt using the normal IPC interrupt protocol. The input library is used by both graphical systems to handle input events. For Fiasco-UX I had to implement a driver to receive the input events from the console program.

4.3.3 Linux Frame Buffer Driver

Although the con driver stub for L^{4}Linux-2.6 works, the situation was not satisfying. The driver stub for DOpE did not work for L^{4}Linux-2.6 and even stopped working properly for L^{4}Linux-2.4 due to maintenance problems. Furthermore, con and DOpE have different versions of the stub, making it necessary to relink L^{4}Linux with the other stub whenever another graphical system should be used. Additionally, running the X Window System with these stubs needs a special X driver that imposes further compilation and configuration work. To provide a quicker solution, I wrote a frame buffer driver for L^{4}Linux that either uses con or DOpE depending which system is available. Using a frame buffer has the benefit that all Linux programs that use a frame buffer will work, including the X Window System; no special driver is needed as it already comes with a driver for the Linux frame buffer. The main disadvantage is the way the frame buffer window is updated. The text console system in the Linux kernel propagates partial rectangular updates properly to the frame buffer driver, so that the screen only needs to be updated if there is something to update. Unfortunately, this interface is not exported to user level programs, forcing the driver to use regular updates to refresh the screen when graphical program like the X Window System are used. This refresh always consumes processor time, even if the screen did not change at all.

The driver handles input events as well. The input library used by con and DOpE is based on Linux 2.6. Single events match the ones in Linux, so that they can be passed unmodified into the Linux input system.

4.3.4 Other Device Virtualization

Adding an input–output mechanism is an important step but cannot be the only one. To make L^4Linux more generally applicable in restricted environments at least a network virtualization is needed. This way L^4Linux could be used on much bigger file systems compared to RAM disks and data exchange would be made a lot easier. L^4Linux could also benefit from the use of an L4 block device server or a file system running on L4 by proper L^4Linux stubs.

Chapter 5

Evaluations

5.1 First Impressions

In its current state, L^{4}Linux runs fairly well. Both on native hardware as well as on Fiasco-UX. It completely boots through our workstation setups, runs the X Window System with the `fbdev` driver including KDE and other graphical applications. It also runs the AIM benchmark [2] for hours. Nevertheless, it is not really rock stable yet. For example the L^{4}Linux system does not run the LMbench benchmark [34, 33] successfully and crashes early in the Linux Test Project [32] run. This unpleasant behavior must be fixed in the future.

From a performance view the system is usable, even with enabled debugging code in the microkernel as well as in the Linux server. Using the graphical console with bigger graphic modes makes the regular updates noticeable and hampers the responsiveness of the system.

5.2 Source Code Reduction

L^{4}Linux-2.6 has also the goal to reduce the size of certain code components in both the Linux server as well as the user processes. Unfortunately the reduction of code in the Linux server is hard to measure as the `l4` tree originated from the `i386` tree and thus shares a lot of code with it. Additionally, the only basis for a comparison would be the `L4Env` part of L^{4}Linux-2.4 but the `i386` tree did change a lot between Linux 2.4 and 2.6 so that no direct comparison is possible. L^{4}Linux-2.4 also supports more microkernel versions as well as environments and therefore contains more code in generic places than L^{4}Linux-2.6.

Nevertheless a few remarks can be made. The size of code that is mapped in the user space was reduced from about 750 lines of code to about 35. Most of the L4-specific files were shortened by removing obsolete code with the goal to clean up the code base and make further enhancements better achievable.

5.3 Performance Comparisons

5.3.1 Benchmarking Environment

The test system consisted of a Pentium 3 CPU with 500MHz, 256MB of SDRAM, an Intel Ethernet Pro 100 network card, an ATA disk and a Matrox G550 AGP graphics card. The host Linux kernel for Fiasco-UX is Linux-2.4.25.

5.3.2 IPC Performance

Fiasco normally uses an optimized IPC path for the most common IPC operations, which is written in assembly language. The introduction of exception IPC makes it necessary to modify the assembler path but this has not been done up to now. Currently, the C-shortcut-path is used, which is slower but easier to modify. This comparison shows the difference between the two paths.

The pingpong program [35] is used for the benchmark on an unmodified version of Fiasco.

	Native x86		**Fiasco-UX**	
	Intra-AS	Inter-AS	Intra-AS	Inter-AS
asm-SC-path	599	837	38700	39000
C-SC-path	939	1172	46100	46400

Figure 5.1: IPC performance comparisons in CPU cycles; Intra-AS: Intra-address-space short IPC; Inter-AS: Inter-address-space short IPC; Benchmark parameters: Shortcut, Switch to receiver, int30/warm

The values for Fiasco-UX are rounded as they differ too much on several consecutive runs.

The figures show that the C-shortcut adds about 50% on the time for short IPC on native hardware, for Fiasco-UX the figures are higher but not as large as 50% compared to the assembly path.

5.3.3 Page-fault performance

The page-fault path in Fiasco has been modified to fill the UTCB of the page-fault handler with the state of the faulted thread. Although the state is only copied if the pager requests it, it is saved upon every entry. This gives the possibility to measure the overhead for saving the whole state as well as the copy operation to the UTCB of the handler.

		Difference to	
	Cycles	**(1)**	**(2)**
(1) unmodified Fiasco	3678	-	-
(2) modified Fiasco with disabled copying	3827	+145	-
(3) modified Fiasco with enabled copying	5493	+1815	+1666

Figure 5.2: Page-fault performance in CPU cycles; The figures are for Inter address space page faults

The values from Figure 5.2 show that just saving the whole state instead the minimal needed one in the page-fault path and the check whether to copy the state cost about 145 cycles. If the whole state is copied another 1666 cycles are added, summing up to 1815 cycles overhead.

Saving the whole state in the page-fault entry path is only needed when the page-fault handler requests the state. Consequently, the page-fault entry path does not need to save the whole state if it is not needed later. It should be possible to reduce the 145 cycles clearly below 100 cycles. The saved cycles will then be added to the copy sequence so that the 1815 cycles will not change significantly.

The figures for the UTCB copy operation are clearly too high and need to be reduced. The current implementation is a prototypical one that uses a flexible and expensive object oriented approach, it copies all the register bit by bit with a lot of overhead. The copying can be optimized by using a more efficient implementation and by adapting the layout of the UTCB to the layout of an exception frame, giving the possibility to copy the registers in blocks.

5.3.4 Exception Delivery

To compare exception IPC with the traditional way of reflection both methods are measured for intra and inter address space delivery. The benchmark is done with a modified version of the pingpong program. I added a test case to measure the performance of inter address space exception delivery with reflection by emulating the behavior of previous L^{4}Linux versions. The exception handler function in the thread causing the exception copies the state into a memory area shared with the handler in another address space and then does an IPC call to it. When the call returns the state is copied back and the thread continues execution.

Two test cases for intra and inter exception IPC were also added. All measurements were done with the same build of Fiasco.

	native x86		**Fiasco-UX**	
	Intra	Inter	Intra	Inter
reflection	913	2519	60800	109600
exception IPC	3589	3878	99000	100200

Figure 5.3: Exception delivery performance in CPU cycles; Intra: The exception is delivered to a thread in the same address space; Inter: The exception is delivered to a thread in another address space

An exception IPC consists of several steps, the following figures are already known from previous measurements:

State save (at least)	145 cycles
UTCB copy (twice)	1666 cycles
short inter-AS IPC (twice)	1172 cycles
Sum	2983 cycles

The figures for the state saving and the UTCB copy operation are known from the page-fault measurements. The state save costs more than shown as Figure 5.2 shows the overhead to the normal page-fault path, which already saves the only necessary state. The short IPC cost is known from Table 5.1. The steps sum up to 2983 cycles, leaving 895 cycles for other work in the inter address space exception IPC. This includes the additional cycles compared to the 145 cycles to save the full state as well as to get from the instruction that triggers an exception to the start of the handler. Overall, the 3878 cycles for an inter address space exception IPC are within the expectable range, looking at the previous figures.

Compared to the reflection mechanism, exception IPC currently adds 1359 cycles to an exception delivery across address spaces. Compared to pure short IPC, it adds 2706 cycles. Exception IPC in the same address space adds about 2670 cycles compared to the reflection mechanism, which disqualifies its use in this case. There is not much difference between intra and inter address space exception IPC. The same path in the microkernel is taken, except that in the inter address space case the address space must be switched.

One positive effect of the inter address space exception IPC can be seen in the figures for Fiasco-UX. If entries to the microkernel are expensive like in Fiasco-UX, exception IPC benefits from one kernel entry less compared to the reflection mechanism.

The performance of exception delivery must be improved. The key to improve the performance is to adapt the assembly IPC shortcut to be able to use it and to improve the copying of the state to and from the UTCB.

5.3.5 System-Call Performance

The previously measured Fiasco performance has consequences on the performance of L^{4}Linux. Two measurements will be done, first the system-call performance will be inspected, then the overall performance will be estimated.

To test the system-call performance I wrote a benchmark program that measures the following four system calls. For getpid it just measures the call itself. For fork and vfork it measures a call of the function and a waitpid in the parent as well as an exit in the child. The exec test first calls fork followed by execve to execute /bin/true. /bin/true is dynamically linked against the installed C library.

	Fiasco version	**getpid**	**fork**	**vfork**	**exec**
native Linux-2.4	-	359	34951	13226	743733
native Linux-2.6	-	360	37574	9345	882355
L^{4}Linux-2.4-V2	U	2022	319414	280830	2023229
L^{4}Linux-2.4-V2	M	2574	346156	307585	2108729
L^{4}Linux-2.4-L4Env	U	2022	468569	393202	2469525
L^{4}Linux-2.4-L4Env	M	2536	495771	420232	2588159
L^{4}Linux-2.6	M	4332	415241	41554	2655248

Figure 5.4: System-call benchmark in CPU cycles; M denotes a modified Fiasco version; U denotes an unmodified Fiasco version; L^{4}Linux-2.4-V2 denotes an L^{4}Linux-2.4 without L4Env support; L^{4}Linux-2.4-L4Env denotes an L^{4}Linux-2.4 with L4Env support

The numbers for L^{4}Linux-2.4 in Figure 5.4 show that the modified Fiasco version introduces an overhead over the unmodified version. This overhead mainly comes from the difference between the assembly IPC shortcut path and the C shortcut path as well as for the additional code in the modified version. Using the L4Env variant of L^{4}Linux-2.4 also increases the overhead.

Comparing L^{4}Linux-2.4 with L^{4}Linux-2.6 shows that the simplest system call in Linux, getpid, is 2310 cycles slower in L^{4}Linux-2.6. Exception IPC currently adds about 1350 cycles compared to the reflection mechanism as stated in Section 5.3.4. Additionally, IPC is slower and the contents of the UTCB must be copied into the Linux internal pt_regs structure and back again as the two structures do not match. The copying routines still leave room for improvements.

The figures for vfork are interesting. With L^{4}Linux-2.6 user processes that share their address space are put into the same address space, making it not necessary to create a new L4 task for the user process as previous L^{4}Linux had to. This considerably speeds up vfork and related system calls like clone.

5.3.6 Overall System Performance

To get an impression on the overall performance a Linux kernel compile was used. The following sequence was measured on an ext2 partition:

```
tar xzf linux-2.6.4.tar.gz
cp config-2.6.4 linux-2.6.4/.config
make -C linux-2.6.4 ARCH=i386
rm -rf linux-2.6.4
```

The native Linux systems as well as the L^{4}Linux systems were configured with 96MB of main memory.

	Fiasco	**Compile time**	**Compared**	
	version	mm:ss	**to (1)**	**to (2)**
(1) Native Linux-2.4	-	18:45	-	-
(2) Native Linux-2.6	-	19:08	-	-
L^{4}Linux-2.4-V2	U	20:31	+9.4%	-
L^{4}Linux-2.4-V2	M	20:45	+10.6%	-
L^{4}Linux-2.4-L4Env	U	22:25	+19.5%	-
L^{4}Linux-2.4-L4Env	M	22:36	+20.5%	-
L^{4}Linux-2.6	M	22:24	-	+17.0%

Figure 5.5: Linux kernel compile benchmark; M denotes a modified Fiasco version; U denotes an unmodified Fiasco version; L^{4}Linux-2.4-V2 denotes an L^{4}Linux-2.4 without L4Env support; L^{4}Linux-2.4-L4Env denotes an L^{4}Linux-2.4 with L4Env support

Figure 5.5 shows the differences among different Linux and L^{4}Linux versions for Linux kernel compiles. In the L^{4}Linux-2.4 section it can be seen that with the modified Fiasco as well as with the L4Env versions the compile is slower.

Interestingly L^{4}Linux-2.6 is faster than L^{4}Linux-2.2-L4Env. The reasons for this are not really clear as the previous measurements indicate a contrary result. Several facts may contribute to that result. The make utility that is used in the Linux kernel compile uses vfork, which is considerably faster in L^{4}Linux-2.6. The fork system call is also faster than in L^{4}Linux-2.4-L4Env. The work needed in the Linux server to set up a user process are smaller, especially as no emulation library and signal threads are used anymore. Additionally, there may be problems in L^{4}Linux-2.4 that increase its overhead. Nevertheless, it can be seen that the system call performance of L^{4}Linux is not the key to a fast system, it is just a part. Unfortunately there is not enough time left to investigate further into this anomaly. Hopefully, future work will disclose this secret.

Chapter 6

Future Work

Future work on L^{4}Linux includes all areas including core design, microkernel and environment support, different architectures, usage of more L4 services, especially driver servers, upstream version merging and others.

Environments

During this work Linux 2.6 has been ported to L4 using L4Env and Fiasco as the environment, The supported L4 APIs are V2 and X.0. The used environment is not the only one that can be used to run L^{4}Linux on. Possible other environments are:

- Pure V2 and X.0 environment without L4Env. L4Env depends on packages from the DROPS system and requires additional knowledge about the system and also uses system resources. For a pure L^{4}Linux usage this may not be necessary.
- X.2 microkernels. The X.2 interface is largely different from the V2 or X.0 interface. L4Env is currently not available for X.2 systems.

L4 Services

L^{4}Linux usually runs within a system consisting not only of L^{4}Linux itself but also of other L4 components. The L^{4}Linux system needs to interact with these components. This especially becomes clear for multiple L^{4}Linux instances in the system and their driver support. The instances must not interfere with their drivers to the same hardware. Normal computer systems usually do not have multiple identical hardware components like network cards. The solution to this problem is to either use L4 drivers or one special L^{4}Linux with enabled drivers which the other L^{4}Linux instances use with stub-drivers. The most benefit would come from a network driver system, as nearly all needed functionality like network access and filesystems can be used with it. A network stub would also improve the development under Fiasco-UX as it does not even have the possibility to use hardware drivers, it depends on a hardware independent solution. Besides the input–output system other services that can be used in L^{4}Linux, other candidates are the disk and filesystems of DROPS, although they are not as important as the network functionality.

Integration of the Microkernel Modifications

For this work the Fiasco microkernel has been modified. To make L^{4}Linux usable for others, the changes need to be merged in the main Fiasco repository. As features were only added and no behavior was modified, integrating the changes should be possible without conflicts. Nevertheless they should be configurable to avoid the performance penalty in cases where the features are not needed. The only problem is the UTCB implementation, as other projects are also working on UTCBs for Fiasco. Only one implementation should be merged in the kernel.

Tamed L^{4}Linux

Tamed L^{4}Linux has the goal to make L^{4}Linux always interruptible by not using any interrupt disable functions [18] or other privileged instructions. This property is needed for real-time systems like DROPS where interrupt latency should be as small as possible. Secure systems benefit from the inability of a tamed L^{4}Linux to interfere with any hardware devices.

L^{4}Linux-2.2 only offers a semi-tamed mode where the `cli` and `sti` instructions to disable and enable interrupts are exchanged with a semaphore based implementation. Unfortunately, this implementation uses `cli/sti` itself for locking. Additionally, there are still a few other appearances of these instructions left. Frank Mehnert is currently working on a truly tamed implementation that does not use any of these instructions at all, making L^{4}Linux always interruptible. This work needs to be integrated into L^{4}Linux-2.6 and future versions as well.

Hardware Separation

In its current state, L^{4}Linux always sees the hardware of the whole machine, giving the possibility to interfere with other system components. This view can be limited and increase the stability of the system. Firstly, access to the I/O-port space on the x86 architecture can be disabled for user applications and partially enabled with I/O-Flexpages [24]. Whenever a program executes an I/O-port instruction it is trapped by the microkernel, which then sends an I/O-pagefault to the corresponding page fault handler. The handler can then grant the right to use the instruction or deny it. This way the use of I/O-port can be restricted to certain hardware only.

The second point is the PCI bus configuration. Ideally L^{4}Linux should only see the parts of the bus which it is supposed to see. This way L^{4}Linux does not need to be configured specially for a particular setup, it can just be configured what L^{4}Linux can use. The `l4io` component of DROPS can provide the wanted functionality, it offers access to the PCI subsystem as well as to I/O-port space.

Maintenance

Maintenance is not a single goal for future work, it is the duty of the L^{4}Linux maintainer or maintainers to keep L^{4}Linux in a working state while details in the L4 environments are constantly changing. Other work includes merging of new Linux versions, bug fixing as well as stability and correctness improvements.

Performance Tuning

The performance of the current L^{4}Linux system can be further improved, as described in the previous chapters. This includes the implemented features in Fiasco as well as L^{4}Linux itself. For example, one tiny enhancement is to improve the start up times of user processes by mapping more than one page into the address space on the first page fault. The pages holding the `upage`, the initial code and the initial stack can be mapped at once.

Generally, the future has to show if exception IPC is a concept that can be used or if other means are more appropriate. Exception IPC is a convenient mechanism but imposes several peculiarities, especially performance related.

Portability

In the future L^{4}Linux may be ported to other architectures, microkernels for other architectures already exist. The L4 specific part of L^{4}Linux needs to be split into an architecture dependent and independent parts, code should be shared as much as possible to avoid code duplication. As the structure also needs to cope with other parts like different L4 APIs it is needs to be well thought out.

Linux Thread Local Storage

Linux 2.6 introduced a new way to access a thread local memory area for threads within an address space. On the register starved x86 architecture a small segment is used to hold the address of the memory area, the segment can be access with the `%gs` segment register [47]. Segments and segment registers can only be modified by privileged programs like the microkernel. Without changes to the microkernel segment registers are not usable by user programs. Furthermore, the X.2 L4 specification defines that the location of the UTCB can be found through `%gs`. The segment register is not available for other uses anymore.

Nevertheless, Linux programs linked against a recent version of the glibc [14] detect the version of Linux on which they run and try to setup and use the segment register for thread local storage. Neither L^{4}Linux nor the underlying microkernel have any support for user accessible segment registers right now. Until the support of any form is added, such programs do not work. Several ways may be taken to get a modern version of glibc working:

- Compile the glibc without NPTL support. NPTL is the Native POSIX Thread Library.
- The microkernel offers user level programs the possibility to set and get the `%gs` and `%fs` segments registers and saves them in context switches. The segment register `%fs` is not needed for TLS but plays a role for other applications.
- X.2 microkernels need to offer the possibility to user programs to either access the UTCB via `%gs` or let them use the segments registers themselves. Programs using the segments registers cannot use the UTCB and vice versa. This restriction does not influence L^{4}Linux as user processes normally do not execute any L4 code and thus do not need their UTCB. L^{4}Linux programs that use L4 functionality are not possible anymore.
- The X.2 specification is changed to use another means for accessing UTCBs. A possibility could be to use `%gs:4` or similar.

Encapsulated L^{4}Linux

In security-aware environments it may be necessary to encapsulate an L^{4}Linux system completely. Such an encapsulation neither allows uncontrolled privileged nor I/O-privileged operation as well as any DMA as long as no IOMMU can be used. Furthermore, user programs must not make use of any of the L4 system calls. The communication of the Linux server itself must be controlled that the server only contacts services it needs to work. These L4 services needs to be secure from attacks of the untrusted L^{4}Linux.

Chapter 7

Summary

This chapter shall give a summary on the results achieved in my work. The main goal was to port Linux-2.6 to L4, which has been achieved. The work was not started from scratch but based on $\mathrm{L^4}$Linux-2.4 and its L4Env port.

The design of the different $\mathrm{L^4}$Linux versions has been quite unmodified from the initial port up to version 2.4. Certain design decisions had been made to accommodate to the L4 design, a few of them to circumvent restrictions of the L4 interface. With $\mathrm{L^4}$Linux-2.6 a few of the restrictions of L4 were lifted, which simplified the design of $\mathrm{L^4}$Linux. I proposed several changes to the microkernel interface, a few were implemented in the context of this work.

$\mathrm{L^4}$Linux-2.6 is quite stable, runs on the modified microkernel and boots through a complete workstation setup including the X Window System and desktop environments like KDE. Nevertheless improvements can be made. $\mathrm{L^4}$Linux-2.6 contains bugs that need to be found and fixed. The performance must be further improved. Other goals are better integration in DROPS by adding stub drivers to communicate with other L4 services.

Appendix A

Glossary

APIC Advanced Programmable Interrupt Controller, Controller chip used to service interrupts and propagate them to the CPU.

ATA Advanced Technology Attachment, technology to connect disks and other storage devices to computers.

DMA Direct Memory Access, device can write data to the main memory without the help of the CPU.

DROPS Dresden Real-time OPerating System [7].

ELF Executable and Linking Format. Format describing the layout of binary and libraries files.

Fiasco Microkernel implementing the L4 interface.

Frame buffer A region of memory that is used by the graphics to store the contents of the screen. Can also be used to offer separate frame buffer windows to clients.

IDT Interrupt Descriptor Table. Table storing entry points of handles that are called on interrupt events.

IOMMU I/O Memory Management Unit. An MMU for device I/O memory, needed to map from 64 bit address spaces to 32 bit device memory space, or to protect the host memory from malfunctioning devices that write to arbitrary locations in the host memory.

IPC Inter-Process Communication.

ISP Internet Service Provider, companies that offers internet connectivity and internet related services to customers.

L4Ka::Pistachio Microkernel implementing the L4 specification X.2 and supporting multiple platforms.

MMU Memory Management Unit. Unit between the memory and the CPU to translate between virtual and physical addresses.

PCI Peripheral Component Interconnect, bus system to connect components to computer systems.

Process Denotes an L^{4}Linux user program (in the context of this work).

RPC Remote Procedure Call, call a function in another address space or in another computer system, when the systems are connected over a network.

SDL Simple Directmedia Library [40], a library for various systems that provides a frame buffer and other abstraction related to multi-media.

SMP Symmetrical Multi Processing.

Task Other notation for address space.

Thread Execution path within an address space. Multiple threads can exist in an address space.

TLS Thread Local Storage, denotes a thread private memory region in multi threaded applications.

User process Unless stated otherwise denotes a client of the Linux server, that is a Linux program running in user space.

VESA Video Electronics Standards Association, describes a standard set of video modes available in nearly all computers

vmalloc memory Linux reserves an area in its kernel virtual memory region to map virtually contiguous and physically noncontiguous memory.

VMM Virtual Machine Monitor.

Bibliography

[1] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*, pages 93–113, Atlanta, GA, June 1986.

[2] AIM multiuser benchmark. URL: `http://sourceforge.net/projects/aimbench/`.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, pages 164–177, Bolton Landing, NY, October 2003.

[4] The Bochs Emulator. URL: `http://bochs.sf.net/`.

[5] Kerstin Buchacker and Volkmar Sieh. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects. In *Third IEEE International High-Assurance Systems Engineering Symposium, HASE 2001*, pages 95–105, Boca Raton, Florida, 2001.

[6] F. B. des Places, N. Stephen, and F. D. Reynolds. Linux on the OSF Mach3 microkernel. In *Conference on Freely Distributable Software*, Boston, MA, February 1996. Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111.

[7] Dresden Realtime OPeration System. URL: `http://os.inf.tu-dresden.de/drops/`.

[8] The exec system call. URL: `http://www.opengroup.org/onlinepubs/007904975/functions/exec.html`.

[9] The FAUmachine project. URL: `http://www.faumachine.org/`.

[10] Norman Feske and Hermann Härtig. DOpE — a Window Server for Real-Time and Embedded Systems. Technical Report TUD-FI03-10-September-2003, TU Dresden, 2003.

[11] The Fiasco microkernel. URL: `http://os.inf.tu-dresden.de/fiasco/`.

[12] Fiasco-UX, the Fiasco User-Mode Port. URL: `http://os.inf.tu-dresden.de/fiasco/ux/`.

[13] Martinus Flöck. Portierung des Mikrokernels P4 auf einen ARM-Prozessor. Master's thesis, Fachhochschule Koblenz, Fachbereich Elektrotechnik und Informationstechnik, September 2002. In German.

[14] The GNU C library. URL: `http://www.gnu.org/software/libc/`.

[15] GNU Public License. URL: `http://www.gnu.org/copyleft/gpl.html`.

[16] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[17] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[18] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[19] Christian Helmuth. Ein Konsolensystem für DROPS. Term paper, TU Dresden, August 2000. In German, Available from URL: `http://os.inf.tu-dresden.de/papers_ps/helmuth-beleg.pdf`.

[20] Christian Helmuth and Frank Mehnert. DROPS Console System. Available from `http://os.inf.tu-dresden.de/local/project/manuals/drops/con/refman/`, 2002.

[21] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: `http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/`.

[22] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems.* PhD thesis, TU Dresden, Fakultät Informatik, September 2002.

[23] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999.

[24] Bernhard Kauer. I/O-Flexpages in L^{4}Linux. Praktical course work, November 2002.

[25] Implementations of the L4 microkernel interface. URL: `http://os.inf.tu-dresden.de/L4/impl.html`.

[26] L4 Environment Concept Paper. Available from URL: `http://os.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.ps`, June 2003.

[27] L^{4}Linux. URL: `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`.

[28] Adam Lackorzynski. L^{4}Linux on L4Env. Term Paper, TU Dresden, December 2002. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/adam-beleg.ps`.

[29] 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.

[30] 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.

[31] J. Liedtke. μ-kernels must and can be small. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 152–155, Seattle, WA, October 1996.

[32] The Linux Test Project. URL: `http://ltp.sf.net`.

[33] LMbench benchmark. URL: `http://www.bitmover.com/lmbench/`.

[34] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[35] L4 pingpong benchmarking program. Available from remote CVS at `http://os.inf.tu-dresden.de/drops/download.html`, path: `l4/pkg/pingpong`.

[36] The L4Ka::Pistachio microkernel. URL: `http://www.l4ka.org/projects/pistachio/`.

[37] Plex86 x86 virtual machine project. URL: `http://plex86.sf.net/`.

[38] QEMU CPU Emulator. URL: `http://fabrice.bellard.free.fr/qemu/`.

[39] Philipp Roebrock. Portierung des P4-Mikrokernels auf eine PowerPC Plattform. Master's thesis, Fachhochschule Gelsenkirchen, August 2002. In German.

[40] Simple Directmedia Library. URL: `http://www.libsdl.org/`.

[41] SID simulator framework. URL: `http://sources.redhat.com/sid/`.

[42] Volkmar Sieh and Kerstin Buchacker. Testing the Fault-Tolerance of Networked Systems. In *International Conference on Architecture of Computing Systems ARCS 2002*, pages 37–46, Karlsruhe, Germany, 2002.

[43] Simics Simulator. URL: `http://www.virtutech.com/products/simics.html`.

[44] Udo Steinberg. Fiasco microkernel User-Mode Port. Term Paper, TU Dresden, December 2002. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/steinberg-beleg.ps`.

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

[46] SYSGO AG. URL: `http://www.sysgo.de/`.

[47] ELF Handling for Thread-Local Storage. URL: `http://people.redhat.com/drepper/tls.pdf`.

[48] User-Mode-Linux. URL: `http://user-mode-linux.sf.net/`.

[49] The vfork system call. URL: `http://www.opengroup.org/onlinepubs/007904975/functions/vfork.html`.

[50] Virtual PC. URL: `http://www.microsoft.com/windowsxp/virtualpc/`.

[51] VMware. URL: `http://www.vmware.com/`.

[52] Version X.0 specification of the L4 API. URL: `http://www.l4ka.org/documentation/files/l4-86-x0.pdf`.

[53] Version X.2 specification of the L4 API. URL: `http://www.l4ka.org/documentation/files/l4-x2.pdf`.

[54] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.

[55] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the fifth symposium on Operating systems design and implementation*, pages 195–209. USENIX Association, 2002.

[56] Alexander Züpke. Linux-Portierung auf den P4 Mikrokernel. Master's thesis, Fachhochschule Gelsenkirchen, July 2003. In German.

[57] z/VM. URL: `http://www.vm.ibm.com/pubs/redbooks/`.