# Großer Beleg: "L$^4$Linux on L4Env"

Adam Lackorzynski <adam@os.inf.tu-dresden.de>
Dresden University of Technology
Department of Computer Science
Operating Systems Group

December 2002

# Contents

# 1 Introduction

Research at the Operating Systems Group in Dresden [OS02] focuses on $\mu$-kernel architectures, real time systems with quality of service requirements as well as secure systems. The $\mu$-kernel architecture is based on the second generation L4 interface, allowing to construct a small, fast and efficient $\mu$-kernels. To build a useful operating system that meets the defined goals, powerful applications on top of the $\mu$-kernel are needed. The DROPS project is aimed to provide such applications. L$^4$Linux is one part of the DROPS project and provides a UNIX-like interface to L4 systems. It can be used for such basic things as developing and debugging L4 components as well as in security architectures where crucial and important applications like financial software run besides "Internet applications" which may come from untrusted sources and should not, under any circumstances, interfere with the critical applications. This separation can be achieved with the help of L4 and the surrounding projects of DROPS.

As the L4 interface itself is quite lean and simple, an environment called L4Env is developed in Dresden to help programmers in common tasks and unify the different existing $\mu$-kernel implementations to one interface. L4Env places an abstraction layer between the $\mu$-kernel itself and its applications. There is still research work to be done on these components but parts of L4Env can already be used in applications without problems. The main goal of the work was to add L4Env support to L$^4$Linux which promises several advantages such as $\mu$-kernel independence over the traditional approach working directly on the $\mu$-kernel. Besides that several other issues like the code breakup and cleanup were pursued.

## 1.1 Overview

In this document I will give an overview of L$^4$Linux-2.4 and the prerequisites that are needed for L$^4$Linux to run. The next chapter will give an introduction to operating systems and $\mu$-kernels in particular. Following this, the $\mu$-kernels I work with are explained. The DROPS system and the L4Env environment is explained along with all the modules needed for L$^4$Linux. The chapter is completed with an overview of the history of L$^4$Linux. The third chapter devotes to the design and will focus on the L$^4$Linux internal abstraction library, the proposed directory structure of the source tree and L$^4$Linux on L4Env. The following chapter will explain some implementation details, focusing on the library with special regard to L4Env, the directory layout and finally a small additional project, the tracing of L$^4$Linux. Following this, a few preliminary benchmarking results will be presented to allow a first evaluation of L$^4$Linux/L4Env within the other version of L$^4$Linux. Chapter 6 describes some issues for the future which still need to be done in L4Linux. Finally, this document is completed with a summary.

# 2 Prerequisites and Related Work

This chapter gives an introduction to operating systems in general and to the $\mu$-kernel architecture in particular. Following this, the Drops [DRO02] project and the relevant parts to L$^4$Linux will be introduced. Furthermore L$^4$Linux in general is not a new project so that already available work will be shown as well. Readers familiar with some or all sections presented in this chapter may safely skip those sections or go straight to the next chapter.

## 2.1 Operating Systems

An operating system manages the resources of the hardware it runs on and provides an interface to applications to use the resources of the system. Besides that, it has to ensure that the programs running on the system are protected from each other so that no program can interfere with others. The resources available on the system need to be shared reasonably among the programs. Currently two major paradigms exist in the operating systems design. The first and mostly used one is the monolithic-kernel approach where the kernel-provided functionality like hardware drivers or file systems are run in kernel mode with all privileges accessible on a particular hardware. As a consequence, a faulty file system implementation can harm the whole system, for example by overwriting some important data structure in memory, as no protections mechanisms exist inside the kernel. Additionally, mostly due to their size, monolithic kernels have disadvantages in the areas of security, maintainability and flexibility.

In contrast to the monolithic approach, the $\mu$-kernel approach only runs the absolutely necessary parts of an operating system in kernel mode at the highest privilege level. These parts are usually handling of different address spaces and threads, management of the physical memory and interrupt handling. All other functionality like file systems or hardware drivers are provided by programs running non-privileged in separate address spaces. This way it is not possible that a faulty driver or file system implementation can harm the system as a whole. For communication purposes between different address spaces the kernel provides an IPC mechanism. Additionally, $\mu$-kernels are easier to handle and maintain from a software-architectural point of view as they are much smaller than traditional monolithic kernels and much less prone to errors. The biggest penalty of $\mu$-kernel based systems, compared to traditional monolithic kernels, is their performance. All the modules have to communicate with each other by using IPC, which is slower than communication in a single address space, but the additional benefits in security and maintainability outplay the small performance impact [HHL$^+$97].

## 2.2 Microkernel Architectures

The $\mu$-kernel paradigm is not new in itself. First steps where done within the Mach project [ABB+86] at Carnegie Mellon University. Although Mach did not have much success in the past it is used in some places like in the Hurd project or in Mac OS X. It basically failed because the architectural improvements could not outplay the dramatically poor performance of Mach compared to other available systems.

In 1996, Jochen Liedtke developed a second generation $\mu$-kernel [Lie96]. Instead of stripping down an existing kernel like it was done with Mach, he designed a new $\mu$-kernel interface, called L4, and provided the first implementation on the x86 architecture. The interface provides the minimally required functionality only and allows to construct highly efficient and optimized $\mu$-kernels. Since then the original L4 API was developed further and new APIs were created. The oldest version still in use is the V2 version of the API. It is implemented in the original version developed by Jochen Liedke and in the Fiasco [Fia02] $\mu$-kernel developed by Michael Hohmuth [MH] and others in Dresden.

Over the years a lot of experiences with the V2 API were gathered and L4 was ported to new hardware platforms. As the V2 API was initially developed for the x86 platform, it is not well suited for other platforms, as it puts restrictions on the API which would not be necessary on newer platforms. For example, newer platforms provide significantly more registers which cannot be used reasonably by the V2 API as it is tailored to the few registers of a x86 platform. Furthermore, research showed that some constructs in the L4 $\mu$-kernel paradigm needed to be changed to overcome conceptual problems. Consequently, new versions of the L4 API were developed. The successor of the V2 API is the VX0 API and currently the VX2 API is being designed and implemented. The VX0 API is implemented in the Hazelnut [Haz02] kernel and the VX2 API will be implemented in the Pistachio [Pis02] kernel. Both versions are developed by the Systems Architecture Group at the University of Karlsruhe [L4K02].

Although the main development of L$^4$Linux is done on Fiasco, L$^4$Linux runs on Hazelnut and Pistachio as well.

### 2.2.1 Available APIs on different hardware platforms

The $\mu$-kernel s mentioned above were ported to or developed for different hardware platforms. The current situation, which is interesting in the context of L$^4$Linux, is displayed in Table 1.

| $\mu$-kernel | API | Platform |
|---|---|---|
| Fiasco | V2 | X86, (IA64, StrongARM) |
| Hazelnut | VX0 | X86, StrongARM |
| (Pistachio) | VX2 | (X86, IA64, Power4, StrongARM) |

Table 1: $\mu$-kernels on different platforms interesting to L$^4$Linux.

The platforms or $\mu$-kernels in parentheses are currently planned or in an early implementation phase and not yet available.

For a complete list, please refer to [L4I02].

## 2.3 Inside L$^4$Linux

This section describes how L$^4$Linux works internally but it will not go into too much detail. For a deep coverage of the internals please refer to [Hoh96b].

L$^4$Linux is an L4 application running in user space, providing a binary compatible environment where unmodified Linux applications can be run in. However, L$^4$Linux needs more privileges then an ordinary user space application, mainly with respect to hardware access.

### 2.3.1 Hardware Interrupts

In the Linux kernel, interrupt handling is divided in two parts, top halves and bottom halves. The top half is executed when an interrupt arrives, acknowledging it and marking the corresponding bottom half for later execution. The bottom half then does the actual work. This scheme ensures that interrupts are blocked only for a short time, the execution time of the top half. Furthermore, to ease synchronization, top halves can only be preempted by other top halves, not by any other code (including bottom halves). Additionally, bottom halves can only be preempted by top halves, not by any other code.

In L4, interrupts are delivered through IPC messages. To receive such a message, a thread needs to wait for a specific interrupt by invoking the appropriate IPC call. It is only possible to wait for one interrupt at a time. Consequently, one thread per interrupt is used. The criteria which code can be interrupted, can be solved easily by using static priorities of the L4 system. In this case the L4 scheduler ensures that higher prioritized processes cannot be interrupted by lower ones. This mechanism will not work unmodified on SMP systems but as no stable multiprocessor capable $\mu$-kernel exists up to now the lack of SMP support is tolerable.

### 2.3.2 Scheduling

In a native Linux system the scheduler is invoked regularly. When it is executed it decides which Linux process to run next. This decision includes several factors like the priority of the process, the amount of previously granted timeslices and its interactivity. L4 systems also have a scheduler using static priorities. In L$^4$Linux, all Linux processes have the same L4 priority and are scheduled by the L4 scheduler. The internal Linux scheduler is only used when a task enters the L$^4$Linux server. This approach has the disadvantage that the Linux specific scheduling scheme (e. g. interactive vs. long running job) cannot be emulated. “Preemption Handlers” are supposed to solve this problem but unfortunately they are not implemented in any $\mu$-kernel at this time. If they get implemented, the usage in L$^4$Linux will be evaluated. Another approach to emulate the Linux scheduling would be to continuously adjust the priorities of the tasks in such a way that the Linux scheduling strategy can be enforced. Unfortunately this technique has not been evaluated in detail nor implemented up to now.

### 2.3.3 Signals

In a UNIX like system signals must be delivered to user processes. Signals are normally delivered when a user process returns to user mode after having entered kernel mode. As discussed in the previous section, a user process only enters the L$^4$Linux server explicitly and not regularly in the timer interrupts as it does under native Linux. This way it cannot be enforced that a signal is delivered in a timely manner, furthermore it may happen that a user process never enters the kernel. To solve this issue an additional thread within the Linux process is introduced, which waits for signals and forces the user task to enter the kernel to get the delivered signals. One signal thread per task is needed since thread manipulations are restricted to threads in the same task. The approach with the signal thread has the disadvantage that a malicious user task could kill or modify its signal thread and so be unresponsive to delivered signals. Nevertheless the `SIGKILL` and `SIGSTOP` signals have to work even with these user processes and so they need special treatment. The `SIGKILL` signal does not need any help of the user process and so the process is killed straight away in the L$^4$Linux server. The semantics of the `SIGSTOP` signal is hard to achieve with the current model but will be solved with the introduction of preemption handlers.

### 2.3.4 System Calls and Emulation Library

L$^4$Linux is binary compatible to a native Linux so that Linux applications can be run on L$^4$Linux without modifications. When executing a system call, Linux systems use an "`int $0x80`" call. In L4 systems this call generated an invalid exception which is trapped by the $\mu$-kernel and sent back to the thread causing it. This exception is then received by a special emulation library which is linked to every Linux program. It contains the necessary code to enter the L$^4$Linux kernel via IPC. This technique allows to use an unmodified Linux user space environment with L$^4$Linux. To avoid the exception and to speed up system calls the user space can be modified to directly use the L$^4$Linux way of entering the kernel. In the usual environment only the C library needs to be modified to enter the kernel directly via IPC.

### 2.3.5 Memory Management

Native Linux manages its physical memory itself, each port needs to provide an implementation if the page tables interface in Linux so that the virtual address spaces of Linux can be managed. The L4 mechanism for recursive address space construction is suitable to implement the memory model needed by Linux. Pages handled by Linux need to be mapped to user address spaces, they need to be revoked and page faults have to be handled.

On startup, L$^4$Linux acquires a fixed amount of memory for itself and its user processes. Then, this memory is handled by virtual memory management of the Linux server. Additionally, in an L4 environment, every thread needs a pager thread which deals with page faults. For L$^4$Linux user tasks there is a pager thread in the L$^4$Linux server which does that. As the L$^4$Linux server itself uses the virtual memory management it can cause page faults and therefore needs a proper pager as well. Because in the original version of L4 tasks cannot map pages in their same address space a separate pager task for the L$^4$Linux server is needed. This task is called "ping pong task" in L$^4$Linux.

## 2.4 Drops

Drops is abbreviated for Dresden Realtime OPerating System. With the Drops project the Operating Systems Group at the University of Dresden [OS02] pursues several goals. One goal is to deploy modern $\mu$-kernel architectures along with an environment for applications with quality of service requirements and normal time sharing applications on the same system. Within this scenario, L$^4$Linux comes into play. On the one side Drops can run native L4 applications which satisfy hard real time requirements and on the other side has the ability to run a fully fledged UNIX environment with unmodified Linux binaries.

The Drops system can currently be used with V2 (Fiasco) and VX0 (Hazelnut) $\mu$-kernels.

## 2.5 Drops system basics

To get a Drops system up and running some prerequisites are needed. The first is a boot loader which loads the $\mu$-kernel and all other needed modules into memory, preferably over a network, and finally starts the system. A minimum of two modules are mandatory for L4 systems, the RMGR and Sigma0. The Drops project uses a derived version of GRUB, the GRand Unified Boot loader [GRU02]. In addition to loading and starting the system, it sets the video mode for the Drops console (see Section 2.7.1).

RMGR, the L4 System Resource Manager, is a simple system resource manager. RMGR is also used as a bootstrapper for the $\mu$-kernel so that RMGR is actually a two stage program. It acts as a pager for the tasks it starts and additionally handles main memory, all interrupts and all task numbers. The first stage loads the $\mu$-kernel and the second one manages the resources.

The second module which is loaded besides the $\mu$-kernel is Sigma0. Though Sigma0 is a user task, it is an essential part of each L4 based system. It acts as the root of the memory mapping hierarchy and firstly receives all memory from the $\mu$-kernel. Successive user supplied pagers implement the desired memory management policy on top of it by getting memory from Sigma0 via the Sigma0 protocol.

## 2.6 L4Env

L4Env is a part of Drops and aims to provide a uniform interface to the $\mu$-kernel primitives used by programs. It is the goal of L4Env that programs written for it are independent from the underlying $\mu$-kernel. In an ideal world it is even possible that the source codes of those applications are independent from the underlying hardware platform as well provided they do not use any hardware specific code. The functionality provided by L4Env is implemented in server tasks and libraries which are linked to applications. Currently, L4Env is available for V2 and VX0 $\mu$-kernels.

L4Env includes a physical memory management server, a memory mapping and thread management library, a naming service, an I/O server, an application loader and execution system, a console and windowing system for output, an input library and network support.

One goal of my work was to add L4Env support to L$^4$Linux so that it can be run as an L4Env application. This can be seen as a new API L$^4$Linux is ported to. As a result, L$^4$Linux will support four different APIs, namely V2, VX0, VX2 and L4Env.

Porting L$^4$Linux to L4Env has several goals. Firstly, most important is the evaluation of L4Env in the context of a large application which is intended to run on different $\mu$-kernel APIs. While working on L$^4$Linux problems with L4Env and its API will show up and can be discussed with the developer of the corresponding L4Env module. Eventually, a solution needs to be found. Secondly, once L$^4$Linux is an L4Env application it can benefit from the L4Env architecture, for example running on colored cache memory [LHH97], without special support for these features in L$^4$Linux itself. Thirdly, when L4Env will finally be ported to all necessary $\mu$-kernel APIs, support for the plain $\mu$-kernel APIs could be abandoned and the maintainability of L$^4$Linux could be improved. Further, L$^4$Linux does not depend on a specific kernel interface anymore, it is runnable on every system where L4Env is already working. The downside of the L4Env approach is that it will be slower than the other versions of L$^4$Linux since more IPC with L4Env components are needed, hence all versions of L$^4$Linux will probably survive. Some performance evaluations between L4Env and non-L4Env L$^4$Linux will be shown in Section 5.

In the next sections I will introduce some parts of L4Env which are used by L$^4$Linux.

### 2.6.1 File Provider

The file provider service is responsible for loading files from a TFTP server off the net. It is currently implemented as a server which includes a TFTP client and fetches the files on behalf of the clients calling the file provider. It is used by the L4 Loader to load programs and by L$^4$Linux to get its initial RAM disk, if requested.

The server comes in two flavors. As a certain resource like a network card can only be accessed by one program at a time, L$^4$Linux with network support and the L4 TFTP server would interfere with each other and make network access impossible. To avoid this situation it is possible to replace the L4 TFTP server with a version running as a Linux process using the Linux infrastructure for network access.

### 2.6.2 L4 Loader

The L4 Loader is a server which starts and stops programs. Programs can be loaded at any time and not only on the startup of the system. They are loaded into their own virtual address space and are not directly mapped to the physical memory as it has to be done when loading programs with RMGR only. This avoids the situation where multiple programs overlap when loaded into the system and need to be recompiled with another link address.

Furthermore, the L4 loader provides a shared library against which the L4Env applications are linked when loaded, avoiding that every application has to be linked statically with the some of the basic libraries and needs to be recompiled if the library changes some of its internal code. The loader retrieves the programs with the help of the file provider mentioned above.

Old applications which are linked to certain static addresses are also supported by the loader with the same restrictions they always had. In case such an old style application needs any additional modules the loader loads those as well. This can be the case when a non-L4Env L$^4$Linux needs a RAM disk to run on. The documentation of the L4 Loader is available online at [Meh02].

### 2.6.3 dm_phys

dm_phys is short for L4Env Physical Memory Dataspace Manager and manages all the physical memory of a system. It is an implementation of a dataspace manager [APJ$^+$01]. Other implementations which provide different features such as paging or "copy-on-write" memory or use other techniques regarding memory management may be developed in the future. Please refer to the online documentation at [Reu02c] for further details. The L4Env version of L$^4$Linux uses dm_phys to get memory from the system.

### 2.6.4 Region Mapper

L4RM, the region mapper, manages the virtual address space of a task by maintaining a region map, which contains mainly the information which virtual memory region is managed by which data space manager. It is implemented as a library which is linked to the application. The region mapper fulfills two purposes. It is the pager for all threads within a task and handles page faults by issuing map calls to the dataspace manager associated with the particular page fault address. The other purpose is virtual memory allocation, L4RM has to be used for to obtain virtual memory. For details please refer to the online manual at [Reu02a].

### 2.6.5 Thread Library

The thread library provides a uniform interface for thread handling and some other miscellaneous functions that are independent from the underlying $\mu$-kernel. It considerably eases the handling of threads for application programmers, for example by implementing often used code for thread creation and stack handling. Furthermore the library makes the program more portable across different $\mu$-kernels. The Thread library will be used for the kernel threads in L$^4$Linux. For more information please consult the online reference manual at [Reu02b].

### 2.6.6 Task server

The task server creates and kills tasks as well as manages all or a certain amount of tasks in a system. Every program which wants to create a task needs to get the right to do so from the task server. In $L^4$Linux, the task server will be used to manage user space tasks.

### 2.6.7 L4IO Server

The I/O server occupies all PCI memory regions as well as all interrupts. It then hands out those resources to other programs on request so that multiple programs cannot access the same resource in parallel and usually hardly detectable errors due to incorrect hardware access can be avoided. For further details please refer to the online manual at [Hel02].

## 2.7 Other Components

The following two components are not parts of the L4Env suite but are used in the $L^4$Linux context so that they are mentioned here.

### 2.7.1 Drops Console

Running multiple programs in parallel in a multi tasking environment requires a possibility to give every program a separate input and output channel. The console provides a graphical virtual console system giving every program its own console for output and input. Users can switch between every console with a key combination. Only one console can be displayed at any given instant. To support the application programmer a library provides the access the console server. More information can be obtained online at [HM02].

### 2.7.2 DOpE Windowing System

The DOpE is quite equal to the console system except that it provides a windowing system which is able to display multiple console windows at a time similar to common window managers used in the X Window System.

## 2.8 History of $L^4$Linux

$L^4$Linux is one of the oldest and most long-standing projects at the Operating Systems Group in Dresden. It was initially started by Michael

Hohmuth [MH] and Jean Wolter [JW]. Michael Hohmuth has written his Diploma thesis about the work done on the initial L$^4$Linux port [Hoh96b]. This work was done with Linux version 2.0. Later when Linux 2.2 had been released, L$^4$Linux was ported to it by the same people. After the initial work the maintenance on the 2.2 tree was done by Frank Mehnert [FM]. Later revisions were done by me [AL].

The initial work on the Linux 2.4 port was done by Volkmar Uhlig [VU] to have a multi processor capable testing application for an SMP $\mu$-kernel. After the first porting efforts the main work was handed over to me. While this work was developed, Volkmar Uhlig contributed the VX2 adoption to L$^4$Linux so that L$^4$Linux also runs on VX2 $\mu$-kernels.

# 3 Design

This section covers the goals of my work and how they are accomplished. It will go into detail on the directory structure layout, the introduced L$^4$Linux-internal abstraction layer and some other issues.

## 3.1 Internal Abstraction Layer

In contrast to previous L$^4$Linux versions, L$^4$Linux-2.4 is targeted to at least four APIs as well as multiple different hardware platforms[1]. Table 1 on page 10 contains a list of different $\mu$-kernel currently available. To be prepared for the upcoming porting efforts, the L$^4$Linux source code tree layout needs to be put in a proper shape so that new ports can easily plug in the code they need to modify and reuse as much as possible of the generic code without scattering API specific code into generic code and defacing the code with `#ifdef` statements.

The currently used L$^4$Linux-2.2 was originally targeted at V2 L4-$\mu$-kernels only and thus $\mu$-kernel specific calls were directly put in the code. Now, if L$^4$Linux should run on multiple different L4 APIs and multiple hardware platforms those specific calls have to vanish from the generic code. The code has to be split into a generic part which handles functionality every version of L$^4$Linux needs to have. $\mu$-kernel specific parts have to be extracted from the old code and put into a separate place, ideally a library.

This library then needs to provide an API which allows to implement the $\mu$-kernel dependent code in a way it fits for every $\mu$-kernel and is generic enough for the other parts of the code.

Parts of L$^4$Linux, which can obviously be put into a separate library, are task and thread management. In this case the API can be designed in the spirit of the threadlib and the task server from L4Env. Of course, the specific parts of the library can also implement additional functionality needed by L$^4$Linux if that is useful. Ideally, in the L4Env case, the library should only be a wrapper to the appropriate L4Env functionality. That of course means that the L$^4$Linux internal library reimplements some parts from L4Env for other APIs.

Furthermore, the library will be written in a way that it could be used across different L$^4$Linux version like L$^4$Linux-2.2 or future L$^4$Linux versions. It will use generic Linux functionality like `kmalloc` and will depend on L4 as well as L4Env. It is not planned to make the library dependent on the L4 specific part of L$^4$Linux.

[1] Currently only the X86 platform is used but others are planned.

For the documentation of the latest version of the library please have a look at [Lac02].

The following sections will explain the task and thread parts of the library in more detail.

### 3.1.1 Tasks

The handling of L4 tasks is quite different in V2/VX0, VX2 and L4Env so that it makes sense to split this code and put it into the library. The different characteristics describe as follows:

**V2/VX0** In a V2 or VX0 environment tasks are either created through RMGR or directly using the appropriate L4 system call.

**VX2** Using a VX2 $\mu$-kernel the tasks are created directly.

**L4Env** All tasks in the system are handled by a task server.

The library needs to provide the following functionality for L$^4$Linux regarding tasks:

- Allocate a task number.
- Create a task with a previously allocated task number.
- Delete a task.
- Free a task number of an idle task.
- Setup (two) threads for each user task.
- Keep track of used/free tasks.

For V2, VX0 and VX2 a simple bit vector is used to keep track of free and used tasks. In the V2 and VX0 case the vector is initialized by querying RMGR for used tasks, reserving free tasks for later use and storing the availability status in the bit vector. Theoretically, the vector would not be needed as RMGR could be asked every time a task needs to be created. This had the additional benefit that no tasks need to be reserved in the L$^4$Linux startup and thus would be available to other L4 applications as well. Unfortunately the communication with RMGR places an additional overhead on every task creation and deletion so that the solution with the bit vector promises better performance. When using L4Env we cannot maintain a vector for the tasks, as we have to communicate with the L4 task server for task handling. This way we can run other L4Env applications besides L$^4$Linux (such as other L$^4$Linux instances) without restricting every L$^4$Linux instance to a too small window of possible task IDs.

### 3.1.2 Threads

The thread handling is another part of L$^4$Linux which can be factored out into the library as it is quite independent of other L$^4$Linux parts and dependent on the underlying $\mu$-kernel or system. Only kernel internal threads (e. g. interrupt threads) are handled by the library. The different ways of handling threads describe as follows:

**V2/VX0** Threads are created and destroyed directly using the `l4_thread_ex_regs` L4 system call. It can be configured whether thread priorities are set directly or with the help of RMGR.

**VX2** As there is no resource manager in the VX2 environment, all threads handling is done directly via the appropriate $\mu$-kernel system calls.

**L4Env** In a L4Env system threads are handled through the threadlib, no direct communication with the $\mu$-kernel itself is needed.

Before the library was used within L$^4$Linux, every thread creation in L$^4$Linux hand coded, including the allocation of memory for the stack of every thread. As the scheme of the thread creation was almost always the same, the stack handling has also been included in the library. Doing a uniform stack handling implies that every stack has the same size which may waste memory in certain cases. As the stack needs to be available from the beginning, it is located in the BSS segment of the L$^4$Linux server itself.

To summarize, the library needs to provide the following functionality:

- Start a thread with a given thread ID or with an unspecified one, in which case the library has to find a free one.
- Destroy a thread.
- Set and get the priority of a given thread.
- Compare two thread IDs for equality.

The stack handling is quite similar among the different implementations, so this will be implemented in a generic part and used from the specific implementations.

## 3.2 Directory structure

A rather minor issue but also an important one for the further development of L$^4$Linux is the directory structure of the source tree. Separation of generic and special code should also be reflected in the directory structure. This becomes even more important when ports to other hardware platforms are added to the tree. The tree should be prepared for such code integrations.

Additionally the L$^4$Linux port should follow the common rules for Linux ports so that different and common code parts between the L$^4$Linux port and the port from the L$^4$Linux port originally derived can easily be found. It is more easy to track changes in the original linux tree in this case as well.

### 3.2.1 Implementation Files

For the above mentioned reasons it makes sense to have `'kernel'`, `'lib'` and `'mm'` directories in the `'arch/l4'` directory. Those directories contain the code which implements the functions every port has to supply for the rest of the kernel. For an older version of this API refer to [Hoh96a], an up to date version is currently not available but may be created in the future.

**Library Implementation** The files implementing the library go to `'arch/l4/l4lxlib'`. This directory then contains subdirectories for implementations the library contains. They are named after the $\mu$-kernel API they implement, so that usually means `V2`, `VX0`, `VX2` and `l4env` are used. Each of these implementation directories then contains the code which implements the library API. It is advised that even the main file names are the same so that one can easily navigate through the different implementations.

To summarize, the layout of the `'arch/l4'` directory is shown in Figure 1 on the next page.

`boot,emulib,kernel,lib,mm` contain the functionality every port has to provide for the kernel to work. As the code in the `'emulib'` directory is API specific it is likely that the code in this directory will be moved to the appropriate library directory and the directory is removed.

`arch-<ARCH>` Those directories contain platform specific code, for example assembler or special initialization code.

`l4lxlib` This directory contains a subdirectory for every API the L$^4$Linux internal library supplies code.

### 3.2.2 Header Files

The layout of the `'include/asm-l4'` directory is shown in Figure 2 on page 23.

| | | |
|---|---|---|
| arch/l4 | boot | |
| ⋮ | emulib | |
| | kernel | |
| | lib | |
| | mm | |
| | ⋮ | |
| | arch-x86 | |
| | arch-ia64 | |
| | arch-ppc | |
| | ⋮ | |
| | l4lxlib | V2 |
| | ... | VX0 |
| | | VX2 |
| | | l4env |

Figure 1: Layout of arch/l4

**include/asm-l4** This directory contains auto generated header files which include the appropriate header files with the same file name from 'include/asm-l4/ arch-<arch>'. If this file does not exists, the header file from 'include/ asm-<arch>' is included. This schema ensures that header files which can be taken unmodified from the appropriate 'asm-<arch>' directory need not to be copied into the 'asm-l4' directory, they are "copied" at build time.

**l4linux** The 'l4linux' directory contains generic header files which are not special to any architecture or API version.

**api-generic** The 'api-generic' directory could contain code which is not generic to all APIs but where code duplication across similar APIs can be avoided (e.g. the V2 and VX0 API are quite similar). Architecture-dependent code should be placed in appropriate 'arch-<ARCH>' directories. The file names in this directory should be considered as well. They should be prefixed with an API specific string, for example 'v2vx0_'. If files with the same prefix get more they should be moved to a directory named as the prefix. Then the prefix of the files must be removed.

**api-<API>** The 'api-<API>' directories contain API specific code and when needed subdirectories for specific architecture dependant code (e.g. C-Bindings).

**api-l4env** L4Env is in fact another API but due to the design of it, no archi-

```
include/asm-l4/
⋮         l4linux
          api-generic
          api-V2
          api-VX0
          api-VX2
                    arch-x86
                    arch-ia64
                    arch-arm
                    ...
          api-l4env
          ...
          arch-x86
          arch-ia64
          arch-ppc
          ...
          l4lxapi
```

Figure 2: Layout of include/asm-l4

tecture dependant subdirectories are necessary. Ideally, this directory only contains header files which wrap the appropriate ones in the L4 directory.

**arch-<ARCH>** The 'arch-<ARCH>' directories in the 'asm-l4' directory contain architecture dependent code which was modified from the appropriate version in 'include/arch-<ARCH>'. They are included from the auto generated header files under point 1.

**l4lxapi** The 'l4lxapi' directory contains the header files for the L$^4$Linux internal library and thus the definition of the API of the library.

### 3.3 Using L4Env

Up to now L$^4$Linux runs directly on the $\mu$-kernel which means that it must be aware all the different APIs those $\mu$-kernels provide. L4Env provides a consistent view to the underlying system an L4Env application runs on. Furthermore it already implements often used functionality which relieves programmers from writing often needed code again and again. Other parts manage the resources of a system so that they cannot be used in parallel by different programs, not knowing anything about each other.

The goal is to add a port to L$^4$Linux which does not run on a specific L4

$\mu$-kernel but on L4Env. This has several benefits in comparison to the traditional L$^4$Linux approach:

- Through the usage of `dm_phys` it is possible to use specially prepared memory like cache colored memory or even paged one. The way how `dm_phys` or any other dataspace manager works is completely transparent to L$^4$Linux so that the type of memory used for L$^4$Linux can be changed without changing L$^4$Linux itself. One just replaces the used dataspace manager with another one.
- Run multiple L$^4$Linux instances in parallel without the need to do special hacks (e. g., link every instance to another address). This can be useful for security architectures, for example run special applications which need superuser privileges in their own Linux environment. Another use could be the migration of user space applications to another L$^4$Linux instance to exchange the currently running L$^4$Linux with a newer version. These migration techniques are still the subject of current research. See [SMi02] for the project page.
- Be independent of the underlying $\mu$-kernel. Ideally, only L4Env needs to be ported to any new $\mu$-kernel architecture and L$^4$Linux would run unmodified or with very little effort on this new kernel. To accomplish this goal the internal library needs proper abstractions and all used L4Env parts need to be examined if they can be used on all $\mu$-kernel APIs.
- Use the existing infrastructure already provided by L4Env.
- Proof of concept: See if the API and functionality of L4Env is sufficient to use it for a big application such as L$^4$Linux.
- Does L4Env provide the full functionality of the underlying $\mu$-kernels?

In addition to the specific parts addressed shortly, the L4Env support in L$^4$Linux needs another entry code as the traditional L$^4$Linux. L4Env applications are expected to have a `main(int, char **)` function with the usual two parameters for the arguments which a traditional L$^4$Linux does not have. Fortunately this alternative entry code is rather small.

### 3.3.1 `dm_phys`

Originally L$^4$Linux runs directly on the memory provided by Sigma0 and maps it one to one to its virtual address space. The information which section of memory to take for L$^4$Linux is taken from the BIOS in the same way as a native Linux does. When using `dm_phys` (see 2.6.3 on page 15) we simply allocate a region of memory and give it to L$^4$Linux in the same way as the BIOS memory. As a result, L$^4$Linux believes it has its "physical" memory mapped one to one to its virtual memory although in reality this memory comes from wherever `dm_phys` or any other dataspace manager takes it. This also means that L$^4$Linux still gets a fixed amount of memory on startup of the kernel.

### 3.3.2 Task server

Each L$^4$Linux user task is encapsulated in a separate address space which has to be created, managed and destroyed. In a L4Env system this is done by a task server. It offers an interface for task management. As a result L$^4$Linux will create and destroy tasks through the task server. The usage of the task server will be implemented in the internal L$^4$Linux library.

### 3.3.3 Threads

In the L$^4$Linux server a lot of threads are used for various tasks, for example for every interrupt in use an interrupt thread is assigned. All these threads need space for their stacks and need to be set up. As mentioned in Section 3.1.2 on page 20 the internal library implements this functionality for the various $\mu$-kernels and for L4Env. In the L4Env case the library implementation is rather a wrapper for the functionality provided by the threadlib as it offers a nearly identical interface.

### 3.3.4 L4IO Server

The I/O server handles all device memory for memory mapped I/O as well as all interrupts. Certain device drivers need memory mapped I/O to communication with their hardware. Without L4Env L$^4$Linux makes this memory available by adding page table entries to the physical memory itself. In an L4Env system the I/O server is used for this purpose.

### 3.3.5 DROPS console/DOpE support

L$^4$Linux with or without L4Env can either run with the normal textual or serial output or display its output on a console of the DROPS console system. To output something L$^4$Linux has to communicate with the console server. This is done with a driver which is plugged into L$^4$Linux either statically or as a Linux module. Through this code a virtual console is opened and the console output of L$^4$Linux is redirected to the console server. The code of this module was originally developed for L$^4$Linux-2.2 by Frank Mehnert [FM] but with a slight adaptation it can also be used for L$^4$Linux-2.4. This adaptation mainly consist of some code changes for the L$^4$Linux internal library and some modifications to the Linux console code since there were some changes between Linux 2.2 and 2.4. Furthermore, to support DOpE, some further changes to the Linux module and to DOpE itself had to be done.

With this DROPS console driver module and a special driver module for the XFree86 X server it is also possible to run the X Windows System in a console window.

# 4 Implementation

In this section I will explain some aspects of the implementation work I did. To reiterate, L$^4$Linux-2.4 was not initially ported by me but when I adopted the code lots of adaption, bug fixing and clean ups needed to be done to get it in a usable shape. Some of the fields I worked on were:

- Make the V2 $\mu$-kernel API part work, as the initial work was done using a VX0 $\mu$-kernel.
- Initially sending signals to user space applications did not work (reliable) and needed fixing.
- The interrupt handling code was modified to resolve some issues like auto probing.
- The kernel was brought up to date with the main line kernel tree multiple times.
- Various other small bits and pieces which are too small to be mentioned here.

Besides the actual work on L$^4$Linux itself, some work had to be done on various DROPS components as well. To use the L4 IO server in L$^4$Linux, the DROPS console also has to use this server as it needs to access the video memory and the I/O server allocates this resource beforehand. As a consequence L4IO support for the console system was added.

Another problem was that initially the L4IO server did not map I/O memory correctly so that certain graphic or network interface cards did not work. I removed this deficiency as well.

Solely for documentation purposes a console snapshot program for L$^4$Linux was written which takes screenshots of DROPS consoles.

## 4.1 Internal Abstraction Layer

The internal abstraction layer was mostly implemented as described in Section 3.1 on page 18. Up to now I implemented the task and thread functionality which is being used within L$^4$Linux and the currently used external modules. Common code which could be shared among different implementations was put into appropriate `generic` directories so that code duplication could be mostly avoided.

The next subsections will show some details on task and threads and will describe some problems with `L4Env` or the `L4Env`-API respectively.

### 4.1.1 Tasks

In a non-L4Env system, L$^4$Linux allocates a predefined number of tasks, usually all tasks available in the system. These tasks are then only usable in L$^4$Linux.

This approach cannot be used with L4Env because it must be possible to run other applications besides L$^4$Linux. Those other applications, which can even be other L$^4$Linux instances, need to be able to create tasks as well. It would be possible that every application which needs to create tasks gets a fixed amount task numbers but obviously this approach is really inflexible and needs an explicit configuration to be adjusted to the local setup. Consequently, a central entity needs to handle the tasks for all applications in the system. Two possible solutions were considered for this.

In a first implementation, RMGR was used as a task server. This was quite uncomplicated except that the `RMGR_TASK_FREE` function of RMGR which had to be implemented in RMGR itself first. This function should have been available looking at the interface of RMGR but was probably not implemented because it was not needed up to now.

The RMGR approach was finally discarded as it is planned to discontinue the use of RMGR, though slowly. Instead, applications running with L4Env should use the L4 task server for their task handling. To make that work, the L4 task server needed some modifications to support the allocation and deallocation of task numbers independently of the actual creation and deletion. This is for example needed in cases where the creation of a task fails and a previously allocated task number needs to be freed again or where the task number needs to be known before the task is created. Knowing the task ID before the creation of the task is needed to set up the environment of the task prior it is started.

Furthermore, the task server of L4Env exhibits a disadvantageous format of its abstract task type. Instead of being independent from the various L4 APIs, the `l4_ts_taskid_t` type is a normal V2 `l4_threadid_t` type in the current implementation. Consequently, the client needs to know about that to work with this type of variables. In the future, a more generic type has to be introduced, mainly also looking at the VX2 interface.

### 4.1.2 Threads

Besides the actual thread management like thread creation or deletion, a stack handling mechanism had to be implemented. Another problem was to have a possibility to transfer some data to a newly created thread. This was solved by codifying the layout of the thread function as follows:

```
void thread_func(void *data);
```

The `data` pointer points to a piece of memory where the parent thread can put data and the child thread can receive it. To avoid races and to guarantee that this memory is always available it is stored on the stack. As the stack is usually 8 KByte in size this implies that the size of the memory pointed to by the `data` pointer is below that. This memory and the normal stack usage have to fit in the whole stack memory.

**Getting Thread IDs** There is another optimization which was considered for V2 and VX0 environments. The `l4_myself()` system call is called away relatively often throughout the kernel, so it makes sense to optimize it away. This is done by putting the thread ID of each thread on the top of the stack when creating it. Getting the thread ID is then a simple operation, one just gets the current stack pointer and delivers the thread ID from the top of the stack. No system call is needed anymore for this task. L4Env and VX2 systems already implement a fast way to access the thread ID so that this optimization is not used in these cases.

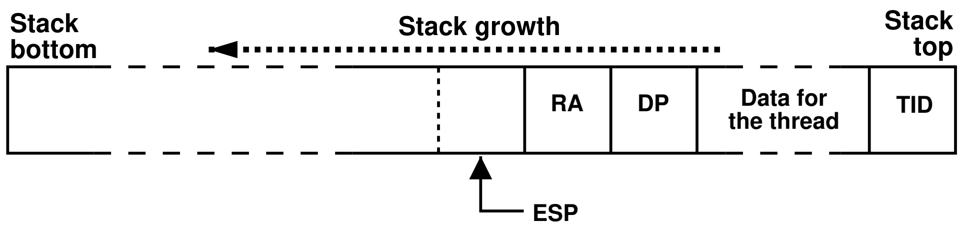
Created threads must not rely on the layout of the stack, which means that in future versions of the library the data memory may be in a completely different place.

**Stack Allocation** The stack memory for the threads is generally handled by the L$^4$Linux itself, except for the L4Env case where the threadlib provides such functionality. In L$^4$Linux, some threads are created very early in the boot process of the kernel which makes it necessary to have the stack memory available before any kernel internal memory handling is set up. Therefore, all stacks are placed in the BSS memory where the amount of stacks and their size can only be configured at compile time.

Through this approach, the stack memory is easily available but a lot of memory is wasted as well. Theoretically, L$^4$Linux may use all interrupts of a system it runs on and so needs a stack for every interrupt thread. There can be up to 16 interrupts in the system (with the use of the IO-APIC of modern systems even more) so L$^4$Linux needs to be prepared with enough stacks at compile time. But usually, only very few interrupt threads are actually used and so lots of memory is wasted.

To prevent this wastage, the thread allocation mechanism will be divided into two parts in a future version of L$^4$Linux. The first part uses BSS allocated stacks, the second part will supersede the first part when the kernel memory management is set up and use the `kmalloc` function for memory allocation. Memory allocated by `kmalloc` is continuous and fulfills the same requirements as the BSS memory. When this technique is used, stack memory is usually allocated through `kmalloc` and only very few stacks need to be in the BSS memory because they are needed before the memory management was set up.

To illustrate, the stack layout for non-L4Env system is displayed in Figure 3:




```
ESP ... Stack pointer for the new thread.
RA  ... Fake return address.
DP  ... Pointer to the data memory (function parameter).
TID ... Thread ID of the new thread.
```

Figure 3: Stack layout of a non-L4Env thread

**Thread Data** Another minor problem occurred with the threadlib in conjunction with the data pointer which is given to thread functions [Reu02b]. The threadlib provides only the data pointer and the thread creator, the thread itself is expected to appropriately handle the memory that the data pointer points to. In the L$^4$Linux mechanism, where the data is being put onto the stack, the creator of the thread can use its own memory right after it called the thread creation function as the actual data was copied to a safe place. Likewise, the created thread can access this memory every time as it is in its own memory.

When using the threadlib, the thread creator needs to allocate memory for the data it wants to give to the new thread and this memory may not be freed until the created thread is completed or the creator is informed by the created thread that it is safe to free this memory or use it for other things. The threadlib provides a one time call back mechanism which can be used for this purpose but is generally not needed in L$^4$Linux. Instead, in the L4Env case, the data will be copied to a separate memory when the thread is created. For this purpose the `kmalloc` stack mechanism will be used which is described before. Additionally, when using `kmalloc`, only the needed size of memory will be allocated and not memory of the whole stack size as in the BSS case.

**Dynamic Thread IDs** Additionally it is planned to remove support for statically assigned thread IDs. Originally, $L^4$Linux only had static thread IDs because dynamically chosen ones were not needed at that point. Later, the dynamic approach was implemented and used side by side with the static one. Using both variations needs preallocation of thread IDs and poses other problems in conjunction with the threadlib where free IDs are hardly known beforehand. Therefore, I will convert $L^4$Linux to a dynamic only thread ID scheme in the future as statically allocated thread IDs have no real need. The one benefit of them, the static relation of the a thread with a fixed number which helps debugging, will be replaced by a mechanism which maps thread IDs to names. With the help of the names one can find a certain thread IDs quite easily.

One other minor problem is that although the threadlib provides "sleep" functions it does not provide one to suspend a thread or to sleep forever. Such a function can be easily added to the threadlib.

## 4.2 Directory Structure

Currently the directory structure looks like designed in Section 3.2 on page 20 but not everything was implemented and some modifications were made.

Firstly, architecture dependent code was not extracted from the generic code and put into a separate architecture directory. It has yet to be shown how multiple architectures work out in the L4 directory of $L^4$Linux. These decisions should be made when an actual port to another architecture is planned or in progress.

Secondly, as the V2 and VX0 API are nearly equal they were put into the same directory. In this case the contamination with `#ifdef` statements is minor compared to the code duplication otherwise. Furthermore "generic" directories were introduced to host code which can be used by multiple implementations of the library. This helps to avoid code duplication and improves maintainability of the library. The generic part of the library is documented in the same way as the other parts, the online version of the documentation can be found at [Lac02].

## 4.3 Integration of L4Env support

The port of $L^4$Linux to L4Env was accomplished by taking the V2 part of $L^4$Linux as a source and modifying the code step by step by replacing V2 code with the code for L4Env. This method makes it possible to have a working kernel nearly all the time and helps developing, testing and debugging a lot. In the current state, the L4Env port of $L^4$Linux only runs on V2 and VX0

$\mu$-kernels as the generic code has not been modified completely. As L4Env is only available for V2 and VX0 $\mu$-kernels so far, the complete independence is not that important right now. Nevertheless it should be the aim to make the L4Env port independent of the underlying $\mu$-kernel. Furthermore it should be mentioned that the L4Env port should run on a VX0 $\mu$-kernel but this was not tested as all the needed L4Env modules need to be recompiled for VX0 and there was not enough time for this up to now.

### 4.3.1 dm_phys

As already stated in Section 3.3.1 on page 24, dm_phys memory replaces the traditional physical memory in L$^4$Linux. With this replacement there is one difference with respect to drivers. The physical memory in L$^4$Linux used to be mapped one to one to the virtual kernel memory and drivers could simply access their hardware without any translation. Now, with a dataspace manager like dm_phys, the relation between physical and virtual memory is not one to one anymore so that a physical address needs to be calculated from the virtual one. dm_phys offers a function to get the offset of a virtual to a physical address. As the memory from the dataspace manager may be noncontinuous this offset is not a simple value but a table to continuous memory regions with their own offsets. This translation was implemented by modifying the virt_to_phys and phys_to_virt functions of Linux.

Using non contiguous physical memory places a problem with the DMA usage within Linux. Theoretically it may happen that a driver within Linux requests a DMA memory region which is not fully covered by physical memory as the dataspace manager may split it into several parts. L$^4$Linux does not know about that and assumes its memory is contiguous. There are several ways to handle the situation. The first and most convenient way is to only request contiguous memory from dm_phys so that the physical memory is not split into parts. This has the drawback that the system may not have the requested memory as a contiguous chunk so that L$^4$Linux cannot run although the needed memory would be available in the system. Another way could be to split the memory from dm_phys in the same way the physical memory is so that L$^4$Linux itself knows about it. This solution requires some more changes and could be implemented at a later point. The last possible solution would be that L$^4$Linux does not need to use DMA capable memory at all. Using DMA memory within secure systems is questionable anyway since malfunctioning or malicious hardware may write anywhere into the memory. Furthermore it is envisioned within Drops that L$^4$Linux does not need to access any hardware directly but will use services offered by L4 servers which access the hardware. Unfortunately, developing drivers is a time consuming and error prone job and so not many drivers for L4 systems exist up to now. Nevertheless more and more drivers for the Drops project are developed which may change the situation in the future, at least for common hardware. Finally, the first solution was taken as it is the easiest for now.

### 4.3.2 L4IO Server

The L4IO server provides access to device I/O memory as well as hardware interrupts [Hel02]. Currently, L$^4$Linux only makes usage of the device I/O memory features of the L4IO server. This was done be modifying the `ioremap` and `iounmap` functions to call appropriate functions in the L4IO server. The interrupt features in the server need to be disabled so that L$^4$Linux can use them itself. Support for L4IO hardware interrupts is planed for a later stage.

## 4.4 Tracing

Another currently ongoing project at the Operating Systems Group is the tracing of the Fiasco $\mu$-kernel as well as the user space applications running on it. Regarding the $\mu$-kernel itself events like context switches and interrupt are interesting. User level programs can focus on function call traces, system calls, IPC as well as communication relationships.

L$^4$Linux was chosen as a test object for the tracing project since it is quite big and complex compared to other applications and was likely to trigger bugs in the tracing tools.

The most interesting part of these tests were the function call traces from L$^4$Linux. To get the traces every function of the Linux kernel needs to be instrumented with special enter and exit function. The GNU C Compiler provides the "`-finstrument-functions`" option for this purpose. The code for these functions is provided by the tracing project in form of a library which is linked to the application. The only code which needs to be written especially for L$^4$Linux is the provision of memory for the trace buffers. This is a bit complicated as the memory is needed very early in the boot process and thus kernel internal memory functions cannot be used. Furthermore, all functions which are used within the tracing library must not be instrumented by the compiler. Using kernel internal memory functions would require that all functions called directly or indirectly need a tag that they are not instrumented. Additionally, the memory needs to be aligned in a certain way. The current solution adds some code to the memory initialization of Linux and takes away the needed buffers. Every thread within L$^4$Linux needs its own trace buffer. Another place for the trace buffers could be the BSS memory but this was disregarded because the buffers can be megabytes big.

The next problem occurs when compiling the kernel. `extern inline` functions do not have code for the pure function and thus a pointer to such a function is not resolvable when linking. But this is needed for the enter and exit functions. To solve this issue, every `extern inline` function needs to be converted to a `static inline` function. This conversion is done by a script on a copy of the L$^4$Linux source. The script originated from the "Linux Trace Toolkit Project" [LTT02] which aims to trace the Linux kernel as well but in a different way.

When running a tracing enabled L$^4$Linux, the gathered data is transmitted to another host via the log server. Currently it has to be minded that L$^4$Linux is not allowed to use the same network card as the log server as they would interfere with each other. This situation will hopefully be resolved in the future. When the tracing data is available on the other host, it has to be converted by several tools until the result can be used as an input file for the VAMPIR tool [VAM02]. VAMPIR is used to visualize the function call traces as well as other data which can be retrieved from the gathered data.

Finally it can be said that the tracing tools still need some enhancements as an instrumented L$^4$Linux produces a huge amount of trace data in a short time. When the data is displayed within VAMPIR it is quite hard to navigate through it and find interesting parts. The tools need some mechanism to filter out most of the information to concentrate on the essential parts. Furthermore, VAMPIR showed some instabilities with big input files which further hinder the work with the trace data.

# 5 Performance Comparisons

This chapter will present some performance related figures comparing different versions of L4Linux. These figures will only give a rough idea, as not enough time was available for extended testing. For example, only Fiasco was used, other $\mu$-kernel s could also be evaluated and compared.

## 5.1 Scenario

The benchmarking scenario looks as displayed in figure 4.

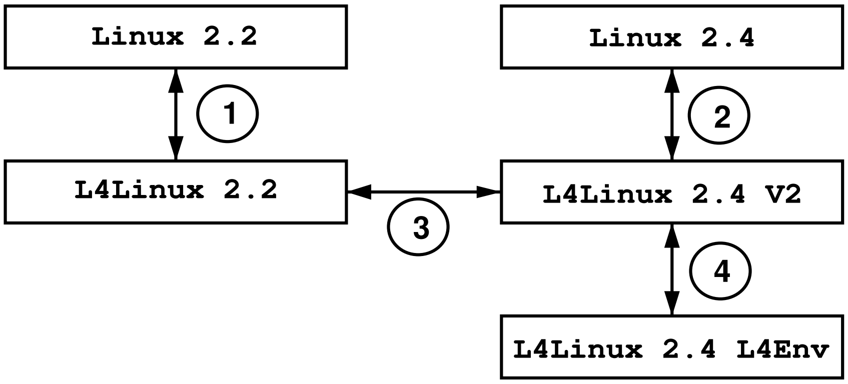



Figure 4: Benchmarking scenario.

1. Compare Linux 2.2 with L4Linux-2.2 to have results to compare L4Linux-2.4 against.
2. Compare Linux 2.4 with L4Linux-2.4 compiled for V2. These results can be compared with the results from the first benchmark.
3. Compare the results for L4Linux from benchmarks one and two against each other. This will show how much better or worse L4Linux-2.4 is compared to L4Linux-2.2.
4. Additionally benchmark L4Linux-2.4 for `L4Env` (under V2) to see what the additional layer `L4Env` costs for the performance of L4Linux.

## 5.2 Benchmark

The benchmark consists of a small program measuring CPU cycles for certain tasks. One is a system call, `getpid()`, the other is a `fork()` which creates a child and thus a new task. The most interesting part of the comparison is the difference between the V2 and the `L4Env` variant of L$^4$Linux-2.4. The values were measured on a Pentium Pro processor at 200 MHz and are displayed in Table 2. While measuring the system was idle otherwise and the figures are the average of 5,000,000 calls of `getpid()` and 5,000 calls of the `fork()` code block. All Linux versions have been compiled with the same version of the build tools (C compiler, binutils). They also have been configured in the same way as much as possible. Furthermore, no special optimization were done to any component although that would be possible, for example the trampoline code for system calls could be avoided by modifying the C library [BHWH97].

| Linux version | `getpid()` | `fork()` |
|---|---:|---:|
| Linux 2.2.22 | 370 | 91,145 |
| Linux 2.4.19 | 377 | 102,123 |
| L$^4$Linux 2.2.22 | 2,049 | 994,508 |
| L$^4$Linux 2.4.19 V2 | 2,078 | 1,022,591 |
| L$^4$Linux 2.4.19 `L4Env` | 2,133 | 1,084,195 |

Table 2: Benchmarking results showing CPU cycles.

### 5.3 Results

Comparing plain Linux versions shows that Linux 2.4 is a bit slower than Linux 2.2, both for the plain system call as well as for the `fork` code. These observations have to be considered when comparing the different L$^4$Linux versions.

Linux 2.2.22 and L$^4$Linux 2.2.22 were measured to get a basis for L$^4$Linux-2.4 and to see the tradeoff in an existing and stable version of L$^4$Linux. Looking at the `getpid()` case, one can see that L$^4$Linux-2.2 and L$^4$Linux-2.4 V2 only differ in a few cycles but the native Linux version also differ a bit, so that may be a correlated behavior. The same argumentation may apply to the `fork()` case where the native versions differ by about 11,000 cycles and the L$^4$Linux versions by 28,000. The most interesting part of the measurements are the differences of the V2 and the `L4Env` version of L$^4$Linux-2.4. In both cases the `L4Env` version is a bit slower which is mainly the tradeoff introduced by the additional `L4Env` components. The slowdown of the `getpid()` case cannot be explained reasonably up to now as no `L4Env` component directly influences this path, the `fork()` slowdown is because of the task server and other introduced parts. Overall, these figures need more investigation to find out why they differ and how to lower them. Nevertheless, the differences between L$^4$Linux-2.4 V2 and `L4Env` are in the expected range so that the `L4Env` approach of L$^4$Linux can be seen as feasible.

## 5.4 Future Benchmarking

As the measurements in the previous sections were only small, other benchmarks need to be done in the future. These benchmarks can cover several areas:

- Use other $\mu$-kernels like Hazelnut [Haz02] or Pistachio [Pis02].
- Use different processor models like Pentium 3 and Pentium 4 as the Pentium 4 implements some functions in a different way than previous Pentium models.
- Use a modified C library which directly sends system calls to the $L^4$Linux server instead of using the trampoline code.
- The Fiasco $\mu$-kernel can be optimized in various ways.
- Implement sysenter and -exit support in $L^4$Linux to speed up L4 system calls.

All the data gathered under the different mentioned points can then be compared to each other and conclusions can be drawn.

# 6 Prospective Work

L$^4$Linux is one of the most important parts in DROPS and lots of work still needs to be done. Firstly, the work started with this thesis needs to be completed. That means the L$^4$Linux internal library needs to be extended and the generic code parts need to be made more generic so that $\mu$-kernel dependent code can be fully replaced by generic code. Furthermore, the code still needs some restructuring and clean ups but that usually is an all time goal.

Then there are some more points to consider for the future. It was proposed that the bindings in L$^4$Linux against the different $\mu$-kernel APIs should be generated from an IDL description so that not many different implementations for different $\mu$-kernels need to developed and maintained. Instead, the different bindings would be generated by the appropriate IDL compiler for each $\mu$-kernel API. To make that happen a proper abstraction for the generic code of the L$^4$Linux port code needs to be found. As this is also the goal of the internal library of L$^4$Linux a later integration of yet another port should not be too difficult.

Concerning ports there are several plans and ideas although they are still quite rough. As displayed in table 1 on page 10, the used $\mu$-kernels run on different hardware platforms and L$^4$Linux would be a great application there as well. Possible usage for it could be PDAs and other small devices where valuable and critical applications are used on the one side and programs from the Internet and other untrusted sources on the other. L$^4$Linux could host the latter and the critical applications run completely separated from L$^4$Linux in a safe environment [Här02]. As L$^4$Linux only exists for the IA32 platform currently it would need to be ported to typical PDA platforms such as (Strong)ARM. Other currently considered platforms are PowerPC and IA64.

Fiasco/SMP is the Fiasco $\mu$-kernel for SMP system, currently on the IA32 architecture. To run L$^4$Linux on this $\mu$-kernel some modification to L$^4$Linux needs to be done. These have been done for L$^4$Linux-2.2 and still need to be done for L$^4$Linux-2.4.

Another currently ongoing project at the Operating Systems Group in Dresden is the port of the Fiasco $\mu$-kernel [Fia02] to the user mode in a UNIX environment, called Fiasco/UX. There are ideas to port L$^4$Linux to Fiasco/UX although the usefulness is still questionable, it is probably useful for debugging.

Writing Linux drivers or other "connectivity code" for DROPS components is another topic. Currently, there are some drivers for L4 in development or are even available, among them a network server with NIC drivers, a USB driver, an IDE driver and a SCSI driver. These drivers are L4 servers, run in user space and have the appropriate privileges to access the hardware they control. To access these drivers in L$^4$Linux a Linux driver is needed which does not access the hardware directly but instead communicates with the corresponding L4 server. User space applications can then access these resources in a normal way. But not only pure "hardware" drivers need to be written. The Drops system also has a file system which could be accessed in L$^4$Linux by a Linux filesystem driver which queries the L4 filesystem internally.

Network transparent IPC for L4 systems was the topic of a diploma thesis at the Dresden group. The goal of this work was to extend the IPC mechanism to work across machine boundaries. With the help of this work, L$^4$Linux could be extended to run user applications on other systems instead of the one L$^4$Linux runs itself. This could work in the spirit of Mosix [Mos02].

When the plans of the Linux core developer come to true, Linux 2.6 (or however the version number is) is not too far away. If it is released, it should be considered to upgrade L$^4$Linux to the next Linux version. It may be reasonable to wait with L$^4$Linux ports to other architectures until this happens. Non IA32 architectures in Linux are usually under heavy development, especially considering IA64. The next Linux version may be a better starting point.

After these rather separate topics there are points which target L$^4$Linux itself or Drops components. The ones affecting L$^4$Linux are on my list as well and will hopefully be added or fixed in the near future:

- Sysenter/-exit support for L4 syscalls, speeds up L4 syscalls.
- Drop static thread number allocation and use a pure dynamic approach. Add a debugging aid to find thread numbers by names.
- Use a two stage approach for thread stack allocation, first stage through BSS, second one through `kmalloc` (also see 4.1.2 on page 28).
- Remove the "Ping-Pong-Task", as all the currently used $\mu$-kernels do not have the limitation like the original version that they can't map pages in the same task.
- When using multiple L$^4$Linux instances it would be beneficial for memory usage if they could use "Copy-on-Write" (COW) memory. This includes the L$^4$Linux server where the code is the same as well as the RAM disk image where nearly the whole image never changes. When using the DROPS RAM disk of 16MByte, which has 15MByte of usually readonly data on it, COW memory would save about 17MByte (RAM disk and L$^4$Linux server) for every additional instance of L$^4$Linux. The COW memory would need to be implemented in a dataspace manager like `dm_phys` or in another dataspace manager.

- Add support for the Omega0 interrupt protocol [LH99] to access interrupts in a hardware and architecture independent manner. The Omega0 protocol is implemented by the L4IO server as well as the Omega0 server. For L$^4$Linux the L4IO will be used as it is already used for I/O memory.
- The oshkosh driver currently in L$^4$Linux-2.2 needs to be ported to L$^4$Linux-2.4. I expect that with this work it will be possible that multiple concurrently running L$^4$Linux instances can have network access. Then it would be possible as well that all L$^4$Linux instances use NFS for their root file system instead of a RAM disk. Process migration [SMi02] over the network would work on a single machine which should help debugging and testing.
- Fully integrate the VX2 port. Currently, the VX2 port has duplicated and modified some code for its efforts. This code duplication needs to be removed and the code merged. This work will be done with the further work on the library.
- Keep up to date with the main line Linux tree and incorporate changes into the L4 tree of L$^4$Linux.

Finally, if L$^4$Linux-2.4 proves to be stable and most of the currently existing bugs are fixed, it could be considered to replace L$^4$Linux-2.2 as the standard L$^4$Linux within Drops.

# 7 Summary

The main goal of this work was to integrate L4Env support into L$^4$Linux-2.4 and to incorporate an abstraction layer to L$^4$Linux to be able to cope with the different configurations more sanely. Besides that smaller issues were addressed as well. Lots of bugs have been fixed and code quality has been improved but there is still some amount of work to do. Additionally the directory structure has been defined and tracing support for the L4 Tracing environment has been added. In the context of this work bugs in some Drops components could be fixed as well as some small features could be added.

As a result, L$^4$Linux-2.4 has progressed a lot lately but there is still a huge amount of work to do.

# A Glossary

**$\mu$-kernel** Microkernel, an operating system only containing the absolutely necessary functionality in kernel mode. Other services are implemented in user mode running in separate tasks.

**API** Application Programming Interface

**BSS** A segment within an address space where variables of a program are stored.

**COW** "Copy On Write", used for initially shared read/write memory, on the first write access, the memory is copied

**CPU** Central Processing Unit

**DMA** Direct Memory Access

**DROPS** Dresden Realtime OPerating System [DRO02]

**Hazelnut** See [Haz02]. A $\mu$-kernel implementing Version X.0 of the L4 specification [VX002].

**IPC** Inter Process Communication

**kernel mode** Code running at the highest privilege level available.

**L4** An interface description to user space of a $\mu$-kernel.

**L4Env** An user level environment for L4 systems consisting of several modules.

**NFS** Network File System

**NIC** Network Interface Card

**Omega0** A portable interface to interrupt hardware for L4 systems. [LH99]

**preemption handler** A mechanism in L4 to schedule in user space.

**Pistachio** See [Pis02]. A $\mu$-kernel implementing Version X.2 of the L4 specification [VX202].

**RMGR** Resource Manager

**Sigma0** The initial pager for L4 systems.

**SMP** Symmetrical Multi Processing

**USB** Universal Serial Bus

**user mode** Code running in a domain of its own, unauthorized access to other applications or data is not possible.

**V2** Version 2 of the L4 API specification. [Lie96]

**VX0** Version X.0 of the L4 API specification. [VX002]

**VX2** Version X.2 of the L4 API specification. [VX202]

# B Bibliography

## References


[ABB+86] 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.

[AL] Adam Lackorzynski. `http://os.inf.tu-dresden.de/~adam/`.

[APJ+01] M. Aron, Y. Park, T. Jaeger, J. Liedtke, K. Elphinstone, and L. Deller. The SawMill Framework for Virtual Memory Diversity. In *6th Australasian Computer Architecture Conference*, Gold Coast, Australia, January 2001.

[BHWH97] Martin Borriss, Michael Hohmuth, Jean Wolter, and Hermann Härtig. Portierung von Linux auf den $\mu$-Kern L4. In *Int. wiss. Kolloquium*, Ilmenau, September 1997.

[DRO02] Dresden Realtime OPeration System. `http://os.inf.tu-dresden.de/drops/`, 2002.

[Fia02] Fiasco $\mu$-kernel, 2002. `http://os.inf.tu-dresden.de/fiasco/`.

[FM] Frank Mehnert. `http://os.inf.tu-dresden.de/~fm3/`.

[GRU02] GRUB, the GRand Unified Boot loader. `http://www.gnu.org/software/grub/`, 2002.

[Här02] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[Haz02] The Hazelnut VX0 $\mu$-kernel. `http://www.l4ka.org/projects/hazelnut/`, 2002.

[Hel02] Christian Helmuth. L4Env Generic I/O Reference Manual. Available from `http://os.inf.tu-dresden.de/~ch12/doc/generic_io/`, 2002.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of $\mu$-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[HM02] Christian Helmuth and Frank Mehnert. DROPS Console System. Available from `http://os.inf.tu-dresden.de/local/project/manuals/drops/con/refman/`, 2002.

[Hoh96a] M. Hohmuth. *Linux Architecture-Specific Kernel Interfaces*. TU Dresden, March 1996. Available from URL: `http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/`.

[Hoh96b] 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/`.

[JW] Jean Wolter. `jean.wolter@inf.tu-dresden.de`.

[L4I02] Different L4 Implementations. `http://os.inf.tu-dresden.de/L4/impl.html`, 2002.

[L4K02] L4KA Homepage. `http://www.l4ka.org/`, 2002.

[Lac02] Adam Lackorzynski. L$^4$Linux Internal Library API Documentation. Available from `http://os.inf.tu-dresden.de/~adam/l4lx/apidoc/html/`, 2002.

[LH99] J. Löser and M. Hohmuth. Omega0 – a portable interface to interrupt hardware for L4 systems. In *Proceedings of the First Workshop on Common Microkernel System Platforms*, Kiawah Island, SC, USA, December 1999.

[LHH97] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

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

[LTT02] "Linux Trace Toolkit" project. `http://www.opersys.com/LTT/`, 2002.

[Meh02] Frank Mehnert. L4 loader Reference Manual. Available from `http://os.inf.tu-dresden.de/~fm3/doc/loader/`, 2002.

[MH] Michael Hohmuth. http://os.inf.tu-dresden.de/~hohmuth/.

[Mos02] Mosix project. http://www.mosix.org/, 2002.

[OS02] Operating Systems Group. http://os.inf.tu-dresden.de/, 2002.

[Pis02] The Pistachio VX2 $\mu$-kernel. http://www.l4ka.org/projects/pistachio/, 2002.

[Reu02a] Lars Reuther. L4 Region Mapper Reference Manual. Available from http://os.inf.tu-dresden.de/~reuther/doc/l4rm/, 2002.

[Reu02b] Lars Reuther. L4 Thread Library Reference Manual. Available from http://os.inf.tu-dresden.de/~reuther/doc/thread/, 2002.

[Reu02c] Lars Reuther. L4Env Physical Memory Dataspace Manager Reference Manual. Available from http://os.inf.tu-dresden.de/~reuther/doc/dm_phys/, 2002.

[SMi02] "Service Migration in Linux Environments" project. http://os.inf.tu-dresden.de/SMiLE/, 2002.

[VAM02] The VAMPIR tool. http://www.pallas.com/e/products/vampir/, 2002.

[VU] Volkmar Uhlig. http://i30www.ira.uka.de/~uhlig/.

[VX002] Version X.0 specification of the L4 API. http://www.l4ka.org/documentation/files/l4-86-x0.pdf, 2002.

[VX202] Version X.2 specification of the L4 API. http://www.l4ka.org/documentation/files/l4-x2.pdf, 2002.