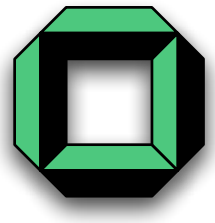


Universität Karlsruhe (TH)

Forschungsuniversität · gegründet 1825

Improving Virtualization on L4

The System Architecture Group
Prof. Dr. Frank Bellosa
Faculty of Computer Science
University of Karlsruhe

by

Thomas Blattmann

Advisor:

Prof. Dr. Frank Bellosa
Joshua LeVasseur

Thesis start: August, 15th 2006
Thesis end: February, 14th 2007

II

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

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, February 14, 2007

Contents

Abstract

A variety of classes of hypervisors run virtual machines. One approach is to use specialized, thin kernels as a base for virtual machine construction. A second approach is to build virtual machines on the higher-level abstractions offered by platform-independent operating system APIs. Virtual machines have also been successfully set up on top of μ-kernels. A μ-kernel is a thin hypervisor with a minimally trusted computing base. Unlike pure hypervisors, however, it is not specialized to one specific purpose. Instead, common μ-kernels are designed to perform well as generic platforms upon which many and various systems can be built. This generality, on the downside, allows pure hypervisors to outperform μ-kernels in their virtualization specific application area.

This work investigates L4, a second generation μ-kernel, with respect to virtual machine construction. It identifies several L4 characteristics that complicate virtual machine construction and cause a performance overhead, respectively. Also, it proposes μ-kernel extensions that are well suited to the usage patterns of a virtual machine but still conform to L4's design philosophy. Finally, the thesis provides a reference implementation that demonstrates the new L4 interface extensions. It reveals that many challenges can be overcome.

Chapter 1

Introduction

Virtualization describes a software abstraction layer that multiplexes a single physical machine interface to create the illusion of many virtual computers, also known as virtual machines. Each one appears to have its own set of hardware, including a processor, memory, and devices. In fact, these items can be real (e.g., if a real network interface is dedicated to a virtual machine for its exclusive use) but they can also be shared among several virtual machines or completely simulated by software. Virtual machines allow you to run several operating systems in isolation on the same physical machine as if they were executing on real hardware.

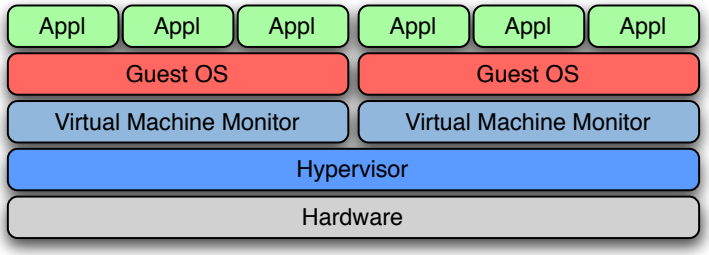


Figure 1.1: Virtual environment abstraction layers

In Figure 1.1 virtualization is shown as a stack of hardware and software layers. The privileged software layer with immediate access to and complete control of the physical hardware is called the hypervisor. It multiplexes physical resources and exports an interface that allows the creation of virtual machines on top of it. Some virtualization projects use specialized thin kernels, which are optimized to execute hypervisor specific tasks. Others construct virtual machines on top of μ-kernels or standard operating systems such as Linux or Microsoft Windows. The latter is commonly referred to as hosted approach.

Another frequently used term related to virtualization is that of a virtual machine monitor, in short VMM. It embraces the entirety of non-privileged software that builds on top of the hypervisor to create virtual machines.

There are several approaches to virtualization based on how complete the hardware platform is implemented. Full virtualization describes a technique that provides a complete simulation of the underlying hardware and allows unmodified operating systems to be run. With Para-Virtualization, the virtual machine's interface is similar but not identical to that of the underlying architecture. It requires operating systems to be ported to comply with that interface. Pre-Virtualization is a novel approach similar to Para-Virtualization. Unlike traditional Para-Virtualization, it highly automates the porting process and avoids dependencies between the guest operating system and a specific hypervisor.

1.1 L4's Problem for Virtualization: Missing Transparency

This work deals with different kinds of challenges the L4 μ-kernel poses to virtual machine construction when using Para- and Pre-Virtualization techniques. It particularly focuses on the transparency of the hypervisor's high-level abstractions on which a VMMs sets up.

Virtualization specific hypervisors and μ-kernels share a great deal of architectural similarities, including reliability, security and isolation. However, L4 was designed to be used as a generic platform upon which many different systems can be built. Its API is architecture neutral and, relating to platform independence, rather comparable to those APIs offered by monolithic multi-programming operating systems. Their abstractions are usually not very transparent, meaning that lots of capabilities that are provided by the underlying hardware are not exported to be usable by software layers that build on top. Other features can only be accessed via indirections.

Platform independent high-level operating system abstractions are simple to use, not very prone to making errors, and are well suited to the common user application usage patterns. Virtual machine monitors, however, export interfaces that either match, or resemble that of the underlying hardware. Obviously, they are particularly sensitive to transparency issues. A highly transparent hypervisor interface matches well to a VMM's usage pattern and allows for efficient virtual machine construction. The more hardware features a VMM can directly access, the less indirection and emulation it requires.

As of yet, there are two projects on virtualization on top of L4, L^4Linux [16] and Joshua LeVasseur's L4Ka Virtual Machine Environment [19]. They share

similar problems with all other hosted virtual machines: Their hypervisor's interface does not efficiently support virtual machine construction as well as a highly transparent, specialized hypervisor does.

1.2 Approach

Peter M. Chen, Samuel T. King and George W. Dunlap [17] managed to remarkably improve the performance of a Linux based virtual machine. Among other things, they introduced new system calls and increased transparency of the Linux API by providing VMMs a way to make use of IA-32 segmentation. This allowed them to safely co-locate operating systems and their applications in the same address space. Motivated by their success, this work identifies and analyzes the weak points of the L4 API with respect to virtualization. Based on the results, it makes proposals on how to make the API more transparent (and thus more suited to virtual machine construction) without interfering with existing μ-kernel concepts.

1.3 Structure of this Thesis

The remainder of this thesis is organized as follows: Chapter 2 covers different virtualization techniques and the concept of abstraction layers. It introduces L4 and points out the challenges that L4's API poses to virtual machine construction based on two related projects. Chapter 3 proposes several API extensions to simplify virtualization on L4 and a new virtual machine design. Chapter 4 describes a reference implementation for both, the extended μ-kernel, and the redesigned virtual machine. Finally, Chapter 6 and 7 compare and analyze the performance results of the different virtual machines as well as the extended μ-kernel.

Chapter 2

Background And Related Work

2.1 Current Virtualization Techniques

The idea of virtual machines is not new but goes back into the mainframe era in the mid 1960s with IBM's VM/370 [13, 8] being the best known example. At that time VMMs provided a compelling way to multiplex the expensive and scarce resources among multiple applications. With the emergence of modern multitasking operating systems in the 1980s and 1990s and the simultaneous drop in hardware costs, however, the value of VMMs eroded somewhat. This development caused hardware designers to draw their attention away from virtualization to support the upcoming developments of the time.

The recent revival of virtual machines unveiled that certain modern architectures have made virtualization complex or even impossible. As popular ones as Intel's IA-32, also called the x86-architecture, make it difficult, inefficient or even impossible for a VMM to export an interface that is identical to real hardware (cp. [27]). In ref. [23], Popek and Goldberg derived a set of sufficient conditions for a computer architecture to efficiently support virtualization. Even though their work goes back to 1974, it is general enough to remain valid on today's architectures and provides standards for the design of virtual machines. In their terminology, a virtual machine must have three essential characteristics. First, any program (which includes of course operating systems) should exhibit a behavior identical with that experienced when running on a real machine directly (equivalence). Second, the VMM must be in complete control of all hardware resources (resource control). Finally, a statistically dominant subset of the virtual processor's instructions must be executed directly by the real processor without VMM intervention (efficiency).

The problem Popek and Goldberg then deal with is to deduce characteristics that an architecture must have in order to create VMMs with the above mentioned traits. They derive requirements for a model of what they call "third generation architectures" (e.g., DEC PDP-10, IBM 360, or Honeywell 6000), which can, however, be easily adapted to modern machines. Their prototype has a processor that can operate in two different privilege modes (user mode and system mode) and requires that a subset of operations is only available in system mode. The processor's instruction set is classified into three different groups:

1. *Privileged* instructions can only be executed in system mode and cause the processor to trap if executed in user mode.
2. *Control sensitive* instructions affect, or potentially affect, the configuration of resources in the system.
3. *Behavior sensitive* instructions are instructions whose result depends on the configuration of system resources.

Popek and Goldberg's analysis finally results in one basic theorem:

> For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

In simple terms, a virtual machine can be constructed if all instructions that could affect the correct functionality of a VMM (i.e., the virtualization sensitive ones) always trap when executed in less privileged mode. Trapping here means that hardware detects a protection violation and passes control to the hypervisor. It thereby allows to enforce the resource control property. Non-privileged instructions are executed natively (efficiency) and the holding of the equivalence property also follows.

The conditions stated by the theorem are sufficient but not necessarily required. Virtual machines have been constructed on non-virtualizable architectures (in the Popek and Goldberg's sense) at the expense of efficiency, increased complexity or architectural equivalence. The following now introduces three modern approaches to virtualization.

2.1.1 Full Virtualization

Full Virtualization makes use of the architectural separation between privileged operating systems and less privileged applications. Provided that all sensitive instructions are privileged, the VMM can easily catch the traps caused

by guest operating systems, simulate those instructions in software and pass control back. This requires no modifications to guest operating system binaries. The guest is totally unaware of the fact that it executes in a virtual machine. However, trapping into the hypervisor on each sensitive instruction is expensive. Further, not all of today's processors are fully virtualizable since certain sensitive instructions do not trap when executed in restricted mode (cp. [27]). Rather they complete or fail silently. Therefore, sophisticated techniques to catch and emulate the execution of virtualization sensitive instructions had to be developed.

VMware's Workstation [25], Microsoft's Virtual Server [5], IBM's virtualization technology [7, 6], and the latest versions of Xen [12] support full virtualization. Xen uses virtualization hardware support that has recently emerged for Intel compatible CPUs [9, 27]. IBM has co-designed cooperating mainframe architecture and system software, which allows hundreds of guests to coexist and to even setup virtual machines recursively without much performance penalty. VMware takes a different approach that does not rely on hardware extensions. Their solution is called binary translation: Where direct execution is not possible, critical parts of the guest operating system are dynamically rewritten for emulation code [1] . It is further required to install several drivers in the host OS. Other virtual machines that use similar translation techniques include Microsoft's Virtual Server and Virtual PC [4].

2.1.2 Para-Virtualization

A different approach is to port the guest operating system to a virtual machine whose interface is similar but not identical to the underlying physical machine. The modified guest then calls the virtual machine's emulation functions to perform low-level operating system tasks, including the execution of virtual sensitive instructions.

With Para-Virtualization, the virtual machine does not need to provide the complexity of entirely simulating a computer architecture and can thus be simplified to be more efficient. Some of the virtualization logic is moved into the modified operating system. Performance is another good reason for this approach. Many expensive context switches between a hypervisor and the guest OS in user mode can be avoided by having parts of the emulation code mapped into the guest's address space. Certain requests can then be processed solely in user mode. Others can be batched for later processing while the hypervisor gets only involved where it is unavoidable.

On the downside, Para-Virtualization requires source code access as well as thorough knowledge of the operating system to be ported and causes high engineering efforts. The guest operating system gets bound to a particular

hypervisor's interface and will no longer execute on bare hardware.

Xen is likely to be the most popular virtualization project to support this technique. Others include virtualization on L4 [19, 16], User Mode Linux [11] or UMLinux [3].

2.1.3 Pre-Virtualization

Pre-Virtualization is an enhancement to Para-Virtualization proposed by Joshua LeVasseur in ref. [19]. It aims to overcome Para-Virtualization's drawbacks without sacrificing the performance advantages it has over Full-Virtualization.

With Pre-Virtualization, modifications made to the guest kernel do not create any strong dependencies between the operating system and a particular hypervisor. The virtualization-aware OS binary is capable of running in any Pre-Virtualization supporting environment. Other than that it can still execute on bare hardware. When the prepared binary is loaded to run in a virtual machine, its neutral interface is automatically adapted to the algorithms offered by the virtual machine monitor.

J. LeVasseur designed and implemented a reference Pre-Virtualization environment. One solution, that is referred to as Compiler Afterburning, is introduced in detail in Section 2.3.2.

2.1.4 Hosted vs. Hypervisor Approach

The previous sections classified virtualization approaches according to how closely the VMM's interface resembles that of the underlying physical machine. Another categorization is based on the platform upon which the virtual machine is constructed. This can be either on physical hardware or on top of the abstractions provided by a host operating system. There are good arguments for either version but there are disadvantages, too.

As operating systems are designed and optimized to execute on bare hardware, the interface a VMM exports should be as close to the hardware as possible. The more it differs or abstracts, the more changes are required to the guest operating system (Para-Virtualization or Pre-Virtualization). A virtual machine that is built right on top of physical hardware can take full advantage of all architectural capabilities. It can pass them up to higher software layers and can provide an interface to guest operating systems whose abstractions not only closely resemble the underlying architecture but also map directly to hardware components. Such a transparent VMM interface allows operating systems with little or no modifications to efficiently use the hardware they are

designed for. The resulting performance is practically equivalent to that of a native system (cp. [12]). On the downside, such kinds of interfaces are highly specialized to virtualization purposes and mostly inappropriate for other applications. If they also export hardware specific features, several ports for the various platforms need to be provided.

Other virtualization projects including User Mode Linux [11] and UMLinux [3] set up on Linux, a general purpose operating system. Just like other operating systems, Linux was not intended to be used as hypervisor in the first place. Its abstractions are platform neutral, hardly transparent, and differ from the hardware on which they build. Guests running on top of a host operating system cannot directly use the underlying hardware mechanisms. Rather, the VMM has to map low-level guest operations to high-level host abstractions in oder to achieve the required functionality.

The main benefit of the hosted approach is due to the simple set of abstractions an operating system offers. As described in [17], such abstractions map well to all parts of the virtual machine: A host process provides a stream of execution comparable to a processor, guest interrupts can be mapped to host signals, virtual devices to host devices and the host's memory mapping and protection provides functionality similar to that of a virtual MMU. All this allows one to implement a VMM as a common user process with little effort. On the other hand, the hosted approach does not achieve the performance results of VMMs that set up on specialized hypervisors. Its high-level host API commonly requires it to modify the guest OS. It further entails additional computing steps and hides certain architecture specific details, which could otherwise be efficiently used by the guest.

2.2 Abstractions and The Concept of Transparency

This section discusses abstraction layers in the design of hierarchically structured systems. It focuses on operating system abstractions, the transparency of abstractions, and the impact of abstraction layers on virtual machine construction.

2.2.1 The Usage of Abstraction Layers

Software abstraction layers, as introduced in Djikstra's THE [10], serve various purposes and can be found in different areas in software engineering. They are commonly set up on low-level, complex, or heterogeneous software and hardware interfaces to export a consistent, easily usable interface. They may also be interposed to hide low-level features from being accessed by higher-level

layers for reasons of security, integrity, and stability.

An operating system API serves all these goals. It exports platform independent, consistent abstractions, which allow the same software to be executed on different architectures. Its abstractions are simple to use and understand. They decrease user level software complexity and help to avoid errors. Further, the API hides many hardware features that must not be accessible to user applications. If they were, user applications could break out of their protection domains or maliciously harm the overall system.

This matter can be explored in more detail by examining Linux's file system related system calls. Linux has an in-kernel software abstraction layer commonly referred to as Virtual Filesystem Switch [2]. It deals with all system calls related to a standard Unix file system and provides a generic high-level interface to several kinds of file systems (such as Linux's Ext3, and ReiserFS, Microsoft's VFAT, and NTFS, or network file systems like NFS). It thereby hides specifics such as how the file system is organized on the storage medium, and whether it is located on a local or remote device. The uniform interface provides a common, simple file abstraction, which makes working with files convenient and consistent to user applications.

The VFS also hides hardware functionality: Its user interface provides no means that would allow applications to directly send commands to the disk, while the kernel is permitted to do so. The kernel can order a disk controller to move the read/write head to a certain position, or to write data to arbitrary disk sectors etc.; user applications cannot. Rather, if a user wants to access a certain file, he must use the corresponding system calls. The kernel then looks up the position of that file on the hard disk and asks the controller to move the disk head, and to return the required data. The main point to highlight here is that functionality, which is offered by lower hardware and software layers, has become inaccessible in higher abstraction layers. Parnas [22] refers to this as a loss of transparency. However, as he describes, such a loss of transparency between different interfaces is mostly to be considered a desirable feature of the abstractions. One of the goals of introducing file related system calls was to prevent undesirable states. If a user application had control of the disk drives it could easily destroy the whole file system. On the other hand, as Parnas further explains, abstractions can also introduce all sorts of inefficiencies when the interface is improperly designed, i.e., if lower layer functionality that could be efficiently and safely used by higher layers is hidden behind abstractions.

Platform independent operating system interfaces such as POSIX [15] or the L4 API [26] are certainly high-level when compared to the low-level hardware interface they build on. They not only provide common, simple to use abstractions but also hide platform specifics that could be safely used by user-level software. Nevertheless, such APIs have proven to be a convenient platform

upon which user applications can be efficiently set up. A redundant increase of transparency would not only complicate their usage but can also downgrade the performance of other workloads that cannot efficiently use the provided extras. A less transparent interface does not imply improper design. One rather has to compare the goals of its designers to how it is eventually used.

2.2.2 The Impact of Abstractions on Virtual Machines

Virtual machines are particularly sensitive to transparency. Highly transparent hypervisor interfaces match well to a VMM's usage patterns, for they allow to take full advantage of the hardware's abilities. The small number of software layers incurs minimal overhead and makes virtualization very efficient. However, pure hypervisors and μ-kernels have been designed with different objectives and motivations in mind and their interfaces differ with respect to transparency. In ref. [21], Liedtke characterizes a μ-kernel as an attempt to "minimize the kernel and to implement whatever possible outside of the kernel" in response to large monolithic kernels. The L4 μ-kernel hides the peculiarities of different platforms behind a simple, common, and hardware independent set of little transparent abstractions. It thereby allows the reuse of software components across a wide variety of architectures. In contrast, work on virtual machines resulted from the need to improve hardware utilization by securely multiplexing the underlying hardware across several operating system instances that execute on a single physical machine. Unlike μ-kernel interfaces, the one a VMM exports per definition closely resembles processor hardware to allow guests with little or no modifications to be run. Pure hypervisor interfaces are thus designed to be highly transparent in order to suit well to the needs of a VMM.

The resulting problem of μ-kernels and other hosted approaches to virtual machine construction are interfaces that are not specifically geared to the needs of VMMs. Instead, they are reduced to the common denominator of features that are offered by all architectures and thus do not reach the same virtualization performance. Regarding the API design, it boils down to a trade-off between simple, hardware independent abstractions that are designed to be used for a wide range of applications, and specialized, transparent interfaces that are well suited to specific applications. Due to the architectural differences, there cannot be a perfect API that combines the best of both worlds.

2.3 Virtualization on L4

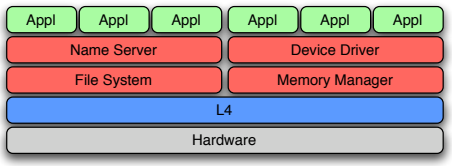
This section introduces the L4 μ-kernel, its user interface as well as two virtualization environments that are built on top of L4. The following section then

analyzes L4's problems with virtualization.

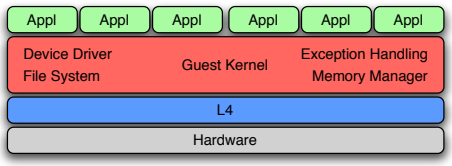
2.3.1 The L4 μ-kernel

The concept of μ-kernels

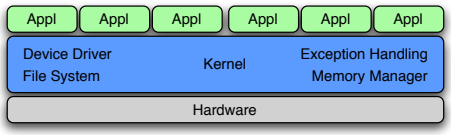
Currently, two different models exist in operating system design. The most widely used is the monolithic kernel approach where most operating system services such as virtual memory management, device drivers and the file system execute in the processor's privileged mode within the same protection domain as the kernel (see Figure 2.1(c)). A μ-kernel based system, in contrast, only keeps whatever is necessary to build a secure system on top in its privileged part. Such services commonly comprise thread and address space management, physical memory management, and interrupt handling. Everything else is implemented as user programs (called servers), which execute as separate tasks at the same privilege level as other user applications. Figure 2.1(a) shows how a μ-kernel based multi-server operating system may look like.



(a) μ-kernel based Multi-Server System



(b) μ-kernel based Monolithic System



(c) Monolithic System

Figure 2.1: Different operating system architectures. Blue layers are privileged while user applications (green) as well as certain OS services in μ-kernel based systems (red) execute in user mode. Services in the same box execute within the same protection domain.

First generation μ-kernels such as the Mach [14] project, or Chorus [24] turned out to be insufficiently flexible and were often criticized for a complex IPC system with too much overhead. L4 [20, 21] is a second generation μ-kernel and results from a re-examination of the entire μ-kernel concept with careful attention to performance and architecture-specific design. The initial Intel i386 kernel was originally designed and implemented by J. Liedtke and has

since then been further optimized and ported to many of today's widely used architectures. The kernel offers only the very basic hardware abstractions threads, address spaces and IPC. Most policy making operating system services including a pager, device drivers and networking are to be implemented as user-level services in order to get the functionality of a complete operating system. Inter-process communication (IPC) is a key characteristic of L4 and has been simplified and highly optimized to match and even outperform the IPC efficiency of monolithic systems.

μ-kernels have several advantages over monolithic kernels. First, they have a fairly small code base. L4/IA-32 has around 17K lines of code while Linux or Microsoft's Windows XP both have several millions. This makes them easy to maintain, verify, and port to new architectures, as well as less likely to contain errors. Second, as most operating system services execute outside the kernel, crashes of single services will not affect the whole system. Also, working on clearly separated system components is considerably easier as code is not entangled with other kernel sources but concentrated in a dedicated server. Finally, microkernels are highly flexible and not restricted to be used as base of a standard operating system. They have been successfully deployed in various application areas besides virtualization, including real time systems and embedded environments.

The major handicap of a μ-kernel based system is its performance. The various servers that implement operating system functionality at user level need to frequently communicate. The kernel has to transfer those messages between different address spaces, which is slower than communication within a single protection domain. The design further entails a good deal more application switches, which can be very expensive on certain architectures.

L4's API in a Nutshell

The following gives a short introduction to the L4 API. It particularly deals with L4 specifics that impact virtual machine construction. For the complete specification refer to the official L4 API reference manual [26].

L4 offers two basic abstractions to user applications, threads and address spaces. Threads are defined as independent flow of execution inside an address space. An address space can be regarded as a protection domain. L4 guarantees that threads running in a certain address space cannot have malicious effects on threads running in others. A single address space can hold one or several concurrently executing threads.

Inter-process communication (IPC) is the key to efficient communication between threads. L4 offers synchronous, message based IPC via its system call *IPC* and *LIPC*. Per-thread message registers describe the message to be sent.

This can either be a limited number of untyped words, a contiguous or noncontiguous sequence of bytes (string item) or a flex page (map or grant item), i.e., a memory page in the sender's address space to be mapped into the receiver's address space.

Per-thread buffer registers are set by the receiving thread to specify what typed messages it is willing to accept and where the received item should be stored at (in case of a string item) or mapped to (in case of a map or grant item). Both, message and buffer registers are virtual registers provided by the kernel. They can map to real processor registers but can also map to memory locations. For the latter, the kernel maps a per-thread User Thread Control Block (UTCB) into each application's address space. It holds the virtual register set as well as thread specific information, including the thread's exception handler, IPC error codes, and the thread's global identifier.

Besides the per-thread UTCB mapping, each application address space contains a kernel mapping called the Kernel Interface Page, or in short the KIP. The KIP contains API and kernel version information, system descriptors (including memory descriptors) and system-call links. The location of both kernel mappings, UTCB and KIP, can be specified upon creation of the thread and address space, respectively.

When L4 starts up it initializes up to three user level address space servers, σ_0, σ_1 and the root server. The latter is the first real user application and is intended to boot the higher-level system. σ_0 initially owns all available physical memory. It contains the first user level pager thread to handle page faults that are caused by the root task. Threads in one of the three initial spaces are special as only they can execute L4's privileged system calls *ThreadControl, SpaceControl, ProcessorControl* and *MemoryControl*. *ThreadControl* and *SpaceControl* allow to create, modify, or delete other L4 threads and address spaces, respectively.

Other than σ_0, newly created address spaces are empty, meaning that their virtual addresses are not yet assigned to any real physical memory. Any read or write operation that involves memory will result in a page fault. However, the kernel itself does not handle page faults. Rather, each L4 user thread other than σ_0 has a pager thread assigned. On page faults, L4 halts the offending thread and sends a page fault IPC (consisting of the faulting address and the faulting instruction pointer) to the attendant pager thread. It is then up to this thread to select one of its memory mappings and to either map or grant it to the faulting thread in response. The kernel intercepts the response, updates the faulting thread's address mappings and finally resumes its execution. Memory management is thus solely controlled by user-level pagers. They by degrees recursively back virtual address spaces with memory mappings.

The page fault protocol maps page faults to common L4 IPC. The same approach is taken with the interrupt and exception protocol. Other than a few special cases, which are handled by L4 internally, user exceptions cause L4 to synthesize an exception message and to send it to the offending thread's exception handler. The received message contains the originator's instruction pointer (IP) and a couple of architecture dependent registers. Exception handlers can modify these registers in reply to exception messages. L4 then resumes the originator with the new register state. The same mechanism is used for hardware interrupt handling. A single L4 thread can register with L4 to receive an Interrupt IPC on hardware interrupts.

The last thing to discuss here is L4's scheduling policy. Unlike paging, exception, or interrupt handling (which is mostly controlled by dedicated user threads) scheduling is done by an L4 kernel-level scheduler. It, however, can be influenced by user level schedulers via the *Schedule* system call. L4 implements a priority-based round robin scheduling with a total of 255 different priority levels. Each thread has an associated scheduler thread, which is capable of adjusting the thread's time slice and priority.

2.3.2 L^4Linux and the L4Ka Virtual Machine Environment

L^4Linux

L^4Linux [16] is a μ-kernel based, monolithic Linux operating system. Linux was adapted to run on top of L4. However, its API is fully binary compliant with Linux/x86. The initial version was implemented to show that μ-kernel based systems can reach a performance comparable to native systems. Though it was tried to keep the porting effort as low as possible. Only architecture-specific parts of the kernel were modified while structural changes were avoided. In total, the porting required to add and modify around 6500 lines of source to the guest. L4 was used without modification to see whether its interface was flexible and general enough to implement a high-performance, conventional operating system on top.

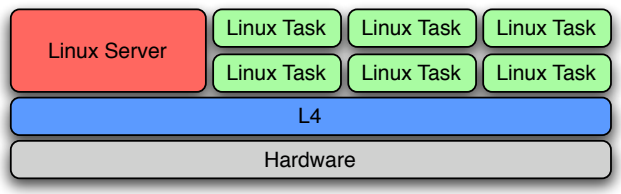


Figure 2.2: L^4Linux with a single Linux server and a bunch of user tasks

The single-server approach as shown in Figure 2.2 executes Linux as L4 server in a separate L4 address space. The server task uses several threads to execute the actual Linux code, to handle interrupts and to handle page faults. Each Linux task runs in its own L4 address space and consists of two L4 threads, one to execute the user binary and a second one to forward signals. The Linux server acts as pager for the user processes it creates.

The authors of [16] report a performance overhead of about 7% compared to a native system when compiling a Linux kernel in L^4Linux. Microbenchmarks, including Linux's *getpid* system call overhead and the lmbench suite, revealed different results, ranging from almost no performance penalty to an overhead of more than 100%.

That original version ran Linux server threads in a small L4 space (using L4's IA-32 small space optimization) and is thus not directly comparable to a virtualization environment that supports the concurrent execution of several virtual machines. For the latter, the 512 MB small space area is clearly too small. Ref. [19] compares the performance of a more recent version of L^4Linux without small space optimization to that of the L4Ka Virtual Machine Environment. The former approach performs only slightly better, but this can be traced back to the fewer guest operating system modifications when compared to L^4Linux.

The L4Ka Virtual Machine Environment

Joshua LeVasseur designed and implemented a reference Pre-Virtualization environment [19] for IA-32 with Xen and L4 as hypervisor to demonstrate the versatility of his virtualization techniques. He adapted several Linux versions by means of the automated compiler afterburning process to be used as guests. The compiler afterburner locates virtualization-sensitive instructions in the operating system binary. It automatically pads them with scratch space for runtime instruction rewriting, and annotates their location. However, certain structural changes are still being researched and to date have had to be done manually. J. LeVasseur also wrote emulation code (called the in-place VMM, IPVMM or wedge), which is mapped into each guest operating system's address space. At load time, the in-place VMM and the guest Linux kernel are linked together, i.e., the guest's virtual sensitive operations are replaced by calls to the wedge's emulation functions as is shown in Figure 2.3. The wedge maps sensitive instructions into abstractions that are provided by the underlying hypervisor. It further intercepts upcalls from the hypervisor such as interrupts, page faults, as well as exceptions and redirects the guest's control flow as appropriate.

Using L4 as hypervisor, guest user applications and the guest operating system

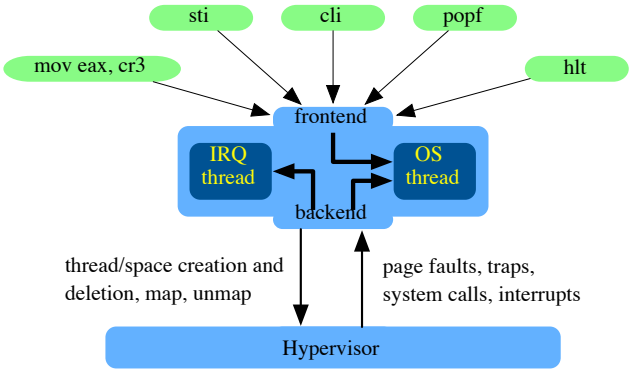


Figure 2.3: In-place virtual machine monitor

execute in separate L4 address spaces as shown in Figure 2.4. This is different from a non-virtualized IA-32 Linux running directly on real hardware, where the kernel maps itself into a reserved area in the top area of each of its user address spaces. Such a mapping, however, is not possible when using L4 abstractions as there are no means that would protect the guest kernel from its user applications.

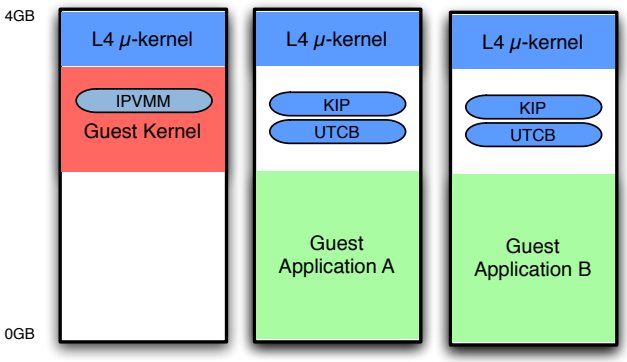


Figure 2.4: Virtual machine with separate guest OS space and two applications

Linux was re-linked to execute at a lower address for the fourth gigabyte is reserved by L4. It now occupies the third gigabyte in its own L4 address space. Other than on a native system where the application is mapped below the operating system area, the lower two gigabyte remain mostly unused. Sometimes, Linux touches memory in that area where it assumes the current application's code and data. The required application mappings are then mapped on-demand in and flushed to the next address space switch. Linux was further configured to provide applications a two gigabyte virtual address

space. The third gigabyte in each application space remains unused, too. This area would normally hold the operating system mappings but is now used for the KIP and UTCB mappings.

Common L4 user threads execute the guest operating system and its applications. Hence, each application space contains at least one L4 thread, which can cause page faults, exceptions, or invoke Linux system calls just as common Linux application threads on a native system. All these events result in an L4 page fault or exception IPC, which is forwarded to the guest operating system. Upon arrival of such messages, the VMM synthesizes an exception in the guest operating system and redirects the thread to execute the corresponding exception handler. Once handled, the guest OS reactivates its user application.

A second thread in the guest kernel's address space is a local pager. It handles page faults that are caused by the operating system thread. If the operating system requires physical memory mappings or causes page faults in the wedge, the request is forwarded to the resource monitor. The resource monitor is a separate L4 task. Initially it grabs all available memory from σ_0 and reserves a user defined amount of memory to each virtual machine. This memory reservation constitutes a VM's physical memory pool and is mapped on demand into the virtual machines. This all happens transparently to the guest kernel.

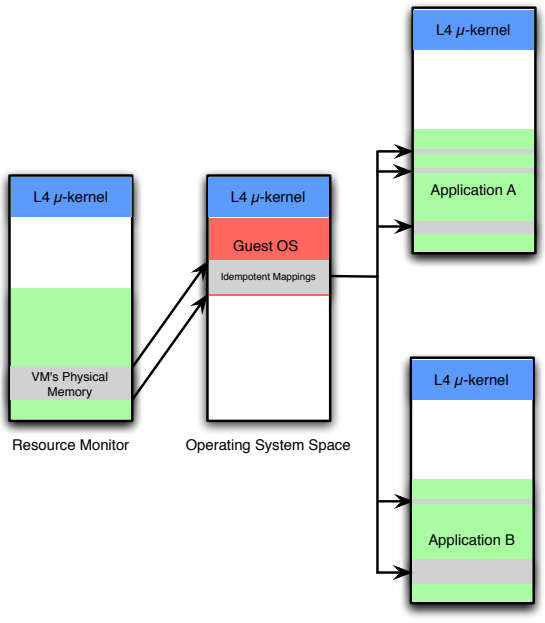


Figure 2.5: Physical Memory Management

While a guest kernel space thus receives its mappings directly from the resource

monitor, user application spaces do not. Rather, they receive their mappings from within the guest kernel's address space, more precisely, from the idempotent physical memory mapping in the one gigabyte guest kernel area. This mechanism is shown in Figure 2.5 and becomes possible since Linux has the first up to 896MB of physical memory contiguously mapped there. Given a physical address less than 896MB, the corresponding mapping in the guest operating system space can thus be calculated by simply adding a fixed offset (2 gigabyte here). To illustrate this, the following describes how application page faults are handled: First, a page fault IPC, which contains the faulting virtual address, is sent to the operating system. Subsequently, the page table is consulted. If Linux has set up a mapping at the faulting address, the page tables contain the assigned VM physical address. The L4 page mapped in reply to the page fault is the one in the guest kernel's address space at 2GB plus the physical address as per the page table, i.e., the one that corresponds to the physical page. This is a fairly limited approach, as it does not work with virtual machines that have more than 896MB RAM configured. The alternative would be to have all applications receive their mapping directly from the resource monitor, too. This, however, requires additional collaboration between the resource monitor and the wedge and would decrease performance.

Finally, the last thread in the guest OS's address space handles interrupts. It registers with L4 to receive interrupt IPCs on hardware interrupts and forwards them to the guest kernel. The wedge therefore provides logic to emulate an interrupt control unit.

2.4 Missing Transparency in the L4 API

This section identifies several L4 characteristics that interfere with the construction of a virtual machine on top of the μ-kernel. Most of them can be traced back to missing transparency (cp. Section 2.2), i.e., the desired features are provided by the underlying hardware but not exported via L4's API to be usable by applications .

2.4.1 Missing Privilege Modes

As described in Section 2.3.1, address spaces are a core L4 abstraction. Threads sharing the same address space can maliciously affect each other, e.g., one can smash another's stack, data, or write into its UTCB. On the other hand, threads running in different L4 address spaces are mutually protected from each other. Most contemporary architectures have support, which allows operating systems to efficiently provide the address space abstraction. In addition, they combine another concept with address spaces, one that is commonly

referred to as privilege modes. Privilege modes allow you to shield parts of an address space from access by its less-privileged threads. They allow you to safely co-locate privileged and less privileged execution entities in the same address space.

The safe co-location is a strong feature that is efficiently made use of by most modern operating system kernels, including Linux or L4. Both kernels map themselves into a privileged region of each user application space. On 32-bit systems, this is commonly the highest gigabyte in the four gigabyte address space. The remaining three gigabytes constitute the application's address space. The two major advantages of co-location are fast transitions between application and kernel as well as easily accessible user memory. Privilege modes make sure that less privileged user threads cannot access or even modify the kernel area.

The guest operating system obviously needs to protect itself from user access. Another simple, working approach to assure this, is to have the operating system run in its own address space, such as done by L^4Linux. However, switches between kernel and application execution then result in continuous address space switches and incur a large overhead. Other than that, only co-location gives operating systems access to application memory by using the application's virtual addresses. Therefore, almost all processors offer at least two privilege modes, one to be used by the operating system and one for its unprivileged applications. Certain architectures, such as IA-32, have additional privilege levels. Since operating systems commonly only use two of them, additional ones are normally not required. When constructing a virtual machine, though, it can be very useful to have them. Xen, for example, makes use of an additional IA-32 privilege level and uses segmentation to safely co-locate all three instances, the hypervisor, the guest operating system, and its applications, in a single virtual address space. This way the guest kernel has access to its application's memory and the hypervisor has full access to the guest operating system and applications. Transitions between them do no require a full hardware address space switch and cause little overhead.

The L4 API supports neither privilege levels nor segmentation in user mode, even if both features are provided by the underlying hardware. The single user privilege mode needs to be multiplexed between the guest operating system and its applications. Consequently both projects, L^4Linux, and the L4Ka Virtual Machine Environment had to put the guest kernel into its own, separate L4 address space. Besides a more expensive transitioning and user memory accessing, the design requires you to pay special attention to non-global mappings (from the guest's point of view) in the operating system space. On guest application switches, those mappings need to be flushed, which results in page faults in the kernel space the next time the application is scheduled.

2.4.2 Expensive Access Bits Virtualization

Unlike other hypervisors, L4 does not allow the guest kernel to directly access hardware page tables. It therefore requires you to virtualize them by introducing shadow page tables. The difficulty is to provide correct semantics of accessed and dirty bits since the processor automatically updates these bits only in the hardware page tables. With the current L4 API and Para-Virtualization techniques, there are two different approaches, which are both rather expensive. Via its *Unmap* system call, L4 allows you to retrieve access bits of up to 64 pages. The guest operating system can thus be modified to directly call the hypervisor to ask for the required access bits. An alternative would be to set permission rights in the hardware page tables so that every memory access causes a page fault in the virtual machine. Status bits can then be emulated at page fault time, i.e., the VMM marks the corresponding shadow page table entry as referenced.

2.4.3 Imprecise Scheduling

Unlike other operating system services, including pagers and exception handlers, the scheduler is still part of the L4 μ-kernel. It uses static priorities and schedules threads at the same priority in a round-robin fashion. A thread's time slice and priority can be controlled by its scheduler, i.e., by another L4 thread that was specified as scheduler via L4's *ThreadControl*.

Other operating systems use different scheduling techniques. Linux, for example, uses a heuristic algorithm based on the past behavior of its user processes and dynamically changes their priorities. They try to give interactive tasks a higher priority than highly CPU bound ones. The L4Ka Virtual Machine Environment and L^4Linux, however, assign the same static priority to all application threads. Also, they allow multiple user tasks to be running simultaneously beyond the guest's scheduler, i.e., while Linux expects one thread to run at a time, L4 executes whatever thread is runnable. To avoid this, L4 priorities of application threads would have to be continuously adjusted or threads would have to be continuously stopped and resumed. Both approaches, however, were regarded as too expensive and were therefore discarded. The current method, besides the fact that the guest scheduler almost completely loses control, also requires synchronization logic inside the VMM to hide the disordered execution (from Linux's point of view) from the guest. Further, the guest operating system is not notified when L4 schedules or preempts one of its user threads. The point of preemption and rescheduling is simply hidden behind L4's abstractions. Neither the VMM nor the guest operating system can therefore account for the processor time their operating system and application threads actually had.

Another shortcoming of the L4 scheduler is that scheduling domains for different VMs are hard to emulate. In an ideal system, L4 would allocate a fixed time slice to each virtual machine, the VM would then pass it around its processes (hierarchical scheduling). Each VM would be given a time slice according to the policy of the virtual machine. However, such a scheduling policy is not supported by the L4 scheduler, which makes inter-VM scheduling imprecise and complex to emulate.

2.4.4 Inaccessible Processor State

An operating system that executes on real hardware has complete access to the hardware's state, including the processors register set. When a user thread enters the kernel (on interrupts, system calls, exceptions, etc.) the operating system stores the user's state, for it needs to be restored after exception handling. This state as well as other processor registers may also be required by the kernel internally, e.g., to pass along signals. The VMM must thus be capable of passing all desired registers to the guest kernel.

L4 implements the concept of exception IPCs: A thread that causes an exception sends a message to its exception handler. This message contains the thread's instruction pointer and an architecture dependent number of processor state registers. However, in general, this only comprises a selection of registers required to change the control flow of a thread. Other than that no registers are sent along page fault exceptions at all. Rather, they are almost completely hidden behind L4's abstractions.

It is possible for a VMM to partly retrieve the missing registers. One way is to force the target thread to execute stub code, which stores all user accessible registers in memory and passes it over to the VMM. However, this indirection is quite inefficient and still does not allow it to get the information from processor registers that are inaccessible in user mode.

2.4.5 Inadequate Control Transfer Support

The VMM needs to change the guest operating system's execution flow on asynchronous events including all kinds of exceptions. In turn, the guest operating system sometimes needs to redirect its application's control flow, e.g., to deliver signals. The current API allows it to easily do this in reply to an Exception IPC. In cases where no Exception IPC is sent, however, control flow redirection becomes intricate, in particular if it takes place across L4 address spaces. But even within the same address space, control transfer must be synthesized by means of stub code, a temporary stack that contains the target thread's new register state, and a redirection via L4's *ExchangeRegisters*.

2.4.6 Inefficient Address Space Construction

The L4 API is inefficient for address space construction that is based on shadow page tables. To create new mappings and to unmap pages via L4's API requires virtual addresses. This is a problem as the guest's page tables have VM physical addresses. For proper handling, page faults have to be resolved by a separate L4 task that represents a virtual machine's physical memory. Following, the guest operating systems and their physical memory tasks must collaborate, which adversely affects performance.

2.4.7 Indistinguishable Execution Modes

L4 classifies user threads into privileged and unprivileged ones. The only difference between them is that privileged threads are allowed to execute privileged L4 system calls while non-privileged ones are not. The latter, however, can still make use of all non-privileged system calls.

This L4 characteristic raises a problem in virtual machine construction. Being able to use L4 system calls, both, guest applications and the operating system do not exhibit a behavior identical with that experienced when running on a non-virtualized system. Another issue is the kernel mappings (KIP and UTCB) in each L4 address space, which cannot be hidden. All this is a minor problem with respect to the guest kernel, as one can use Para-Virtualization techniques to make the guest kernel aware of this fact. For its unmodified user applications, though, this remains an issue. Up to now it has relied on the fact that Linux user applications simply do not use L4 system calls and the kernel mappings are put into an otherwise unoccupied memory region (cp. Figure 2.4).

On the other hand, it can sometimes be quite beneficial to allow guest application to interact with L4, too. As of yet, application system calls are initiated by common software interrupts, which result in an expensive L4 exception. It would be more efficient if the user applications used conventional IPC instead. Provided that this is possible without modifications to the application binaries. Linux, for example, allows you to do so. It maps a page similar to L4's KIP into each application address space. The mapping is commonly referred to as *vsyscall page* and contains functions that which user applications call to perform a system call. With Para-Virtualization, these functions can be modified to send an L4 IPC to the operating system thread.

2.5 Related Work

Peter M. Chen, George W. Dunlap and Samuel T. King [17] examined the reasons for the large overhead in the hosted approach. They focused their research on UMLinux [3], a VMM developed by researchers at the University of Erlangen-Nuernberg for use in fault-injection experiments. It builds on Linux as a host operating system and has to modify the machine-dependent parts of the guest to comply with the VMM (Para-Virtualization). UMLinux uses only two host processes. The first executes the guest operating system and all guest applications. They both share the same 4GB address space. To protect the guest operating system from its applications, the guest kernel area is only mapped into the address space in kernel mode. It is continuously mapped in and out on kernel entries and exits via the host's *map*, *unmap*, and *mprotect* system call. The same mechanism is used to switch between guest applications, i.e., the mappings in the application area that belong to the outgoing process are revoked and the incoming application is mapped in.

Compared to a standalone system, the original UMLinux was hundreds of times slower for null system calls and context switches. It was 18 times slower in compiling the host kernel, 8 times slower on the SPECweb99 benchmark and incurred an overhead of about 10% on POV-Ray. Three reasons were identified to cause the large overhead. All of them could be eliminated by simple modifications to the host system. First, the design with two separate host processes caused an excessive number of context switches. Moving the VMM process's functionality into the host kernel improved performance by a factor of around 3. A second modification was to use IA-32 segmentation to protect the guest kernel area without the expensive memory mappings when switching between guest user and kernel mode. This optimization gained a performance boost by a factor of 3 to 5 depending on the benchmark conducted. Finally, the host operating system API was augmented to allow single processes to support various address space mappings. A new system call was introduced to switch between the different mappings, thereby reducing context switch overhead by a factor of 13. All three host OS optimizations together yielded a VMM that runs all macrobenchmarks with a performance overhead of no more than 14% to 35% relative to running outside a virtual machine.

User Mode Linux [11] is similar to UMLinux. The main difference between both approaches is that the original version of User Mode Linux has separate host processes for each guest application. The guest kernel is always present in each address space, and, by default not protected from its application. To fix this obvious security problem and to get rid of an inefficient signal delivery mechanism on guest application system calls and interrupts, its designers decided to add support in the host Linux kernel. A new execution mode called skas (Separate Kernel Address Space) was introduced. It allows you to efficiently run the UML guest kernel in an entirely different host address space

and to improve signal delivery. In skas mode, only one host thread executes all UML application code. It does so by switching between host address spaces on UML context switches.

In [18], Lackorzynski proposes changes to L4 to simplify L^4Linux. He proposes to extend L4's *ExchangeRegisters* to get rid of the cumbersome L^4Linux signal delivery. L4's API restricts this system call to be executable only by threads in the same address space as the target thread. It is proposed to additionally allow a thread's pager (in case of L^4Linux, the Linux server) to use *ExchangeRegisters*. He thereby enables the Linux server to force its application threads to enter the kernel and thereby greatly simplified signal handling. Another proposal he make is to have a user thread's state visible in the exception handler's and pager's UTCB on exceptions and page faults, respectively. The register state can thus be efficiently passed to the server.

Chapter 3

Proposed Solution

The previous chapter identified several L4 characteristics that complicate virtual machine construction on top of L4. Most of them are caused by a lack of transparency in L4's API, which hides the desired features behind high-level, operating-system-like abstractions. This section addresses several of these issues and proposes API extensions to improve virtualization on L4. The guiding principle behind all modifications is to conform to L4's philosophy and to minimize impact where something is architecture specific.

3.1 Thread Limits

As explained in Section 2.4.1, user applications on native systems commonly share their virtual address space with the operating system. The operating system reserves itself parts of each address space that is made unaccessible to user applications.

This co-location is hardware supported by means of privilege levels, with most modern computer architectures offering at least two of them. The operating system commonly reserves the most privileged level for itself, which leaves a less privileged one for user applications. If more than two privilege levels are available, the additional ones often remain unused. They are not exported via the API to be usable by applications. For virtual machine construction, however, that might be very beneficial as it would allow the co-location of the guest operating system and its applications in the same user address space.

Rather than extending the L4 API to export privileges, the extension proposed here achieves nearly the same result by offering per-thread address space limits. It allows one to restrict a thread's accessible area in its L4 address space to a contiguous, variable sized memory region starting at address zero. The limit

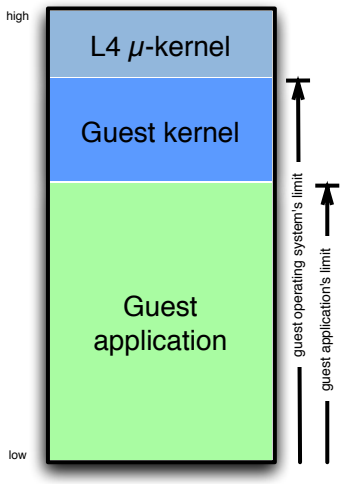


Figure 3.1: L4 address space with co-located guest application and operating system

can be defined at thread creation via an parameter added to L4's system call *ThreadControl* (cp. Appendix A.3) and can also be changed later on.

As shown in Figure 3.1, per-thread limits allow to safely co-locate guest operating systems and user applications in the same L4 address space. The guest operating system thread can access the full 3GB area. A guest application's address space, however, is restricted from the kernel area; applications can thus not interfere with the operating system.

There are, however, a couple of limitations to this approach. One is that the guest operating system is assumed to reside at a higher address than its applications. Secondly, operating systems sometimes make certain pages in the kernel area user accessible, such as Linux's vsyscall page (cp. Section 2.4.7). With a fixed limit set, it is not possible for user applications to access that memory. However, Para- and Pre-Virtualization allows the solution of this problem by relinking user accessible pages to the beginning of the guest kernel area and extending the application thread's limits accordingly. Another downside is that an application's virtual address space shrinks down. Further, as will be explained later in Section 4.4, the extension can only be efficiently implemented if the CPU offers support. Finally, the protection is incomplete as even restricted user application threads can execute L4 system calls, including *ExchangeRegisters*, and thereby compromise the guest kernel. Yet even without co-location this might pose major problems.

3.2 Control Transfer Messages

Control transfer messages address the problem of intricate control flow redirection as described in Section 2.4.5. They allow VMs to efficiently handle asynchronous events that entail a redirection of a thread's control flow. Other than that control transfer items were integrated into further extensions as will be described below.

A control transfer item is a new IPC message type, which specifies a part of a thread's register state. Typically it contains the register state required to change the control flow of a thread, e.g., instruction pointer, stack pointer etc. Its length is architecture specific. The item serves two different purposes. First, it enables threads to change the register set of other threads, similar to L4's *ExchangeRegister* system call, but is not restricted to be used among threads in the same address space. For a transfer to succeed, the receiving thread must set a bit in its acceptor to indicate the willingness to accept a new state. Secondly, it offers an elegant way of allowing threads to pass a register state to other threads. If a control transfer item is sent to a thread whose acceptor bit is cleared, the item is copied into the receiver's message registers, however, the receiving thread's state remains unchanged.

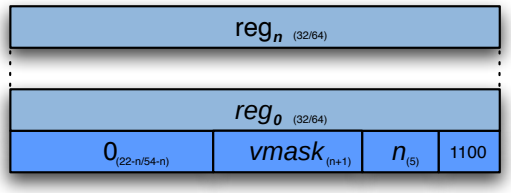


Figure 3.2: Control transfer item layout

The general layout of a control transfer item is shown in Figure 3.2. The value n is an architecture dependent number of register values sent minus one, and *vmask* is a bitmask to indicate which registers of the accepting thread should be set to the values in the message. Only if bit k is set, the value reg_k is written into the corresponding register. Otherwise the value is ignored.

3.3 Preemption Messages

Preemption messages increase transparency of the kernel's internal scheduling and address the problems described in Section 2.4.3. They offer user-level threads a way to trace which L4 thread was actually assigned CPU time and provide information about the time of preemption. The extension allows a

virtual machine to avoid the influence of the L4 scheduler and enables the guest OS to fully control the scheduling of its applications.

A preemption message is a reliable IPC call from a just-preempted thread to its scheduler. It can be activated and deactivated on a per-thread basis using L4's *Schedule* system call. If activated for a thread, L4 synthesizes a message and sends it to the thread's scheduler in one of the following cases:

1. The thread's time slice has expired and L4 chooses to schedule another thread.

2. A higher priority thread woke up (to which L4 instantly switches).

3. A hardware interrupt was triggered and L4 decides to continue with another thread.

Subsequently the preempted thread waits for the receiver's response, i.e., L4 will not reschedule the thread unless it receives the schedulers reply message.

Preemption messages include the time of preemption. The kernel can further be configured to support control transfer and to attach the suspended thread's execution state in terms of a control transfer item as shown in Figure 3.3. In this case, L4 sets the preempted thread's control transfer acceptor bit and thereby allows the scheduler to set a new state in reply.

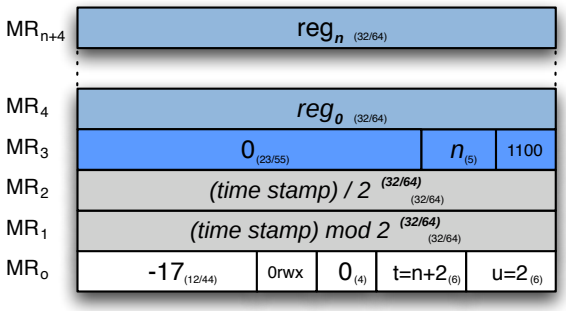


Figure 3.3: Preemption message synthesized by L4

On the downside, a problem with the extension might become the extra overhead that is caused by frequent preemption messages, in particular if L4 is configured to pass the user register state along.

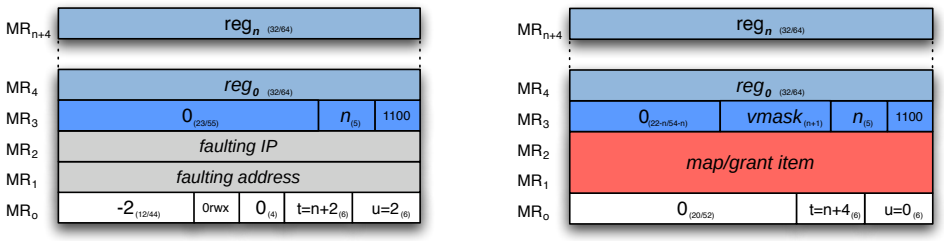
3.4 Zero Time Slices

This simple extension allows the configuration of threads with a time slice of zero. Because of this, the L4 scheduler will not schedule them unless they execute on donated time slices. This can be used to more accurately emulate scheduling domains by setting guest operating system and application thread's time slices to zero. Thus external, inter-VM scheduling can be controlled by a global authority via time slice donation.

3.5 An Extended Page Fault Protocol

The problem addressed here is that an application's register state is unaccessible in user mode, or inefficient to retrieve. As by the original L4 API, a page fault message consists of two untyped words, the faulting address and instruction pointer. Although other exception messages contain additional thread state information such as the thread's general-purpose register set. An immediate access to that information on page faults would simplify virtual machine construction on L4 (cp. Section 2.4.4).

The μ-kernel was therefore extended to send the additional information along with page fault messages as well. If both, this extension and control transfer messages are enabled, the state is passed by means of a control transfer item as shown in Figure 3.4(a). The item contains the faulting thread's state at the point where the exception occurred. The kernel also sets the faulting thread's control transfer item acceptor bit, thereby allowing its pager to modify the thread's execution state in reply.



(a) Page fault message synthesized by L4

(b) Page fault message reply

Figure 3.4: Page fault message layout

Control transfer items increase transparency between L4's API and the physical hardware. The register set of a user thread at the time when a page fault was generated is made accessible to other user threads. With respect to virtualization, a guest operating system can now easily be passed the required

exception context. Further, the guest operating system is offered a nice and efficient way to change the applications control flow, e.g., Linux could run a signal handler if the page faults could not be handled properly.

The proposed extension is well suited to the demands of the IA-32 L4Ka Virtual Machine Environment, in particular as IA-32 has a small general purpose register set. However, on the downside it is not generally applicable to all architectures. Especially on those with a large register set (such as IA-64), passing the whole state in 64 message registers is infeasible and would incur a large page fault penalty. Further, it does not provide an efficient way to retrieve other non-general-purpose registers, including floating point registers. A more general approach could be a new system call, which allows you to demand a selected register set of a specific thread. This, however, requires the complete register set to be stored in the kernel for all threads, even though that might not even be required by L4 otherwise. Another problem is the additional time needed for kernel entries and exits on the different platforms. On IA-32, with very expensive kernel entries and exits and a small register set that frequently needs to be passed to the guest kernel, sending the state along messages is likely to be the most efficient way. On other architectures it is a tradeoff between the register set commonly required and the overhead such system calls entail.

3.6 A new Wedge Design

To see whether the proposed L4 extensions pay off, J. LeVasseur's in-place VMM and parts of the resource monitor (both described in Section 2.3.2) were redesigned. Major changes include the co-location of guest applications and the guest operating system in the same L4 address space, a more accurate scheduling policy and signal handling as well as the elimination of several inefficiencies inside the wedge. The redesign also required further modifications to the guest operating system.

Co-Located Guest Application and Operating System

Figure 3.5 shows the general layout of the redesigned virtual machine, which makes a separate guest operating system address space redundant. The guest kernel is mapped into each user application's address space. The KIP and UTCB mappings are put into the first megabyte of the guest kernel area that Linux doesn't use. Application threads have their limit set to cover the 2GB application space. Optionally it can be extended to further include the KIP and parts of the UTCB area.

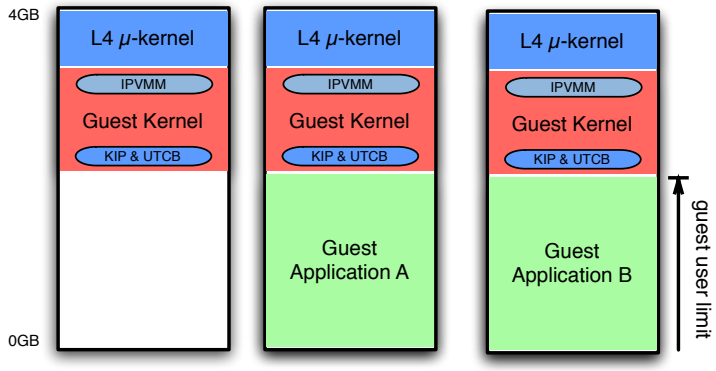


Figure 3.5: The Linux kernel resides in a logical space that spans all three hardware address spaces

Each application address space holds at least three L4 threads. The two privileged ones are the guest operating system and its pager thread. One or several non-privileged ones execute the application. Other than application spaces, there is a single master address space. It is the first address space created in each VM and executes the IRQ thread.

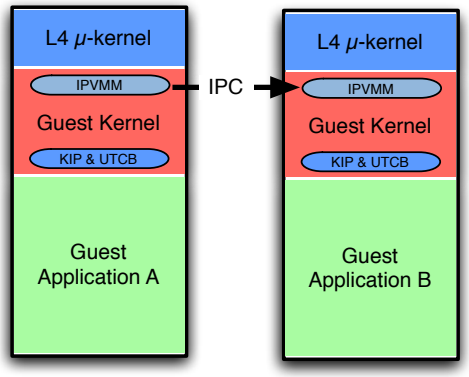


Figure 3.6: A guest address space switch causes the VMM to switch execution to another L4 space

Even though there may be hundreds of application spaces in a virtual machine, only one guest operating system thread is runnable at any point in time. The "active" space corresponds to the guest operating system's current task. On each guest kernel task switch, the VMM stops execution in the outgoing address space and resumes the guest operating system thread of the incoming one. This is done by means of an L4 IPC as shown in Figure 3.6.

All threads in a non-active space are waiting for an IPC. In particular operating system threads are waiting to be activated by operating system threads in other application spaces. On each guest address space switch, the guest operating system thread executes emulation code. It sends an IPC to activate the target space and subsequently waits to get activated again. The VMM preserves the continuity across such switches, i.e., it makes sure that the new guest operating system thread resumes execution with the same register set and the same kernel stack.

Improved Scheduling

In an "active" space, either the guest kernel or a guest application thread is runnable, but never both. With the introduction of preemption messages, there is no concurrent execution of L4 threads beyond the guest's control any more.

Preemption messages are enabled for all application threads. Consequently they are not executed by L4 unless their scheduler, the guest operating system thread, activates them by sending a preemption message reply. Linux is thus enabled to fully control scheduling of its user applications, as opposed to the old wedge design. Further, several guest application threads in the same address space can be executed by a single L4 thread and lots of synchronizing logic that was required to hide the concurrent execution of L4 threads could be removed.

Preemption messages are as well enabled on all guest operating system threads with the IRQ thread as their scheduler. This not only allows the support of a new interrupt handling (which is explained further below) but can also be used to implement a virtual clock. Such a virtual clock becomes useful when several virtual machines execute simultaneously on the same system and thus preempt each other. As each virtual machine is notified on preemption, it can easily account for the lost time.

Improved Page Fault Handling

With the original L4 API, it was too expensive to retrieve an application's register state on page faults. Linux's page fault handler was therefore passed an invalid register set. The new wedge makes use of the user state, which is sent along L4 page fault messages and allows to virtualize page faults properly.

Improved Exception, Signal and Interrupt Handling

Control transfer messages allow for the efficient change of a thread's control flow. They are used to redirect control flow on asynchronous events while the guest operating system thread executes. Further, they are used to redirect the application's control flow to deliver Linux signals.

The original L4 kernel required the VMM to cumbersomely redirect control flow on asynchronous events by means of stub code, a temporary stack and L4's *ExchangeRegisters*. Signals to user applications could only be delivered in reply to a user exception since only exception messages allowed a new thread state to be set in reply. Unless the application caused such an exception, Linux signal handling failed.

The new VMM makes use of control transfer in combination with preemption messages to allow for an improved handling of these events. With preemption messages enabled on all application threads, signals can now be delivered at any point in time when the operating system thread executes. A control transfer can be sent to the application in reply to the formerly received preemption, exception, or page fault message. The VMM can further easily redirect the operating system thread's control flow.

Chapter 4

Implementation

This chapter describes a reference implementation for all L4 extensions and the redesigned VMM as proposed by this thesis. The extended μ-kernel is based on L4's IA-32 port while the redesigned VMM emanates from J. LeVasseur Pre-Virtualization reference implementation.

4.1 Control Transfer Messages

Control transfer messages allow you to change the control flow of threads by means of a common IPC (cp. Section 3.2). It is a simple L4 extension and did not require any substantial modifications to the kernel's internals. Control transfer items on IA-32 as shown in Figure 4.1 are treated as typed message elements just as strings, flexpage-map, or grant items. Adjustments to L4's IPC mechanism could thus be restricted to a single function that gets only invoked by the IPC logic when typed message items are involved. A typed item is identified via its tag, i.e., the lowermost four bits of its first word. For control transfer items the bit sequence 1100_b was chosen. Other than that, the highly optimized IPC logic remained untouched.

As explained in Section 2.3.1, acceptors specify what kind of typed items a thread is willing to accept when receiving a message. One of the reserved bits in the acceptor was thus used for control transfer items. Apart from this, acceptors remain unchanged. The new acceptor is shown in Appendix A.1.

To allow for a control transfer, the receiver must set the corresponding bit in its acceptor and perform a receiving IPC system call. When entering the kernel as a result of the system call, L4 stores the thread's context on its kernel stack to be restored when it is rescheduled. The current implementation therefore simply overwrites the thread's context frame according to the received control

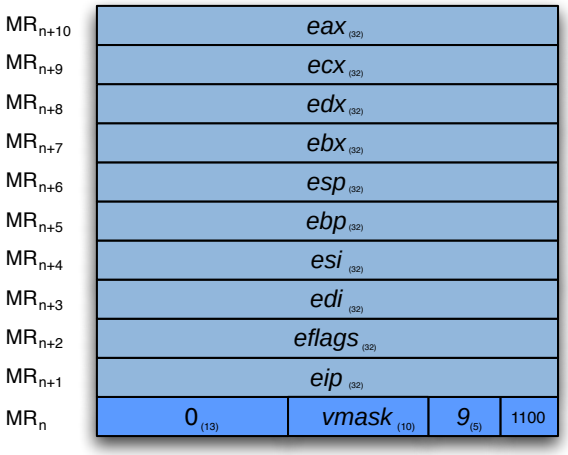


Figure 4.1: Control transfer item on IA-32

transfer item, provided the receiver accepts. If the receiver's acceptor bit is cleared, though, no control transfer takes place. However, the item is copied into the receiver's message registers.

4.2 Preemption Messages

Preemption messages required more modification of the kernel, including a new internal thread state, a different startup protocol as well as changes to the kernel's internal control flow. The extension can be enabled on individual threads via L4's *Schedule* system call. Detailed information on how to turn it on and off can be found in Appendix A.5.

For threads with preemption messages enabled, a message containing the time stamp and, if L4 is appropriately configured, the execution context at the point of preemption is sent to the thread's associated scheduler on asynchronous preemption other than page faults (cp. Section 3.3). The current implementation therefore uses L4's IPC mechanism without modification. First, L4 composes the message in the preempted thread's message registers to send it to the scheduler. Subsequently, L4 completes the IPC operation provided that the receiver was ready to receive a message or if not, temporarily halts the sending thread and reschedule. The latter case results in L4's original scheduling behavior and does not require any further changes. However, if the scheduler is ready to receive when a preemption message is sent, L4's IPC logic will unconditionally grant the sender's remaining time slice to the receiver, i.e., the scheduler, to which it instantly switches. This is the appropriate behavior

in case the receiver's priority is the highest amongst all currently runnable threads. However, if the thread that caused the sender's preemption has a higher priority, the proper behavior would be to schedule it rather than the receiver.

A way around this priority inversion is to have L4 reschedule after each IPC operation. This, however, requires changes to the optimized IPC process as well as additional scheduling decisions that would incur unacceptable IPC overhead in the general case. Therefore, a hook was implemented to redirect the receiver's control flow in this special case.

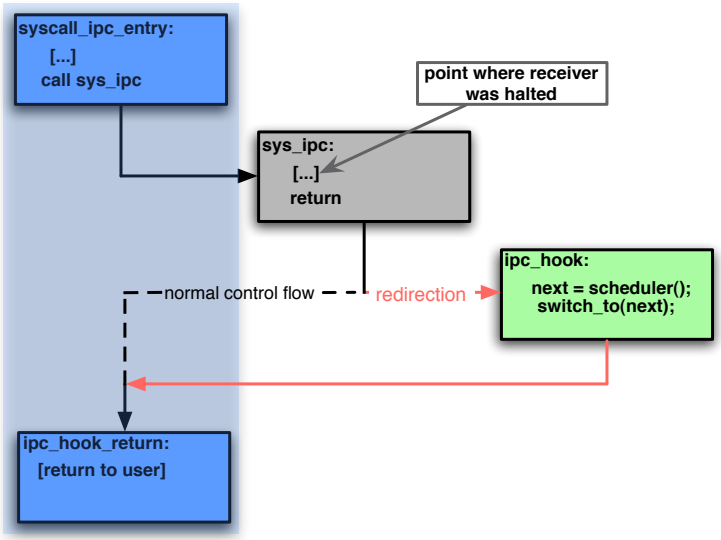


Figure 4.2: Receiver thread control flow redirection on preemption messages

To illustrate how the hook works, Figure 4.2 shows a thread's control flow during an IPC. Label *syscall_ipc_entry* marks the IPC entry point. A call to the architecture independent IPC main method *sys_ipc* follows and causes the caller's return address to be pushed on the stack. This return address is at a fixed offset from the receiver's stack bottom and can thus be located. Before it starts the IPC operation to send a preemption message, L4 checks whether the destination thread is waiting. If so, the receiver must have performed an IPC call at some earlier point. It was then interrupted somewhere within the IPC process, more precisely in *sys_ipc*. The current implementation overwrites the stored return address. It thereby redirects the receiver when it continues on the donated time slice to execute a function that instantly reschedules. The reschedule function never returns to its caller but jumps to the instruction at the replaced address and thus returns to normal control flow. This hook allows the proper implementation of preemption messages without changing a single line of code in the IPC logic.

Another situation that required special attention is when preemption messages are enabled on threads that have not yet received a startup message. As by the L4 API this special message must be sent by the thread's pager and has to contain the initial stack and instruction pointer. L4 subsequently sets the thread's registers according to the values passed in and starts executing it.

A new thread state (a thread waiting for a startup message) was introduced to catch this case. The startup protocol then changes. Rather than having the thread waiting for a startup message from its pager, it then waits for a startup message from its scheduler. The thread further accepts a control transfer if L4 is configured to support control transfer messages. This way it is put into a state equivalent to as if it was preempted earlier, waiting to be re-activated by its scheduler. The startup message to be sent by the scheduler has to contain an instruction and stack pointer (in form of two untyped words) or may consist of a complete control transfer item with instruction and stack pointer set accordingly.

4.3 Page Fault Extension

Page fault messages are synthesized by L4 on behalf of the faulting thread and sent to the faulting thread's pager. This is done transparently to the originator. Since the kernel uses the faulting thread's message registers to compose and send the message, it needs to restore them after the page fault handling has completed.

The extended page fault protocol passes additional user state information along with page fault messages. Its implementation required simple adjustments to existing kernel methods that synthesize and send page fault messages. On page faults, L4 stores the desired user state on the faulting thread's kernel stack. The stored user state can thus be accessed and simply appended to the original page fault message, either in terms of untyped words, or (if the kernel is configured to support control transfer) in form of a control transfer item. Figure 3.4(a) depicts how the sender's message register set finally looks like. It is then copied into the receiver's message registers during the IPC process. The kernel has to be able to finally restore the original message register content. The required number of modified registers is thus saved in the sender's thread control block (TCB), which was accordingly extended.

There is an as yet unimplemented optimization to avoid saving, restoring and copying those message registers that are used to pass a thread's execution context. One could have the IPC logic read the sender's processor registers from the kernel stack and directly write them into the receiver's message registers. With this optimization the sender would only send minimal information (e.g., a bit set in the message tag) to indicate that the user context still needs to

be attached. When the actual message register copying takes place, the kernel can access the sender's context and append it to the receiver's message. Compared to the current implementation, this saves three memory copies of the size of an control transfer item. However, it requires additional logic for the case that some other thread modifies the faulting thread's state via L4's *ExchangeRegisters* before the IPC operation has completed. Otherwise the receiver will get the execution context not at the point of the exception but at the modified one.

4.4 Thread Limits

Thread limits are implemented by means of IA-32 segmentation. Segmentation is an IA-32 specific feature that is well suited to the desired functionality. However, this feature is not provided by all computer architectures.

The following gives a brief overview of those parts of IA-32's memory management that are used by the implementation. Subsequently, implementation details on per-thread limits are provided and alternatives are proposed on how to achieve the same functionality on other architectures.

4.4.1 IA-32 Protected Mode Memory Management

Starting with the 80286, x86-compatible CPUs offer a two-level logical to physical address translation. The processor's memory management unit transforms a user's virtual address (commonly referred to as logical address in conjunction with IA-32) consisting of a 16-bit segment selector and a 32-bit offset into a 32-bit linear address. Subsequently a paging unit transforms the linear address into the 32-bit physical address. Figure 4.3 shows the first step taken by the segmentation unit to translate a logical address.

IA-32 has six segment registers (called selectors) to be used as an index into a kernel protected, memory resident table of segment descriptors. Each segment descriptor is 8 bytes long and specifies a segment (a region of memory) within the 4GB linear address space by its base address, its limit, and a required privilege level to access memory in that area. At system startup, the operating system sets up the table with descriptors for kernel and user segments. Only processes running at privilege levels higher or equal to the descriptor's privilege are allowed to reference the covered memory. There are global as well as per process (local) descriptor tables. The entirety of segments that can be referenced by an application in either the global or its local descriptor table constitute its part of the linear address space. Any attempt by non-privileged threads to access linear addresses outside their dedicated memory segments

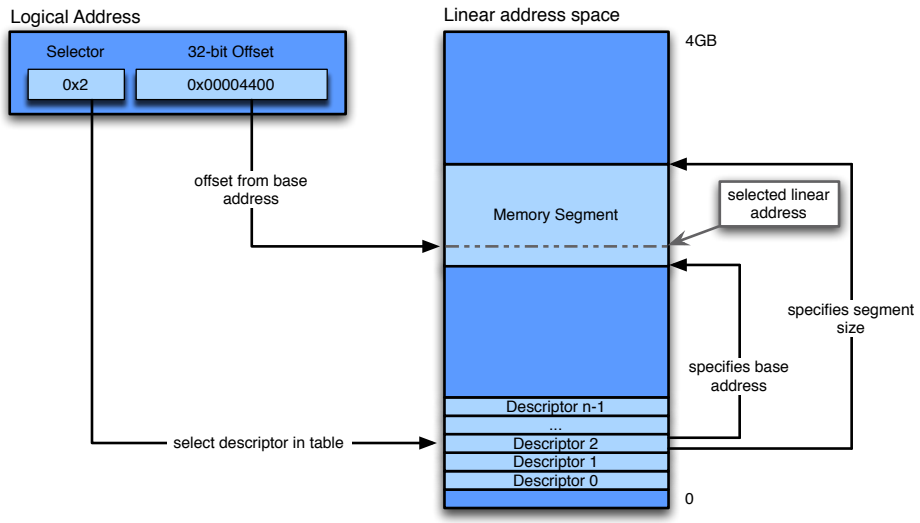


Figure 4.3: Translating a logical address

causes the processor to raise an exception (segmentation fault, or a general protection fault if a segment register is loaded with illegal indices) and to pass control back to the privileged kernel.

A programmer does not necessarily have to explicitly state the full logical address (consisting of a segment selector and an offset) when reading from or writing to memory. Instructions have implicit segments they refer to when being executed. For example a *move* instruction used to read or write from or to memory implicitly uses the data segment (DS) selector. Operations involving the stack such as *push* or *pop* refer to the stack in the segment pointed to by the stack segment (SS) selector and take the stack pointer (SP) as offset. The CS (code segment selector) register points to the segment that holds the processes' executable code and is implicitly used by instructions such as procedure calls or jumps. ES is commonly used by string copy operations while FS and GS are free to point to further data segment descriptors.

Segmentation can be used to assign each process a different part of the 4GB linear address space. Linear addresses, however, are not directly mapped one to one to physical addresses. Rather, a second memory management component maps memory blocks in the linear address space to blocks of real physical memory. The mappings are controlled by the operating system by means of so called page tables. It is common for the operating system to have one page table for each user address space. Mappings for different address spaces can thus be established by changing the page table on context switches. This mechanism is commonly referred to as paging.

Unlike segmentation, paging is available on all modern computer architectures. Paging can map the same linear address to different physical ones. It is more flexible than segmentation and allows you to multiplex the linear address space among all user applications. Therefore, Linux, L4, and most other contemporary operating systems use a memory model called flat segmentation. With flat segmentation, segment descriptors are set to cover 4GB segments starting at address zero, thus allowing each process to access the whole linear address space. Code data and stack segments overlap. Paging is used to multiplex the linear address space and to prevent user threads from accessing those virtual memory regions that are reserved by the operating system.

4.4.2 Thread Limit Implementation

IA-32 supports 4GB virtual address spaces starting at address zero. The L4 kernel reserves the 4th gigabyte in each address space, which leaves a 3GB user area to its user threads. Per-thread limits aim at further restricting the user accessible part for certain threads. This can be achieved by adapting the segment limits of all user accessible segments on thread switches.

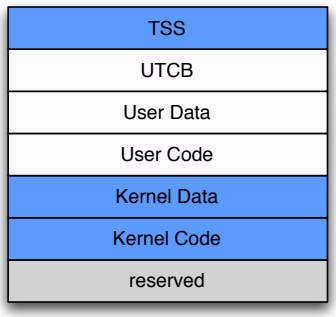


Figure 4.4: L4's Global Descriptor Table

Figure 4.4 shows L4's global descriptor table with three segment descriptors for user threads and the kernel each. The first entry is reserved and must not be used by the operating system. The following two descriptors constitute flat, privileged 4GB memory segments. They can only be referenced in kernel mode and allow you to access the whole 4GB of linear memory. Two non-privileged segments, *User Code* and *User Data* are thought to be used by application threads to access their program code and data. Their limits depend on how the kernel is configured. Without IA-32's small space optimization, both segments are flat as well. L4 protects the 4th GB by means of paging. With small address space support enabled, the user segment descriptor limits range between a minimum of 4MB up to 3GB. The reason for this is that L4 then puts several user address spaces inside the same hardware address space.

A single large 3GB user space starts at logical address zero and several small user spaces share (but do not overlap) the following area covering 512MB. When L4 executes threads in a large space, it sets the user segment limits to 3GB (with the base at address zero) to protect the small space area. When it switches to a thread in a small address space, the segment limits are set according to the space's size. The segment bases point to the range inside the 512MB small space area that is reserved for the respective space.

The third user accessible descriptor, denoted as *UTCB*, covers a tiny user accessible part in the linear address space above 3GB, which provides the current thread's UTCB address at offset zero. Finally, *TSS* is a kernel protected segment to hold a special x86 structure with information about the current task.

To allow for per-thread limits, the extension adjusts the user's code and data segment limits on thread switches. Non-restricted threads experience the same virtual address space as before. But when L4 switches to a restricted thread both segment limits are reduced accordingly. To avoid needless, expensive GDT updates, the kernel first compares the new limit to a per-CPU global variable with the ceiling currently set.

Limits are supported in steps of 4KB and stored in each thread's TCB. A thread's limit must be at least 4KB and may expand up to 3GB. For threads running in small spaces, the actual limit written into the GDT is the minimum of the small space's limit and the thread's limit. Even though it would be more efficient to have the minimum value calculated once and stored in each thread's TCB for fast access on thread switches, the limit needs to be re-determined on each thread switch. The reason for that is that L4 provides no mapping from address spaces to threads running inside them. Thus, if the small space limit is reduced, it is not possible to update the stored limits in the affected thread's TCBs.

4.4.3 Implementation Alternatives

Segmentation is quite IA-32 specific and does not allow arbitrary virtual memory regions to be protected. An alternative implementation, which is supported by other architectures as well, is to use paging, i.e., to make certain pages in an address space inaccessible to restricted threads. However, the costs highly differ from architecture to architecture. On IA-32 with hierarchical page tables that are directly accessed by the processor, they are unacceptably high. The kernel would have to walk through the page tables and mark selected pages inaccessible on each thread switch from or to a restricted thread. That would significantly increase the thread switch overhead. Other architectures, including Mips, have a software-loaded TLB, i.e., the processor raises a page

fault exception if no valid mapping can be found in the TLB. System software then looks up the translation in its own mapping structures and updates the TLB accordingly. In such a scheme it is way simpler and more efficient to make regions in an address space inaccessible. All the page fault handler needs to do, is to check whether the faulting thread is allowed to access the faulting page or whether it is part of a protected area. If it is allowed to do so, the mapping is written into the TLB, otherwise an exception is triggered.

4.5 Wedge Design

The redesigned VMM is based on J. LeVasseur's design, which was augmented to use the additional features provided by the extended L4 μ-kernel. The following sections describe how the major modifications were implemented.

4.5.1 Address Space Construction

As opposed to the original VMM where the guest operating system ran in a separate L4 address space, the redesigned VMM maps the guest operating system into each application's address space (cp. Section 3.6). The VMM thereby needs to make sure the different guest kernel areas, even though they are distributed across several L4 address spaces, constitute an identical logical address space. Changes in one kernel area must become immediately effective in all other kernel areas. Consequently, page faults in the same VM and at the identical VM physical address must result in the same memory page to be mapped in, independent of the specific L4 address space. This was achieved by having physical page requests from all guest operating system threads resolved by the identical process, the resource monitor. The resource monitor doesn't care about which guest operating system thread originated the request. All it factors in is the virtual machine the requester belongs to and the physical page called for.

Similar to the old VMM, the contiguous physical memory mappings in the kernel areas are used to back application memory mappings. However, since application page faults are now handled by the local operating system thread, each application's mappings also originate in the local address space as shown in Figure 4.5.

To allow for the original VMM's address space construction via shadow page tables, Linux was modified to execute emulation code in the wedge whenever it establishes or deletes memory mappings (i.e., whenever is modifies a page table or directory entry) in either an application or the kernel space. A pointer to the modified page table entry is passed along these functions. It allows

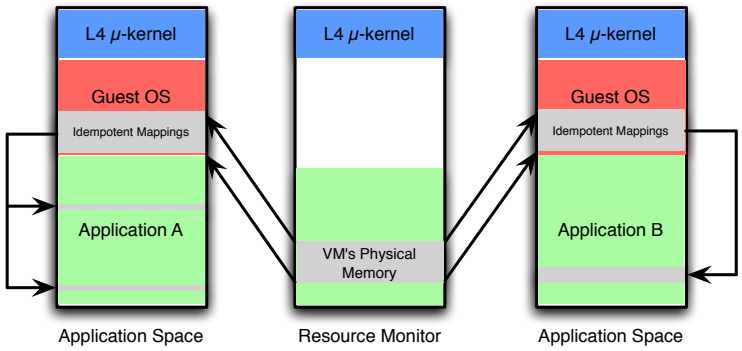


Figure 4.5: Distributed Physical Memory Mapping

you to retrieve the affected VM physical memory page. The original VMM could then easily flush invalidated physical page mappings from all application spaces. It simply performed an unmap operation on the page in the idempotent mapping area corresponding to the physical one. As a result L4 unmaped the affected VM physical page from all application spaces, no matter what address space the page table entry actually referred to. However, these page table hooks turned out to be insufficient for the redesigned VMM's distributed mapping scenario. Doing the same unmap operation would result in the corresponding physical page being unmapped only from the current application's address space, but not from others. The problem encountered here is that the emulation functions do not provide information about which address space the page table entries actually belong to. If they did, it would be easy as you could switch to the target address space to revoke the desired mapping in the same way as the original VMM did. As it turns out though, Linux does not provide the needed context information when it calls the hooks.

That problem finally required additional modification to the guest kernel to intercept Linux's TLB flush events. The idea was that L4 mappings have TLB semantics, meaning that whenever Linux deletes or overwrites a mapping entry in the page table, the respective entry could still be cached in the TLB. So there is no need to instantly flush the corresponding L4 mappings. Rather, this can be delayed until Linux removes that entry from the TLB. The newly introduced TLB flush hooks provide the desired address space identifier. Unlike the original VMM, flushing is now deferred and done in a lazy manner.

Another challenge was to avoid a burst of page faults in the guest kernel areas. With the new design, each application has its own L4 address space with a user and a guest operating system area. Page faults in the user area are unavoidable and occur in both VMM designs at the same frequency. The number of page faults in the guest kernel area, however, differs significantly. Most mappings in there are global and constant. The original VMM caused

initial page faults, but eventually the kernel area in the separate operating system space was backed with memory mappings and the number of page faults in that area dropped down markedly. The new VMM can have hundreds of application spaces, each one with its own kernel area. Such application spaces are frequently created and deleted and along with them the mappings in the kernel area. The continuous page faults by the operating system would cause a huge performance penalty.

The introduction of an address space recycling mechanism helped to alleviate the problem. When Linux deletes an application, the corresponding L4 address space is put into a pool of free address spaces. Only the user area mappings and local memory mappings in the guest kernel area are deleted. When Linux creates a new application, a recycled address space out of the pool can be reused. Most kernel mappings and the in-place VMM are still there and way less page faults are triggered.

The handling of local memory mappings in the Linux kernel area was simplified by the distributed scenario. As opposed to global mappings, such local mappings differ from application to application. The old VMM with a dedicated operating system task required to keep track of them and to flush them on each address space switch. Following, when switching to the next application, the flushed local kernel mappings cause inevitable page faults in the operating system address space. The new design allows to preserve such mappings across address space switches.

4.5.2 Address Space Switch

With the original wedge design, the guest Linux operating system ran in its own L4 address space. One single thread executed the guest operating system as well as those VMM emulation functions the modified kernel called. The new design maps the identical guest operating system binary and in-place VMM into each guest application space. It thus requires one operating system thread per guest application space. However, at any point in time only one of them can be active. All other threads are waiting for an IPC.

When the guest kernel switches execution to another application, the guest operating system thread in the outgoing space needs to be halted and the incoming one must resume execution. The VMM has to preserve control flow across such a switch, which needs to be done transparently to the guest kernel. Consequently, the activated guest operating system thread must continue executing at the instruction following the IPC system call. Further, it has to adopt its predecessor's register state.

This high-level thread switch was implemented by means of an additional switch stack and L4 IPC. An address space switch causes the guest operating

system thread to execute an emulation function in the wedge. The thread then stores its register set (including the current stack pointer) on a separate switch stack that is used during the thread switch. Subsequently, it sends an empty message to the operating system thread in the address space to be activated and waits for an IPC. The now activated thread restores its state with the one from the switch stack and continues execution normally.

Operating system threads in newly created address spaces must be set up to wait for an activation IPC just as operating system threads in other inactive address spaces do. This is done in collaboration with their local pager thread and requires several privileged and non-privileged L4 system calls. In particular the privileged ones are expensive, for they can only be performed in collaboration with the privileged root task, i.e., the resource monitor. Thus, when recycling application spaces, the guest operating system thread, its pager thread, and one application thread are recycled as well. They are all properly set up to be easily reusable without further configuration.

4.5.3 Exception, Signal and Interrupt Handling

In combination, preemption messages, and control transfer allowed for an improved interrupt, signal, and exception handling in the virtual machine (cp. Section 3.6). Each Linux application thread has preemption messages enabled and has the local guest operating system thread for its scheduler. Signals can now be delivered at any point in time when the operating system thread executes. The required control transfer can easily be sent to the application in reply to the formerly received preemption, exception or page fault message.

Guest operating system threads, have the IRQ thread for their scheduler, with preemption messages enabled. Upon interrupts, while the guest kernel is active, L4 synthesizes an interrupt message and a preemption message to the IRQ thread. It thereby guarantees that interrupt messages arrive prior to preemption messages.

When receiving an interrupt message, the IRQ thread has two cases. If the guest kernel is idle (i.e., waiting for an IPC), no preemption message will follow. In this case, IRQ sends an IPC to the operating system, informing the thread to activate the guest kernel's interrupt handler. Otherwise, it marks the interrupt pending but waits to see the subsequent preemption message. Once the preemption message arrives, the IRQ thread replies with a new state in terms of a control transfer item and thereby forces the operating system to branch to its interrupt handler. Finally, in case of an interrupt while a Linux application is active, L4 sends an interrupt IPC to the IRQ thread and a preemption message to the local guest operating system thread. Following both, the IRQ and the operating system thread wake up. Since IRQ runs at

a higher priority and interrupt messages are guaranteed to arrive first, the IRQ thread executes first but only marks the interrupt pending. When the operating system thread continues, the interrupt is already marked pending and can easily be synthesized in the guest kernel.

Chapter 5

Evaluation

5.1 Measurement Methodology

All benchmarking was done on the same x86 test machine to get comparable and reproducible results. The computer was equipped with a 1.5 GHz Pentium 4, with 1 GB RAM and a 3Com 3c905C-TX Tornado 100 Mbit network interface card. An afterburned Linux 2.6.9, compiled with GCC version 3.3.5, was used as a guest operating system. It ran Debian 3.1 from a local ExcelStore J340 ATA disk. The extended L4Ka::Pistachio kernel is based on the CVS version from September 2006. Pistachio was configured to use the new mapping database while the fast IPC was turned off. The μ-kernel benchmarks in Section 5.2 are the median of five trials. The virtual machine benchmarks in Section 5.3 are reported with a 95% confidence interval (five independent benchmarks run) with no more than +/- 1% error.

5.2 μ-kernel benchmarks

To see how the virtualization extensions described in Chapter 3 affect the μ-kernel's IPC performance, both versions were compared by a standard L4 IPC benchmark called pingpong. Pingong sends short messages between two threads and measures the number of processor cycles and time as μ-seconds elapsed. The two communicating threads can execute in different L4 address spaces (Inter-AS) or share a common space (Intra-AS). The small space optimization on IA-32 allows them to execute in different virtual address spaces but within the same hardware address space (Small-AS).

Figure 5.1 compares L4, with small space optimization disabled, to the modified L4 kernel with all virtualization extensions enabled and small spaces

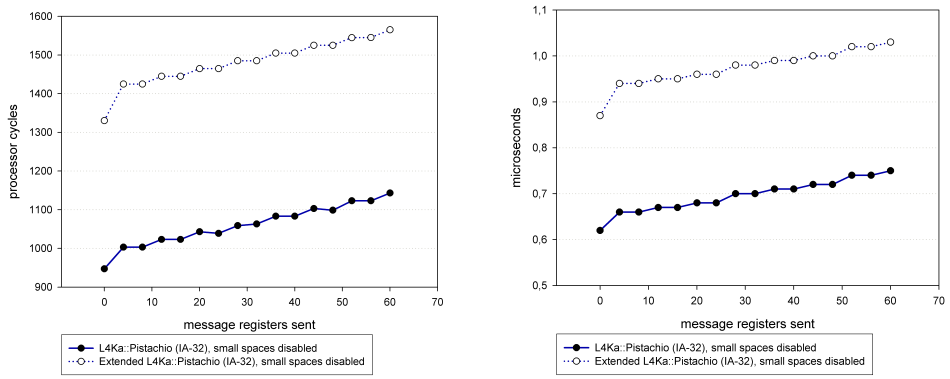


Figure 5.1: L4 μ-kernel with small spaces disabled (Inter-AS IPC)

turned off. All user threads in the system were set up to have the same 4GB address space limit. The graph shows the result of an inter-address space IPC. As the benchmark reveals, all extensions together add an average overhead of between 37% and 42%.

The reasons for this overhead could be easily identified by another test run, which compared L4 with an extended API and small spaces enabled to the unmodified Pistachio kernel, this time with small space optimization.

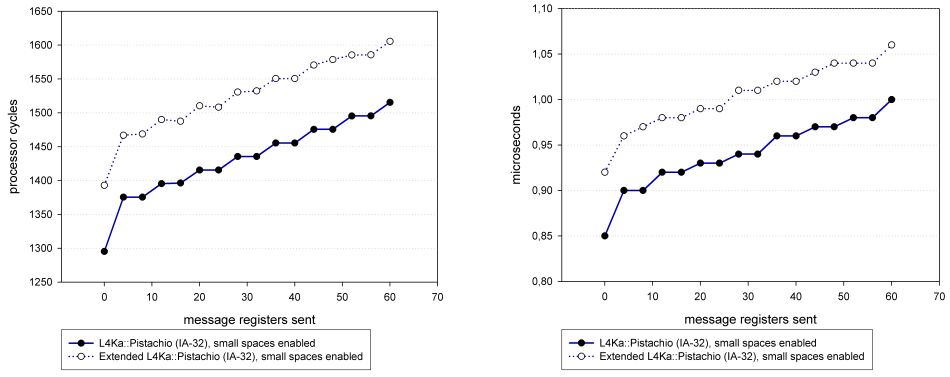


Figure 5.2: L4 μ-kernel with small spaces enabled (Inter-AS IPC)

As shown in Figure 5.2, the performance gap for inter-address space IPC then shrinks down to an overhead of between 6% and 8%. The overhead for intra-address space IPC is no more than 17% (Figure 5.3) while inter-address space IPC between threads in different small spaces reaches about the same performance as the original kernel (Figure 5.4). This is not surprising as the per-thread limit extension required similar kernel modifications to those used for L4's small space optimization. Even for the unmodified kernel, turning on this IA-32 specific feature causes an overhead of 32% to 37% for inter space

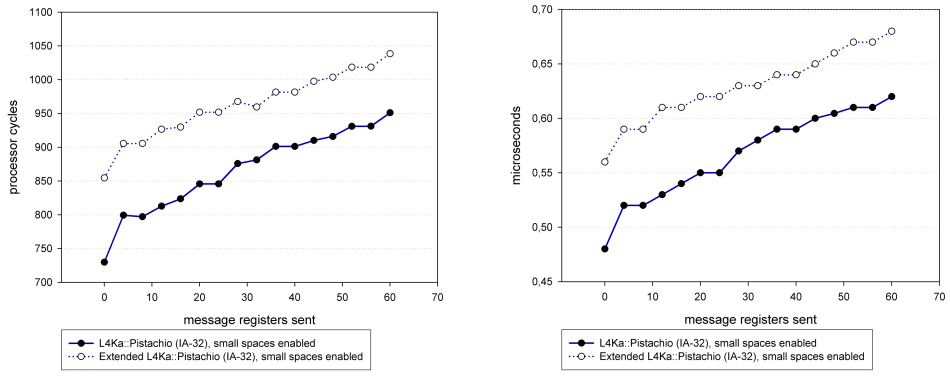


Figure 5.3: L4 μ-kernel with small spaces enabled (Intra-AS IPC)

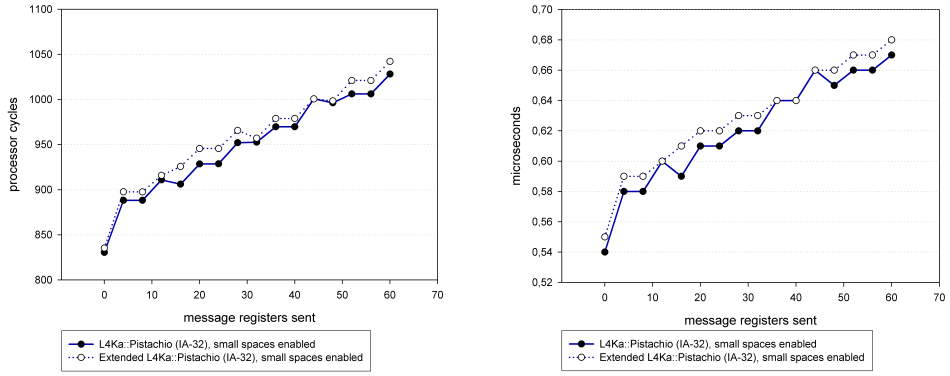


Figure 5.4: L4 μ-kernel with small spaces enabled (Small-AS IPC)

IPC. There are several reasons to be identified for this performance decrease:

1. L4 uses a GDT with privileged and user-accessible entries as is shown in Figure 4.4. User-accessible segments comprise a code, a data as well as a UTCB segment. Without the small spaces optimization, both user code and data segment are flat and cover the whole logical address space. Consequently, user code and data descriptors differ only with respect to their privilege levels from the kernel's. L4 uses paging to protect its 1GB kernel area in the upper virtual memory. The current IA-32 implementation thus avoids reloading data segment registers on kernel entries and exits. It simply stays with the user's and thereby decreases the overhead of kernel entries and exits that would be caused by the processor's expensive segment register privilege checks.

   This optimization cannot be used with the small space feature enabled. In that case the user's segment descriptors are no longer flat but are restricted to not exceeding the 3GB limit. L4, however, expects access

to the complete logical address spaces in kernel mode. It thus requires reloading the data segment registers on kernel entries/exits to point to the kernel's flat data segment. Indeed, it would be better to do so without small spaces enabled, too. With the current implementation, a malicious user could set DS to reference the UTCB segment descriptor before entering the kernel. In that case L4, as currently implemented, crashes.

2. With the small space optimization, L4 is forced to use a special kernel exit trampoline when using fast kernel entries and exits via *sysenter/sysexit*.

3. Small spaces further require additional thread switch logic as the kernel needs to update segment descriptor limits in the GDT on each switch to or from a thread in a small space.

All three points apply to the per-thread limit extensions as well and thus explain the major performance degradation. The remaining overhead is due to additional logic required inside the thread switch code. Further benchmarking revealed that a negligibly small IPC overhead is caused by all other extensions, including control transfer and preemption messages or the additional user state passed on each page fault.

5.3 Virtual Machine Benchmarks

This section compares the results achieved by the different virtual machine designs from the perspective of pure Linux applications. The interesting question to answer here is how L4's extensions affect the performance, and in particular whether co-location pays off. To answer it, the redesigned virtual machine was benchmarked in two different development stages. In the first stage, the original virtual machine was redesigned to make use of all extensions other than per-thread limits. Without co-location, the VM's general layout remained unchanged, i.e., there is one dedicated guest operating system space and several application spaces as shown in Figure 2.4. In the second stage, per-thread limits are enabled and the guest operating system is mapped into each application's address space (see Figure 3.5).

The virtual machine's performance results are compared to the guest operating system running natively on raw hardware. As the starting point ran a single operating system on hardware with direct device access. In order to obtain comparable results, the same configuration was used in the virtual machine environment. A single VM entity ran afterburned Linux as guest with direct device access. All benchmarks used the slow legacy *int 80* system call invocation.

The resulting runtimes were derived from the processor's time-stamp counters. The number of system calls invoked was traced by *strace*, which was configured to include child processes as they are created.

5.3.1 Microbenchmarks

To measure the system call overhead, *getpid* was repeatedly called in a loop. Table 5.1 shows the consumed time and the number of cycles per invocation. The overhead relative to the native system is shown in the last column.

System	Time	Cycles	Overhead
Native Linux	1.022 μsec	1530	-
Original VMM	4.102 μsec	6155	4.01x
Redesigned VMM Stage 1	4.422 μsec	6637	4.33x
Redesigned VMM Stage 2	3.250 μsec	4875	3.18x

Table 5.1: *getpid* system call costs on the different implementations

The second microbenchmark allows to estimate the time required for an address space creation and its deletion. It is a small program that continually creates and deletes child processes using Linux's *fork* system call. The child processes, when started, call *exit* right away. Table 5.2 shows the results of this benchmark.

System	Time	Overhead
Native Linux	0.128 msec	-
Original VMM	1.225 msec	9.59x
Redesigned VMM Stage 1	1.296 msec	10.15x
Redesigned VMM Stage 2	0.720 msec	5.64x

Table 5.2: Costs for the creation and deletion of a single process

The two benchmark results reveal what could be expected for the different VMM stages. The original design entails a large system call overhead. System calls via *int 0x80* cause L4 to synthesize an exception IPC to send it to the guest operating system in its separate address space. Subsequently, the VMM synthesizes a software interrupt in Linux before the guest operating system thread can execute the associated exception handler. After the interrupt is handled, the guest kernel returns to the application. In total, each system call requires at least two L4 address space switches.

The first stage of the redesigned VMM is about 5.8% slower than the original VMM. The reasons for this overhead are easy to identify. First, L4 has preemption messages enabled and the wedge activates them on all application and guest operating system threads. It takes time to compose and send the

additional messages. Second, page faults as well as preemption messages now contain the complete IA-32 general-purpose register set. Thus, the time L4 requires to compose and transfer the larger messages increases. The additional overhead is acceptable, as one has to keep in mind that the original wedge didn't handle page faults or signals properly. Trying to do so without kernel support would have a greater adverse effect on performance (cp. Section 2.4.4).

The second redesigned VMM reduces system call overhead for *getpid* by about 21%. Since there is only a single application space, address space recycling cannot affect the result. The performance boost is an immediate consequence of co-location, i.e., of the reduced kernel entry and exit overhead. The second benchmark performs 1.7 times faster than the original VMM. The improvement results from the combination of fast kernel entries/exits and address space recycling. The latter one plays the major role. With address space recycling turned off, the benchmark performs several times slower.

5.3.2 Macrobenchmarks

Three complex application benchmarks, which exercise the whole system, have been employed to investigate the performance under different workloads. The first one, Povray, is a raytracing program. It calculates the results of light rays that bounce around in a virtual world. It comes with various example scenes. In this benchmark, Povray Version 3.6 rendered an advanced scene called "chess2" with standard configuration. It is a highly CPU-bound benchmark that generated only one single process and no more than about 3.5K system calls.

System	**Time**	**Overhead**
Native Linux	89,05 sec	-
Original VMM	89,66 sec	1.01x
Redesigned VMM Stage 1	90,04 sec	1.01x
Redesigned VMM Stage 2	89,50 sec	1.01x

Table 5.3: Povray results

Table 5.3 shows that the results are almost identical and yield no information about how the μ-kernel extensions and the redesigned VMM affect performance.

The second benchmark, Netperf, is an I/O-intensive benchmark. It was configured to transfer 1GB of data at standard Ethernet packet size to a second host in the local network. The remote host was a multi-processor workstation, configured with two 2GHz AMD Opteron processors, 2GB of RAM, a gigabit ethernet controller and ran native Linux 2.6.18. During the send phase about

64K system calls were invoked.

The third benchmark compiles the Linux kernel Version 2.6.9 with minimal configuration using GCC Version 3.3.5. It creates around 3.5K processes, exercising *fork* and *exec* and thereby strains the memory subsystem. Further it generates about 847K system calls. The respective results are shown in Table 5.5 and Table 5.4.

System	**Time**	**Overhead**
Native Linux	88,09 Mb/sec	-
Original VMM	86,44 Mb/sec	1.02x
Redesigned VMM Stage 1	85,77 Mb/sec	1.03x
Redesigned VMM Stage 2	78,85 Mb/sec	1.12x

Table 5.4: Netperf results

System	**Time**	**Overhead**
Native Linux	243,35 sec	-
Original VMM	295,96 sec	1.22x
Redesigned VMM Stage 1	305,84 sec	1.26x
Redesigned VMM Stage 2	324,73 sec	1.33x

Table 5.5: Linux kernel build results

The performance hit by the first redesigned VMM is for the same reasons as those stated above. The results achieved by the second stage are particularly surprising as it performs worse than all others. Since the first stage completes remarkably better, the deterioration must have been caused by one of the following modifications that were additionally made to the second stage:

1. The guest operating system is mapped into each application's address space. Each address space holds an additional operating system and a pager thread. The number of L4 threads in the system thus greatly increases.

2. The application switch is more complicated compared to the original VMM design. Each one results in an IPC and requires the VMM to preserve continuity across the address space switch.

3. Per-thread limits are activated in L4.

4. The distributed physical memory management is more complex. Unlike with the old wedge, memory mappings in a specific application's space can only be unmapped from within that space. Further, TLB flush hooks had to be introduced. They were required to inform the VMM what address space a page needing to be unmapped belongs to (cp. Section 4.5.1).

To start with the first one, the number of L4 threads in the system was traced during a Linux kernel build. The maximum number of simultaneously existing L4 thread's during the benchmark was less than one hundred. The same holds for Netperf. L4, however, was designed to support up to 2^{17} user threads on IA-32. It can thus be ruled out that a possibly huge number of user threads decreased the μ-kernel's overall performance. The second difference, which calls for closer investigation, is the application switch. However, measuring the time spent in the switch routines (using the Pentium performance counters) revealed that this cannot be the reason for the large overhead. During the Linux kernel build (which lasts between 4 and 5.5 minutes depending on the VMM), the time spent there was less than a second.

The overhead is thus likely to be caused by the per-thread limit extension as well as the required modifications to the memory management. In fact, turning off the per-thread limit extension (while the guest operating system remains unprotectedly mapped into application spaces) perceivably improved the Netperf benchmark result. Additionally, earlier μ-kernel benchmarks support the thesis that L4's performance, and thus the VMM, suffers from that extension (cp. Section 5.2). In particular it has negative effects to IPC-bound workload. The more interesting finding, however, is related to the VMM's distributed memory management. Table 5.6 prints the number of three different kinds of page faults counted during a Linux kernel build. Application page faults are those caused by all application threads in their respective 2GB space. Kernel-space page faults include all page faults caused by operating system threads. The last column summarizes the number of physical memory requests that were sent to the resource monitor.

System	**Application**	**Kernel**	**ResMonitor**
Original VMM	2,782,000	858,000	214,000
Redesigned VMM Stage 2	7,600,000	853,000	140,000

Table 5.6: Page fault statistics

The interesting point here is the conspicuously high number of application page faults created by the second stage VMM. Compared to the original VMM, there are almost three times as many. The only conceivable explanation is that L4 mappings in application spaces are much more frequently revoked. In fact, the unmap mechanism has changed. While the old wedge pro-actively removes L4 mappings as soon as Linux changes the corresponding page table entries, the redesigned VMM is forced to use the TLB hooks for the reasons described in Section 4.5.1.

Consequently, the next step of investigation was to determine how frequently each of the different TLB hooks is called. Ideally, Linux would invoke TLB hooks to selectively remove translation entries that correspond to those previously invalidated in the page tables. It would also batch adjacent page

invalidations and eventually call a hook to flush the affected area. However, it doesn't. Rather, it frequently flushes the whole TLB even though only few page table entries have changed. It thereby forces the VMM to revoke all mappings in a particular application space. This naturally results in a huge number of subsequent page faults. As it turned out, Linux for IA-32 is not optimized to support selective TLB flushes, as opposed to other ports for architectures whose TLB entries are tagged by additional address space identifiers. Reducing the number of expensive page faults to a reasonable value would require additional modifications to Linux/IA-32. The TLB flushing needs to be done in a more fine-grained manner.

In summary, it is the adverse effect of per-thread limits and Linux/IA-32's TLB policy that is likely to cause the overhead. A final proof of this assumption requires additional Linux adaptation. An earlier optimization, however, illustrates how adversely such an extraordinary number of page faults can affect the virtual machine's performance. Without the address space recycling optimization, as introduced in Section 4.5.1, every new application startup will be delayed by frequent page faults for the guest kernel area. A continuous creation and deletion of applications thus causes an inordinate number of page faults by guest operating system threads in the virtual machine. In fact, the Linux kernel build finished in no less than 380 seconds without the optimization compared to 324 seconds otherwise.

Chapter 6

Conclusion

The following sections briefly summarize this thesis and point out its open questions. They also give ideas for future work that is based on the results.

6.1 Summary

This thesis was motivated by comparable work on how to alleviate the overhead of host-based virtual machines. It claims that L4's interface lacks transparency and thus does not allow to construct virtual machines with the same efficiency as specialized hypervisors.

The thesis proposed transparency-increasing extensions to improve virtualization on L4. Thread limits offer mechanisms that are similar to a processor's privilege modes and allow for natural co-location of the guest operating system and its applications. Preemption messages and zero time slices allow virtual machine monitors to avoid the influence of the L4 scheduler and enable the guest OS to fully control scheduling its applications. The additional user state passed along page fault and preemption messages can be efficiently passed to the guest operating system. Finally, control transfer messages provide a novel way to redirect control flow in reply to synchronous and asynchronous events.

The L4Ka Virtual Machine Environment was redesigned to make use of L4's extended API. A reference implementation for IA-32 was also provided. Its evaluation and the μ-kernel's performance was two-sided. On one hand, the extensions allow for a more complete and correct virtual machine construction. Additionally, the redesigned virtual machine can reduce the kernel entry/exit overhead. On the other hand, L4 forfeits IPC performance in consequence of how thread limits are implemented. Consequently, the more transparent API is well suited to a VMM's usage patterns but adversely affects other systems

that build on top of L4. However, it also became apparent that L4's IA-32 port is currently faster than it should be. With respect to the virtual machine's performance, macrobenchmarks revealed that complex user applications with a high memory consumption suffer significantly from the distributed memory scenario in conjunction with Linux's TLB flush policy.

6.2 Future Work

This work proposed extensions to solve or alleviate most, but not all difficulties L4 poses to virtual machine construction. In particular, access bit virtualization and address space construction that rely on shadow page tables remain an open issue. Some of the proposed extensions turned out to have adverse effects on the μ-kernel's overall performance. Other extensions worked out for specific architectures but are generally insufficient. Based on the results of this thesis, future work is required to improve the value of these extensions and to reduce their overhead. In particular IPC, as the key L4 mechanism, must not significantly suffer from any virtualization supporting modification.

Some μ-kernel enhancements, as implemented, lack support for fast IPC. However, it should not be difficult to add the required logic. The scheduler of the baseline L4/IA-32 turned out to not properly handle delayed preemption. L4 threads with delayed preemption that execute on donated time slices are spuriously preempted. Additionally, the preemption flags are not always set correctly. For the evaluation of this work, some of the errors were tentatively fixed, but the scheduler needs further revision to make sure it does handle all cases as expected.

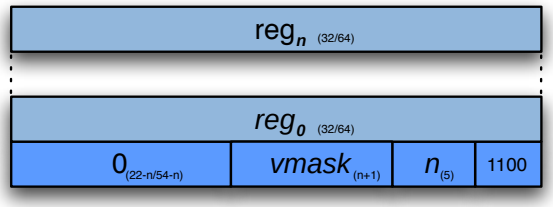
Future work is required to modify the guest operating system. The TLB flushes are too coarse-grained and cause an extraordinarily high number of application page faults in the virtual machines. Unless the number of page faults in the virtual machines are not roughly equal, it is impossible to conclude whether or not co-location pays off for a wide range of applications. Another optimization, which would likely improve performance, is to outsource the interrupt thread into its own task and to concurrently use L4's small space optimization when available. On IA-32, the additional overhead caused by the small space logic should be negligible as it can piggyback on the costs of the thread limit extension. Hardware interrupts then would not imply hardware address space switches, further accelerating the virtual machine.

Appendix A

API Version *X.2* Extensions

A.1 Control Transfer Items

A control transfer item is a typed message item that specifies parts of the register state of a thread. Typically it contains registers required to change the control flow of a program (e.g., instruction pointer and stack pointer).



n	The number of register values in the control transfer item - 1.
$r_0..r_n$	The register values to overwrite the register state with. The register corresponding to a specific register value r_k is architecture dependent. A register value r_k is ignored if bit k in *vmask* is cleared.
vmask	Bitmask indicating which register contents to modify. If bit k is set the register value r_k is written into the corresponding register. if bit k is cleared the register value r_k is ignored.

Control transfer messages allow to reset a thread's registers by means of common IPC. To allow for a control transfer, the receiver must set a corresponding

bit in its acceptor and perform a receiving IPC system call. The new acceptor was accordingly complemented.

c	Control transfer is accepted iff $c = 1$.

If the kernel is configured to support control transfer, L4 automatically accepts control transfer items in reply to page fault, exception, and preemption messages.

A.2 Preemption Messages

A preemption message is a reliable IPC call from a just-preempted thread to its scheduler. Threads with preemption messages activated case L4 to synthesize a message and to send it to the thread's scheduler in all of the following events:

1. The thread's time slice has expired and L4 chooses to schedule another thread.

2. A higher priority thread woke up (to which L4 instantly switches).

3. A hardware interrupt was triggered and L4 decides to schedule another thread.

Subsequently the preempted thread waits for the receiver's response, i.e., L4 will not reschedule the thread unless the scheduler replies with an empty message.

Preemption messages include the time of preemption. The kernel can also be configured to attach the suspended thread's execution state, either in terms of untyped words or as a control transfer item.

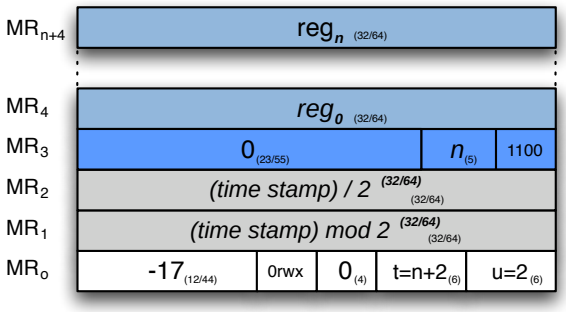


Figure A.1: Preemption message layout

A.3 ThreadContol

ThreadId	*dest*	$\longrightarrow$	*Word result*
ThreadId	*SpaceSpecifier*		
ThreadId	*scheduler*		
ThreadId	*pager*		
*void**	*UtcbLocation*		
Word	*limit*		

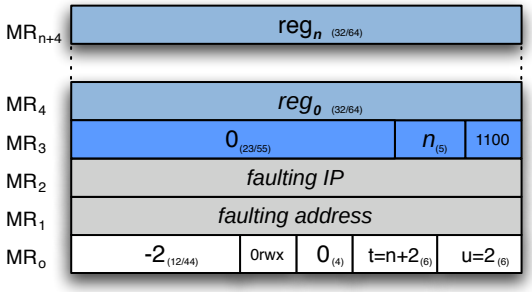
The system call has an additional parameter *limit*. It allows to restrict the destination thread's accessible memory region within its virtual address space. All other input parameters and the result are not affected.

Input Parameter *limit*

limit specifies the upper limit of the thread's accessible memory region, which starts at address zero. A value of -1 indicates no restriction. Otherwise, the limit is a multiple of 4K blocks, i.e., the thread's highest accessible memory address is 4K * (limit + 1) - 1. For specific processors, it can be a multiple of a larger memory unit. A thread trying to access memory above its limit causes an exception.

A.4 Page Fault Protocol

The kernel can be configured to support an extended page fault protocol. Other than the faulting address and the instruction pointer, the extended page fault message additionally contains an architecture dependent set of the faulting thread's register state.



If the kernel is configured to support control transfer, the format of the additional state passed in message registers matches a control transfer item. The kernel guarantees that no real control transfer will take place, even if the pager has its acceptor set appropriately.

A.5 Schedule

ThreadId	*dest*	$\longrightarrow$	*Word result*
Word	*time control*		*Word time control*
Word	*processor control*		
Word	*prio*		
Word	*preemption control*		

The system call allows to activate preemption messages on a per-thread basis.

Input Parameter *preemption control*

If p is set, preemption messages are enabled on the target thread. Otherwise they are disabled.

Preemption messages can be enabled or disabled at any point in time. Modifications become immediately effective. If preemption messages are enabled on threads that have not yet received their startup message the thread start protocol (cp. [26]) changes. A newly created thread with preemption messages enabled starts by receiving a message from its scheduler, as opposed to its pager. The startup message format remains unchanged, i.e., it contains an initial instruction and stack pointer. If the kernel is configured to support control transfer, the message can alternatively be a control transfer item to specify the thread's startup state.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM Press.

[2] Neil Brown et al. The Linux Virtual File-system Layer, 1999.

[3] K. Buchacker and V. Sieh. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects, 2001.

[4] Microsoft Corp. Microsoft Virtual PC 2004 technical overview, 2005. *http://www.microsoft.com/windows/virtualpc/evaluation/techoverview.mspx*.

[5] Microsoft Corp. Microsoft Virtual Server 2005 Technical Resource Overview. *http://www.microsoft.com/windowsserversystem/virtualserver/techinfo/default.mspx*.

[6] IBM Corporation. Dynamic Logical Partitioning. *http://www-03.ibm.com/servers/eserver/iseries/lpar/*.

[7] IBM Corporation. z/VM built on ibm virtualization technology. *http://www.vm.ibm.com/techinfo/*.

[8] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[9] Advanced Micro Devices. AMD64 Virtualization Codenamed "Pacifica" Technology. *Secure Virtual Machine Architecture Reference Manual*, 2005.

[10] Edsger W. Dijkstra. The structure of the the multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[11] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[12] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[13] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[14] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.

[15] The Open Group. IEEE Std 1003.1,2004 Edition. *http://www.unix.org/version3/ieee_std.html*.

[16] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, St. Malo, France, October 5–8 1997.

[17] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *USENIX Annual Technical Conference, General Track*, pages 71–84, 2003.

[18] Adam Lackorzynski. L^4Linux Porting Optimizations, Diploma Thesis, Faculty of Computer Science, Technische Universität Dresden. 2004.

[19] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-Virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[20] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[21] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[22] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Commun. ACM*, 18(7):401–408, 1975.

[23] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[24] Marc Rozier, Vadim Abrossimov, François Armand, I. Boule, Michel Gien, Marc Guillemont, F. Herrmann, Claude Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[25] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine monitor.

[26] The L4Ka Team. L4 eXperimental Kernel Reference Manual. Version X.2, August 2006.

[27] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.