Fiasco μ-Kernel User-Mode Port

Udo Steinberg
us15@os.inf.tu-dresden.de

Dresden University of Technology
Institute of System Architecture

December 19, 2002

Linux is a registered trademark of Linus Torvalds. UNIX is a registered trademark of The Open Group. Solaris is a registered trademark of Sun Microsystems Inc. Windows is a registered trademark of Microsoft Corporation. VMware is a registered trademark of VMware, Inc. All other product names and trademarks are recognized, and are the property of their respective owners.

Contents

1 Introduction **5**

2 Fundamentals **7**

2.1 Process Tracing with the ptrace Interface 7

2.1.1 Enabling Process Tracing 7

2.1.2 Tracing Modes 7

2.1.3 Inspecting and Modifying Registers 8

2.1.4 Inspecting and Modifying Program Memory 8

2.2 User Mode Linux 9

2.2.1 UML Design 9

2.2.2 Ring Transitions 10

2.2.3 UML System Calls 10

2.2.4 Memory Protection 10

2.2.5 Task Switches 11

2.2.6 Interrupts 11

2.2.7 Performance 12

3 Design **13**

3.1 Virtual-Memory Layout 14

3.2 FIASCO-UX Startup 14

3.2.1 Constructors 14

3.2.2 Physical Memory 15

3.2.3 Loading of ELF Modules 15

3.2.4 Kernel Initialization 16

3.3 Changes to the Kernel Core 16

3.3.1 Tasks and Host Processes 16
3.3.2 Page Tables 17
3.3.3 Page Fault Handling 17
3.3.4 User Memory Access 17
3.3.5 Privileged Instructions 18
3.3.6 Kernel Lock 18
3.3.7 FPU Handling 19
3.4 Limitations 19

4 Implementation 21
4.1 Task Creation 21
4.2 Address-Space Manipulation 22
4.3 Interrupts, Signals 23
4.4 Signal Stack 24
4.5 Kernel Mode 24
4.5.1 Privileged Instructions 25
4.5.2 Interrupts in Kernel Mode 26
4.5.3 Kernel Mode Page Faults 26
4.5.4 Ring Transitions 26
4.6 User Mode 27
4.7 Race Conditions 28

5 Performance Analysis 29

6 Conclusions 31

7 Summary 33

A Glossary 35

B Bibliography 37

Chapter 1

Introduction

Today's operating systems can be divided into three main groups. There are **monolithic systems** such as Linux, Solaris and Windows which implement all operating system abstractions, including device drivers, in kernel space. Contrary to this approach, the **exokernel systems** such as Aegis [EKT94] are almost devoid of any functionality and only implement a few stubs in kernel space. Exokernels hide no resources behind abstraction layers or a set of trusted servers. Instead applications can directly request physical memory pages or blocks on a disk; the exokernel merely ensures the protection of these resources or makes sure they are free. Typically exokernel systems are augmented by a library operating system which user mode applications are linked against. Multiple such libraries can coexist in an exokernel system, for example one library providing a Unix API and another providing a Windows API.

The research at Dresden University of Technology [TUD] is focused on the third group, **microkernel systems**, in particular the FIASCO [Hoh98] L4 μ-kernel. Such μ-kernels only implement a minimal set of necessary abstractions in kernel space upon which operating systems can be built. The remaining functionality is implemented in user space. This approach makes such systems more robust, because a single failing service can simply be restarted without having to take the entire system down. Additionally, very little code actually runs with kernel level privileges, which shrinks the trusted computing base considerably and makes systems more secure. There are even projects trying to prove the correctness of the kernel code, such as the VFiasco project [HT01]. Furthermore, due to their small size, μ-kernel systems have a lower memory and cache footprint.

The first μ-kernels like Mach [RJO+89] were slow and not very small; however, the first L4 μ-kernel [Lie95] was only a few kilobytes in size. FIASCO is another L4 μ-kernel with realtime properties, written in C++. It implements message-based synchronous IPC[1], external paging mechanisms and security mechanisms based on secure domains (tasks, clans and chiefs).

As with any other software, the development of a μ-kernel requires a lot of testing. Typically kernel developers have a machine for coding and another machine for testing their kernel. During the

[1] Inter Process Communication

development process the test machine needs to be rebooted over and over again which is a tedious and time consuming task. This was one of the reasons which led to the proposal of developing a version of the FIASCO kernel which runs entirely in user mode as a normal Linux program [Sch01]. A user space port of the kernel would allow developing and testing code on the same machine without having to reboot. It even makes it possible to run multiple such kernels simultaneously on the same machine. Anyone with an x86-based Linux machine could then run and test the FIASCO kernel and develop software for it.

The work which is presented in this document is **FIASCO-UX**, a port of FIASCO to the Linux system call interface. Throughout the development of this kernel port I tried to leave most of the original FIASCO core code unchanged. This was achieved by adding a small emulation layer on top of FIASCO which abstracts certain hardware functionality and implements it with Linux system calls. The lack of hardware access also required to make small modifications in other parts of the kernel code.

With the release of FIASCO-UX, the L4 community gains a μ-kernel that runs entirely in user mode and which executes unmodified L4 binaries on virtually any x86-based Linux system.

Terminology

This document deals with the emulation of kernel functionality in user mode. In order not to leave the reader confused, the meaning of several terms used in this document shall be defined as follows:
The **host kernel** is the native Linux kernel under which FIASCO-UX runs. A **task** is an emulated protection domain that consists of an address space and activities, **threads**, which execute in that address space. The representation of a task in the host kernel is called **host process**.
Physical memory refers to the emulated physical memory in the user mode kernel and not to the physical memory in the host kernel. Similarly, **virtual memory** refers to the emulated virtual memory. **Kernel mode** refers to the notion of a privileged context in a user mode kernel, which emulates the kernel context of a native kernel, even though that context actually runs in user mode from the processor's point of view.

When talking about the host kernel's memory or the host kernel's kernel mode, this is explicitly stated in the text.

Chapter 2

Fundamentals

2.1 Process Tracing with the ptrace Interface

All emulators and programs striving to virtualize hardware and kernel functionality either modify the underlying operating system directly, for example by loading kernel modules, or rely on support from the operating system which allows the trapping of function calls that need to be emulated. For this purpose the Linux kernel provides the `ptrace` interface[1], which shall be examined in this chapter.

2.1.1 Enabling Process Tracing

For security purposes the ability to trace processes is limited. A parent process can trace any of its child processes. The child can request to be traced by its parent by using the `PTRACE_TRACEME` option. Alternatively the parent process can attach to a child process via the `PTRACE_ATTACH` option. The latter method also works for unrelated processes if the attaching process has the necessary access rights. In that case the tracing process temporarily becomes the parent process of the traced process, but this fact is hidden from both processes, so that for example `getppid` still returns the expected value. The tracing and its effects can be undone from inside the tracing process using the `PTRACE_DETACH` option. Alternatively the parent process can use `PTRACE_KILL` which delivers a `SIGKILL` signal to the child process. Since this signal cannot be caught by the child process, the child will cease to exist.

2.1.2 Tracing Modes

The ptrace interface offers three different tracing modes to choose from, depending on what level of tracing is desired. After invoking ptrace with one of the three options, execution of the child process resumes. When the child process stops due to a tracing event, the parent process will be notified with a `SIGCHLD` signal. A subsequent call to `wait` or `waitpid` allows the parent to obtain the signal number that caused the child process to stop. The tracing process can choose to cancel the signal by resuming the child process with no continuation signal. In that case the child process will never see

[1] Several other Unix operating systems provide a similar interface, however ptrace is not part of the POSIX standard.

the signal. Alternatively the tracing process can choose to forward or alter the signal by resuming the child process with the desired continuation signal. The following three tracing modes are currently available:

- `PTRACE_CONT`
  The child process continues until it receives a signal. Upon reception of a signal, execution stops and the parent process receives a `SIGCHLD` signal, with `wait` indicating the reception of the child's signal.
- `PTRACE_SYSCALL`
  The behaviour is the same as for `PTRACE_CONT`. Additionally the child process will stop upon execution of a Linux system call (`int 0x80`). Two such tracing events will be generated - one upon entering the kernel and one after exiting from kernel mode. This allows the parent to inspect the system call parameters at the first stop and the return value at the second stop. Both times the parent process will be sent a `SIGCHLD` signal, with `wait` indicating the reception of a `SIGTRAP` signal by the child process.
- `PTRACE_SINGLESTEP`
  The behaviour is the same as for `PTRACE_CONT`; however, the child process will also stop after every single machine instruction. Again the parent process will be notified with a `SIGCHLD` signal, with `wait` indicating the reception of a `SIGTRAP` signal by the child process.

It should also be noted that if the child process stops with a `SIGTRAP` signal, the tracing process cannot easily distinguish between a stop due to a system call, a single step, or an `int3` instruction. It must either explicitly remember which tracing mode is currently used, or perform an opcode analysis at the instruction pointer in the child process.

2.1.3 Inspecting and Modifying Registers

Only when a traced child process has been stopped can the tracing process inspect and modify the register set of the child process. The `PTRACE_GETREGS` option returns a structure containing the entire register set. Similarly the `PTRACE_SETREGS` option allows the parent to set all registers with the values specified in the passed structure. The Linux kernel performs additional sanity checks on the values of these registers. Any attempt to modify the privilege level of a segment selector will result in an error. Modifications to privileged bits in the `EFLAGS` register will be silently discarded by the Linux kernel.
Similar functions exist for floating point registers. The `PTRACE_GETFPREGS` function retrieves the floating point state of a traced process whereas `PTRACE_SETFPREGS` can be used to modify floating point registers.

2.1.4 Inspecting and Modifying Program Memory

Like the options that manipulate the contents of the registers, Linux offers the functionality to read and modify the data and code of a traced process. The `PTRACE_PEEKDATA` option copies a word

from the specified address in the child's address space into the tracing process' memory. The PTRACE_POKEDATA option can be used to copy a word from the parent's memory to the traced child's address space. The options PTRACE_PEEKTEXT and PTRACE_POKETEXT perform the same operations on text segments; however, in Linux they are merely aliases for the first two options, because no distinction is made between text and data segments.

2.2 User Mode Linux

User Mode Linux [UML] was designed and implemented by Jeff Dike. It is a complete Linux kernel that runs entirely in user mode. The UML kernel runs unmodified user code and only differs from native Linux in the way it handles system calls and devices. UML does not support native devices. Instead it provides a number of virtual devices that are emulated entirely in software, such as consoles, block devices, serial devices and network devices.

2.2.1 UML Design

User Mode Linux uses a memory layout that was designed for good system call performance. The upper area of the user address space in each UML process is reserved for the UML kernel as shown in Figure 2.1.

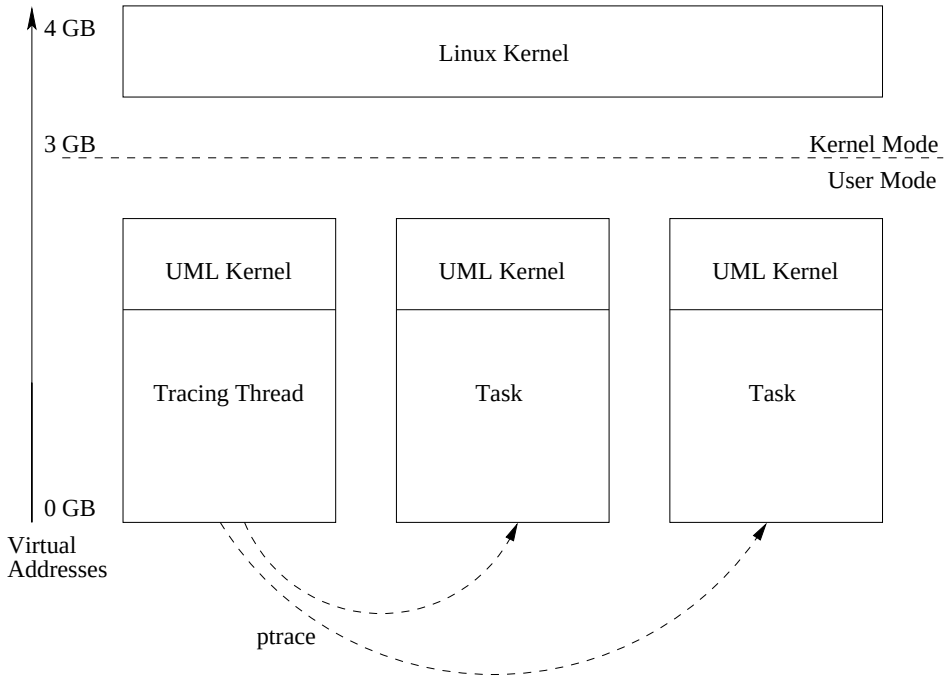

Figure 2.1: UML Tracing Relationship

As a result the usable virtual memory size for user tasks decreases from 3 GB to 2.5 GB. The kernel code and data are shared between all user tasks; each respective virtual memory area is backed by the same physical pages.

2.2.2 Ring Transitions

The notion of a privileged kernel mode and an unprivileged user mode in User Mode Linux is emulated by a special host process, called the UML tracing thread. It traces all other UML tasks. While a task is executing in user mode, it is under tracing control by the tracing thread. When such a task raises an exception, trap, or fault, it will receive a signal from the Linux host kernel, which is intercepted by the tracing thread because it is tracing that task. Additionally the tracing thread will receive a `SIGTRAP` signal when a task attempts to execute a Linux system call. If execution is to continue in kernel mode, the tracing thread exchanges the user mode context of the task with a kernel mode context and turns system call tracing off. The task then executes in `PTRACE_CONT` mode rather than `PTRACE_SYSCALL` mode. The transition back to user mode is also done by the tracing thread, but it is requested by the kernel context. Similar to an `iret` call, the kernel context raises a `SIGUSR1` signal, which signals the tracing thread to switch back to the user context and turn system call tracing back on.

2.2.3 UML System Calls

System calls under User Mode Linux trap directly into the native Linux kernel. However, UML requires that system calls be executed in the UML kernel instead of the native Linux kernel. When a task attempts to execute a system call, the tracing thread receives a `SIGTRAP` signal. Because the system call should not be executed in the native Linux kernel, UML changes the system call number to `getpid`, which is the least expensive and intrusive system call, since it does not change the system state. It is not possible to prevent a system call from trapping into the Linux kernel, because at the time the tracing process receives notification, the task has already entered the Linux kernel. After cancelling the system call in the Linux kernel, UML emulates it in the UML kernel and then passes control back to the task. The `fork` system call is a special case, because a new process must be created in the host. UML changes a `fork` system call into a `clone` system call with the appropriate parameters to duplicate the calling task. The new task starts in a trampoline which sets up the necessary timers and signal handlers and then goes to sleep until the kernel sets up the correct context and tracing for the new task. While a task runs in kernel mode on its kernel stack, system call tracing for that task is turned off by the tracing thread. This allows the kernel context to make system calls directly, without being intercepted by the tracing thread and without the system call being changed to `getpid`.

2.2.4 Memory Protection

Having the kernel and task in the same address space makes kernel entries fast compared to designs which put the kernel in a different address space (and thus in a different host process). All system

calls and interrupts translate to signals whose handlers warp the calling process onto its kernel stack (signal stack) in the kernel area with the help of the tracing thread. Returning from the signal handler is equivalent to returning from kernel to user mode. This design, which simplifies system calls a lot, is also a major security concern. Kernel code and user code reside in the same address space and must be isolated from each other. Otherwise any user program could overwrite kernel code and modify or crash the kernel. When returning from kernel mode, the entire kernel memory area must be protected against read and write access. Similarly, when entering kernel mode the kernel memory area must be unprotected. The `mprotect` operations are very expensive since they require page table entries in memory to be updated. Having to do these protect/unprotect operations on each kernel entry slows down system calls to a crawl, which is why memory protection is turned off by default. It can be enabled by setting the `jail` kernel parameter when starting UML.

2.2.5 Task Switches

Context switches from one task to another happen on the kernel stack of the current process. All currently inactive tasks block on their switch pipe by attempting to read a byte from it. The outgoing task, that is the task which is currently running, writes a byte into the switch pipe of the incoming task, that is the task which is going to run next, and then puts itself to sleep by reading from its own switch pipe. The incoming task wakes up when a byte arrives for reading at its switch pipe. While it was asleep the mapped kernel memory might have changed while executing in a different task. For example, a slab allocator could have mapped a new page in kernel memory. Therefore the incoming task must fix its kernel address space by scanning the page tables that cover the UML kernel's virtual memory, removing any pages that are no longer mapped, adding newly mapped pages and updating page protection attributes. The same scanning must be done for the virtual user memory areas due to lazy page management across tasks. When the kernel adds, modifies or revokes a page in the address space of a different task, the modification is only made in the respective task page table and not immediately carried out as an `mmap`, `mprotect` or `munmap` operation, because that would incur additional context switches. When the kernel switches to a task later on, the page tables are scanned for updates, and modifications are carried out in the current address space.

2.2.6 Interrupts

User Mode Linux implements device interrupts with the `SIGIO` signal. Each device driver that generates interrupt events ensures that, at the same time, a `SIGIO` signal is generated for the currently executing context. To achieve this, each interrupt line is associated with a pseudo terminal. UML enables asynchronous I/O notification on the PTY's read descriptor so that whenever a device driver writes to the write descriptor of the PTY, one process will receive a `SIGIO` signal. Which process receives the `SIGIO` signal is determined by the descriptor owner, which can be set using `fcntl` and which UML always sets to the current task. The reception of a `SIGIO` signal by the currently executing task forces the task onto its signal stack, which functions as the task's kernel stack. The UML kernel then proceeds to call the standard IRQ handlers in the kernel code. The timer interrupt is a special case as it is not implemented using the `SIGIO` signal. The Linux `setitimer` system

call provides access to different timers. `ITIMER_REAL` is a timer which decrements in real time and delivers a `SIGALRM` signal upon expiration, whereas `ITIMER_VIRTUAL` only decrements when the process is executing and delivers `SIGVTALRM` when it expires. The timer interrupt for tasks is implemented using `SIGVTALRM`, so that a timer interrupt occurs based on UML's internal notion of execution time, rather than the host kernel's timing which also accounts for the time consumed by other processes in the host. The idle loop is a special case, because it blocks the kernel using `sleep`. While the kernel sleeps, the virtual timer is not decrementing and the kernel would never wake up. For this reason the kernel's idle loop uses `SIGALRM` as timer interrupt.

During a task switch, the UML kernel changes the ownership of each IRQ line so that the incoming task is forced to enter the kernel instead of the outgoing task when an interrupt occurs. Additionally all pending interrupts in the outgoing task are forwarded to the incoming task by sending the incoming task a `SIGIO` signal in this case.

2.2.7 Performance

Emulating hardware with software always incurs a performance penalty. The most critical performance bottleneck with UML is the protection of kernel memory when switching from kernel mode to user mode and vice versa. The corresponding `mprotect` operation involves walking page tables in the host and changing attributes for each kernel page. There is additional overhead for all kernel entries and exits, which require context switches to the tracing thread. For each system call, which would normally require two ring transitions (to kernel mode and back), there are now four context switches with UML. When a task executes a system call there is one context switch to the tracing thread, which receives a `SIGTRAP` signal. The tracing thread then forwards that signal to the task, which forces the task onto its kernel stack and causes a second context switch as the task resumes. Two similar context switches occur when the system call returns back to user mode. The fact that UML maps and unmaps pages lazily and the consequential address space scan on each context switch further drains performance.

More information about the performance of User Mode Linux is presented in Chapter 5.

Chapter 3

Design

The shortcomings of address-space protection in User Mode Linux led to the decision to use a different design for the implementation of FIASCO-UX. Instead of having the kernel mapped in the top half gigabyte of a task's virtual memory, FIASCO-UX puts the kernel in its own host process as shown in Figure 3.1. FIASCO-UX abandons the idea of a tracing thread and traces all L4 tasks itself.

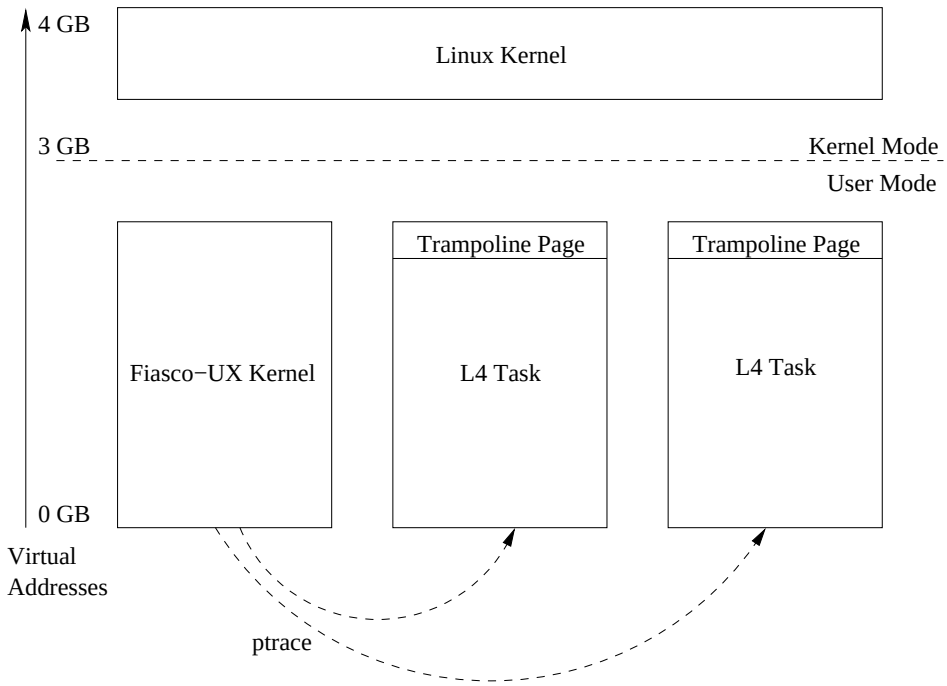

Figure 3.1: FIASCO-UX Tracing Relationship

To perform memory map operations in the different address spaces, FIASCO-UX maps a single page into each task, which is used for trampoline operations. The available virtual memory size for L4 tasks is thus nearly 3 GB.

3.1 Virtual-Memory Layout

The FIASCO kernel uses a memory layout that is unsuitable for use with FIASCO-UX, because it uses virtual addresses beyond 3 GB that are not accessible to Linux programs running in user mode. The memory layout for FIASCO-UX was therefore changed as shown in Figure 3.2.

Virtual Memory Area	Size	Usage
`0x00000000-0x0fffffff`	256 MB	Fiasco-UX Code
`0x20000000-0x3fffffff`	512 MB	Thread Control Blocks, Kernel Stacks
`0x40000000-0x4fffffff`	256 MB	GNU Libc Internals (standard address)
`0x50000000-0x5fffffff`	256 MB	Kernel Structures, IDT, GDT, TSS
`0x60000000-0xbffeffff`	1.49 GB	Mapped Physical Memory
`0xbfff0000-0xbfffffff`	64 KB	Fiasco-UX Linux Process Stack
`0xc0000000-0xffffffff`	1.0 GB	Linux Kernel

Figure 3.2: FIASCO-UX Kernel, Virtual Memory Layout

FIASCO is written in C++ and uses only few functions of the C library. It is therefore linked to a very lean C library from the OSKit, which has been designed for kernel and operating-system development. Due to the fact that the OSKit C library uses functions that operate directly on hardware, such as directly writing to video memory for character output, it cannot be used for programs running under Linux. The Free Software Foundation provides a C library for Linux, glibc, which FIASCO-UX is linked against.

3.2 FIASCO-UX Startup

3.2.1 Constructors

The GNU C library requires proper initialization before some of its functions can be used. Therefore, special care must be taken to run the constructors in the right order. FIASCO-UX requires that `kmem::init()` be executed before any of its constructors, because some constructors allocate memory, such as the mapping database. For a correct startup, the following sequence of initialization must be followed.

1. glibc constructors
2. `kmem::init()`
3. other FIASCO-UX constructors
4. `main()`

However, the constructors must all run at once. For this reason we turned `kmem::init()` into a constructor and introduced a new constructor priority scheme, which allows programmers to specify the exact order in which constructors are called, even across different source files.

3.2.2 Physical Memory

FIASCO-UX needs a backing store for frames of physical memory that it hands out to L4 tasks. Because both an L4 task and the FIASCO-UX kernel need to access the memory, shared memory is the only choice. FIASCO-UX implements physical memory via POSIX memory-mapped files. The kernel maps the entire physical memory into its virtual address space in order to be able to copy directly to and from user-task address spaces (see Figure 3.2). The file resembling physical memory normally resides in `/tmp` and has no file name associated with it, to prevent other (possibly hostile) processes from accessing the physical memory pages of FIASCO-UX.
Because no FIASCO-UX kernel code or data is ever mapped into the virtual address space of an L4 task, there is no need to allocate pages of physical memory for the FIASCO-UX kernel. Even resource manager programs like `rmgr`, which know about the physical memory layout, cannot overwrite kernel code this way and more physical memory is available for applications. A few physical pages are reserved as shown in Figure 3.3. FIASCO-UX also reserves a percentage of physical memory for kernel allocators and page tables, but their addresses are not fixed.

Physical Memory Area	Size	Usage
`0x00000000-0x00000fff`	4 KB	Multiboot Info Structure
`0x00001000-0x00001fff`	4 KB	Task Trampoline Page
`0x00002000-0x00002fff`	4 KB	Fiasco-UX Signal Altstack

Figure 3.3: FIASCO-UX Kernel, Reserved Physical Pages

3.2.3 Loading of ELF Modules

Under normal circumstances it is not the kernel's responsibility to load the L4 tasks that are to be run. At the time the kernel starts, the bootloader will have already loaded the required ELF modules and passed the memory map to the kernel in a multiboot structure. This mechanism cannot be used with FIASCO-UX because the kernel is not booted via a bootloader but is instead started as a process from a shell or script. In order to load the ELF modules into the physical memory file, FIASCO-UX can use one of the following techniques:

1. Use a modified bootloader process to load the ELF modules
2. Implement its own ELF loader
3. Load the modules via `execl()`

The third method is by far the easiest to implement, because it relies on the Linux ELF loader to do most of the work. FIASCO-UX employs a little trick in order to load the modules. For each module the FIASCO-UX kernel forks off a child process and then attaches to it via `ptrace`. When the child process then executes the image of the task to load, the process will stop at the `exec` system call, because the child process is traced by FIASCO-UX. At this point the Linux loader has unpacked the ELF image into memory and the process' instruction pointer points to the image's entry point. After

the instruction pointer has been recorded in the multiboot entry for that module, FIASCO-UX copies the ELF sections from the child's memory area to the respective address range in the physical memory file, using the memory map provided by Linux for each process in `/proc/pid/maps`.

3.2.4 Kernel Initialization

The kernel startup code for FIASCO-UX is slightly more complex than that of FIASCO. FIASCO-UX provides a command line parser, with options to specify the size of the physical memory and which modules to load. Besides remembering the location of the multiboot info structure, FIASCO-UX also has to remember the file descriptor of the physical memory file, needed later to map pages from that file, and the pointer to the first argument vector, argv[0], which stores the process name. The process name is later overwritten in each child process so that the Linux `ps` command shows the task number for each task that runs under FIASCO-UX's control.

FIASCO-UX does not initialize the console, video memory, interrupt controller, GDT, LDT or any of the built-in kernel debuggers, because none of these are used in this port.

3.3 Changes to the Kernel Core

3.3.1 Tasks and Host Processes

FIASCO runs each task in its own address space. Within a task, multiple threads can run concurrently, sharing the same address space. For FIASCO-UX, two possible mappings of tasks and threads to Linux processes are worth considering:

1. Create one Linux host process for each L4 thread. Threads belonging to the same task can share their memory mappings if all of their memory is mapped as shared memory.
2. Create one Linux thread for each L4 thread using `clone` with the option `CLONE_VM`.
3. Create one Linux host process for each task. Threads can run within that particular Linux process by switching instruction pointer and stack pointer, like some user-level thread packages do.

The first method generates more overhead in the host kernel than the other two. Linux would have to allocate process control structures for each thread. Also when mapping or unmapping pages in a task, FIASCO-UX would have to iterate over all threads of that task and remove the pages in every corresponding host process. The second method does not have this overhead, but uses `clone`, a Linux specific system call which is not portable. FIASCO-UX implements the third method with one process per task.

3.3.2 Page Tables

Kernels that run in kernel mode manipulate the address space of a task by means of page tables. The page table hierarchy is accessed by the processor via the page directory base register (`CR3`). By adding or removing entries from a page table the kernel can add or remove pages from a process' address space respectively. However, FIASCO-UX runs in user mode and therefore cannot load the page directory address in `CR3`. Also, due to the address space protection mechanisms employed by Linux, the FIASCO-UX kernel process cannot add or remove pages from other process' address spaces. FIASCO-UX handles these two problems in the following manner:

- FIASCO-UX maintains the same page table data structures the FIASCO kernel uses, however, they are not used by the processor and are only kept by FIASCO-UX for the kernel's internal management of address spaces.
- Whenever a page table entry for an L4 process is changed, the change is not automatically visible in the task's address space, but has to be performed manually. Because only the Linux kernel and a Linux process itself can modify the respective address space, the L4 task has to add and remove pages in its virtual memory area itself. A `ptrace` extension could help with this problem.

3.3.3 Page Fault Handling

When the processor encounters an access to a page that is not mapped or has insufficient access rights, it raises a page-fault exception and executes the kernel's page-fault handler. In order to resolve the fault, the kernel needs to know the fault address, which can be found in the register `CR2` and which is not accessible from user mode. A Linux process that performs an illegal memory access is sent a `SIGSEGV` signal by the kernel. FIASCO-UX installs an extended signal handler for `SIGSEGV` in each L4 task, which causes the Linux kernel to put a ucontext structure onto the process' signal stack, which also contains the fault address. Reading the page fault address of a process in user mode is only possible from within the `SIGSEGV` handler of that process.

3.3.4 User Memory Access

The FIASCO kernel can always access the user memory of the current task, because the task's virtual memory is mapped below 3 GB, whereas the kernel's memory is mapped above 3 GB. Copying from and to pages of that task are therefore simple `memcpy` operations. When a new task is scheduled, FIASCO simply loads a new page directory address and the address space layout of the new task is automatically in effect for the next address translations. FIASCO-UX cannot do the same, because loading new page tables does not automatically make the task's address space visible in the kernel process. To emulate the original FIASCO behaviour, FIASCO-UX would have to unmap the user address space of the old task in the kernel process and then map the entire user address space of the new task at each task switch. This is too expensive and therefore FIASCO-UX copies data via the physical pages that back the virtual pages of the task.

- The kernel translates the task's virtual address into a physical address using the task's page table. If the page table contains no mapping for that virtual page or the page has insufficient access rights, the copy operation raises a page fault for that task.
- The kernel translates the physical address into a virtual address inside the kernel process. This is always possible, because the kernel has all physical memory mapped in its virtual address space.
- The copy operation then copies between the virtual memory areas in the kernel. Special care must be taken when crossing page boundaries, because adjacent virtual memory pages in the task are not guaranteed to be backed by adjacent physical pages. Therefore all copy operations copy data until reaching a page boundary, then do a new address translation before continuing.

This mechanism allows the FIASCO-UX kernel to copy directly between a user address space and the kernel address space. It also makes it possible to copy directly between two user address spaces by performing the aforementioned translation for both tasks. Being able to directly copy between user address spaces means that the I/O window as used by the native FIASCO kernel for Long IPC is no longer necessary. In the FIASCO-UX kernel, all code related to the I/O window has been replaced with appropriate copy routines between user address spaces.

3.3.5 Privileged Instructions

Some parts of the FIASCO kernel operate directly on hardware and thus cannot be used in FIASCO-UX. This includes instructions that are only permitted in ring 0 (kernel mode) and access to control registers which are not accessible from user mode. In general there are two approaches to solving this problem:

1. Replacing the privileged instructions throughout the kernel code with procedures that emulate their behaviour.
2. Attempting to execute the privileged instructions, which allows us to leave the code unchanged. A `SIGSEGV` signal is sent to the process indicating a general protection fault. The kernel must then figure out which instruction led to the fault and emulate it correctly.

FIASCO-UX implements the latter method, because it does not clutter the kernel stack with call frames of the emulation procedures and requires no modifications to the kernel code, except for adding code to emulate three instructions (`cli`, `sti`, `iret`). The disadvantage of this method is the overhead resulting from the `SIGSEGV` handler which traps these instructions. Registers that are not accessible from user mode are substituted with global variables, for example `CR2` with `page_fault_addr` and `CR3` with `page_dir_addr`.

3.3.6 Kernel Lock

The implementation of `kernel_lock_t::test()` and `kernel_lock_t::test_and_set()` in the FIASCO kernel makes use of the interrupt bit (`IF`) in the `EFLAGS` register to determine the

status of interrupts. Because FIASCO-UX runs entirely in user mode, interrupts in the host kernel will always be enabled while FIASCO-UX executes. Consequently the `IF` bit will always be set and cannot be used as an indicator for the status of interrupt delivery. FIASCO-UX uses a global variable `interrupts_disabled` instead.

3.3.7 FPU Handling

The floating point unit can only be used by one thread at a time. When switching between threads, native operating systems save the FPU state of the old process and restore the FPU state of the new process. When using lazy FPU save/restore, the FPU state is only saved when it has been modified and only restored when a thread attempts to use the FPU and thereby generates an exception. However, such an FPU exception is not reflected back to the user mode program, but handled transparently inside the host kernel. This means that FIASCO-UX cannot detect if or when one of its threads uses the FPU. Lazy FPU saving is therefore not possible. Due to the fact that the Linux host kernel handles the FPU context switching between its processes, FIASCO-UX does not have to care about the FPU state when switching between tasks, because each L4 task is represented by a host process. When switching between threads of the same task, the FPU state must be saved and restored. Such an intra-task thread switch can also occur indirectly, for example thread 1 of a task A switches to a thread of task B and task B then switches back to thread 2 of task A. The Linux kernel only knows about task switches. It does not know that inside these tasks, thread switches occur as well. FIASCO-UX handles this issue by remembering the FPU owner thread on a per task basis. Each time someone switches to a thread of that task, that thread is compared with the FPU owner thread. If they are not equal, the FPU context of the owner thread needs to be saved and the FPU state of the new thread must be restored. The new thread then becomes the FPU owner thread.

3.4 Limitations

There are a few limitations in the FIASCO-UX kernel that users should be aware of:

- The Linux interval timer has a granularity of 10 milliseconds. This is also the rate at which FIASCO-UX generates timer ticks. The original FIASCO kernel generates timer ticks every millisecond.
- Intel microprocessor documentation [Intel] states that the `sti` instruction enables interrupts after the instruction following `sti`. The delayed effect of `sti` when followed by a `ret` allows to return from a function with interrupt delivery deferred until after the return. With FIASCO-UX interrupts are enabled as soon as `sti` has been executed. Because neither FIASCO nor FIASCO-UX rely on the special aspect of `sti` this is not an issue.
- FIASCO-UX cannot detect and emulate the behaviour of code changing the `IF` bit (interrupt flag) in the `EFLAGS` register using a combination of `pushf` / `popf` or `pushfd` / `popfd`. This is due to the fact that these instructions do not generate an exception when invoked with

insufficient privileges; instead the privileged bits do not change. Beware of code that does something like this:

```
/* interrupts enabled here */
pushfd
cli
/* critial section, interrupts disabled */
popfd
/* interrupts enabled here */
```

FIASCO-UX will not recognize the effect of `popfd` restoring the previously saved state of the `IF` bit. However, it will execute the `cli` instruction and interrupts will be disabled after this section of code.

- When entering kernel mode, FIASCO-UX disables all maskable interrupts. This is the behaviour of an interrupt gate. Trap gates do not disable interrupts during kernel entry. FIASCO and FIASCO-UX use only interrupt gates, so FIASCO-UX does not distinguish between the different gate types and disables interrupts unconditionally.

- Some functions of the GNU C library (glibc), especially `printf`, `sprintf`, `snprintf` and `vprintf` use more than 2 KB stack when invoked. Because the kernel stacks in FIASCO-UX are smaller than 2 KB any attempt to call such a function directly or indirectly on a kernel stack will result in corrupting the preceeding TCB. FIASCO-UX does not need to call any of these functions in kernel code; there are also less expensive output functions, such as `puts` available. The general problem of stack intensive functions in glibc can probably be avoided by linking against a less bloated C library, such as dietlibc.
  Alternatively the kernel stacks could be made bigger, which would reduce the number of possible tasks and threads. It would also be a fundamental change in kernel memory layout, compared to the native FIASCO kernel.

- The number of pages that can be mapped in an L4 process is limited by the Linux host kernel. Currently a process cannot have more than 65536 VMA[1] mappings. This can be problematic for pagers which map small 4 KB pages one by one instead of using 4 MB superpages. Linux kernels in the 2.2 series merge adjacent VMAs, whereas newer 2.4 kernels do not. In such a case a task can map no more than 256 MB with single 4 KB pages before memory map operations start failing.

- Because there are currently no virtual devices available for FIASCO-UX, there is also no support for I/O ports. Once such device emulation exists, the emulation code can be extended to support I/O ports by performing an opcode analysis at the instruction pointer to detect the presence of `in` and `out` instructions.

[1] Virtual Memory Area

Chapter 4

Implementation

4.1 Task Creation

In this section I will examine the steps that are necessary to create an L4 task under FIASCO-UX. For each new task, a new process has to be created in the host. The normal means by which a Linux process can create new tasks is the `fork` system call[1]. This system call creates an exact copy of the calling process, which then becomes the calling process' child process. The created child process inherits copies of the parent process data space, heap and stack, which are copied on demand using a copy-on-write mechanism. Additionally the child also inherits open file descriptors, user and group IDs, signal masks and signal dispositions. When FIASCO-UX creates a new L4 task, the task should start with an empty address space so the kernel can allocate pages as needed, for example whenever a page fault occurs. Therefore the child process has to unmap its entire virtual address space inherited from the parent, including the page containing the currently executing code. After having unmapped the address space, the task must not execute any instruction anymore, because this would result in an immediate page fault and the termination of the child process, because the memory has just vanished. Because the new task also inherits all signal dispositions and the set of blocked signals from the parent, the FIASCO-UX kernel, all signals must be unblocked and meaningful signal handlers for the child must be installed. The recommended method for installing a signal handler under Linux is to use the `sigaction` function. However, deep within glibc lies code that makes use of undocumented signal handler features and caused numerous crashes during the development of FIASCO-UX. To understand the problem we have to examine the way that Linux signal handlers work:

- When a signal is delivered in a process, the kernel saves the context of the process in a signal frame on the process' stack (or signal stack) and jumps to the function that had previously been declared as signal handler for that signal. When the signal handler returns, the signal frame has to be removed from the stack and the interrupted context must be restored.
- The signal frame is removed by the Linux kernel using the `sigreturn` system call. Because the signal handler does not call `sigreturn` itself, the Linux kernel modifies the return address on the process' stack to point to a few instructions inside the signal frame.

[1] clone and vfork are less common and less portable

- When the signal handler then returns via a `ret` instruction, execution does not resume in the interrupted code, but inside the cleanup code which the kernel put inside the signal frame.
- The cleanup code calls `sigreturn`, and then returns to the interrupted context.
- A rather unknown fact is that Linux provides a possibility to use custom code to clean up the signal frame. The manpage of `sigaction` states that the `sa_restorer` field of the sigaction structure is obsolete. In fact this field can be used to specify a custom restorer function instead of the generic one.
- The GNU C library uses this undocumented feature and provides custom restorer code in order to make GDB aware of signal frames.

Because the L4 task just had all of its memory unmapped, the pages containing the glibc restorer code were no longer present. As soon as the first signal occured and the signal frame was to be cleared from the stack, the kernel jumped into unallocated memory and the host process crashed.
To prevent this problem, FIASCO-UX bypasses the glibc `sigaction` function altogether and installs the signal handlers for the task directly with Linux syscalls. That way the Linux kernel puts the default restorer code onto the signal stack, which is guaranteed to be present.
When all of a task's virtual memory is unmapped upon task startup, the child process keeps the top pages which Linux uses as process stack. At the top of these pages, just below 3 GB, the Linux kernel allocates the process environment, which contains the process name and the argument list. The pointer to the process name resides inside Linux kernel space and cannot be changed. FIASCO-UX overwrites the process name with the task number, so that `ps` shows the task number corresponding to the host process. In addition to the page containing the process name, FIASCO-UX maps the physical page `0x1000` into all its tasks and uses it for trampoline code. The virtual memory layout of all L4 tasks is shown in Figure 4.1.

`0x00000000-0xbfffdfff`	Available
`0xbfff0000-0xbfff0fff`	Trampoline Page
`0xbfff1000-0xbfffffff`	Linux Process Stack, Environment, Name
`0xc0000000-0xffffffff`	Linux Kernel

Figure 4.1: FIASCO-UX: Task Virtual Memory Layout

4.2 Address-Space Manipulation

One unfortunate consequence of having a seperate process for the kernel and address-space protection among all processes is the inability of the kernel process to manipulate the address space of an L4 task directly. Ideally the kernel process should be able to map and unmap pages in a traced process, but that functionality is not provided by Linux. Only a process itself can map or unmap pages in its own address space.
FIASCO-UX uses the trampoline page of a process for address space manipulation. Because the

trampoline page is a fixed physical page (`0x1000`), its virtual address in the kernel (`0x60001000`) is well-known and the kernel can write to it directly. The trampoline works as follows:

- The kernel process retrieves the current register status of the task process using `PTRACE_GETREGS` and saves it for later restoration.
- The syscall opcode `int 0x80` is written to the bottom of the trampoline page.
- The kernel writes the syscall parameters that are passed on the stack directly above the syscall opcode on the the trampoline page.
- The register set of the task is modified, so that `EIP`[2] points to the syscall opcode and `EAX` reflects the Linux syscall number. The modified registers are then committed to the task using `PTRACE_SETREGS`.
- The kernel process then activates the task process and waits for it to return from the system call.
- The task stops twice, once as it enters kernel mode and again when it returns from kernel mode, at which point the syscall is complete.
- The kernel process restores the previous register set for the task.

This scheme can be used with all required syscalls. FIASCO-UX provides trampoline code for the syscalls `mmap`, `munmap` and `mprotect`, to map, unmap and change permissions on a page, respectively. The syscall itself is performed by the task, but under control of the tracing kernel process.

During the development of FIASCO-UX the trampoline code contained a bug that sometimes made trampoline calls fail in mysterious ways. It turned out that it is not sufficient to set up the registers directly required by the Linux syscall. Especially the registers `CS` and `SS` must be loaded with valid selectors. Whenever the selectors happened to be valid, the trampoline call worked; when they were random, the call failed. The current code now copies all of the task's old registers, only overwriting those to be modified, which guarantees that the selectors are valid.

4.3 Interrupts, Signals

FIASCO-UX implements interrupts with the `SIGIO` signal and a byte queue for each interrupt line. Linux allows the `SIGIO` signal to be configured as an asynchronous notification event when data can be read from a file descriptor. Currently Linux supports `SIGIO` on sockets and pseudo terminals, but not on pipes and fifos. FIASCO-UX opens a pseudo terminal (PTY) for each interrupt line, which works like a pipe. For each interrupt line there is one host process that generates interrupt events by writing a byte into the write end of the PTY. The read end of each PTY is shared between the FIASCO-UX kernel and all L4 processes. The Linux `fcntl` system call with the `F_SETOWN` option allows to configure a process as signal owner; that process receives the `SIGIO` signal to indicate

[2] Instruction Pointer

there is data to be read. Whenever the FIASCO-UX kernel schedules an L4 task to run, it changes the `SIGIO` owner to the PID of the active task. When that task enters the kernel, the kernel changes ownership back to the PID of the kernel process. When a task runs and an interrupt comes in, the task will stop immediately with a `SIGIO` signal and the kernel process will gain control and handle the interrupt. It should be noted that the kernel can only stop a task by sending that task a signal. The overhead of sending the task the `SIGIO` signal directly is smaller than only delivering interrupts (`SIGIO` signals) to the kernel process and then sending a stop signal to the task process. With one host process for each interrupt line, interrupts can happen asynchronously from the kernel control flow and different interrupts can occur simultaneously. The kernel can handle interrupts with different priorities by polling the PTY byte queues in the right order. FIASCO-UX currently only implements the timer interrupt, but other interrupt sources can be added by implementing processes that generate their interrupt events.

4.4 Signal Stack

When the kernel or a user-mode process is interrupted by an interrupt in FIASCO, the processor saves the context of three or five words respectively on the current stack to allow the restoration of that context later. Due to the fact that FIASCO-UX uses signals to implement interrupts, the Linux kernel also puts the entire interrupted context on the current stack. However, that context is much bigger, typically several hundred bytes. This is problematic for two main reasons:

- The kernel stack of FIASCO is small and cannot grow because it is stacked in between TCBs. FIASCO-UX makes no changes to the layout of kernel stacks and TCBs and therefore also has limited kernel stack space. With several interrupts occuring in a nested manner and several function call frames already on the stack, the kernel stack can grow down excessively and corrupt the TCB, with unfortunate consequences for the stability of the kernel.
- Secondly, delivering Linux signals on the FIASCO-UX kernel stack means that FIASCO-UX will find a non kernel frame on that stack, which can confuse kernel debuggers that know nothing about the layout of a Linux signal frame and may try to follow the inter-linked call frames.

The limited kernel stack space was the main reason leading to the decision to deliver signals on an alternate stack. Linux allows the registration of such an alternate stack by means of the `sigaltstack` system call. Signals on that alternate stack cannot occur in a nested manner; each signal blocks all other signals and multiple occurances of itself while running in the signal handler.

4.5 Kernel Mode

The following section discusses a crucial part of the FIASCO-UX emulation code - the code which handles the emulation of privileged instructions, interrupts and page faults in kernel mode. Figure 4.2 shows a state-transition chart of the execution path in kernel mode. Most of the emulation

code runs in a signal-handler context, indicated by a black box. Because signal handlers restore the interrupted context and all registers at once, they are ideal to warp the kernel into an entirely different context by modifying the instruction pointer, stack pointer and signal mask in the restore context. The `kernel_entry` function sets up a processor context on the kernel stack and modifies the signal handler context, so that the kernel process falls into the specified interrupt gate when the signal handler returns.

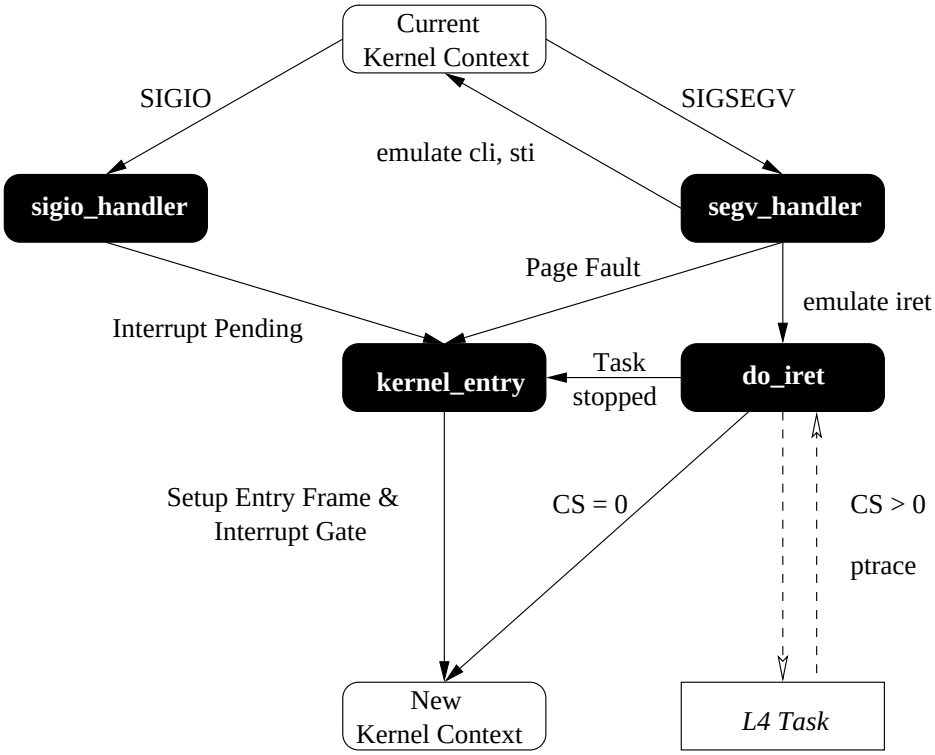

Figure 4.2: FIASCO-UX Emulation Call Graph (Black boxes indicate signal context)

4.5.1 Privileged Instructions

FIASCO and FIASCO-UX use a number of instructions, such as `cli`, `sti`, and `iret`, that can only be executed by tasks with a sufficient privilege level. Because FIASCO-UX is a user mode application, any attempt to execute one of these instructions will result in the Linux kernel delivering a `SIGSEGV` signal to the calling process and executing the handler for that signal. The `segv_handler` receives a pointer to an ucontext structure (see Figure 4.3) as parameter. The ucontext structure contains the trap number, error code and address of the fault. If the trap number is 0xd (general protection fault), then FIASCO-UX attempted to execute an illegal or privileged instruction. In that case the `segv_handler` checks the opcode at the current instruction pointer (`EIP`) to determine which instruction led to the fault. The instructions `cli` and `sti` manipulate the status of interrupt delivery. FIASCO-UX emulates their behaviour by filling or clearing the mask of blocked signals in the interrupted context on the signal stack. The `segv_handler` then returns and the modified mask

of blocked signals is immediately in effect. The `iret` instruction causes a call to `do_iret` and will be discussed in Section 4.5.4.

4.5.2 Interrupts in Kernel Mode

Interrupts can occur asynchronously to the normal control flow in the kernel, unless they have been disabled by the `cli` instruction. When interrupts are enabled in FIASCO-UX and an interrupt is pending in the kernel process, the Linux kernel will deliver a `SIGIO` signal to FIASCO-UX and the `sigio_handler` will be called. This handler polls all interrupt queues and enters the kernel with the pending interrupt of the highest priority through an interrupt gate. It is possible that a stale `SIGIO` signal does not indicate an interrupt activity, namely when the interrupt event had already been dequeued before the `SIGIO` signal was delivered. In this case the handler simply returns.

4.5.3 Kernel Mode Page Faults

FIASCO-UX uses page faults to map certain memory areas on demand, such as TCBs. When the kernel process attempts to read or write to a memory area that is currently not mapped, the Linux kernel will deliver a `SIGSEGV` signal to the FIASCO-UX process and the ucontext structure contains 0xe (page fault) in the trap number. Additional information is also provided in that structure, such as the page fault address (`CR2`) and the error code, which describes if the fault was due to a read or a write operation. FIASCO-UX then enters the kernel through the page fault interrupt gate. The layout of the ucontext structure is shown in Figure 4.3.

	0 … 31	32 … 63	64 … 91	92 … 127
+00	uc_flags	uc_link ptr	ss_sp	ss_flags
+16	ss_size	gs	fs	es
+32	ds	edi	esi	ebp
+48	esp	ebx	edx	ecx
+64	eax	trapno	error	eip
+80	cs	eflags	uesp	ss
+96	fpregs ptr	oldmask	cr2	sigmask

Figure 4.3: ucontext structure

4.5.4 Ring Transitions

The `iret` instruction is used by FIASCO and FIASCO-UX to restore a previously interrupted context or to setup an entirely new context. The interrupted context consists of either three (kernel mode) or five (user mode) words, determined by the value of the code segment (`CS`).

If the code segment indicates privilege level 0 (kernel mode), `do_iret` returns from the signal handler context with the interrupted context restored. If the privilege level is greater than 0 the execution will continue in user mode and an L4 task is scheduled to run.

4.6 User Mode

Execution in user mode happens outside the FIASCO-UX process in the L4 task processes. This means that the kernel has to determine the PID of the host process which corresponds to a particular L4 task. This information is stored in an unused page directory slot. Furthermore, interrupts have to be forwarded to the task process and the register set of the kernel must be transferred to the task process. This is done by reading out the interrupted context on the signal stack and committing the registers to the task process using `PTRACE_SETREGS`. The FIASCO-UX kernel process then activates the task process using `PTRACE_SYSCALL` and waits for it to stop using `waitpid`. The task process stops upon the delivery of a signal at which point the FIASCO-UX process is notified of that event by `waitpid` returning the status of the task. A task can stop due to the following signals:

- When the task performs an L4 syscall (`int 0x30` through `int 0x36`) the Linux kernel will deliver a `SIGSEGV` signal for that task, because the task raised an exception.
- When a page fault occurs in the task, the Linux kernel will also deliver a `SIGSEGV` signal.
- When the task performs an `int3` instruction or a Linux system call, the Linux kernel will deliver a `SIGTRAP` signal for the task.
- Interrupt activity on one of the interrupt queues results in a `SIGIO` signal being delivered to the task.

When the L4 task stops, the task's registers are copied back to the signal handler context, interrupt delivery is changed back to the kernel process and FIASCO-UX emulates a kernel entry for that task.

The event that led to the L4 task entering the FIASCO-UX kernel must be determined in order to jump to the correct interrupt gate. Interrupts have a unique signal (`SIGIO`). The `int3` debug extension and Linux system calls (`SIGTRAP`) can be easily distinguished by examining the opcode at the instruction pointer. However, when the L4 task stops with a `SIGSEGV` signal, the kernel process cannot immediately tell whether this was due to a page fault or an L4 syscall. As stated before, L4 uses `int 0x30` through `int 0x36` for its syscalls. If the task wanted to enter the FIASCO-UX kernel with one of these syscalls, then the instruction pointer (`EIP`) will point to the opcode sequence `0xcd` followed by one byte indicating the interrupt number.
In any other case the task has raised a general protection fault or a page fault. In case of a page fault, the FIASCO-UX kernel needs to know the faulting address (`CR2`), the error code (read or write access) and in all other cases the trap number. This information cannot be read using the `ptrace` interface and is only accessible from a signal handler inside the task process. Because L4 tasks have no notion of signals or signal handlers, FIASCO-UX copies the signal handler code to the trampoline page of that task and then forwards the intercepted `SIGSEGV` signal to the task's process in the host. When the task returns from the signal handler, it will stop again and the FIASCO-UX kernel process can read out the ucontext structure which has been stored on the task's signal stack, the trampoline page. The offset of the ucontext structure on the signal stack is architecture dependent, but the signal handler receives a pointer to it as one of its parameters. The offset calculated from that pointer is

written to a well-known location on the trampoline page to allow the kernel to find the ucontext structure and read out the necessary information.
This trampoline operation is very expensive in terms of CPU cycles, but Linux currently provides no alternative solution.

4.7 Race Conditions

The lack of support for sharing signal masks between Linux processes brings up two race conditions in the interrupt emulation:

- When the FIASCO-UX kernel wants to schedule a user mode task by activating its process in the host, it has to forward interrupts to that process by changing the ownership of the `SIGIO` signal to the PID of the task's process. After the call to `iret` and before the ownership has been changed to the task, an interrupt could come in and would be pending in the kernel process. The task would not receive a `SIGIO` signal for this interrupt and would continue executing until the next interrupt, which would likely be a timer interrupt. The pending interrupt in the kernel process would be delivered as soon as FIASCO-UX unblocks signals after the task's kernel entry. FIASCO-UX works around this race condition by checking for pending interrupts in the kernel after changing the ownership of the PTY descriptor to the task. If an interrupt occured, the task process will not be activated, interrupt delivery will be changed back to the kernel process and FIASCO-UX will proceed with a task kernel entry.

- The second race condition is more serious, as it could lead to interrupts being delayed for long periods or even being lost forever. The problem is similar to the first one. When an L4 task enters the kernel with a syscall and before the kernel can change the PTY ownership back to the kernel process, an interrupt could come in and would be delivered to the task process, which is stopped at that time. If the syscall put the task to sleep forever, the interrupt would be trapped on the task and never be delivered. As with the first problem, the race condition can be fixed by polling all interrupt queues after a task's kernel entry and queueing a `SIGIO` signal for the kernel process if an interrupt occured in the meantime.

These workarounds prevent both race conditions and ensure that no interrupt activity will go unnoticed. There is additional overhead for polling the interrupt queues for activity and processing the interrupt events in both cases. Additional overhead occurs for the delivery of pending `SIGIO` signals whose interrupt events had already been dequeued by one of the workarounds.

Chapter 5

Performance Analysis

This chapter examines the performance of FIASCO-UX. All benchmarks were conducted on an otherwise idle Linux system with minimal workload. It should be noted that running a heavy workload (X, compiler, etc.) can degrade the performance of FIASCO-UX immensely.

The first point of interest is the slowdown ratio of the emulation kernels compared to the native kernels. For this purpose I compared the cheapest syscall on both systems, `getpid` on Linux and `id_nearest` on FIASCO. Both syscalls reflect the cycles (`clk`) required for one kernel entry and exit plus minimal processing overhead in the system call path. FIASCO-UX is about 60 times slower than FIASCO, UML with address space protection at least 120 times slower than Linux. Both User Mode Linux and FIASCO-UX ran under a native Linux 2.4.19 host kernel during this benchmark.

Linux Kernel	`getpid`	AMD Duron 800		Intel Pentium 4 1600	
Linux 2.4.19	native	273 clk	0.3 μs	1650 clk	1 μs
UML 2.4.19-32um	without jail	33500 clk	42 μs	111000 clk	69 μs
UML 2.4.19-32um	with jail	136000 clk	170 μs	207000 clk	129 μs

FIASCO μ-Kernel	`id_nearest`	AMD Duron 800		Intel Pentium 4 1600	
FIASCO	native	305 clk	0.4 μs	1631 clk	1 μs
FIASCO-UX		17600 clk	22 μs	49400 clk	31 μs

Figure 5.1: Native Kernels vs. User Mode Kernels

Figure 5.2 shows the standard L4 pingpong benchmark, which measures Short-IPC performance between threads of the same task (Intra-AS[1]), and between threads of different tasks (Inter-AS). Surprisingly, inter-address-space IPC is faster than intra-address-space IPC. The reason for this strange behaviour is the expensive FPU save/restore operation when switching among threads of the same task. FPU state need not be saved across thread switches among different tasks because the Linux host kernel already performs this task. Again FIASCO-UX is about 50 times slower than FIASCO.

[1] Intra Address Space

Pingpong Benchmark		AMD Duron 800		Intel Pentium 4 1600	
FIASCO	Intra-AS	543 clk	0.7 μs	3280 clk	2.0 μs
FIASCO	Inter-AS	890 clk	1.1 μs	4095 clk	2.5 μs
FIASCO-UX	Intra-AS	46500 clk	58 μs	94500 clk	59 μs
FIASCO-UX	Inter-AS	43500 clk	54 μs	88000 clk	55 μs

Figure 5.2: L4 Pingpong Benchmark, FIASCO vs. FIASCO-UX

Figure 5.3 shows the most expensive operations in the emulation path. For example, an IPC operation costs one `waitpid` call, waiting for the task to stop, one `PTRACE_GETREGS` call to read out the task's current register context, one call to `PTRACE_PEEKTEXT` at the task's instruction pointer to check for an IPC syscall opcode, a call to `kernel_entry` to warp execution to the right interrupt gate and calls to `fcntl` and `poll` to change and poll the IRQ lines. The kernel's syscall path contains one `sti` and one `iret` opcode which are emulated in the kernel's `SIGSEGV` handler. Upon return to user mode there is another call to `fcntl` and `poll` to change ownership on the IRQ lines and check for IRQ activity, a call to `PTRACE_SETREGS` to set up the task's new register state and finally a call to `PTRACE_SYSCALL` to restart the task. These operations cost roughly 17200 of the 17600 cycles measured for `id_nearest` on an AMD Duron 800. The rest of the code in the syscall and emulation paths accounts for the remaining cycles.

Emulation Path Operation		AMD Duron 800		Intel Pentium 4 1600	
`cli/sti/iret`	Kernel Signal Handler	3145 clk	3.9 μs	5556 clk	3.5 μs
`fcntl`	Change IRQ Ownership	437 clk	0.5 μs	1744 clk	1.1 μs
`poll`	Check IRQ Activity	1561 clk	1.9 μs	2472 clk	1.5 μs
`fpu reclaim`	Save/Restore FPU Context	1901 clk	2.4 μs	3868 clk	2.4 μs
`waitpid`	Wait For Child Stop	636 clk	0.8 μs	2056 clk	1.3 μs
`memcpy`	Copy Trampoline Code	398 clk	0.5 μs	508 clk	0.3 μs
`kernel_entry`	Setup Interrupt Gate	352 clk	0.4 μs	360 clk	0.2 μs
`PTRACE_GETREGS`	Read Task Context	842 clk	1.1 μs	2648 clk	1.7 μs
`PTRACE_SETREGS`	Write Task Context	1119 clk	1.4 μs	2880 clk	1.8 μs
`PTRACE_PEEKTEXT`	Read Task Memory	941 clk	1.2 μs	2316 clk	1.4 μs
`PTRACE_SYSCALL`	Resume Task Execution	2958 clk	3.7 μs	8096 clk	5.1 μs
`task_sighandler`	Get Task Fault Context	8000 clk	10 μs	15000 clk	9.4 μs
`magic_mmap`	Map Page In Task	18000 clk	22.5 μs	40000 clk	25 μs
`magic_munmap`	Unmap Page In Task	17000 clk	21.3 μs	40000 clk	25 μs

Figure 5.3: Emulation Path Costs

The mapping or unmapping of a page in a task costs two calls to `PTRACE_GETREGS`, two calls to `PTRACE_SETREGS`, two calls to `PTRACE_SYSCALL`, two calls to `waitpid`, a call to `mmap` or `munmap` in the task and four context switches between the FIASCO-UX kernel and the task.

Chapter 6

Conclusions

The development of FIASCO-UX has shown that it is possible to run a complete μ-kernel entirely in user mode. The emulation of merely three assembler instructions (`cli`, `sti` and `iret`) is all that is required to leave major portions of assembler code in FIASCO-UX unchanged. However, the high cost for the emulation of these instructions due to the signal handler overhead provides an opportunity for optimization. For example, `cli` and `sti` could be replaced with appropriate calls to `sigprocmask`. This would require major modifications to assembler code in the kernel using these instructions and has therefore not been done at this point.

However, both FIASCO-UX and User Mode Linux show that user mode kernels cannot compete with their native counterparts in terms of performance. While user code will execute as fast as on a native kernel, ring transitions to and from kernel mode incur high performance penalties. These can be avoided in several ways:

- If security is not an issue, kernel code and user code can share the same address space. Transitions between privilege levels can then be implemented via signal handlers, as in UML, or via calls to `longjmp`. Access to user memory is a trivial `memcpy` operation. When using process tracing (`ptrace`) with this scheme, address space protection is possible as demonstrated in UML's `jail` mode. Without tracing address space protection is impossible and there is no privilege separation at all. It is therefore not a recommended solution.

- Emulators like VMware [VMW] benefit from the loading of kernel modules. All emulator functionality that requires support from the host kernel has full access to all kernel control structures without having to rely on time consuming trampoline calls. Because the loading of kernel modules requires root access on the machine and can compromise the security of the host kernel, it is not a favourable solution either. Furthermore, kernel interfaces are often changing, which makes kernel modules much less portable.

- A far more interesting solution is the addition of more options to the `ptrace` interface in Linux. Because that interface has been designed for the debugging of tasks, it can trivially be extended to support emulators like User Mode Linux and FIASCO-UX in a much better fashion.

In a discussion between the author of User Mode Linux, Jeff Dike, and myself on the Linux Kernel Mailing List, we identified that the following extensions to the `ptrace` interface and signal handling would sufficiently boost the performance of both UML and FIASCO-UX:

- The ability to map, unmap and modify pages in a traced process from inside the tracing process would completely avoid the overhead of trampoline calls in FIASCO-UX and the address space scan in UML. The number of context switches for such operations would decrease from four to zero, the number of ring transitions from nine to two. Such options could for example be called `PTRACE_MMAP`, `PTRACE_MUNMAP` and `PTRACE_MPROTECT`.

- An extension to `clone` that would permit multiple processes to share the same pending signal mask. The first process to accept (via `sigwaitinfo`) or deliver (via a signal handler) an interrupt signal would clear it from the pending mask and the signal would then no longer be pending in any of the other processes. This would eliminate all race conditions relating to interrupt delivery and completely avoid the need to forward pending signals during context switches.

- Another high performance penalty, reading out a task's fault context, would be eliminated with an addition to the siginfo structure for the `SIGSEGV` signal. Currently the fault address is passed in the `si_addr` field of the siginfo structure in the task's signal handler, whereas there is currently no way to obtain the error code of the fault from that structure, which is necessary to distinguish between read faults and write faults. The error code is only available in the ucontext structure on the faulting task's signal stack.
  If a system had the aforementioned shared pending signal masks, the tracing process could then, in case of a task's fault, simply dequeue the task's `SIGSEGV` signal from the shared signal queue using `sigwaitinfo`, which returns the siginfo structure, but not the ucontext structure. This approach completely avoids all overhead associated with signal handlers. The fault information including the error code would be available from the siginfo structure returned. Apart from two ring transitions for the `sigwaitinfo` system call, no other context switches occur in such a scenario.

It remains to be seen what new `ptrace` functionality will be included in future versions of the Linux kernel. Current ongoing work in User Mode Linux introduces a new `PTRACE_SWITCH_MM` call to Linux which looks very promising.

Further work on FIASCO-UX should concentrate on implementing missing functionality, such as the support for I/O ports and virtual devices. Since FIASCO-UX can be run inside GDB, it is unclear whether the FIASCO kernel debugger should be ported to run under FIASCO-UX. For debugging user mode L4 tasks under FIASCO-UX it is necessary to implement a GDB stub similar to that in UML, which allows GDB to attach to L4 tasks, even though the FIASCO-UX kernel is already tracing them.

Chapter 7

Summary

The result of this port is an L4 compliant μ-kernel that can be used for the development of L4 applications under any Linux system. With GDB, there exists a powerful debugger, which is compatible with FIASCO-UX, and which can be a useful tool in the further development of the FIASCO μ-kernel.

The experience I gained during the development of FIASCO-UX has been very helpful in my understanding of some Linux and FIASCO internals.

Finally, the native FIASCO kernel has also benefitted from this user mode port. Due to its different timing behaviour, FIASCO-UX exposed a few bugs in the native kernel which had not been discovered before. Some new abstractions introduced during this port will also make it easier to port FIASCO to other architectures.

The source code of FIASCO-UX has been merged into the main FIASCO tree and is available from remote CVS at: `http://os.inf.tu-dresden.de/drops/download.html`

Acknowledgments

I would like to thank all people who supported me during the development of FIASCO-UX by providing moral support, answering my questions and helping me to find bugs. My special thanks goes to Dr. Michael Hohmuth, my tutor, and to the entire FIASCO kernel team for many useful discussions and helpful advice.

Appendix A

Glossary

API	Application Programming Interface. Set of system defined routines through which an application calls into the operating system or a library
Context	Program state. Typically consists of the register state of the CPU and the program's stack
CPU	Central Processing Unit, Processor
ELF	Executable and Linking Format. Binary interface definition that extends across multiple operating environments
FPU	Floating Point Unit, Coprocessor. Used for mathematical operations
Gate	Predefined entry point in the operating system through which tasks can call certain services
GDB	The GNU Debugger. A program which can be used to monitor the execution of other programs
GDT	Global Descriptor Table. A memory management table that describes system-wide memory segments
IPC	Inter-Process Communication. Exchange of data between one process and another according to a certain protocol
L4	Second generation API for μ-kernels
LDT	Local Descriptor Table. A memory management table that is used to describe memory segments for each non-kernel process
Long IPC	Inter-Process Communication that involves copying memory between two threads
μ-kernel	Small kernel which implements only basic services according to a chosen policy

Page Table	Data structure maintained by the operating system for the mapping of virtual pages to physical pages
Paging	Mechanism to map and unmap physical pages in the virtual address space of processes or the kernel
PID	Process ID. Identifies each process in an operating system
POSIX	Portable Operating System Interface, that allows both BSD-based and AT&T-based Unix systems to share a common system call interface
PTY	Pseudo Terminal. Paired device where one end acts like a terminal and the other is typically controlled by a program
Selector	An opaque handle referring to a code, data or text segment in an address space
Short IPC	Fast Inter-Process Communication which exchanges data only through processor registers
Signal	Synchronous or asynchronous event which interrupts the current program in order to call a service routine
Task	Protection domain that consists of an address space and one or more threads
TCB	Thread Control Block. Management structure in a kernel which contains thread state information
Thread	Sequence of instructions which run in parallel with other activities in a task
Trampoline	Small piece of code that calls a context which otherwise cannot be called from the original context
x86	Processor architecture originally designed by Intel. Now used on most desktop computers

Appendix B

Bibliography

[EKT94] D.R. Engler, M.F. Kaashoek, J.W. O'Toole Jr., *The Operating System Kernel as a Secure Programmable Machine*. ACM SIGOPS European Workshop, 1994

[Hoh98] M. Hohmuth, *The Fiasco Kernel, Requirements Definition*. TU Dresden, Technical Reports, 1998

[HT01] M. Hohmuth, H. Tews, *VFiasco, Towards a provably correct microkernel*. USENIX Annual Technical Conference, 2001

[Intel] IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference, Intel Corporation, 2002

[Lie95] J. Liedtke *On μ-kernel Construction*. 15th ACM Symposium on Operating System Principles (SOSP), 1995

[RJO+89] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M.B. Jones, *Mach: A system software kernel*. Proceedings of the 1989 IEEE International Conference, COMPCON, 1989

[Sch01] S. Schönberg, *A User Mode L4 Environment*. 2nd Workshop on microkernel-based systems, 2001

[TUD] Dresden University of Technology, Faculty of Computer Science, Operating Systems Group Webpage: `http://os.inf.tu-dresden.de/`

[UML] The User Mode Linux Kernel, Sourcecode, Tutorials, Documentation, Papers, `http://user-mode-linux.sourceforge.net/`

[VMW] The VMware Virtual Machine, `http://www.vmware.com/`