TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

TUD–FI98–12 — Dezember 1998

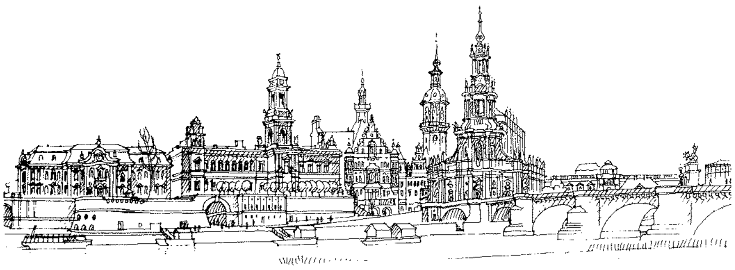
Michael Hohmuth

Institut für Betriebssysteme, Datenbanken und Rechnernetze

**The Fiasco Kernel:
Requirements Definition**

Technische Berichte
Technical Reports

ISSN 1430-211X



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

The Fiasco Kernel: Requirements Definition

Michael Hohmuth*
Dresden University of Technology

December 10, 1998
Revision : 1.4

Portions written by Jochen Liedtke

Technische Berichte — Technische Universität Dresden, Fakultät Informatik
ISSN 1430-211X
TUD–FI–12 — Dezember 1998

Abstract

Fiasco is a new implementation of the L4 microkernel interface for the x86 architecture. It is intended to be a drop-in replacement for L4/x86.

Fiasco is intended to have good real-time properties, and it should be small, fast, embeddable, maintainable, and well-documented. It does not need to be portable, but it should be able to serve as a model for future L4 implementations. It will be distributed under a freeware (open-source) license.

This document summarizes the preparations for the Fiasco implementation. It contains a definition of the problem which triggered the construction of Fiasco, a requirements definition, and pointers to documents containing further implementation prerequisites.

*Dresden University of Technology, Dept. of Computer Science, Operating Systems Group, D-01099 Dresden, Germany; Email: hohmuth@innocent.com

Contents

1 Why a New µ-Kernel?

1.1 The Problem

One foundation of the Dresden Real-Time Operating System (DROPS) [4] is the L4/x86 microkernel (µ-kernel). L4 is a µ-kernel interface defined by Jochen Liedtke, and there exist implementations for the x86 architecture (by Jochen Liedtke), the MIPS CPU (by UNSW [2]), and the Alpha CPU (by Sebastian Schönberg [18]). DROPS presently uses the x86 implementation exclusively.

Recently, a number of significant problems have cropped up with respect to Jochen's L4 implementations:

Maintainability. L4/x86 is not easily maintainable because it has been implemented in highly-optimized assembly language.

License and availability. L4/x86 is not freely redistributable. This has lost us a number of interesting industry and research partnerships.

Real-time properties. L4/x86 uses interrupt disabling as its sole kernel synchronization scheme. Under certain circumstances, this can lead to interrupts being disabled for time intervals which are unacceptably long. [6]

These problems make the L4/x86 unsuitable as a base for the DROPS operating system. Also, it would be desirable to use the µ-kernel in education, but the first two problems make this impossible.

1.2 The Solution

The L4/x86 µ-kernel shall be replaced with a new implementation of the L4 interface for the x86 architecture. This new µ-kernel, to be named "Fiasco," shall be a drop-in replacement for Jochen Liedtke's L4 implementation.

Fiasco shall have good real-time properties, and it shall be small, fast, embeddable, maintainable, and well-documented. It does not need to be portable, but it should serve as a model for future L4 implementations. It shall be distributed under a freeware (open-source) license.

Prospective users of Fiasco include the DROPS project and potentially industry users building embedded applications.

1.3 Structure of This Document

The following section, Section 2, formally defines the requirements for the Fiasco kernel. Section 3 contains pointers to documents containing further implementation prerequisites. The appendices contain detailed specifications concerning the kernel's functionality and interface.

2 Requirements Definition for the Fiasco Kernel

2.1 Functional Specification

The Fiasco kernel shall implement the functionality of an L4 µ-kernel as defined in the The L4 Reference Manual [16], Sections 1 (L4 in General), reprinted in this document as Appendix A (L4 Functional Specification).

2.2 Application Binary Interface

The application binary interface (ABI) shall consist of the following parts: system call interface, Int3 interface, and Sigma0 interface.

System Call Interface. The L4 Reference Manual [16], Sections 2.1–2.6, incompletely specifies L4's application binary interface (ABI). These sections are reprinted in this document as Appendix B (L4/x86 ABI) on page 26.

Fiasco shall implement the ABI as defined in this manual. Moreover, Fiasco shall offer a compatibility mode which implements all features of the latest available version of L4/x86 not defined in the L4 Reference Manual as they become known. These features must be documented in the user manual.

Fiasco may offer differing functionality in native (non-compatibility) mode. This functionality must be documented in the user manual.

Additional features may be required in future versions of this specification. In particular, changes in the following areas need to be planned for: new system calls; new system call parameters; and new kernel objects, like ports, as new alternative addresses for inter-process communication (IPC) messages.

Int3 Interface. Fiasco shall implement the L4 kernel's Int3 interface which can be used by application programs to print debug messages or to trap into the kernel debugger. This interface is specified in Libl4sys' `<l4/kdebug.h>` header file (see Appendix C on page 57 for a copy).

Sigma0 Interface. The Sigma0 interface (sometimes also written as "σ_0 interface") is an IPC protocol to communicate with an L4 task called "Sigma0" (or "σ_0") which is created by the kernel at boot time. This protocol is specified in the L4 Reference Manual [16], Section 2.7—reprinted in this document as Appendix B.7 (σ_0 RPC protocol) on page 54.

Fiasco shall be accompanied with a user-level implementation of Sigma0 which implements this interface. This Sigma0 implementation shall be made compatible with the L4/x86 µ-kernel (that is, it shall be possible to boot L4/x86 with this Sigma0 implementation instead of its own, native one).

Fiasco shall export to Sigma0 the data structure known as the *kernel info page* which is defined in the L4 Reference Manual [16], Section 2.5—reprinted in this document as Appendix B.5 (The Kernel-Info Page) on page 51. The physical memory address of this data structure shall be the first word on Sigma0's stack when Sigma0 is started up by the Fiasco kernel at boot time.

2.3 Hardware

Fiasco shall run on the x86 family of CPUs, up from (and including) the Intel i486 CPU.

Fiasco shall support standard personal computer (PC) hardware. Minimum hardware requirements shall be minimal, because Fiasco is intended to be run in embedded environments, but may include a VGA console output device. VGA and MGA (or Hercules) console adapters should be supported.

The kernel should support applications using floating point unit (FPU) instructions.

The number of CPUs supported in a given system shall be 1.

It is likely that this specification will be extended in the future, requiring support of symmetric multiprocessing (SMP) machines with more than 1 CPU. This needs to be planned for, and a strategy for changing the implementation for SMP support must be documented in the implementation notes.

2.4 Interface to External Software

Booting. Fiasco shall be booted using the L4 resource manager RMGR. Fiasco shall accept and, to the extent possible, evaluate boot options valid for L4/x86 [8]. The interface between RMGR and Fiasco (for booting purposes) is not specified, but must be documented in the implementation notes.

Source-Level Debugging. Fiasco shall be source-level debuggable remotely with a GNU Debugger `gdb` attached via a serial interface using `gdb`'s "`target remote`" command.

Profiling. Fiasco shall support kernel profiling using the GNU Profiler `gprof`. The profiling data (in `gmon.out` format) shall be dumped via a serial interface upon request. The transfer encoding of the profiling data is unspecified but must be documented in the user manual.

2.5 User Interface

The Fiasco kernel does not have a user interface per se. It only interacts with a user on special kernel events: debugging breakpoints, panics, and so on.

User interaction in these cases shall be possible using a local console (both VGA and MGA [Hercules] video adapters shall be supported), a serial terminal, or a remote `gdb` attached via a serial interface.

The details of the user interaction are not specified. It would be nice if the user interface was compatible with L4/x86's kernel debugger L4KD, but this is not required.

2.6 Performance

High performance, in particular, low latencies, are vitally important for µ-kernels. The introduction of a µ-kernel into a system architecture induces an inherent overhead because of the increased number of user-kernel-mode switches and address space switches, and because of its memory use. Liedtke argues in [15] that to be acceptable, the µ-kernel overheads must be close to the hardware-induced penalties included in these overheads.

There is, however, a tradeoff between a µ-kernel's performance and its maintainability. Because maintainability is a major concern for Fiasco (as discussed in Section 2.11 [Maintainability and Portability] on page 10), an additional high-level–programming-language penalty is acceptable.

The metric of acceptability for a given penalty is pragmatically defined as follows:

First let us define *critical kernel operations* as those operations which contribute to a majority of 80 percent of total CPU running time spent in the kernel, not including the idle loop. (According to the 80–20 rule, it is expected that about 20 percent of the kernel code spend these 80 percent of time in the kernel.) We now start from the assumption that the L4/x86 kernel implements these operations optimally with regard to our purposes.

The Fiasco kernel shall implement critical kernel operations with a latency that is at most twice as high as the corresponding L4/x86 latency. Non-critical kernel operation shall be implemented with a latency that is at most ten times as high as the corresponding L4/x86 latency.

Critical kernel operations can be determined in an iterative process using profiling. It is expected that the following operations are critical: short (register-only) IPC, short flexpage IPC, page unmapping, timer interrupt handler, interrupt propagation to user threads via IPC, and task and thread switching code (including the secondary cost induced by the kernel's TLB load). The implementation notes must document the kernel operations that are critical.

2.7 Real-Time Properties

The Fiasco kernel shall be preemptible: Interrupts shall be deliverable at virtually any time; specifically, hardware interrupt delivery shall not be disallowed for longer than 10 microseconds.[1]

Fiasco shall guarantee that a waiting thread that has the highest priority in the system will be scheduled no later than 20 microseconds after the event occurred that set it ready to run (for example, after an interrupt).

In general, a high-priority thread must not block because it is competing for resources with a lower-priority thread. The user manual must document all operations which do not guarantee this property, and must propose a work-around which avoids the blocking situation.

2.8 Security

Fiasco shall isolate task address spaces from each other using the CPU's hardware mechanisms (for example, using page tables or protected segments).

Fiasco shall protect resources it manages (memory, input–output ports, hardware interrupts, interrupt disable privilege) from unauthorized accesses. The L4 Reference

[1]Latencies given are relative to a 133 MHz Pentium CPU.

Manual [16] specifies how to authorize memory and port accesses in its Section 1.1.1, and how to authorize usage of hardware interrupts in its Section 2.3 (IPC system call, operation “associate interrupt”).

Usage of the interrupt-disable privilege shall be restricted to *privileged* tasks. A task is privileged if it has read–write access rights for all input–output ports.

Fiasco shall also implement a backward-compatibility mode where all tasks are created privileged. This is required to support legacy software which expects this misbehaviour of many versions of L4/x86.

The Sigma0 server task automatically has mapped one-to-one all of the physical RAM into its virtual address space. It shall ensure that no other task can read or modify the kernel code and data located in RAM. The memory regions reserved for exclusive use by the kernel are indicated in the kernel info page (see Section 2.2 [Application Binary Interface] on page 5, and the L4 Reference Manual [16], Section 2.5).

2.9 Error Recovery

The following error recovery strategies shall be available in Fiasco:

Panic. An error message is printed, and the kernel is halted. If configured, the kernel debugger is invoked. Otherwise, the kernel shall print a diagnostic message and reboot.

Kill Thread. The thread responsible for a fault is suspended.

Return error condition. Indicate to a user task that an error has occurred in the system call handler.

Kernel Event. A diagnostic message is printed. If the kernel debugger has been configured, it is invoked; otherwise, the kernel is suspended until a key has been pressed on the console. If the operator decides so, execution is resumed afterwards.

Failures can fall into one of three classes: hardware failures, kernel software failures, and user software failures.

Hardware Failure. Most hardware errors can not be catched by the Fiasco kernel because the kernel doesn’t contain any device drivers (except for clock device drivers, which are not expected to fail). The only kind of failure detected by the kernel is the RAM error. In this case, the kernel should just panic.

Kernel Software Failure. If the kernel detects an internal error, it should panic.

User Software Failure. Fiasco shall detect two kinds of user errors: invalid system call parameters, and unhandled traps.

If a system call parameter is invalid (e. g., an out-of-bound memory address), in general, the kernel should return an error condition code to the user task.

Failure	Recovery Strategy		
	conservative mode		always
Hardware failure			
Memory error			panic
Software failure (kernel)			
Assertion failed			panic
Unexpected trap			panic
Software failure (user task)			
Invalid system call parameter	kernel event	+	return error or trap
Unhandled trap	kernel event	+	kill thread

Table 1: Summary of error recovery strategies

However, sometimes this is not possible because the system call in question has been specified not to return error conditions. In this case, the affected thread should take a trap instead. For example, if an out-of-bounds instruction pointer has been passed as an argument to the `thread_ex_regs` system call, the affected thread should be modified anyway so that it will take a trap next time it is scheduled.

To handle traps, threads can install a thread-local interrupt descriptor table (IDT), as described in Appendix B.4 (Processor Mirroring) on page 50. If a trap occurs for which a handler has been installed, the handler shall be run. Otherwise, the thread shall be killed.

User tasks are expected to make few errors because they need to manage on their own the µ-kernel resources they use (e. g., task and thread numbers). Indeed, many error conditions detected in the kernel hint at a problem in the user code. That's why, to aid debugging of user tasks, Fiasco shall also implement a *conservative mode* that produces a kernel event every time a user software failure is detected.

Table 1 summarizes the error recovery strategies discussed in this section.

2.10 Resource Requirements

Main Memory. Fiasco shall have a small main memory footprint. The size of the kernel's code and static data shall be smaller than 100 kilobytes. The kernel's dynamic data size shall be smaller than 10 percent of the total RAM size on systems with more than 64 megabytes of RAM.

Disk Space. Fiasco shall take up less than 100 kilobytes of disk space.

CPU Time. Fiasco's CPU requirements are discussed in Section 2.6 (Performance) on page 6.

2.11 Maintainability and Portability

Because of its intended fields of application (research, education, commercial employment), maintainability is a major concern for Fiasco.

The Fiasco kernel shall be written in a high-level language. Where required to implement low-level operations or to meet the performance requirements spelled out in Section 2.6 (Performance) on page 6, Assembler source code is permissible. The relative amount of Assembler in the Fiasco source code shall not exceed 5 percent.

The source code of Fiasco shall be organized in subsystems with clean programming interfaces so that it is easy to replace subsystem implementations.

The high-level programming language source code is not required to be portable across platforms. However, the source code shall be clean and understandable enough to be able to serve as a model for new implementations of the L4 µ-kernel interface.

2.12 Documentation

Fiasco shall be accompanied with extensive documentation that allows it to be used and altered seamlessly. In particular, the following documents are required:

- This requirements specification.
- User manual. This manual shall document Fiasco's L4 ABI, ABI extensions and clarifications, command line options, real-time properties, and the debugging and profiling processes.
- System architecture handbook. This document shall describe the high-level design of the Fiasco kernel.
- Implementation notes. This manual shall describe implementation details of the Fiasco kernel to make it easy to adapt it to new applications.

2.13 License

Fiasco shall be distributed under a freeware (open-source) license. The Open Source Definition[2] defines the requirements for open-source licenses.

From this requirement it follows that Fiasco shall not contain components which are not freeware, and that all tools and documentation necessary to build and use Fiasco should also be freeware.

[2] URL: `http://www.opensource.org/osd.html`

3 Further Reading

A companion document to this one, the Fiasco system architecture specification [7], contains information on more implementation prerequisites. It describes the Fiasco kernel's architecture in detail, chooses a programming language and explains programming conventions which should be used for the implementation.

A L4 Functional Specification

This appendix is a reprint of Section 1 of the L4 Reference Manual [16], by Jochen Liedtke, and is used here with kind permission.[3]

The following subsections define the functionality of an L4 μ-kernel, independent from implementation and architecture.

A.1 Basic Concepts

The following section contains excerpts from [14, 13, 15].

We reason about the minimal concepts or "primitives" that a μ-kernel should implement.[4] The determining criterion used is functionality, not performance. More precisely, a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

We assume that the target system has to support interactive and/or not completely trustworthy applications, i.e. it has to deal with protection. We further assume that the hardware implements page-based virtual memory.

One inevitable requirement for such a system is that a programmer must be able to implement an arbitrary subsystem S in such a way that it cannot be disturbed or corrupted by other subsystems S'. This is the principle of independence: S can give guarantees independent of S'. The second requirement is that other subsystems must be able to rely on these guarantees. This is the principle of integrity: there must be a way for S_1 to address S_2 and to establish a communication channel which can neither be corrupted nor eavesdropped by S'.

Provided hardware and kernel are trustworthy, further security services, like those described by [3], can be implemented by servers. Their integrity can be ensured by system administration or by user-level boot servers. For illustration: a key server should deliver public-secret RSA key pairs on demand. It should guarantee that each pair has the desired RSA property and that each pair is delivered only once and only to the demander. The key server can only be realized if there are mechanisms which (a) protect its code and data, (b) ensure that nobody else reads or modifies the key and (c) enable the demander to check whether the key comes from the key server. Finding the key server can be done by means of a name server and checked by public key based authentication.

A.1.1 Address Spaces

At the hardware level, an *address space* is a mapping which associates each virtual page to a physical page frame or marks it 'non-accessible'. For the sake of simplicity,

[4]Proving minimality, necessarity and completeness would be nice but is impossible, since there is no agreed-upon metric and all is Turing-equivalent.

we omit access attributes like read-only and read/write. The mapping is implemented by TLB hardware and page tables.

The µ-kernel, the mandatory layer common to all subsystems, has to hide the hardware concept of address spaces, since otherwise, implementing protection would be impossible. The µ-kernel concept of address spaces must be tamed, but must permit the implementation of arbitrary protection (and non-protection) schemes on top of the µ-kernel. It should be simple and similar to the hardware concept.

The basic idea is to support recursive construction of address spaces outside the kernel. By magic, there is one address space σ_0 which essentially represents the physical memory and is controlled by the first subsystem S_0. At system start time, all other address spaces are empty. For constructing and maintaining further address spaces on top of σ_0, the µ-kernel provides three operations:

Grant. The owner of an address space can *grant* any of its pages to another space, provided the recipient agrees. The granted page is removed from the granter's address space and included into the grantee's address space. The important restriction is that instead of physical page frames, the granter can only grant pages which are already accessible to itself.

Map. The owner of an address space can *map* any of its pages into another address space, provided the recipient agrees. Afterwards, the page can be accessed in both address spaces. In contrast to granting, the page is not removed from the mapper's address space. Comparable to the granting case, the mapper can only map pages which itself already can access.

Flush. The owner of an address space can *flush* any of its pages. The flushed page remains accessible in the flusher's address space, but is removed from all other address spaces which had received the page directly or indirectly from the flusher. Although explicit consent of the affected address-space owners is not required, the operation is safe, since it is restricted to own pages. The users of these pages already agreed to accept a potential flushing, when they received the pages by mapping or granting.

Reasoning

The described address-space concept leaves memory management and paging outside the µ-kernel; only the grant, map and flush operations are retained inside the kernel. Mapping and flushing are required to implement memory managers and pagers on top of the µ-kernel.

The grant operation is required only in very special situations: consider a pager F which combines two underlying file systems (implemented as pagers f_1 and f_2, operating on top of the standard pager) into one unified file system (see figure 1). In this example, f_1 maps one of its pages to F which grants the received page to *user A*. By granting, the page disappears from F so that it is then available only in f_1 and *user A*; the resulting mappings are denoted by the thin line: the page is mapped in *user A*, f_1 and the standard pager. Flushing the page by the standard pager would affect f_1 and *user A*, flushing by f_1 only *user A*. F is not affected by either flush (and cannot flush itself), since it used the page only transiently. If F had used mapping instead of

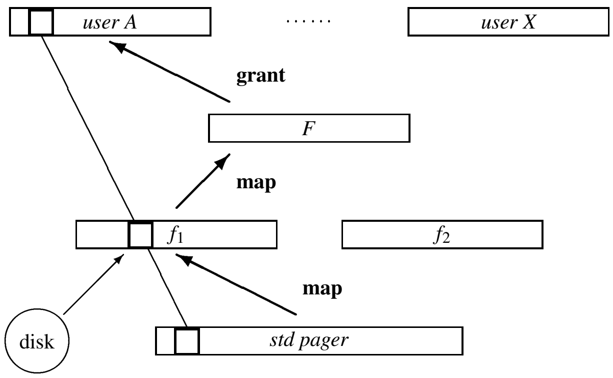


Figure 1: *A Granting Example.*

granting, it would have needed to replicate most of the bookkeeping which is already done in f_1 and f_2. Furthermore, granting avoids a potential address-space overflow of F.

In general, granting is used when page mappings should be passed through a controlling subsystem without burdening the controller's address space by all pages mapped through it.

The model can easily be extended to access rights on pages. Mapping and granting copy the source page's access right or a subset of them, i.e., can restrict the access but not widen it. Special flushing operations may remove only specified access rights.

I/O

An address space is the natural abstraction for incorporating device ports. This is obvious for memory mapped I/O, but I/O ports can also be included. The granularity of control depends on the given processor. The 386 and its successors permit control per port (one very small page per port) but no mapping of port addresses (it enforces mappings with v=v'); the PowerPC uses pure memory mapped I/O, i.e., device ports can be controlled and mapped with 4K granularity. Controlling I/O rights and device drivers is thus also done by memory managers and pagers on top of the µ-kernel.

An Abstract Model of Address Spaces

We describe address spaces as mappings. $\sigma_0 : V \rightarrow R \cup \{\phi\}$ is the initial address space, where V is the set of virtual pages, R the set of available physical (real) pages and ϕ the nilpage which cannot be accessed. Further address spaces are defined recursively as mappings $\sigma : V \rightarrow (\Sigma \times V) \cup \{\phi\}$, where Σ is the set of address spaces. It is convenient to regard each mapping as a one column table which contains $\sigma(v)$ for all $v \in V$ and can be indexed by v. We denote the elements of this table by σ_v.

All modifications of address spaces are based on the replacement operation: we write $\sigma_v \leftarrow x$ to describe a change of σ at v, precisely:

$$\text{flush}\,(\sigma, v) \quad ; \quad \sigma_v := x \quad .$$

A page potentially mapped at v in σ is flushed, and the new value x is copied into σ_v. This operation is internal to the µ-kernel. We use it only for describing the three exported operations.

A subsystem S with address space σ can *grant* any of its pages v to a subsystem S' with address space σ' provided S' agrees:

$$\sigma'_{v'} \leftarrow \sigma_v \quad , \quad \sigma_v \leftarrow \phi \quad .$$

Note that S determines which of its pages should be granted, whereas S' determines at which virtual address the granted page should be mapped in σ'. The granted page is transferred to σ' and removed from σ.

A subsystem S with address space σ can *map* any of its pages v to a subsystem S' with address space σ' provided S' agrees:

$$\sigma'_{v'} \leftarrow (\sigma, v) \quad .$$

In contrast to grant, the mapped page remains in the mapper's space σ and *a link to the page in the mapper's address space* (σ, v) *is stored in the receiving address space* σ', instead of transferring the existing link from σ_v to $\sigma'_{v'}$. This operation permits to construct address spaces recursively, i.e. new spaces based on existing ones.

Flushing, the reverse operation, can be executed without explicit agreement of the mappees, since they agreed implicitly when accepting the prior map operation. S can *flush* any of its pages:

$$\forall_{\sigma'_{v'} = (\sigma, v)} : \sigma'_{v'} \leftarrow \phi \quad .$$

Note that $\leftarrow$ and *flush* are defined recursively. Flushing recursively affects also all mappings which are indirectly derived from σ_v.

No cycles can be established by these three operations, since $\leftarrow$ flushes the destination prior to copying.

Implementing the Model

At a first glance, deriving the phyical address of page v in address space σ seems to be rather complicated and expensive:

$$\sigma(v) = \begin{cases} \sigma'(v') & \text{if } \sigma_v = (\sigma', v') \\ r & \text{if } \sigma_v = r \\ \phi & \text{if } \sigma_v = \phi \end{cases}$$

Fortunately, a recursive evaluation of $\sigma(v)$ is never required. The three basic operations guarantee that the physical address of a virtual page will never change, except by flushing. For implementation, we therefore complement each σ by an additional table P, where P_v corresponds to σ_v and holds either the physical address of v or ϕ. Mapping and granting then include

$$P'_{v'} := P_v$$

and each replacement $\sigma_v \leftarrow \phi$ invoked by a flush operation includes

$$P_v := \phi \quad .$$

P_v can always be used instead of evaluating $\sigma(v)$. In fact, P is equivalent to a hardware page table. μ-kernel address spaces can be implemented straightforward by means of the hardware-address-translation facilities.

The recommended implementation of σ is to use one mapping tree per physical page frame which describes all actual mappings of the frame. Each node contains (P, v), where v is the according virtual page in the address space which is implemented by the page table P.

Assume that a grant-, map- or flush-operation deals with a page v in address space σ to which the page table P is associated. In a first step, the operation selects the according tree by P_v, the physical page. In the next step, it selects the node of the tree that contains (P, v). (This selection can be done by parsing the tree or in a single step, if P_v is extended by a link to the node.) Granting then simply replaces the values stored in the node and map creates a new child node for storing (P', v'). Flush lets the selected node unaffected but parses and erases the complete subtree, where $P'_{v'} := \phi$ is executed for each node (P', v') in the subtree.

A.1.2 Threads and IPC

A *thread* is an activity executing inside an address space. A thread τ is characterized by a set of registers, including at least an instruction pointer, a stack pointer and a state information. A thread's state also includes the address space $\sigma^{(\tau)}$ in which τ currently executes. This dynamic or static association to address spaces is the decisive reason for including the thread concept (or something equivalent) in the μ-kernel. To prevent corruption of address spaces, all changes to a thread's address space ($\sigma^{(\tau)} := \sigma'$) must be controlled by the kernel. This implies that the μ-kernel includes the notion of some τ that represents the above mentioned activity. In some operating systems, there may be additional reasons for introducing threads as a basic abstraction, e.g. preemption. Note that choosing a concrete thread concept remains subject to further OS-specific design decisions.

Consequently, cross-address-space communication, also called inter-process communication (IPC), must be supported by the μ-kernel. The classical method is transferring messages between threads by the μ-kernel.

IPC always enforces a certain agreement between both parties of a communication: the sender decides to send information and determines its contents; the receiver determines whether it is willing to receive information and is free to interpret the received message. Therefore, IPC is not only the basic concept for communication between subsystems but also, together with address spaces, the foundation of independence.

Other forms of communication, remote procedure call (RPC) or controlled thread migration between address spaces, can be constructed from message-transfer based IPC.

Note that the *grant* and *map* operations (section A.1.1) need IPC, since they require an agreement between granter/mapper and recipient of the mapping.

Interrupts

The natural abstraction for hardware interrupts is the IPC message. The hardware is regarded as a set of threads which have special thread ids and send empty messages (only consisting of the sender id) to associated software threads. A receiving thread concludes from the message source id, whether the message comes from a hardware interrupt and from which interrupt:

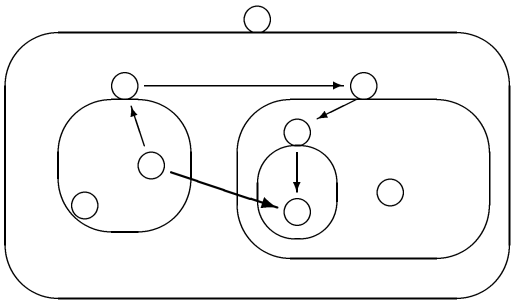
```
driver thread:
    do
        wait for (msg, sender) ;
        if sender = my hardware interrupt
            then   read/write io ports ;
                   reset hardware interrupt
            else   . . .
        fi
    od .
```

Transforming the interrupts into messages must be done by the kernel, but the μ-kernel is not involved in device-specific interrupt handling. In particular, it does not know anything about the interrupt semantics. On some processors, resetting the interrupt is a device specific action which can be handled by drivers at user level. The `iret`-instruction then is used solely for popping status information from the stack and/or switching back to user mode and can be hidden by the kernel. However, if a processor requires a privileged operation for releasing an interrupt, the kernel executes this action implicitly when the driver issues the next IPC operation.

A.1.3 Clans & Chiefs

Within all systems based on direct message transfer, protection is essentially a matter of message control. Using access control lists (acl) this can be done at the server level, but maintenance of large distributed acls becomes hard when access rights change rapidly. So [5] have proposed that object (passive entity) protection be complemented by subject (active entity) restrictions. In this approach the kernel is able to restrict the outgoing messages of a task (the subject) by means of a list of permitted receivers.

The clan concept [12] is an algorithmic generalization of this idea:



A *clan* (denoted as an oval) is a set of tasks (denoted as a circle) headed by a *chief* task. Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's chief. This chief may inspect the message (including the sender and receiver ids as well as the contents) and decide whether or not it should be passed to the destination to which it was addressed. As demonstrated in the figure above, these rules apply to nested clans as well. Obviously subject restrictions and local reference monitors can be implemented outside the kernel by means of clans. Since chiefs are tasks at user level, the clan concept allows more sophisticated and user definable checks as well as active control. Typical clan structures are

Clan per machine: In this simple model there is only one clan per machine covering all tasks. Local communication is handled directly by the kernel without incorporating a chief, whereas cross machine communication involves the chief of the sending and the receiving machine. Hence, the clan concept is used for implementing remote ipc by user level tasks.

Clan per system version: Sometimes chiefs are used for adapting different versions. The servers of the old or new versions are encapsulated by a clan so that its chief can translate the messages.

Clan per user: Surrounding the tasks of each user or user group by a clan is a typical method when building security systems. Then the chiefs are used to control and enforce the requested security policy.

Clan per task: In the extreme case there are single tasks each controlled by a specific chief. For example these one-task-clans are used for debugging and supervising suspicious programs.

In the case of intra-clan communication (no chief involved), the additional costs of the clan concept are negligible (below 1% of minimal ipc time). Inter-clan communication however multiplies the ipc operations by the number of chiefs involved. This can be tolerated, since (i) L3 ipc is very fast (see above) and (ii) crossing clan boundaries occurs seldom enough in practice. Note that many security policies can be implemented simply by checking the client id in the server and do not need clans.

A.1.4 Unique Identifiers

A µ-kernel must supply unique identifiers (uid) for something, either for threads or tasks or communication channels. Uids are required for reliable and efficient local communication. If S_1 wants to send a message to S_2, it needs to specify the destination S_2 (or some channel leading to S_2). Therefore, the µ-kernel must know which uid relates to S_2. On the other hand, the receiver S_2 wants to be sure that the message comes from S_1. Therefore the identifier must be unique, both in space and time.

In theory, cryptography could also be used. In practice, however, enciphering messages for local communication is far too expensive and the kernel must be trusted anyway. S_2 can also not rely on purely user-supplied capabilities, since S_1 or some other instance could duplicate and pass them to untrusted subsystems without control of S_2.

A.1.5 Flexibility

To illustrate the flexibility of the basic concepts, we sketch some applications which typically belong to the basic operating system but can easily be implemented on top of the µ-kernel.

Memory Manager. A server managing the initial address space σ_0 is a classical main memory manager, but outside the µ-kernel. Memory managers can easily be stacked: M_0 maps or grants parts of the physical memory (σ_0) to σ_1, controlled by M_1, other parts to σ_2, controlled by M_2. Now we have two coexisting main memory managers.

Pager. A Pager may be integrated with a memory manager or use a memory managing server. Pagers use the µ-kernel's grant, map and flush primitives. The remaining interfaces, pager – client, pager – memory server and pager – device driver, are completely based on IPC and are user-level defined.

Pagers can be used to implement traditional paged virtual memory and file/database mapping into user address spaces as well as unpaged resident memory for device drivers and/or real time systems. Stacked pagers, i.e. multiple layers of pagers, can be used for combining access control with existing pagers or for combining various pagers (e.g. one per disk) into one composed object. User-supplied paging strategies [11, 1] are handled at the user level and are in no way restricted by the µ-kernel. Stacked file systems [10] can be realized accordingly.

Multimedia Resource Allocation. Multimedia and other real-time applications require memory resources to be allocated in a way that allows predictable execution times. The above mentioned user-level memory managers and pagers permit e.g. fixed allocation of physical memory for specific data or locking data in memory for a given time.

Note that resource allocators for multimedia and for timesharing can coexist. Managing allocation conflicts is part of the servers' jobs.

Device Driver. A device driver is a process which directly accesses hardware I/O ports mapped into its address space and receives messages from the hardware (interrupts) through the standard IPC mechanism. Device-specific memory, e.g. a screen, is handled by means of appropriate memory managers. Compared to other user-level processes, there is nothing special about a device driver. No device driver has to be integrated into the µ-kernel.[5]

[5]In general, there is no reason for integrating boot drivers into the kernel. The booter, e.g. located in ROM, simply loads a bit image into memory that contains the micro-kernel and perhaps some set of initial pagers and drivers (running at user mode and *not* linked but simply appended to the kernel). Afterwards, the boot drivers are no longer used.

Second Level Cache and TLB. Improving the hit rates of a secondary cache by means of page allocation or reallocation [9, 17] can be implemented by means of a pager which applies some cache-dependent (hopefully conflict reducing) policy when allocating virtual pages in physical memory.

In theory, even a software TLB handler could be implemented like this. In practice, the first-level TLB handler will be implemented in the hardware or in the µ-kernel. However, a second-level TLB handler, e.g. handling misses of a hashed page table, might be implemented as a user-level server.

Remote Communication. Remote IPC is implemented by communication servers which translate local messages to external communication protocols and vice versa. The communication hardware is accessed by device drivers. If special sharing of communication buffers and user address space is required, the communication server will also act as a special pager for the client. The µ-kernel is not involved.

Unix Server. Unix[6] system calls are implemented by IPC. The Unix server can act as a pager for its clients and also use memory sharing for communicating with its clients. The Unix server itself can be pageable or resident.

Conclusion. A small set of µ-kernel concepts lead to abstractions which stress flexibility, provided they perform well enough. The only thing which cannot be implemented on top of these abstractions is the processor architecture, registers, first-level caches and first-level TLBs.

[6]Unix is a registered trademark of UNIX System Laboratories.

A.2 Data Types

A.2.1 Unique Ids

Unique ids identify tasks, threads and hardware interrupts. They are also unique in time. Unique ids are 64-bit values.

A.2.2 User-Level Operations on Uids

a = b :	a = b
task(a) = task (b) :	(a AND NOT lthread mask) = (b AND NOT lthread mask)
chief(a) = chief (b) :	(a AND NOT chief mask) = (b AND NOT chief mask)
site(a) = site (b) :	(a AND NOT site mask) = (b AND NOT site mask)
lthread no(a) :	(a AND lthread mask) SHR lthread shift
thread(a,n) :	(a AND NOT lthread mask) + (n SHL lthread shift)
task no(a) :	(a AND task mask) SHR task shift
chief no(a) :	(a AND chief mask) SHR chief shift
site no(a) :	(a AND site mask) SHR site shift

A.2.3 Fpages

Fpages (Flexpages) are regions of the virtual address space. An fpage consists of all pages actually mapped in this region. The minimal fpage size is the minimal hardware-page size.

An fpage of size 2^s has a 2^s-aligned base address b, i.e. b mod 2^s=0. An fpage with base address b and size 2^s is denoted by the 32-bit value

$$b + 4s.$$

On x86 processors, the smallest possible value for s is 12, since the hardware page size is 4K.

A.2.4 Messages

S :: snd ; EMPTY .

R :: rcv ; EMPTY .

EMPTY :: .

S R message:	rcv fpage option , size dope , S R msg dope , S R mwords , S R string dopes .	
rcv fpage option:	rcv fpage:fpage ; zero:word.	
size dope:	reserved:byte , string dope number:5bits , mwords number:19bits .	 $= S$ $= W$
snd R msg dope:	undefined:byte , string dope number:5bits , mwords number:19bits .	 $= s \quad s \leq S$ $= w \quad w \leq W$
rcv msg dope:	undefined:word .	
snd R mwords:	$w \times$ send receive word , $(W - w) \times$ receive word ; $m \times$ snd fpage receive double word , $w - 2m \times$ send receive words , $(W - w) \times$ receive word .	 $2m \leq w$
rcv mwords:	$W \times$ receive word .	
snd R string dopes:	$s \times$ snd R string dope , $(S - s) \times$ R string dope .	
rcv string dopes:	$S \times$ rcv string dope .	

snd rcv string dope:	snd addr:word ,		
	snd size:word ,	$\leq$ 4MB	
	rcv addr:word ,		
	rcv size:word .	$\leq$ 4MB	
snd string dope:	snd addr:word ,		
	snd size:word ,	$\leq$ 4MB	
	undefined:word ,		
	undefined:word .		
rcv string dope:	undefined:word ,		
	undefined:word ,		
	rcv addr:word ,		
	rcv size:word .	$= s_r$	$s_r \leq$ 4MB
snd map fpage:	grant flag:1bit ,		
	write flag:1bit ,		
	snd base:30bits ,		
	snd fpage:fpage .		

A.3 μ-Kernel Calls

ipc (dest option, snd descriptor option, rcv descriptor option, timeouts) $\longrightarrow$ (source option, result code)

CALL (dest, snd descriptor, closed rcv descriptor, timeouts) $\longrightarrow$ (dest option, result code)

SEND/RECEIVE (dest, snd descriptor, open rcv descriptor, timeouts) $\longrightarrow$ (source option, result code)

SEND (dest, snd descriptor, –nil– , timeouts) $\longrightarrow$ ($\sim$, result code)

RECEIVE FROM (source, –nil– , closed rcv descriptor, timeouts) $\longrightarrow$ (source option, result code)

RECEIVE ($\sim$, –nil– , open rcv descriptor, timeouts) $\longrightarrow$ (source option, result code)

RECEIVE INTR (intr, –nil– , closed rcv descriptor, timeouts) $\longrightarrow$ (source option, result code)

SLEEP (–nil– , –nil– , closed rcv descriptor, timeouts) $\longrightarrow$ ($\sim$, result code)

id_nearest (dest id) $\longrightarrow$ (nearest id)

fpage_unmap (fpage, map mask) $\longrightarrow$ ()

thread_switch (dest) $\longrightarrow$ ()

lthread_ex_regs (lthread no, SP, IP, int preempter, pager)
$\longrightarrow$ (FLAGS, SP, IP, int preempter, pager)

thread_schedule (dest, prio, timeslice, ext preempter)
$\longrightarrow$ (prio, timeslice, state, ext preempter, partner, time)

task_new (dest task id, mcp/new chief, SP, IP, pager id) $\longrightarrow$ (new task id)

B L4/x86 ABI

This appendix is a reprint of Sections 2.1–2.7 of the L4 Reference Manual [16], by Jochen Liedtke, and is used here with kind permission.[7]

The following subsections define the ABI of L4/x86.

B.1 Notational conventions

∼	If this refers to an input parameter, its value is meaningless. If it refers to an output parameter, its value is undefined.
EAX,ECX...	denote the processor's general registers.
$\langle$SP+$n\rangle$	denotes the word on the user stack addressed by SP+n, where SP represents the user-level stack pointer.

B.2 Data Types

B.2.1 Unique Ids

Unique ids identify tasks, threads and hardware interrupts. Each unique id is a 64-bit value which is unique in time. An unique id in x86 format consists of two 32-bit words:

	nest (4)	chief (11)	site (17)		+4
thread id	ver1 (4)	task (11)	lthread (7)	ver0 (10)	0

	nest (4)	chief (11)	site (17)		+4
task id	ver1 (4)	task (11)	0 (7)	ver0 (10)	0

	∼ (32)		+4
interrupt id	0 (24)	intr + 1 (8)	0

[7] Copyright © 1995–1996 GMD German National Research Center for Information Technology, Copyright © 1996–1998 IBM T. J. Watson Research Center

	$\sim_{(32)}$	+4
nil id	$0_{(32)}$	0

	$\sim_{(32)}$	+4
invalid id	FFFFFFFF $_{(32)}$	0

B.2.2 Fpages

Fpages (Flexpages) are regions of the virtual address space. An fpage consists of all pages actually mapped in this region. The minimal fpage size is 4 K, the minimal hardware-page size.

An fpage of size 2^s has a 2^s-aligned base address b, i.e. b mod 2^s=0. On the x86 processors, the smallest possible value for s is 12, since hardware pages are at least 4K. The complete user address space (base address 0, size $2^{32} - K$, where K is the size of the kernel area) is denoted by $b = 0, s = 32$. An fpage with base address b and size 2^s is denoted by a 32-bit word:

fpage $(b, 2^s)$	$b/4096_{(20)}$	$0_{(4)}$	$s_{(6)}$	$\sim$	$\sim$
fpage $(0, 2^{32} - K)$	$0_{(20)}$	$0_{(4)}$	$32_{(6)}$	$\sim$	$\sim$

B.2.3 IO-Ports

On the 486, IO-ports form a separate address space besides the conventional memory address space. Its size is 64 K and its granularity is 16 bytes. However, IO-ports can only be mapped idempotently, i.e. physical port x is either mapped at the address x in the task's IO address space or it is not mapped.

L4 handles IO-ports like memory, i.e. as fpages. IO-fpages can be mapped, granted and unmapped like memory fpages. However, since IO-ports can only mapped idempotent, always the complete IO space (64 K) should be specified as receive fpage.

An IO-fpage of size 2^s ($4 \leq s \leq 16$) has a 2^s-aligned base address p, i.e. p mod 2^s=0. An fpage with base port address p and size 2^s is denoted by a 32-bit word:

IO-fpage $(p, 2^s)$	F0 $_{(8)}$	$p_{(12)}$	$0_{(4)}$	$s_{(6)}$	$\sim$	$\sim$
IO-fpage $(0, 2^{16})$	F0 $_{(8)}$	$0_{(12)}$	$0_{(4)}$	$16_{(6)}$	$\sim$	$\sim$

B.2.4 Messages

⋮

	dword 1 $_{(32)}$			+16
msg dwords:	dword 0 $_{(32)}$			+12
msg snd dope:	dwords $_{(19)}$	strings $_{(5)}$	$\sim_{(8)}$	+8
msg size dope:	dwords $_{(19)}$	strings $_{(5)}$	$\sim_{(8)}$	+4
msg rcv fpage option:	fpage $_{(32)}$			0

	*rcv string $_{(32)}$	+12
	rcv string size $_{(32)}$	+8
	*snd string $_{(32)}$	+4
string dope:	snd string size $_{(32)}$	0

	snd fpage $_{(30)}$	w	g	+4
snd fpage:	snd base $_{(32)}$			0

B.2.5 Timeouts

Timeouts are used to control ipc operations. The *send timeout* determines how long ipc should try to send a message. If the specified period is exhausted without that message transfer could start, ipc fails. The *receive timeout* specifies how long ipc should wait for an incoming message. Both timouts specify the maximum period of time *before message transfer starts*. Once started, message transfer is no longer influenced by send or receive timeout.

Pagefaults occuring during ipc are controlled by *send* and *receive pagefault timeout*. A pagefault is translated to an RPC by the kernel. In the case of a pagefault in the receiver's address space, the corresponding RPC to the pager uses *send pagefault timeout* (specified by the sender) for both send and receive timeout. In the case of a pagefault in the sender's address space, *receive pagefault timeout* specified by the receiver is taken.

Besides the special timeouts 0 (do not wait at all) and ∞ (wait forever), periods from 1 µs up to approximately 19 hours can be specified. The complete quadruple is packed into one 32-bit word:

m_r $_{(8)}$	m_s $_{(8)}$	p_s $_{(4)}$	p_r $_{(4)}$	e_s $_{(4)}$	e_r $_{(4)}$

Note that for efficiency reasons the highest bit of any mantissa *m* *must* be 1, except for *m*=0.

$$
\text{snd timeout} \quad = \quad \begin{cases} \infty & \text{if} \quad e_s{=}0 \\ 4^{15-e_s} m_s \ \mu s & \text{if} \quad e_s{>}0 \\ 0 & \text{if} \quad m_s{=}0, e_s{\neq}0 \end{cases}
$$

$$
\text{rcv timeout} \quad = \quad \begin{cases} \infty & \text{if} \quad e_r{=}0 \\ 4^{15-e_r} m_r \ \mu s & \text{if} \quad e_r{>}0 \\ 0 & \text{if} \quad m_r{=}0, e_r{\neq}0 \end{cases}
$$

$$
\text{snd pagefault timeout} \quad = \quad \begin{cases} \infty & \text{if} \quad p_s{=}0 \\ 4^{15-p_s} \mu s & \text{if} \quad 0{<}p_s{<}15 \\ 0 & \text{if} \quad p_s{=}15 \end{cases}
$$

$$
\text{rcv pagefault timeout} \quad = \quad \begin{cases} \infty & \text{if} \quad p_r{=}0 \\ 4^{16-p_r} \mu s & \text{if} \quad 0{<}p_r{<}15 \\ 0 & \text{if} \quad p_r{=}15 \end{cases}
$$

approximate timeout ranges		
e_s, e_r, p_s, p_r	snd/rcv timeout	pf timeout
0	∞	∞
1	256 s … 19 h	256 s
2	64 s … 55 h	64 s
3	16 s … 71 m	16 s
4	4 s … 17 m	4 s
5	1 s … 4 m	1 s
6	262 ms … 67 s	256 ms
7	65 ms … 17 s	64 ms
8	16 ms … 4 s	16 ms
9	4 ms … 1 s	4 ms
10	1 ms … 261 ms	1 ms
11	256 µs … 65 ms	256 µs
12	64 µs … 16 ms	64 µs
13	16 µs … 4 ms	16 µs
14	4 µs … 1 ms	4 µs
15	1 µs … 255 µs	0
$m{=}0, e{>}0$	0	—

B.3 µ-Kernel Calls

This section describes the 7 system calls of L4:

• ipc	int 30
• id_nearest	int 31
• fpage_unmap	int 32
• thread_switch	int 33
• thread_schedule	int 34
• lthread_ex_regs	int 35
• task_new	int 36

ipc

snd descriptor	EAX		EAX	*msg.dope + cc*	/	*cc*
timeouts	ECX		ECX	∼	/	∼
msg.w0	EDX		EDX	*msg.w0*	/	∼
msg.w1	EBX	— INT 0x30 ⟶	EBX	*msg.w1*	/	∼
rcv descriptor	EBP		EBP	∼	/	∼
dest.low	ESI		ESI	*source.low*	/	∼
dest.high	EDI		EDI	*source.high*	/	∼
virtual sender id.low / ∼	⟨SP⟩					
virtual sender id.high / ∼	⟨SP+4⟩					

This is the basic system call for inter-process communication and synchronization. It may be used for intra- as inter-address-space communication. All communication is synchronous and unbuffered: a message is transferred from the sender to the recipient if and only if the recipient has invoked a corresponding ipc operation. The sender blocks until this happens or a period specified by the sender elapsed whithout that the destination became ready to receive.

Ipc can be used to copy data as well as to *map* or *grant* fpages from the sender to the recipient. For the description of messages see section B.2.4.

8-byte messages (plus 64-bit sender id) can be transferred solely via the registers and are thus specially optimized. If possible, short messages should therefore be reduced to 8-byte messages.

A single ipc call combines an optional send operation with an optional receive operation. Whether it includes a send respectively a receive is determined by the actual parameters. If the send or receive address is specified as *nil* (0xFFFFFFFF), the corresponding operation is skipped.

No time is required for the transition between send and receive phase of one ipc operation.

Parameters

snd descriptor

“nil”

0xFFFFFFFF (32)

Ipc does not include a send operation.

“mem”

*snd msg/4 (30)	*m*	*d*

Ipc includes sending a message to the destination specified by *dest id*. *snd msg must point to a valid message. The first two 32-bit words of the message (*msg.w0* and *msg.w1*) are *not* taken from the message data structure but must be contained in registers EDX and EBX.

“reg”

0 (30)	0	*d*

Ipc includes sending a message to the destination specified by *dest id*. The message consists solely of the two 32-bit words *msg.w0* and *msg.w1* in registers EDX and EBX.

m=0 Value-copying send operation; the dwords or the message are simply copied to the recipient.

snd descriptor *m*=1 Fpage-mapping send operation. The dwords of the message to be sent are treated as ’send fpages’. The described fpages are mapped (respectively granted) into the recipient’s address space. Mapping/granting stops when either the end of the dwords is reached or when an invalid fpage denoter is found, in particular 0. The send fpage descriptors and all potentially following words are also transferred by simple copy to the recipient. Thus a message may contain some fpages and additional value parameters. The recipient can use the received fpage descriptors to determine what has been mapped or granted into its address space, including location and access rights.

d=0 Normal send operation. The recipient gets the true sender id.

d=1 Deceiving send operation. A chief can specify the *virtual-sender id* which the recipient should get instead of the chief’s id. The *virtual-sender-id* parameter on the user stack is only required if *d*=1. Recall that deceiving is secure, since only *direction-preserving deceit* is possible. If the specified *virtual-sender id* does not fulfil this constraint, the send operation works like *d*=0.

rcv descriptor

“nil”

0xFFFFFFFF $_{(32)}$

Ipc does not include a receive operation.

“mem”

*snd msg/4 $_{(30)}$	0	*o*

Ipc includes receiving a message respectively waiting to receive a message. *rcv msg must point to a valid message. The first two 32-bit words of the received message (*msg.w0* and *msg.w1*) are *not* stored in the message data structure but are returned in registers EDX and EBX.

“reg”

0 $_{(30)}$	0	*o*

Ipc includes receiving a message respectively waiting to receive a message. However, only messages up to two 32-bit words *msg.w0* and *msg.w1* are accepted. The received message is returned in registers EDX and EBX.

“rmap”

rcv fpage $_{(30)}$	1	*o*

Ipc includes receiving a message respectively waiting to receive a message. However, only send-fpage messages or up to two 32-bit words *msg.w0* and *msg.w1* are accepted. The received message is returned in registers EDX and EBX. If a map message is received, “rcv fpage” describes the receive fpage (instead of “rcv fpage option” in a memory message buffer). Thus fpages can also be received without a message buffer in memory.

o=0 Only messages from the thread specified as *dest id* are accepted (“closed wait”). Any send operation from a different thread (or hardware interrupt) will be handled exactly as if the actual thread would be busy.

o=1 Messages from any thread will be accepted (“open wait”). If the actual thread is associated to a hardware interrupt, also messages from this hardware interrupt can arrive.

dest id

≠*nil* Sending is directed to the specified thread, if it resides in the sender’s clan. If the destination is outside the sender’s clan, the message is sent to the sender’s chief. If the destination is in an inner clan (a clan whose chief resides in the sender’s clan), it is redirected to that chief. (See also ‘nchief’ operation.) If no send part was specified (*snd descriptor* =*nil*), *dest id* specifies the source from which messages can be received. (However recall that the receive restriction is only effective if $o = 0$.)

=*nil* (*nil*=0) Although specifying *nil* as the destination for a send operation is illegal (error: ‘destination not existent’), it can be legally specified for a receive-only operation. In this case, ipc will not receive any message but will wait the specified *rcv timeout* and then terminate with error code ‘receive timeout’. (However recall that the receive restriction is only effective if $o = 0$.)

source id — If a message was received this is the id of its sender. (If a hardware interrupt was received this is the interrupt id.) The parameter is undefined if no message was received.

msg.w0 + w1

“snd” — First two 32-bit words of message to be sent. These message words are taken directly from registers EDX and EBX. *They are not read from the message data structure.*

“rcv” — First two 32-bit words of received message, undefined if no message was received. *These message words are available only in registers EDX and EBX.* The μ-kernel does not store it in the receive message buffer. The user program may store it or use it directly in the registers.

msg.dope + cc

mwords $_{(19)}$	strings $_{(5)}$	*cc* $_{(8)}$

Message dope describing received message. If no message was received, only *cc* is delivered. *The dope word of the received message is available only in register EAX.* The μ-kernel does not store it in the receive message buffer. The user program may store it or use it directly in the register. (Note that the lowermost 8 bits of msg dope and size dope in the message data structure are undefined. So it is legal to store EAX in the msg-dope field, even if $cc \neq 0$.)

cc

ec $_{(4)}$	*i*	*r*	*m*	*d*

d=0	The received message is transferred directly (“undeceived”) from *source id*.
d=1	The received message is “deceived” by a chief. *source id* is the virtual source id which was specified by the sending chief.
m=0	The received message did not contain fpages.
m=1	The sender mapped or granted fpages. The sender’s fpage descriptors were also (besides mapping/granting) transferred as mwords.
r=0	The received message was directed to the actual recipient, not redirected to a chief. I.e. sender and receiver a part of the same clan. The *i*-bit has no meaning in this case and is zero.
r=1	The received message was redirected to the chief which was next on the path to the true destination. Sender and addressed recipient belong to different clans.
i=0	If r=1: the received message comes from outside the own clan.
i=1	If r=1: the received message comes from an inner clan.

ec

= 0 — *ok*: the optional send operation was successful, and if a receive operation was also specified (*rcv descriptor* $\neq$*nil*) a message was also received correctly.

$\neq 0$ — If ipc fails the completion code is in the range 0x10…0xF0. If the send operation already failed, ipc is terminated without the potentially specified receive operation. s specifies whether the error occurred during the receive $(s=0)$ operation or during the send $(s=1)$ operation:

1 — *Non-existing* destination or source.

$2+s$ — *Timeout*.

$4+s$ — *Cancelled* by another thread (system call lthread_ex_regs).

$6+s$ — *Map failed* due to a shortage of page tables.

$8+s$ — *Send pagefault timeout*.

$A+s$ — *Receive pagefault timeout*.

$C+s$ — *Aborted* by another thread (system call lthread_ex_regs).

$E+s$ — *Cut* message. Potential reasons are (a) the recipient's mword buffer is too small; (b) the recipient does not accept enough strings; (c) at least one of the recipient's string buffers is too small.

1…5 — The according operation was terminated before a real message transfer started. No partner was directly involved.

6…F — The according operation was terminated while a message transfer was running. The message transfer was aborted. The current partner (sender or receiver) was involved and got the corresponding error code. It is not defined which parts of the message are already transferred and which parts are not yet transferred.

timeouts

This 32-bit word specifies all 4 timouts, the quadruple (*snd, rcv, snd pf, rcv pf*). For A detailed description see section B.2.5. Frequently used values are

	snd	rcv	snd pf	rcv pf
0x00000000	∞	∞	∞	∞
0x00000001	0	∞	∞	∞
0x00000011	0	0	∞	∞

"snd" — If the required send operation cannot start transfer data within the specified time, ipc is terminated and fails with completion code 'send timeout' (0x18). If ipc does not include a send operation, this parameter is meaningless.

"rcv" — If ipc includes a receive operation and no message transfer starts within the specified time, ipc is terminated and fails with completion code 'receive timeout' (0xA0). If ipc does not include a receive operation, this parameter is meaningless.

"spf" — If during sending data a pagefault *in the receiver's address space* occurs, *snd pf* specified by the sender is used as send and receive timeout for the pagefault RPC.

"rpf" — If during receiving data a pagefault *in the sender's address space* occurs, *rcv pf* specified by the receiver is used as send and receive timeout for the pagefault RPC.

Basic Ipc Types

CALL

**snd msg / 0*	EAX		EAX	*msg.dope + cc*
timeouts	ECX		ECX	∼
msg.w0	EDX		EDX	*msg.w0*
msg.w1	EBX	— INT 0x30⟶	EBX	*msg.w1*
**rcv msg / 0*	EBP		EBP	∼
dest id.low	ESI		ESI	*unchanged*
dest id.high	EDI		EDI	*unchanged*

This is the usual blocking RPC. *snd msg* is sent to *dest id* and the invoker waits for a reply from *dest id*. Messages from other sources are not accepted. Note that since the send/receive transition needs no time, the destination can reply with *snd timeout* = 0.

This operation can also be used for a server with one dedicated client. It sends the reply to the client and waits for the client's next order.

SEND/RECEIVE

**snd msg / 0*	EAX		EAX	*msg.dope + cc*
timeouts	ECX		ECX	∼
msg.w0	EDX		EDX	*msg.w0*
msg.w1	EBX	— INT 0x30⟶	EBX	*msg.w1*
**rcv msg+1 / 0+1*	EBP		EBP	∼
dest id.low	ESI		ESI	*source id.low*
dest id.high	EDI		EDI	*source id.high*

snd msg is sent to *dest id* and the invoker waits for a reply from any source. This is the standard server operation: it sends a reply to the actual client and waits for the next order which may come from a different client.

SEND

**snd msg / 0*	EAX		EAX	*cc*
timeouts	ECX		ECX	∼
msg.w0	EDX		EDX	∼
msg.w1	EBX	— INT 0x30⟶	EBX	∼
0xFFFFFFFF	EBP		EBP	∼
dest id.low	ESI		ESI	∼
dest id.high	EDI		EDI	∼

snd msg is sent to *dest id*. There is no receive phase included. The invoker continues working after sending the message.

RECEIVE FROM	*0xFFFFFFFF*	EAX		EAX	*msg.dope + cc*
	timeouts	ECX		ECX	∼
	∼	EDX		EDX	*msg.w0*
	∼	EBX	— INT 0x30⟶	EBX	*msg.w1*
	**rcv msg / 0*	EBP		EBP	∼
	dest id.low	ESI		ESI	*unchanged*
	dest id.high	EDI		EDI	*unchanged*

This operation includes no send phase. The invoker waits for a message from *source id*. Messages from other sources are not accepted. Note that also a hardware interrupt might be specified as source.

RECEIVE	*0xFFFFFFFF*	EAX		EAX	*msg.dope + cc*
	timeouts	ECX		ECX	∼
	∼	EDX		EDX	*msg.w0*
	∼	EBX	— INT 0x30⟶	EBX	*msg.w1*
	**rcv msg+1 / 0+1*	EBP		EBP	∼
	∼	ESI		ESI	*source id.low*
	∼	EDI		EDI	*source id.high*

This operation includes no send phase. The invoker waits for a message from any source (including a hardware interrupt).

RECEIVE INTR	*0xFFFFFFFF*	EAX		EAX	*msg.dope + cc*
	timeouts	ECX		ECX	∼
	∼	EDX		EDX	∼
	∼	EBX	— INT 0x30⟶	EBX	∼
	**rcv msg / 0*	EBP		EBP	∼
	intr + 1	ESI		ESI	*unchanged*
	0	EDI		EDI	*unchanged*

This operation includes no send phase. The invoker waits for an interrupt message coming from interrupt source *intr*. Note that interrupt messages come *only* from the interrupt which is currently associated with this thread.

The *intr* parameter is only evaluated if *rcv timeout* = 0 is specified, see 'associate intr'.

ASSOCIATE INTR	*0xFFFFFFFF*	EAX		EAX	*msg.dope + cc*
	rcv timeout = 0	ECX		ECX	~
	~	EDX		EDX	~
	~	EBX	— INT 0x30⟶	EBX	~
	**rcv msg / 0*	EBP		EBP	~
	intr + 1	ESI		ESI	*unchanged*
	0	EDI		EDI	*unchanged*

The *intr* parameter is evaluated if *rcv timeout* = 0 is specified. If no (currently associated) interrupt is pending, the current thread is (1) detached from its currently associated interrupt (if any) and (2) associated to the specified interrupt provided that this one is free, i.e. not associated to another thread. If the association succeeds, the completion code is *receive timeout* (0x20) and no interrupt is received.

If an interrupt from the currently associated interrupt was pending, this one is delivered together with completion code *ok* (0x00); the interrupt association is *not* modified. If the requested new interrupt is already associated to another thread or is not existing, completion code *non existing* (0x10) is delivered and the interrupt association is not modified.

Getting rid of an associated interrupt without associating a new one is done by issuing a receive from *nilthread* (0) with *rcv timeout* = 0.

SLEEP	*0xFFFFFFFF*	EAX		EAX	*cc* = 0*xA*0
	timeouts	ECX		ECX	~
	~	EDX		EDX	~
	~	EBX	— INT 0x30⟶	EBX	~
	0	EBP		EBP	~
	0	ESI		ESI	~
	0	EDI		EDI	~

This operation includes no send phase. Since *nil* (0) is specified as source, no message can arrive and the ipc will be terminated with 'receive timeout' after the time specified by the *rcv-timeout* parameter is elapsed.

id_nearest

~	EAX		EAX	*type*
~	ECX		ECX	~
~	EDX		EDX	~
~	EBX	— INT 0x31 →	EBX	~
~	EBP		EBP	~
dest id.low	ESI		ESI	*nearest id.low*
dest id.high	EDI		EDI	*nearest id.high*

If *nil* is specified as destination, the system call delivers the uid of the current thread. Otherwise, it deliveres the nearest partner which would be engaged when sending a message to the specified destination. If the destination does not belong to the invoker's clan, this call delivers the chief that is nearest to the invoker on the path from the invoker to the destination.

- If the destination resides outside the invoker's clan, it delivers the invoker's own chief.
- If the destination is inside a clan or a clan nesting whose chief C is direct member of the invoker's clan, the call delivers C.
- If the destination is a direct member of the invoker's clan, the call delivers the destination itself.
- If the destination is *nil*, the call delivers the current thread's id.

Concluding: nchief (*dest id* $\neq$ *nil*) delivers exactly that partner to which the kernel would physically send a message which is targeted to *dest id*. On the other hand, a message from *dest id* would physically come from exactly this partner.

Parameters

dest id		Id of the destination.
type		Note that the *type* values correspond exactly to the completion codes of ipc.
	=0	Destination resides in the same clan. *dest id* is delivered as *nearest id*.
	=C	Destination is in an inner clan. The chief of this clan or clan nesting is delivered as *nearest id*.
	=4	Destination is outside the invoker's clan. The invoker's chief is delivered as *nearest id*.
nearest id		Either the current thread's id or the id of the nearest partner towards *dest id*.

fpage_unmap

fpage	EAX		EAX	~
map mask	ECX		ECX	~
~	EDX		EDX	~
~	EBX	— INT 0x32→	EBX	~
~	EBP		EBP	~
~	ESI		ESI	~
~	EDI		EDI	~

The specified *fpage* is unmapped in all address spaces into which the invoker mapped it directly or indirectly.

Parameters

fpage Fpage to be unmapped.

map mask

f	0 $_{(29)}$	*w*	0

w=0 Fpage will partially unmapped. Already read/write mapped parts will be set to read only. Read only mapped parts are not affected.

w=1 Fpage will be completely unmapped.

f=0 Unmapping happens in all address spaces into which pages of the specified fpage have been mapped directly or indirectly. The *original* pages in the own task remain mapped.

f=1 Additionally, also the original pages in the own task are unmapped (flushing).

thread_switch

$\sim$	EAX		EAX	$\sim$
$\sim$	ECX		ECX	$\sim$
$\sim$	EDX		EDX	$\sim$
$\sim$	EBX	— INT 0x33 $\longrightarrow$	EBX	$\sim$
$\sim$	EBP		EBP	$\sim$
dest id.low	ESI		ESI	$\sim$
$\sim$	EDI		EDI	$\sim$

The invoking thread releases the processor (non-preemtively) so that another ready thread can be processed.

Parameters

dest id	*=nil*	(=0) Processing switches to an undefined ready thread which is selected by the scheduler. (It might be the invoking thread.) Since this is "ordinary" scheduling, the thread gets a new timeslice.
	≠nil	If *dest id* is ready, processing switches to this thread. In this "extraordinary" scheduling, the invoking thread donates its remaining timeslice to the destination thread. (This one gets the donation additionally to its ordinary scheduled timeslices.) If the destination thread is not ready, the system call operates as described for *dest id =nil*.

thread_schedule

param word	EAX		EAX	*old param word*
∼	ECX		ECX	*time.low*
∼	EDX		EDX	*time.high*
ext preempter.low	EBX	— INT 0x34⟶	EBX	*old preempter.low*
ext preempter.high	EBP		EBP	*old preempter.high*
dest id.low	ESI		ESI	*partner.low*
dest id.high	EDI		EDI	*partner.high*

The system call can be used by schedulers to define the *priority*, *timeslice length* and *external preempter* of other threads. Furthermore, it delivers thread states. Note that due to security reasons thread state information must be retrieved through the appropriate scheduler.

The system call is only effective, if the current priority of the specified destination is less or equal than the current task's *maximum controlled priority (mcp)*.

Parameters

dest id — Destination thread id. The destination thread must currently exist and run on a priority level less than or equal to the current thread's *mcp*. Otherwise, the destination thread is not affected by this system call and all result parameters except *old param word* are undefined.

param word

valid

m_t (8)	e_t (4)	0 (4)	small (8)	prio (8)

prio — New priority for destination thread. Must be less than or equal to current thread's *mcp*.

small — (Only effective for Pentium.) Sets the *small address space number* for the addressed *task*. On Pentium, small address spaces from 1 to 127 currently available. A value of 0 or 255 in this field does not change the current setting for the task. This field is currently ignored for 486 and PPro.

m_t, e_t — New timeslice length for the destination thread. The timeslice quantum is encoded like a timeout: $4^{15-e_t} m_t$ µs.
The kernel rounds this value up towards the nearest possible value. Thus the timeslice granularity can be determined by trying to set the timeslice to 1 µs. However note that the timeslice granularity may depend on the priority.
Timeslice length 0 ($m_t = 0, e_t \neq 0$) is always a possible value. It means that the thread will get no ordinary timeslice, i.e. is blocked. However, even a blocked thread may execute in a timeslice donated to it by ipc.

"inv" — (0xFFFFFFFF) The current priority and timeslice length of the thread is not modified.

ext preempter

valid — Defines the external preempter for the destination thread. (Nilthread is a valid id.)

"inv" — (0xFFFFFFFF,∼) The current external preempter of the thread is not changed.

old param word

valid

m_t (8)	e_t (4)	ts (4)	∼ (8)	prio (8)

prio — Old priority of destination thread.

m_t, e_t — Old timeslice length of the destination thread: $4^{15-e_t} m_t$ μs.

$ts =$ — Thread state:

- $0+k$ — *Running.* The thread is ready to execute at user-level.
- $4+k$ — *Sending.* A user-invoked ipc send operation currently transfers an outgoing message.
- $8+k$ — *Receiving.* A user-invoked ipc receive operation currently receives an imcoming message.
- C — *Waiting* for receive. A user-invoked ipc receive operation currently waits for an incoming message.
- D — *Pending* send. A user-invoked ipc send operation currently waits for the destination (recipient) to become ready to receive.
- E — Reserved.
- F — *Dead.* The thread is unable to execute.

- $k=0$ — *Kernel inactive.* The kernel does not execute an automatic RPC for the thread.
- 1 — *Pager*. The kernel executes a pagefault RPC to the thread's pager.
- 2 — *Internal preempter.* The kernel executes a preemption RPC to the thread's internal preempter.
- 3 — *External preempter.* The kernel executes a preemption RPC to the thread's external preempter.

"inv" — (0xFFFFFFFF) The addressed thread does either not exist or has a priority which exceeds the current thread's *mcp*. All other return parameters are undefined (∼).

old ext preempter — Old external preempter of the destination thread.

partner — Partner of an active user-invoked ipc operation. This parameter is only valid, if the thread's user state is *sending, receiving, pending* or *waiting* (4...D). An invalid thread id (0xFFFFFFFF,∼) is delivered if there is no specific partner, i.e. if the thread is in an open receive state.

time

m_w (8)	e_w (4)	e_p (4)	T_{high} (16)	EDX
T_{low} (32)				ECX

T Cpu time (48-bit value) in microseconds which has been consumed by the destination thread.

m_w, e_w Current user-level wakeup of the destination thread, encoded like a timeout. The value denotes the still remaining timeout interval. Valid only if the user state is *waiting* (C) or *pending* (D).

e_p Effective pagefault wakeup of the destination thread, encoded like a 4-bit pagefault timeout. The value denotes the still remaining timeout interval. Valid only if the kernel state is *pager* ($k = 1$).

lthread_ex_regs

lthread no	EAX		EAX	*old EFLAGS*
ESP	ECX		ECX	*old ESP*
EIP	EDX		EDX	*old EIP*
int preempter.low	EBX	— INT 0x35 ⟶	EBX	*old preempter.low*
int preempter.high	EBP		EBP	*old preempter.high*
pager.low	ESI		ESI	*old pager.low*
pager.high	EDI		EDI	*old pager.high*

This function reads and writes some register values of a thread in the current task.

It also creates threads. Conceptually, creating a task includes creating all of its threads. Except lthread 0, all these threads run an idle loop. Of course, the kernel does neither allocate control blocks nor time slices etc. to them. Setting stack and instruction pointer of such a thread to valid values then really generates the thread.

Note that this operation reads and writes the *user-level* registers (ESP, EIP and EFLAGS). Ongoing kernel activities are not affected. However an ipc operation is cancelled or aborted. If the ipc is either waiting to send a message or waiting to receive a message, i.e. a message transfer is not yet running, ipc is cancelled (completion code 0x40 or 0x50). If a message transfer is currently running, ipc is aborted (completion code 0xC0 or 0xD0).

Parameters

lthread no

v	$0_{(24)}$	lthread $_{(7)}$

Number of addressed lthread (0...127) inside the current task.

	$v = 0$	If ESP and EIP are valid, they specify 32-bit protected-mode values. The addressed thread will execute in 32-bit protected mode afterwards.
	$v = 1$	If ESP and EIP are valid, they specify 16-bit V86-mode values. The addressed thread will execute in V86 mode afterwards.
ESP	valid	New stack pointer (ESP) for the thread. It must point into the user-accessible part of the address space.
	"inv"	(0xFFFFFFFF) The existing stack pointer is not modified.
EIP	valid	New instruction pointer (EIP) for the thread. It must point into the user-accessible part of the address space.
	"inv"	(0xFFFFFFFF) The existing instruction pointer is not modified.
int preempter	valid	Defines the internal preempter used by the thread. (*Nil* is a valid id.)
	"inv"	(0xFFFFFFFF,∼) The existing internal preempter id is not modified.

pager	valid	Defines the pager used by the thread.
	"inv"	(0xFFFFFFFF,∼) The existing pager id is not modified.
old EFLAGS		Flags of the thread. The *VM* flag specifies whether the thread currently runs in 32-bit protected mode ($VM = 0$) or in V86 mode ($VM = 1$). Note that this flag determines the format of the delivered old ESP and EIP.
old ESP		Old stack pointer (ESP) of the thread.
old ESP		Old instruction pointer (EIP) of the thread.
old int preempter		Id of the thread's old internal preempter (may be nilthread).
old pager		Id of the thread's old pager.

V86-mode Pointers

ESP, old ESP	SS $_{(16)}$	SP $_{(16)}$
EIP, old EIP	CS $_{(16)}$	IP $_{(16)}$

Example

Signalling can be implemented as follows:

```
signal (lthread) :
    esp := receive signal stack ;
    eip := receive signal ;
    mem [esp – –] := 0 ;
    lthread ex regs (lthread, esp, eip, eflags, –, –) ;
    mem [esp – –] := eflags ;
    mem [esp – –] := eip ;
    mem [idle stack – wordlength] := esp .
```

```
receive signal :
    push all regs ;
    while mem [esp + 8 × wordlength] = 0 do
        thread switch (nilthread)
    od ;
    pop all regs ;
    pop (esp) ;
    jmp (signal eip) .
```

task_new

mcp / new chief	EAX		EAX	∼
initial ESP	ECX		ECX	∼
initial EIP	EDX		EDX	∼
pager.low	EBX	— INT 0x36 →	EBX	∼
pager.high	EBP		EBP	∼
dest task.low	ESI		ESI	*new task.low*
dest task.high	EDI		EDI	*new task.high*

This function deletes and/or creates a task. Deletion of a task means that the address space of the task and all threads of the task disappear. The cputime of all deleted threads is added to the cputime of the deleting thread. If the deleted task was chief of a clan, all tasks of the clan are deleted as well.

Tasks may be created as *active* or *inactive*. For an active task, a new address space is created together with 128 threads. Lthread 0 is started, the other ones wait for a "real" creation by lthread_ex_regs. An inactive task is empty. It occupies no resources, has no address space and no threads. Communication with inactive tasks is not possible. Loosely speaking, inactive tasks are not really existing but represent only the right to create an active task.

A newly created task gets the creator as its chief, i.e. it is created inside the creator's clan. Symmetrically, a task can only be deleted either directly by its chief (its creator) or indirectly by a higher-level chief.

Parameters

dest task — Task id of an *existing* task (active or inactive) whose chief is the current task. If one of these preconditions is not fulfilled, the system call has no effect. Simultaneously, a new task *with the same task number* is created. It may be active or inactive (see next parameter).

pager $\neq$ *nil* — The new task is created as *active*. The specified pager is associated to lthread 0.

$=$ *nil* — (0,∼) The new task is created as *inactive*. Lthread 0 is not created.

ESP — Initial stack pointer for lthread 0 if the new task is created as an active one. Ignored otherwise.

EIP — Initial instruction pointer for lthread 0 if the new task is created as an active one. Ignored otherwise.

mcp — Maximum controlled priority (mcp) defines the highest priority which can be ruled by the new task acting as a scheduler. The new task's effective mcp is the minimum of the creator's mcp and the specified mcp.
EAX contains this parameter, if the newly generated task is an *active* task, i.e. has a pager and at least lthread 0.

new chief		Specifies the chief of the new inactive task. This mechanism permits to transfer inactive ("empty") tasks to other tasks. Transferring an inactive task to the specified chief means to transfer the related right to create a task. Note that the task number remains unchanged. EAX contains this parameter, if the newly generated task is an *inactive* task, i.e. has no pager and no threads. EAX contains only the lower 32 bits of the new chief's task id. (The chief must reside in the same site.)
new task id	*≠nil*	Task creation succeeded. If the new task is active, the new task id will have a new version number so that it differs from all task ids used earlier. Chief and task number are the same as in *dest task*. If the new task is created inactive, the chief is taken from the *chief* parameter; the task number remains unchanged. The version is undefined so that the new task id might be identical with a formerly (but not currently and not in future) valid task id. This is safe since communication with inactive tasks is impossible.
	=nil	(0,∼) The task creation failed.

B.4 Processor Mirroring

B.4.1 Segments

L4 uses a flat (unsegmented) memory model. There are only two segments available: *user_space*, a read/write segment, and *user_space_exec*, an executable segment. Both cover (at least) the complete user-level address space.

The values of both segment selectors *are undefined*. When a thread is created, its segment registers SS, DS, ES, FS and GS are initialized with *user_space*, CS with *user_space_exec*. Whenever the kernel detects a general protection exception and the segment registers are not loaded properly, it reloads them with the above mentioned selectors. From the user's point of view, the segment registers cannot be modified.

However, the binary representation of *user_space* and *user_space_exec* may change at any point during program execution. Never rely on this value.

Furthermore, the LSL (load segment limit) machine instruction may deliver wrong segment limits, even floating ones. The result of this instruction is always undefined.

B.4.2 Exception Handling

#PF (page fault), #MC (machine check exception) and some #GP (general protection) exceptions are handled by the kernel. The other exceptions are mirrored to the virtual processor of the thread which raised the exception.

The mirrored exception handling works as described in the processor manuals.

LIDT [EAX] — This machine instruction is emulated by the kernel and operates per thread. Any thread should install an IDT by this instruction. The length field of the IDT vector is not interpreted. The IDT has always a length of 256 bytes, i.e. covers the Intel-reserved exceptions 0 to 31.

The IDT must have the format described in the processor manuals. However, only trap gates can be used in a user-level IDT. The segment selectors in the IDT are ignored, since all segments describe the flat address space.

Invalid IDT addresses or invalid exception-handler addresses do not raise a double fault; instead, the current thread is shut down.

Note that this mechanism deals only with exceptions, not INT n instructions. Executing an INT n in 32-bit mode will always raise a #GP (general protection). The general-protection handler may interpret the error code ($8n+2$, see processor manual) and emulate the INT n accordingly.

B.4.3 Debug Registers

User-level debug registers exist per thread. DR0...3, DR6 and DR7 can be accessed by the machine insructions `MOV DRx,n` and `MOV r,DRx`. However, only task-local breakpoints can be activated, i.e. bits L0...3 in DR7 cannot be set. Breakpoints operate per thread. Breakpoints are signalled as #DB exception (INT 1).

Note that user-level breakpoints are suspended when kernel breakpoints are set by the kernel debugger.

B.5 The Kernel-Info Page

The kernel-info page contains kernel-version data, memory descriptors *and the clock*. The remainder of the page is undefined. (In fact, it contains kernel code.) The kernel-info page is mapped *read-only* in the σ_0-address space. σ_0 can use the memory descriptors for its memory management. σ_0 can map the page read-only to other address spaces.

L4 version strings	$L \times 16$

+0xC	+8	+4	+0	
~		clock		0xA0
dedicated mem4.high	dedicated mem4.low	dedicated mem3.high	dedicated mem3.low	0x90
dedicated mem2.high	dedicated mem2.low	dedicated mem1.high	dedicated mem1.low	0x80
dedicated mem0.high	dedicated mem0.low	reserved mem1.high	reserved mem1.low	0x70
reserved mem0.high	reserved mem0.low	main mem.high	main mem.low	0x60
~				0x50
~				0x40
~				0x30
~				0x20
~				0x10
~	0 | *L*	version word	“L4µK”	0x00

mem.low Physical address of first byte of region. Must be page aligned.

mem.high Physical address of first byte beyond the region. Must be page aligned. If *mem.high* = 0, the region is empty.

main mem Main memory region.

reserved mem This region must not be used. It contains kernel code (reserved mem0) or data (reserved mem1) grabbed by the kernel before σ_0 was initialized.

dedicated mem	This region contains dedicated memory which cannot be used as standard memory. For example, [640K, 1M] is a popular dedicated memory region.

B.6 Page-Fault and Preemption RPC

Page Fault RPC

kernel sends:

w0 (EDX)	fault address / 4 $_{(30)}$	w	~
w1 (EBX)	faulting user-level EIP $_{(32)}$		

$w = 0$ Read page fault.
$w = 1$ Write page fault.

kernel receives: The receive fpage covers the complete user address space. The kernel accepts mappings or grants into this region as well as a simple 2-word copy message. The received message is ignored!

timeouts used for pagefault RPC	PF at user level	PF at ipc in receiver's space	PF at ipc in sender's space
snd	∞	sender's snd pf	receiver's rcv pf
rcv	∞	sender's snd pf	receiver's rcv pf
snd pf	∞	sender's snd pf	receiver's rcv pf
rcv pf	∞	sender's snd pf	receiver's rcv pf

Preemption RPC

kernel sends:

w0 (EDX)	user-level ESP $_{(32)}$
w1 (EBX)	user-level EIP $_{(32)}$

kernel receives: The kernel accepts only a simple 2-word reply. Its content is ignored!

timeouts used for preemption RPC	
snd	∞
rcv	∞
snd pf	∞
rcv pf	∞

B.7 σ_0 RPC protocol

σ_0 is the initial address space. Although it is *not* part of the kernel, its basic protocol is defined with the μ-kernel. Special σ_0 implementations may extend this protocol.

The address space σ_0 is idempotent, i.e. all virtual addresses in this address space are identical to the corresponding physical address. Note that pages requested from $sigma_0$ continue to be mapped idempotent if the receiver specifies its complete address space as receive fpage.

σ_0 gives pages to the kernel and to arbitrary tasks, but only once. The idea is that all pagers request the memory they need in the startup phase of the system so that afterwards σ_0 has spent all its memory. Further requests will then automatically denied.

Kernel Communication

σ_0 receives from a *kernel* thread:

w0 (EDX)	FFFFFFFF/4 $_{(30)}$	0	0
w1 (EBX)	$\sim_{(32)}$		

Intended Action: For reply, σ_0 should *grant* a page (4K) to the kernel thread. The page might be located at an arbitrary position but must contain ordinary memory. If no more memory is available, σ_0 should reply a 0-word instead of a page.

σ_0 receives from a *kernel* thread:

w0 (EDX)	0 $_{(31)}$	1
w1 (EBX)	$\sim_{(24)}$	0 $_{(8)}$

Intended reply:

w0 (EDX)	$k_{(32)}$
w1 (EBX)	$\sim_{(32)}$

k is the number of pages recommended by σ_0 for kernel use (pagetables and other kernel-internal data).

General Memory Mapping

σ_0 receives from a *non-kernel* thread:

w0 (EDX)	FFFFFFFF/4 $_{(30)}$	0	0
w1 (EBX)	$\sim_{(32)}$		

Intended Action: For reply, σ_0 should *map* a page (4K) writable to the requester. The page might be located at an arbitrary position but must contain ordinary memory. If no more free page is available, σ_0 should reply a 0-word instead of a page.
The mapped page must be marked and must not further be mapped or granted by σ_0 to another task. However, multiple mapping to the same requester should be supported.

σ_0 receives from any thread:

w0 (EDX)	address $\leq$ 40000000/4 $_{(30)}$	0	0
w1 (EBX)	$\sim_{(32)}$		

Intended Action: For reply, σ_0 should *map* the specified physical page frame (4K) writable to the requester. If the page is already mapped or is not available, σ_0 should reply a 0-word instead of a page.
The mapped page must be marked and must not further be mapped or granted by σ_0 to another task. However, multiple mapping to the same requester should be supported.

σ_0 receives from any thread:

w0 (EDX)	40000000/4 $< a \leq$ C0000000/4 $_{(30)}$	0	0
w1 (EBX)	$\sim_{(32)}$		

Intended Action: For reply, σ_0 should *map* the physical 4M superpage with address $a - 40000000$ writable to the requester. If the page is already mapped or is not available, σ_0 should reply a 0-word instead of a page.
The mapped superpage must be marked and must not further be mapped or granted by σ_0 to another task. However, multiple mapping to the same requester should be supported.

σ_0 receives from any thread:

w0 (EDX)	0 $_{(31)}$	1
w1 (EBX)	$\sim$ $_{(24)}$	1 $_{(8)}$

Intended Action: For reply, σ_0 should *map read only* the kernel info page to the requester. This page can be mapped multiply.

C <l4/kdebug.h>

```
/*
 * $Id: kdebug.h,v 1.1 1998/08/21 14:33:01 hohmuth Exp $
 *
 * kdebug.h provides some useful makros to acces the functionality
 * of the kernel debugger
 */

#ifndef __L4_KDEBUG_H__
#define __L4_KDEBUG_H__

#include <l4/compiler.h>

#define enter_kdebug(text) \
asm(\
    "int $3 \n\t"\
    "jmp 1f \n\t"\
    ".ascii \""text "\"\n\t"\
    "1: \n\t"\
    )

#define asm_enter_kdebug(text) \
    "int $3 \n\t"\
    "jmp 1f \n\t"\
    ".ascii \""text "\"\n\t"\
    "1: \n\t"

#define kd_display(text) \
asm(\
    "int $3 \n\t"\
    "nop \n\t"\
    "jmp 1f \n\t"\
    ".ascii \""text "\"\n\t"\
    "1: \n\t"\
    )

#define ko(c)  \
  asm( \
      "int $3 \n\t" \
      "cmpb %0,%%al \n\t" \
      : /* No output */ \
      : "N" (c) \
      )

/*
```

```
 * prototypes
 */
L4_INLINE void outchar(char c);
L4_INLINE void outstring(char *text);
L4_INLINE void outhex32(int number);
L4_INLINE void outhex20(int number);
L4_INLINE void outhex16(int number);
L4_INLINE void outdec(int number);

L4_INLINE void outchar(char c)
{
  asm(
      "int $3 \n\t"
      "cmpb $0,%%al \n\t"
      : /* No output */
      : "a" (c)
      );
}

/* actually outstring is outcstring */
L4_INLINE void outstring(char *text)
{
  asm(
      "int $3 \n\t"
      "cmpb $2,%%al \n\t"
      : /* No output */
      : "a" (text)
      );
}

L4_INLINE void outhex32(int number)
{
  asm(
      "int $3 \n\t"
      "cmpb $5,%%al \n\t"
      : /* No output */
      : "a" (number)
      );
}

L4_INLINE void outhex20(int number)
{
  asm(
      "int $3 \n\t"
      "cmpb $6,%%al \n\t"
      : /* No output */
```

```
      : "a" (number)
      );
}

L4_INLINE void outhex16(int number)
{
  asm(
      "int $3 \n\t"
      "cmpb $7, %%al \n\t"
      : /* No output */
      : "a" (number)
      );
}

L4_INLINE void outdec(int number)
{
  asm(
      "int $3 \n\t"
      "cmpb $11, %%al \n\t"
      : /* No output */
      : "a" (number)
      );
}

#endif /* __L4_KDEBUG_H__ */
```

References

[1] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–178, Monterey, CA, November 1994.

[2] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. L4 reference manual—MIPS R4x00, version 1.0, kernel version 70. UNSW-CSE-TR 9709, University of New South Wales, School of Computer Science, 1998.

[3] M. Gasser, A. Goldstein, C. Kaufmann, and B. Lampson. The Digital distributed system security architecture. In *12th National Computer Security Conference (NIST/NCSC)*, pages 305–319, Baltimore, 1989.

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

[5] H. Härtig, O. Kowalski, and W. Kühnhauser. The BirliX security architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

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

[7] Michael Hohmuth. The Fiasco kernel: System architecture. Technical report, TU Dresden, 1998.

[8] Michael Hohmuth. *RMGR, The L4 Resource Manager*. TU Dresden, 1998.

[9] R. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):11–22, November 1992.

[10] Y. A. Khalidi and M. N. Nelson. Extensible file systems in Spring. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 1–14, Asheville, NC, December 1993.

[11] C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: high performance external virtual memory caching. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–164, Monterey, CA, November 1994.

[12] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[13] J. Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[14] J. Liedtke. A persistent system in real use – experiences of the first 13 years –. In *3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 2–11, Asheville, NC, December 1993.

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

[16] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sep 1996; available from URL: `ftp://borneo.gmd.de/pub/rs/L4/l4refx86.ps`.

[17] T. H. Romer, D. L. Lee, B. N. Bershad, and B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–266, Monterey, CA, November 1994.

[18] S. Schönberg. L4 on Alpha, design and implementation. Technical Report CS-TR-407, University of Cambridge, 1996.

Index