

L4.Sec  
**Preliminary** Microkernel Reference Manual

Bernhard Kauer, Marcus Völp  
Technische Universität Dresden  
01062 Dresden, Germany  
{kauer,voelp}@os.inf.tu-dresden.de

Version: 0.2

October 19, 2005



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	In Construction . . . . .	7
1.2	About This Manual . . . . .	7
1.3	Maintainers . . . . .	7
1.4	Legal notice . . . . .	8
1.5	Credits . . . . .	8
1.6	Notation . . . . .	9
1.6.1	Terms and Definitions . . . . .	9
1.6.2	Basic Data Types . . . . .	9
<b>2</b>	<b><i>L4.Sec</i> - An Overview</b>	<b>11</b>
2.1	Local Names and Address Spaces . . . . .	11
2.1.1	Privilege Transfer and Revocation . . . . .	13
2.1.2	Capability Faults and Exceptions . . . . .	14
2.2	Threads . . . . .	15
2.2.1	<code>exchange-registers</code> . . . . .	15
2.2.2	<code>thread-control</code> . . . . .	15
2.3	IPC . . . . .	16
2.4	Scheduling . . . . .	17
2.4.1	Reservation Framework . . . . .	17
2.4.2	Scheduling Control . . . . .	18
2.4.3	Scheduling in Dresden's <i>L4.Sec</i> implementation . . . . .	18
2.5	Object Creation and Kernel Memory Management . . . . .	18
2.5.1	Object Destruction . . . . .	20
2.5.2	Address Space Construction and Startup . . . . .	21
2.6	IRQs . . . . .	21
2.7	Miscellaneous . . . . .	22
2.7.1	Processor Control . . . . .	22
2.7.2	System Startup . . . . .	22
2.8	Summary . . . . .	23
2.8.1	List of Kernel Objects . . . . .	23

2.8.2	List of system-calls . . . . .	23
<b>3</b>	<b>Application Programming Interface</b>	<b>25</b>
3.1	Introduction . . . . .	25
3.1.1	Parameter Words . . . . .	25
3.1.2	System-Call Numbers . . . . .	25
3.1.3	TCRs . . . . .	25
3.1.4	Datatypes . . . . .	26
3.2	Binding . . . . .	27
3.2.1	<code>short-ipc</code> - transfer some words . . . . .	27
3.2.2	<code>exchange-registers</code> - exchange volatile thread state . . . . .	29
3.2.3	<code>unmap</code> - revoke permissions recursively . . . . .	31
3.2.4	<code>create</code> - create kernel objects . . . . .	33
3.2.5	<code>thread-control</code> - read and write thread control register . . . . .	35
<b>4</b>	<b>Application Binary Interface</b>	<b>37</b>
4.1	x86 . . . . .	37
4.1.1	<code>sysenter</code> . . . . .	37
4.1.2	<code>int 0x40</code> . . . . .	37

# List of Figures

- 3.1 Flexpage Format . . . . . 26
- 3.2 Short IPC Input Parameters . . . . . 27
- 3.3 Short IPC Output Parameters . . . . . 28
- 3.4 Ex Regs Input Parameters . . . . . 29
- 3.5 Ex Regs Output Parameters . . . . . 30
- 3.6 Unmap Input Parameters . . . . . 31
- 3.7 Unmap Output Parameters . . . . . 31
- 3.8 Create Input Parameters . . . . . 33
- 3.9 Create Output Parameters . . . . . 34
- 3.10 Thread Control Input Parameters . . . . . 35
- 3.11 Thread Control Output Parameters . . . . . 36



# Chapter 1

## Introduction

### 1.1 In Construction

Please note, this document is currently under construction and will therefore include undefined and forward references to not yet completed section.

### 1.2 About This Manual

It is the primary purpose of this document to provide to the *L4.Sec* application programmer and the *L4.Sec* kernel programmer a precise definition of the *L4.Sec* interface (both API and ABI) and its operational semantics. This definition is in part complemented by additional sections containing formal specifications of the interfaces, design decisions and reasoning, kernel implementation considerations, and, example use cases. The sources of this document can be configured to in- or exclude certain types of these sections.

The following list shows the sections and their highlighting that have been included in this document:

API

ABI

Convenience Interface:

*some convenience interface*

Reasoning:

some reason

Kernel Implementation:

some implementation consideration

Usage Example

some usage example

### 1.3 Maintainers

Bernhard Kauer

Marcus Völp

## 1.4 Legal notice

This document is an experimental draft. As such it may change significantly without further notice. However, an up to date version of this document is available at

Copyright © 2005 Bernhard Kauer, Marcus Völp, Systems Architecture Group, Technische Universität Dresden.

This specification is provided “as is” without any warranties, including any warranty of merchantability, non-infringement, fitness for any particular purpose, or any warranty otherwise arising of any proposal, specification or sample.

Permissions to copy and distribute verbatim copies of this specification in any medium for any purpose without free or royalty is hereby granted. No right to create modifications or derivatives is granted by this license.

## 1.5 Credits

This manual originated from the discussions at and following the Dresden Workshop on Microkernel Security. As a preliminary proposal of extensions to the L4 microkernel this work also builds on the previous microkernel versions including L4.V2, L4.X2 and L4 Strawberry.

Helpful contributions to this manual came from many persons, in particular from Ron Aigner, Marcus Brinkmann, Christian Ceelen, Norman Feske, Hermann Härtig, Christian Helmuth, Michael Hohmuth, Adam Lackorzynski, Jork Löser, Frank Mehnert, Michael Peter, Martin Pohlack, Lars Reuther, Espen Skoglund, Udo Steinberg, Christian Stüble, Neil Walfield, Alexander Warg, Stephan Wagner, Andreas Weigand, and, Jean Wolter

Special thanks go to Jonathan Shapiro for his continuing support in building secure microkernels.

## 1.6 Notation

### 1.6.1 Terms and Definitions

In this document we use the following terms and definitions:

**architecture defined** We say a parameter, behavior or part of this specification is architecture defined if this parameter, behavior or part is valid only for a specific architecture. In particular an architecture defined parameter, behavior or part may vary on a different architecture. In the sections describing the individual architectures, this document defines all architecture defined parameters, behaviors and parts for a respective architecture.

**implementation defined** We say a parameter, behavior or part of this specification is implementation defined if this parameter, behavior or part is valid only for a specific implementation. In particular an implementation-defined parameter, behavior or part may vary in different implementations of the *L4.Sec* micro-kernel. Each implementation must publish a specification of these implementation-specific parameters, behaviors and parts either in this document or in an external document.

**undefined** The behavior of an operation will not be defined and may lead to the destruction of the address space of the invoking thread. Invoking an operation that is undefined must be considered a bug. The micro-kernel, however, guarantees that the immediate effect of this invocation is limited to the invoking thread's address space. Additional threads may be affected by the unavailability of the immediately effected threads.

**unspecified**  $\square\sim$  An in- or output parameter of an operation will not be specified in this manual. Unspecified output parameters may hold arbitrary values prior to a system-call, unspecified input parameters are not considered by the system-call. For upwards compatibility reasons, however, should unspecified input parameters be set to zero prior to invoking a system-call. Reading an unspecified value will not leak any security relevant information.

**ignored**  $\square-$  An input parameter will be ignored by an operation, despite to what value it is set.

**reserved** Reserved parameters or fields are intentionally not used. They may be used in future versions of this specification or without notice in some micro-kernel implementations. Reading or writing a reserved field is undefined.

### 1.6.2 Basic Data Types

This manual describes the *L4.Sec* application-programming and application-binary interface for 32 and 64 bit processors. The data type *Word* always refers to an unsigned integer type of architectural width (i.e, an unsigned 32 bit integer on 32 bit machines, an unsigned 64 bit integer on 64 bit machines). When an unsigned integer data type does not depend on the underlying hardware architecture we write *WordN* (e.g., *Word16*, *Word32* or *Word64*) for an unsigned data type of the respective width N. Even though *Word* describes an unsigned integer we write for short  $-1$  to denote the unsigned integer with all bits set to 1.



## Chapter 2

# *L4.Sec* - An Overview

to do: general L4 and micro-kernel introduction

*L4.Sec* is based on four fundamental abstractions:

- *address spaces*,
- *threads*,
- *reservations*, and,
- *endpoints*.

These abstractions are complemented by a single mechanism: *inter-process communication (IPC)*.

### 2.1 Local Names and Address Spaces

In *L4.Sec* all kernel objects are accessed via address-space local names. An address space (aka task) is a mapping translating local names into pairs of kernel objects and sets of rights. One such pair we call a capability with the intended meaning that the set of rights restricts the operations that are permitted on the referenced kernel object. *L4.Sec* implements capabilities for all first class kernel objects. The following first class kernel objects exist: memory pages, IO ports, CPUs as given by the underlying hardware complemented with address spaces, threads, IPC endpoints and reservations. Interrupts are no kernel objects. *L4.Sec* models interrupts as messages from “hardware” threads.

Reasoning:

In a system with local names control of the mappings of local names to kernel objects is necessary and sufficient to control which process can access which kernel object. In particular local names are required to prevent information flow through the fact that a particular physical object exists. We discussed two alternative solutions for achieving this property and discarded them for the following reasons:

**Virtual addresses of data structures.**

Each kernel object can be identified through a pointer to the data structure (e.g., the C++ Object) implementing it. As such it was proposed to map the memory pages on which this data structure is allocated into the kernel region of the address space and use the virtual address as local name for this object. The expected benefits of this approach is a gain in performance because of removing the indirection inherent with a capability table lookup.

The approach was discarded because of the page granularity being much larger than the size of the objects. Page tables implementing an address space are typically in the order of a page, thread control blocks are in the range of one fourth of the size of a page, however, endpoints are much smaller (typically in the order of a cache line). Also there is a potential performance penalty hidden by not tagging the kernel objects as global: When two objects have previously been accessed, their pages remain in the TLB and we get only one capability miss for the two lookups (assuming dense capabilities). Otherwise, we get two TLB misses to access the two objects.

**Partitioned physical addresses.**

Local names can also be implemented by a priori partitioning the physical address space and by statically assigning these partitions to the threads that are allowed to use the objects in there.

This approach was discarded because of the inflexibility of a priori partitioning.

There is a special capability address —the *invalid capability*— that is never associated with a capability and that can therefore never be invoked.

Reasoning:

We use this invalid capability to express that a system-call will not perform a certain operation on a kernel object.

The *L4.Sec* API makes explicit whether a local name refers to a memory page or to any other type of kernel object. Depending on the processor architecture the API also differentiates local names referring to IO ports (e.g., on Intel x86 based processors). Correspondingly we call the respective parts of an address space the memory space, the IO space and the capability space.

Reasoning:

By partitioning an address space into memory space, IO space and capability space we encapsulate the differing semantics of the referred kernel objects. For example a name in the capability space always refers to one object, in the memory space adjacent names can either address the same large page or two smaller pages; the virtual address of an IO port must be the same as its physical address, etc. Furthermore, the translation table format for the memory and IO space is often specified by the underlying processor architecture, no such restriction exists on the format of the capability translation table.

We discussed the alternative of having per object type spaces instead of a single capability space and discarded it because we expect most tasks to use only a small number of capabilities but of varying object type. Note, however, that we would safe checking the type of a capability when making per object type spaces explicit in the API.

Unlike the other objects, the sender of an IPC-message is not identified through a capability residing in the receiver's address space. Instead, a kernel-protected message value —the *badge*— is transmitted to the receiver. There is a *badge* for each capability referring to an IPC-endpoint (see below) which is transmitted whenever sending a message by invoking this capability. As such the *badges* of an IPC-endpoint establish a name space local to the set of threads that can receive messages from this endpoint.

#### Reasoning:

An alternative model: virtual threads (by Espen Skoglund) addresses a sender through a capability in the receiver space. However, because multiple capabilities may refer to the same sender (aliasing) a sender-side hint is required. We decided against this model because we feel hints expose to much knowledge of the receiver's address-space layout.

#### Reasoning:

Identity of capabilities: to determine whether a the target of a received capability is trusted we introduce the following two experimental functions: `identity(CapabilityAddress c1, CapabilityAddress c2)` and `read_badge(CapabilityAddress c1, CapabilityAddress c2)`. `identity` returns true iff the target of the two capabilities at `c1` and `c2` refer to the very same kernel object and if the capability at `c1` provides compare permissions. `read_badge` returns the badge of the capability at `c2` iff the capability at `c2` refers to an endpoint and the capabilities at `c1` and `c2` are identical.

This way a server can check whether a capability received from a not fully trusted entity refers to one of the kernel objects to which it holds a capability with compare permissions and which it knows whether or not to trust.

### 2.1.1 Privilege Transfer and Revocation

Transfer of rights in *L4.Sec* applies a generalization of the *Recursive Virtual Address-Space Model*[Lie95].

We say a thread possesses a capability if there is a local name referring to this capability in it's address space.

A thread can *map* any of it's capabilities to an address space, provided mapping to that address space is permitted. As a result of a successful map operation a mapping of a local name to a capability is inserted into this address space that comprises access to the same object as the mapped capability and with the minimum of the rights of the mapped capability and those specified in the map operation. Furthermore any thread in the mapper's address space is permitted to later on revoke any mapped right recursively from any address space that directly or indirectly received the capability from the mapper's address space.

In *L4.Sec* mapping of capabilities to an address space is permitted by two means: first, any thread with send and map permissions on an IPC-endpoint can *map* any capability to the receiving thread's address space, provided the receiving thread accepts mappings.

When mapping via an IPC-endpoint, the sending thread specifies the capabilities and rights to be mapped as well as a hot spot. The receiving thread specifies for each space (capability, memory, IO) a region wherein it accepts capabilities. The hot spot is used to determine the exact position within the receive region of the appropriate type (memory space region for capabilities on memory pages, IO space region for capabilities on IO-ports and capability space region for capabilities on other objects). Map is implemented as a special IPC message that allows multiple map items to be transferred together with untyped data or strings. Map items specify the rights to map, the hot spot and a *flexpage* describing the capabilities to map.

A *flexpage* is a region of an address space of some power of 2 size and that is aligned to this size. A *flexpage* selects all capabilities to which the local names in this region refer, respectively the local names at which to map the capabilities on the receiver's side.

In a map operation the hot spot is evaluated to adjust *flexpages* to the same size and to contain the hot spot. For the exact algorithm please refer to Section ??.

With the `unmap` system-call a thread can revoke any right selectively from any address space that directly or indirectly received a mapping from any thread in the revoking thread's address space. With accepting a mapping the receiver implicitly agrees that the mapping can later on be revoked. As such revocation of rights will happen without prior notification and asynchronous to the receiver of a mapping.

The above highlights only the basic functionality of the map and unmap operation. In Section ?? we precisely define the semantics of these two operations.

#### Kernel Implementation:

The implementation of unmap has to fulfil three important consistency properties:

**No privilege elevation** It must not be possible to elevate a thread's privileges an a capability received with restricted privileges unless it receives a capability with elevated privileges from a thread that has such a capability.

**Revoke-ability** Any mapped right must be revocable. In particular this means that the tree of derived mappings must remain traversable until all rights have been revoked in it. Two pass revocation (privileges in the first pass, nodes in the second) or post-order revocation ensure this property.

**Restart-ability** In case unmap returns unsuccessfully, it must be possible to restart a similar unmap operation. Unmap can for example return with no success if the unmapping thread revokes access to the resources with which it is implemented. In this case other threads in the same address space or threads from which the capabilities originated must be able to restart the unmap operation.

#### Reasoning:

Grant and copy have been excluded from this specification and are considered experimental (i.e., they will be added only if a relevant usage scenario cannot reasonably be built without).

Copy transfers a capability like map does, but instead of the mapper, the threads in the address space from which the mapped capability origins gets the right to later on revoke the copied rights. Grant is similar to copy except that the capability is removed from the granting space. Derived mappings may thereby persist the removal of the granted capability and in this case would be revocable by the recipient of the granted capability.

The reasons we excluded grant and copy are the following: implementing grant of a smaller page out of a larger page with the intended semantics that derived mappings persist requires splitting the large page. This operation is difficult and complex to implement and no use case has been found justifying this effort. Copy is less complex to implement but the intended use of copy: to forward capabilities without requiring further resources contradicts our resource model. Precisely, the copying thread must provide the mapping-database resources used to implement the `unmap` system-call.

## 2.1.2 Capability Faults and Exceptions

Whenever a thread invokes a capability referring to an object of mismatching type or comprising insufficient permissions on this object (for invocation purposes, local names referring to no object can be seen as invocations on capabilities referring to an object whose type always mismatches and that comprises no right) the micro-kernel generates a capability fault on behalf and with the resources of the invoking thread and delivers a capability-fault message to this thread's fault handler. In *L4.Sec* a fault handler is a server addressed by a pair of IPC-endpoint capabilities. The kernel attempts to send the capability-fault message to the fault-handler send capability and after delivering this message it atomically sets up a receive operation on the fault-handler receive capability. The capability-fault message contains information on the attempted operation (e.g., a write to a page, an IPC send operation on an IPC-endpoint or an attempt to insert a page or page-table into the memory space), the instruction pointer of this operation and the relevant information needed to resolve this fault (e.g., the memory address of a memory-page fault).

Reasoning:

We discussed the alternative of returning errors instead of capability faults. Both solutions are equally powerful. A non-trivial system on top of *L4.Sec* has to decide which of them should be chosen.

Similar to page faults, exceptions as defined by the underlying hardware architecture (e.g., division by 0) are reflected as IPC-messages and sent to the thread's exception handler send IPC-endpoint. This message contains all the relevant state to resolve the exception which can be done in two ways: first, the thread's state can be modified with the `exchange-registers` system-call, provided the handler has a capability with appropriate permissions on the thread causing the exception. Second, a exception-reply message can be sent from which the kernel extracts the adjusted thread state. *L4.Sec* provides for reflecting capability faults as exceptions by sending a special message as reply to a capability fault (see Capability Fault and Exception Protocols in Section ??).

## 2.2 Threads

Threads are the active entities in *L4.Sec* and are therefore subject to scheduling. Threads can be stopped or active whereby active threads can execute, block waiting for a message transfer, be halted or be preempted (i.e., able to execute but not currently executing). Each thread has at most one address space associated with it at a given time. Threads with no address space associated are halted and must be *bound* to an address space and be resumed to start executing again.

### 2.2.1 exchange-registers

The `exchange-registers` system-call enables any thread possessing a capability with read, respectively write permissions on a thread to read, respectively to modify volatile thread state. For example the referred thread's status and registers including the thread's instruction pointer and the stack pointer. In particular, the `exchange-registers` system-call allows to abort blocked and preempted IPC operations and to halt a thread and resume a halted thread. The operation is atomic with respect to further exchange registers operations and with respect to the execution of this thread.

Reasoning:

Experimental: to implement a system-call that reveals the identity of a thread to itself, `exchange-registers` could be executed by the thread itself without the need of a thread capability.

### 2.2.2 thread-control

The `thread-control` system-call implements the *bind* operation which associates a thread to an address space. Furthermore, it allows to modify thread-control register.

Reasoning:

We discussed thread startup per startup message in the following way: A thread that is bound with the `thread-control` system-call executes a receive operation on the endpoint specified in the `thread-control` operation and therefore blocks until either a startup message arrives from which the kernel extracts the instruction pointer, the stack pointer and initial registers or the message gets aborted with an `exchange-registers` system-call providing these parameters.

Having two different solutions for start/stop and halt/resume seems to be unnecessary. Therefore the startup message is experimental.

## 2.3 IPC

Inter-process communication (IPC) is the mechanism in *L4.Sec* by which threads in different address spaces can interchange messages and capabilities.

### Reasoning:

The ability to transfer both messages and capabilities with the same mechanism was originally motivated to resolve memory page faults with a single kernel primitive: IPC. A page fault requires both a notification of the faulting thread—a message—which makes it restart the faulting operation and a capability transfer (e.g., a memory page) to resolve the fault

Implementing map inside an IPC operation, however, increases the complexity of this message transfer primitive. In particular, when map is such that it removes existing mappings in the destination address space long interrupt latencies and complicated preemptions may occur during message transfer. Currently, alternative solutions that separate both message and capability transfer are discussed but as yet it cannot be foreseen whether or when they will be included into *L4.Sec*.

IPC in *L4.Sec* is over IPC-endpoints. An endpoint abstracts an unidirectional communication channel to which all threads can send that hold a capability on this endpoint with send permissions and from which all threads can receive that hold a capability on this endpoint with receive permissions. Capability transfer requires send and map permissions on the respective endpoint. In cases where multiple senders or multiple receivers are waiting to send to or to receive from one endpoint it is not defined by the *L4.Sec* API which respective sender or which respective receiver gets selected.

### Reasoning:

Previous approaches [Lie96, Lie99, DLSU04, ?] directly address threads and in addition provide for further means to assist in implementing multi-threaded or even multi-address-space servers (i.e., propagation, deceiving, auto propagation and redirection). Instead, we abstract from threads as message recipients and introduce the endpoint as a mean to address a server. The implementation of this server can then be hidden behind endpoints. In particular, one implementation of a server can be single threaded, another can be multi threaded. Note performance results when implementing a non-trivial system on top of *L4.Sec* may show that the overhead of endpoints in comparison to directly addressing threads may be too large. In this case, we will review endpoints as an abstraction of communication channels.

### Usage Example

Two general rules should be regarded when implementing simple client-server scenarios:

1. All threads of the server receiving from the same endpoint should implement the same interface. Because the receiver selection policy is not defined it cannot be controlled which server thread gets a particular request. If such a control is required or if different non functional properties must be enforced, different endpoints should be used.
2. The server should be provided with an endpoint that only the client is receiving from to send the reply.

### Kernel Implementation:

There is potential for future optimizations in not defining a selection strategy. For example a cross processor IPC implementation could start notifying a receiver on a remote CPU but then decide for a CPU local thread when one comes in before the notification arrived on the remote processor.

Message transfer over an IPC-endpoint is synchronous and blocking, that is, the message is only transferred if both the receiver and the sender have performed a corresponding IPC-receive respectively an IPC-send

operation. Otherwise, the sender or the receiver blocks waiting until the respective partner has performed its operation, until a timeout occurs or until the message gets aborted using the `exchange-registers` system-call. Even though multiple receivers can be waiting on a single endpoint at a time, the kernel selects only one receiver where to transmit the message (i.e., there is no multi cast).

#### Kernel Implementation:

Synchronous and blocking IPC avoids any message-buffer management inside the kernel. It further ensures that a message can be processed by a receiver without being overwritten by subsequent messages unless the receiver accepts these messages and vice versa, that the message to be sent is not overwritten before it is sent unless the sender accepts subsequent messages.

Note also, though the message communication is unidirectional, status information flows backwards from the receiver to the sender which could be exploited as a channel.

#### Reasoning:

As a minimal information, synchronous IPC signals to the sender whether the receiver has accepted the message or not. This bit of information is sufficient to construct a channel and when avoiding even this bit of information the message transfer becomes unreliable. Instead of providing for an unreliable communication primitive in the kernel, we decided to maintain the back channel for performance reasons and propose to implement unidirectional communication at user level. For example, a trusted mediator can be inserted that forwards the messages of the sender and suppresses the information sent back from the receiver. Alternatively a direct channel could be established with shared memory that the sender can write but that the receiver can only read (provided countermeasures are in place to prevent the cache timing channel).

## 2.4 Scheduling

*L4.Sec* does not restrict the scheduling algorithm that has to be implemented in the kernel. Instead it defines a framework that scheduling algorithms inside the kernel must implement. The particular algorithm is implementation specific and may vary in different implementations.

#### Reasoning:

So far we did not come up with a reasonable mechanisms that allows us to implement arbitrary scheduling at user level. The implementation described below, for example, lacks support for dynamic scheduling algorithms such as EDF, etc. Also due to performance or security reasons it may be necessary to adopt the primitive in-kernel scheduler to better adhere to the application scenario.

### 2.4.1 Reservation Framework

A *reservation* is an abstraction of CPU time on one processor. The thread that is associated with a particular reservation can consume the CPU time represented by this reservation. The condition when and the amount of CPU time represented by a reservation depends on the scheduling algorithm. For example a reservation for the classic static priority round robin algorithm implemented in previous L4 kernels would define a time-slice and a priority that can be consumed if all higher prioritized threads block and when no thread of the same priority is executing according to the round robin policy.

A user-level scheduler can associate a reservation with a thread. Multiple reservations can be associated with a single thread at the same time but only one thread can be associated to a single reservation. An attempt to associate a second thread to an already associated reservation will break up the prior association. The order in which reservations are associated to a thread may be relevant depending on the scheduling algorithm.

Reasoning:

Quality assuring scheduling requires multiple reservations to be associated to a single thread. The thread itself can switch to the next reservation using the `thread-switch` system-call.

A reservation can be associated with a thread using the `schedule` system-call, provided thread write and thread schedule permissions on the thread and write permissions on the reservation. The `schedule` system-call can also be used to read and write the scheduling parameters in a reservation, provided reservation read, respectively reservation write permissions. It is sufficient to hold a reservation write capability to break up an association.

Reasoning:

The reservation framework prevents an attack on the scheduler that can be mounted if the scheduler does not create the threads it schedules. By revoking the thread capability from the scheduler the scheduler can no longer name and therefore modify the scheduling parameters of a thread. Reservations solve this problem because they can be created by the scheduler independent of the threads it schedules. Even though the thread capability may get revoked the scheduler can still influence the thread's scheduling parameters by modifying the reservation or breaking up the reservation-to-thread association.

A thread can voluntarily release the CPU it is currently running on using the `thread-switch` system-call. Depending on the scheduling policy it may also be possible to specify a thread to which the invoking thread switches, donating its reserved CPU time.

## 2.4.2 Scheduling Control

To set scheduling parameters, the `schedule` system-call takes two reservations capabilities as parameters: a source reservation and a destination reservations. The parameters that can be set in the destination reservation are limited by the parameters of the source reservation such that in total the two reservations obey the same or less CPU time than the source reservation obeyed prior to this *split* operation. The precise semantics of the *split* operation obviously depends on the scheduling policy. For example, for a proportional-share policy the sum of shares of the *split* reservations must be less or equal to the share of the source reservation prior to the split.

At system startup the kernel offers an initial reservation obeying full CPU time.

## 2.4.3 Scheduling in Dresden's *L4.Sec* implementation

The Dresden implementation of *L4.Sec* is meant to implement Jean's and Udo's scheduling framework which will support quality assurance scheduling. **to do: describe scheduling algorithm**

Reasoning:

Donation as described by Steinberg and Wolter [] can not directly be adopted. The problems in adapting their scheme is in identifying the thread to which to donate the reservation. In particular if in a multi threaded server multiple threads are currently servicing requests it is difficult to select the donee with the intention that this thread will become ready to process the donator's request.

## 2.5 Object Creation and Kernel Memory Management

*L4.Sec* provides for managing at user-level any memory used by the kernel to implement a user initiated operation.

Any thread can *convert* a memory page to a *kernel page*, provided convert rights on the memory page using the `convert` system-call. As a result of this operation all access rights on the memory page get preempted (i.e., no operation can be invoked on the memory page while it is a kernel page, however, the rights can still be mapped and unmapped) and the resulting kernel page is associated with a capability address.

Any thread with access to a kernel page can create objects within this kernel page with the intended semantics that the micro-kernel will attempt to allocate the memory it needs to build the object to be created in this page.

Any attempt to access a memory page that has been converted to a kernel page generates a capability fault even though the thread may possess a capability that authorizes this operation.

Any thread in the address space of the threads that converted the kernel page or in an address space from which the converter directly or indirectly received the memory page it converted can unmap the convert right from the memory page. In this case, all objects that have been created in the corresponding kernel page and all objects that inherently depend on these objects get destroyed, the memory page gets cleared and all rights on this page get resumed (i.e., invocations on the memory page won't generate a fault if the access is permitted by the invoked capability). Privileges are resumed at a capability only after the unmap operation completed revoking the rights from this capability.

The `create` operations take as parameter a set of kernel pages specified through a capability flexpage with the intended meaning that the resources for the created object should be allocated in these kernel pages. *L4.Sec* provides a limited control of where in these kernel pages the objects are placed by requiring each implementation to publish the size of the data structures to allocate, their alignment requirements and the placement policy that is applied on the specified flexpages. For example, to create an address space on x86 one requires a first level page-table (size and alignment 4KB), an IO permission bitmap (2x 4KB may be non consecutive) and probably an address-space control block (size in the order of 1KB, arbitrary alignment), the alignment policy for this implementation is first fit.

#### Reasoning:

The reasoning behind this interface is to provide for both an architecture independent manner of creating objects (e.g., in a pool of memory) by kernel-memory management policies that do not care about placement and an architecture and implementation dependent interface for those policies that must control the placement of objects.

Note we discussed various alternatives with varying degrees of controlling placement and allocation strategy. This choice was based on an educated guess of what typical kernel-memory management policies may need. It may thus change after we analyzed kernel-memory management policies in a non-toy system built on top of *L4.Sec*.

First class kernel-objects (i.e., threads, address-spaces, reservations and endpoints) are created explicitly. As a result of a successful create operation the creator is provided with a capability on this object with full permissions. Second-class kernel objects (i.e., page tables and capability tables —the page tables of the capability space— and mapping-database nodes) cannot be named through a capability referring to the object and must therefore be implicitly named by the resources with which they are implemented. As such we introduce an operation that defines the resources with which to implement these second-class objects and determine its use in the same operation.

When transferring capabilities, when converting a memory page or when creating a first class object, mapping-database nodes must be created to prepare for an unmap. The resources in which to create these mapping-database nodes are specified as a parameter of these respective operations whereby it is provided for transferring multiple capabilities to multiple destinations (using multiple map-items) with the same set of kernel pages (i.e., with a single map resource item specifying the resources to use for the capability transfer). The resources for transferring capabilities are provided by the sending thread.

**Reasoning:**

The alternative —receiver provided mapping resources— was discussed and discarded because of the page-fault case where the receiver typically lacks the resources to accept a mapping. Additionally it is currently in discussion whether page and capability tables as first class objects simplify the design.

When creating page tables or capability tables in addition to the resources we have to specify the local names and the address space that these tables should back. *L4.Sec* implements this functionality through special messages containing resource items that determine the local name where to place the created page or capability table in the address space of the receiving thread. As with map this destination can further be limited by the receiver (see Section ??).

### 2.5.1 Object Destruction

Objects get destroyed either when no capability refers to them (i.e., when the capability of the creator is flushed<sup>1</sup>) or when the resources with which the object is implemented are revoked. In both cases, the object and all objects that inherently depend on this object are destroyed. For example, the page-tables and capability-tables of an address space get destroyed and all capabilities associated with the names these tables implement get unmapped when and prior to destroying the address space. The threads executing in this address space are unbound and thereby stopped because for threads to exist no address space is required.

---

<sup>1</sup>This changes when reintroducing grant or copy.

Kernel Implementation:

The sequence by which objects must be destroyed is the following:

**Revoke all permissions on this object, though not the creator’s capability.** As a result of this no method can be invoked on the object to be destroyed. It is, however, important to not remove the name-to-capability association of the creator’s capability because the creator must be able to complete the destroy operation when flushing this capability.

**Stop and De-associate all objects associated with the object to be destroyed.** For example stop and unbind all threads when destroying an address space or all reservations when destroying a thread. After all threads are stopped no further mappings can be made to the task (remember we already revoked task bind permissions).

**Destroy all dependent objects.** For example page-tables and capability-tables inherently depend on an address-space and must therefore be destroyed prior to the task. Note, that all capability mappings need to be flushed prior to destroying a page-table.

**Destroy the object and remove its final capability.** At last the final capability can be removed and the space of the object be freed up.

The above memory management scheme in combination with the privilege management can be traversed as a tree which is rooted in the physical memory mappings in Sigma0. The structure is a tree because to create an object a thread requires a converted kernel page and to convert a kernel page it must have a memory page. Note, however, that in this memory page there might have been allocated a mapping node which serves for mapping the very same memory page to another address space. For this mapping to be performed, the memory page must already be converted because otherwise no mapping-node could be created in it.

Note also, because memory may serve in mapping other memory pages that there might be a cascading effect when unmapping a memory page. This effect occurs when using this memory page a mapping of another memory page has been implemented and so on. User-level protocols should prepare for avoiding these effects non-voluntarily by negotiating which kernel pages can be used for critical mappings.

Also it may be that a thread revokes the memory page with which it was created. In this case the thread is stopped and the destroy operation is aborted such that it can be restarted by another thread of the same address space or by a thread from which the memory page has directly or indirectly been received.

## 2.5.2 Address Space Construction and Startup

In *L4.Sec* a newly created address space is empty (i.e., no capabilities have been transferred to it). Transfer of capabilities in *L4.Sec* is bound to IPC (i.e., there must be a receiving thread accepting capability transfers). When creating a thread it is not bound to an address space so in order to populate an address space with capabilities we must bind a thread to this address space and make it receive further capabilities. In order to perform a receive operation, however, an endpoint capability is required. For this reason we introduced the special case of transferring an endpoint capability to capability address 0. Subsequent capabilities can be mapped together with the startup message to the thread receiving from capability 0. This first startup message should transfer sufficient capabilities for the initial thread to execute (i.e., at least another endpoint to send fault messages, plus the memory pages containing the initial code and data sections for this thread).

## 2.6 IRQs

*L4.Sec* models interrupts as IPC messages from ‘hardware’ threads. When an interrupt occurs the kernel generates an IPC message and sends it to the fault handler of the ‘hardware’ thread and waits for a reply to this message to acknowledge the interrupt. Initially the kernel provides a thread capability to a ‘hardware’ thread for each interrupt line. A ‘hardware’ thread behaves like a normal thread (i.e., it can be bound to address spaces and its fault handler can be set using the `thread-control` system-call) with the exception

that this thread will never execute user level code.

#### Usage Example

To attach to an IRQ (i.e., to prepare for receiving IRQ messages) the ‘hardware’ thread of this respective IRQ must be bound to an address space and the two fault-handler endpoint capabilities must be setup so that the IRQ-handler thread can send and receive messages from these endpoints.

## 2.7 Miscellaneous

### 2.7.1 Processor Control

To control which thread in a system can start, stop or change the frequency and power consumption of a processor, *L4.Sec* introduces CPU capabilities for each CPU in the system. CPU write permissions authorize modifications of CPU state, CPU read permissions authorize reading this state.

### 2.7.2 System Startup

The *L4.Sec* system initially starts one address space with a single thread on the bootstrap processor. All other processors remain to be started using the `processor-control` system-call. The initial address space (called  $\sigma_0$ ) is setup by the micro-kernel to hold memory-page capabilities to all available physical memory pages minus those needed by the micro-kernel (kernel code and initial data) in the largest possible and supported hardware page size. The virtual addresses of these capabilities are the physical addresses of the respective memory pages. Furthermore,  $\sigma_0$  holds capabilities to the following objects: the CPUs, a task capability to the initial address space, a thread capability to the initially created thread, a reservation, thread capabilities to all “hardware” interrupt threads, and, kernel-page capabilities to kernel pages that can be used to create further objects (at least the memory for the mapping-database nodes, page- and capability-tables necessary to retype further kernel pages is provided). For the whole  $\sigma_0$  startup protocol see ??.

## 2.8 Summary

### 2.8.1 List of Kernel Objects

#### First Class Objects

In *L4.Sec* the following objects are first-class (i.e., nameable) kernel objects:

**Memory Page:** a page of physical memory,

**IO-Port:** an IO-port (e.g., on x86),

**Address Space:** implements mapping of local names to capabilities,

**Thread:** active entity,

**Endpoint:** communication channel,

**Reservation:** abstraction of CPU time,

**Kernel Page:** a page of physical memory in which the micro-kernel can allocate the data structures of other kernel objects<sup>2</sup>,

**CPU:** a physical CPU or hyper-thread.

#### Second Class Objects

This list is complemented by the following second-class objects (i.e., objects that cannot be addressed through a capability):

**Mapping-Database Node:** used to implement the `unmap` system-call,

**Page Table:** implement mapping of virtual addresses to memory pages,

**Capability Table:** implement the mapping of capability addresses to kernel objects besides memory pages and IO-ports.

### 2.8.2 List of system-calls

*L4.Sec* implements the following system-calls:

**IPC:** inter-process communication, including capability transfer (i.e., `map`),

**exchange registers:** read and modify a thread's registers, state and parameters,

**thread control:** bind a thread to an address space,

**create:** creates threads, address spaces, endpoints or reservations,

**convert:** convert a memory page into a kernel page,

**unmap:** revoke permissions (`unmap` implicitly destroys objects),

**schedule:** read and modify scheduling parameters in reservation, associate reservation with thread,

**thread switch:** voluntarily release CPU, switch to destination thread,

---

<sup>2</sup>Of course except physical resources like CPU, memory page or IO-port

***processor control:*** start, stop CPU, modify frequency, power.

Reasoning:

The `convert` system-call could be seen as a special version of `create`. It is mentioned here as independent system call since it has a slightly different semantic (it takes a memory-page and returns a kernel object).

# Chapter 3

## Application Programming Interface

### 3.1 Introduction

The following binding named W5 is described in detail in [Kau05].

#### 3.1.1 Parameter Words

Parameter words are an abstraction of input and output parameters of a system-call. They are named  $W_0, W_1, \dots, W_n$ . The number of parameter words could be theoretically unlimited but a practical limit of 256 is assumed. The parameter words have the same width as the hardware registers: 32 bit on x86.

Parameter words are mapped to hardware registers or memory locations. W5 assumes that at least 5 of them are hardware registers.

#### Reasoning:

Parameter Words as virtual register offer a simple interface between API and ABI and they make the binding more portable. This abstraction can be build on top of different architectures or kernel-memory mappings like UTCBs. It is independent of the used kernel-entry method.

#### 3.1.2 System-Call Numbers

The binding distinguish different system-calls according to a system-call number. The following numbers are defined:

0	short-ipc
1	long-ipc ( <i>reserved</i> )
2	unmap
3	create
4	thread-control
5	exchange-registers
6	schedule ( <i>reserved</i> )
15	debug

#### 3.1.3 TCRs

Thread Control Register (TCR) are relatively static data of a thread that resides in the kernel TCB. These are e.g. the pager endpoints or the task of a thread. For efficiency the TCR's are numbered and the number

of TCR's is limited to 64.

---

0	<i>invalid</i>
1	task to bind ( <i>writeonly</i> )
2	user defined value
3	( <i>reserved</i> )
4,5	pager send/receive capability id
6,7	preempter ( <i>reserved</i> )
8,9	exception handler ( <i>reserved</i> )
10	send resources
11	receive resources
12-15	receive windows for 4 spaces ( <i>reserved</i> )
16	timeout0
17	timeout1

### 3.1.4 Datatypes

---

#### Flexpage Format

Flexpages describe an aligned region of a name-space, starting at a base  $B$  with a size of  $2^S$  and include permissions  $P$  for this region. They are used e.g. as parameters for `map` and `unmap` and for regions of resources in `create`. Flexpages have a `type` parameter which allows to distinguish different spaces.

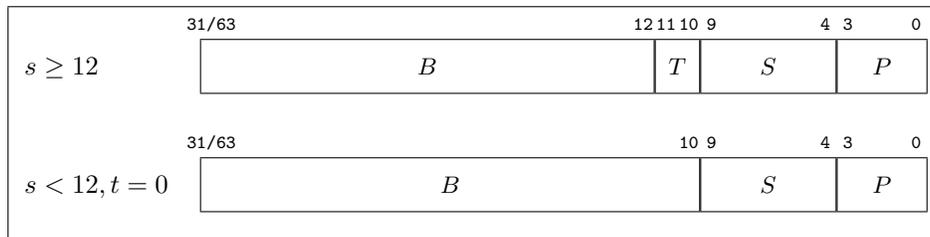


Figure 3.1: Flexpage Format

#### P (permission) field

The permissions of the flexpage.

#### S (size) field

The size of the flexpage.

#### T (type) field

The type of the flexpage.

---

0	Memory Space
1	Capability Space
2	IO Space
3	<i>reserved</i>

#### B (base) field

The base of the flexpage.

## 3.2 Binding

This section shows the W5 binding for the most used system-calls. Other system-calls are not as stable to be defined here.

### 3.2.1 short-ipc - transfer some words

See Section 2.3 for a description of this system-call.

---

#### Short IPC Input Parameters

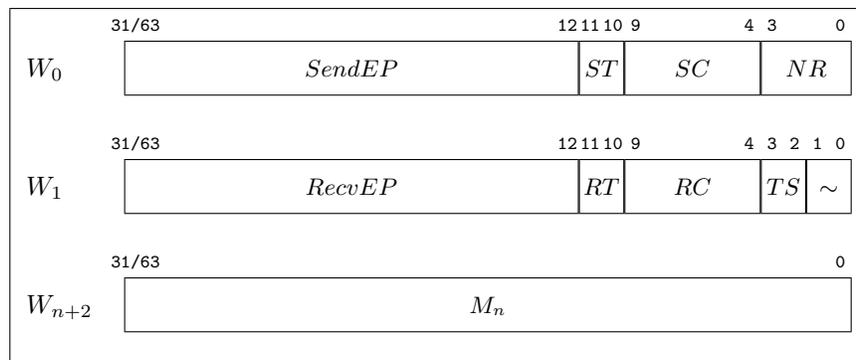


Figure 3.2: Short IPC Input Parameters

#### NR (system-call number) field

The number of the system-call to call. See Section 3.1.2 for the value.

#### SC (send count) field

The number of message words to send. Up to 63 words can be send in a single Short-IPC.

#### ST (send typed) field

The number of typed items to send. Up to 3 mappings can be transfered with this. The typed items are stored at the end of the message to benefit from cache effects while map after message copy.

#### Send EP (send endpoint capability) field

The capability ID of the endpoint the message is send to.

#### TS (timeout number) field

Timeout is either (*never, never*) with  $TS = 0$ , (*zero, never*) with  $TS = 1$  or it is an offset into a Timeout TCR ( $TS > 1$ ).

#### RC (receive count) field

The number of message words to receive. Up to 63 words can be received in a single Short-IPC.

#### RT (receive typed) field

The number of typed items to receive.

#### Recv EP (receive endpoint capability) field

The capability ID of the endpoint a message is received from.

#### M (send message word) fields

The message to send is stored in SC parameter words.

---

### Short IPC Output Parameters

The message to send is stored in SC parameter words.

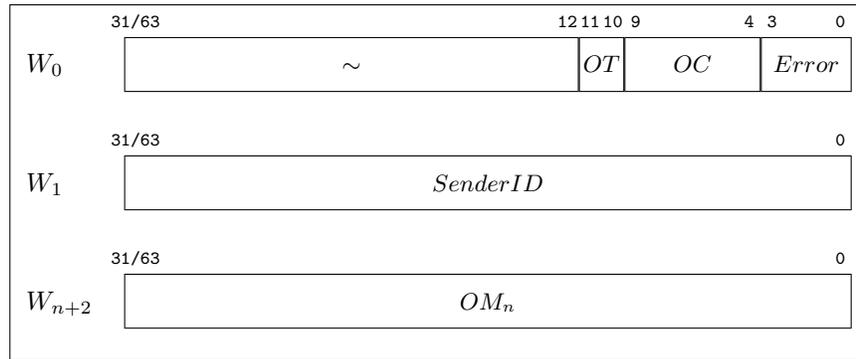


Figure 3.3: Short IPC Output Parameters

#### Error field

The base error code from this system-call. See Section 3.1.2 for values. Some system-calls need more values than the 16, which are possible with this field. They have to use an extended error number to achieve this.

#### OC (received count) field

The number of message words received. This is the minimum of SC, given by the sender, and RC given by this receiver.

#### OT (received typed) field

The number of typed items received.

#### Sender ID field

The sender ID identifies the sender of the message. This is the badge of the endpoint with the Sender-ID TCR appended.

#### OM (received message word) fields

The received message is stored in OC parameter words.

---

### Permissions

#### SendEP

send (w) - to send a message  
 map (t) - to map via typed items

#### RecvEP

recv (r) - to receive a message

### 3.2.2 exchange-registers - exchange volatile thread state

See Section 2.2.1 for a description of this system-call.

to do: add SRH output values

---

#### Ex Regs Input Parameters

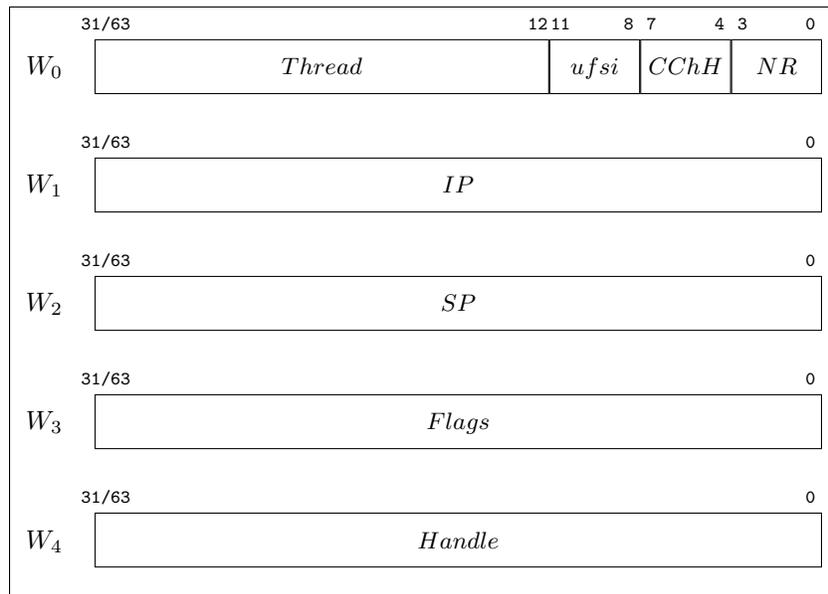


Figure 3.4: Ex Regs Input Parameters

#### NR (system-call number) field

The number of the system-call to call. See Section 3.1.2 for the value.

#### CChH (cancel and halt flags) field

The H-bit allows to halt or resume a thread if  $h = 1$ . case. The CC-bits are interpreted as abort of (no,all,recv,nested) IPC.

#### ufsi (set flags) field

The flags control whether the corresponding parameters are set or simply ignored.

#### Thread (thread capability) field

The capability ID of the thread to change.

#### IP (instruction pointer) field

The instruction pointer to set if  $i=1$ .

#### SP (stack pointer) field

The stack pointer to set if  $s=1$ .

#### Flags (processor flags) field

The processor flags to set if  $f=1$ .

#### Handle (user-defined handle) field

A user defined value to set if  $u=1$ . This sets the user-defined handle thread ctrl register.

---

### Ex Regs Output Parameters

A user defined value to set if u=1. This sets the user-defined handle thread ctrl register.

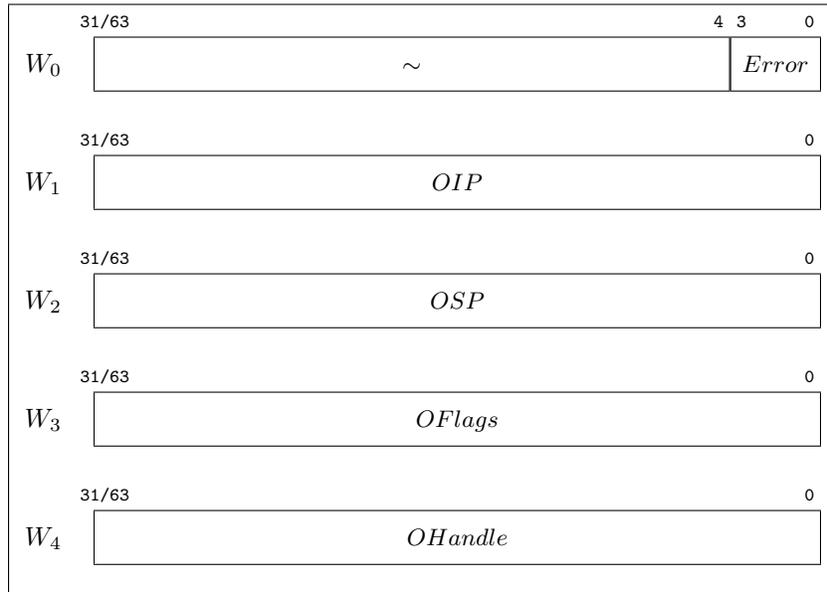


Figure 3.5: Ex Regs Output Parameters

#### Error field

The base error code from this system-call. See Section 3.1.2 for values. Some system-calls need more values than the 16, which are possible with this field. They have to use an extended error number to achieve this.

#### OIP (old instruction pointer) field

Returns the old instruction pointer.

#### OSP (old stack pointer) field

Returns the old stack pointer.

#### OFlags (old processor flags) field

Returns the old processor flags.

#### OHandle (old user defined handle) field

Returns an old user defined value. This returns the user-defined handle thread ctrl register.

---

### Permissions

#### Thread

read (r) - to read thread state

write (w) - to modify thread state

### 3.2.3 unmap - revoke permissions recursively

See Section ?? for a description of this system-call.

#### Unmap Input Parameters

Returns an old user defined value. This returns the user-defined handle thread ctrl register.

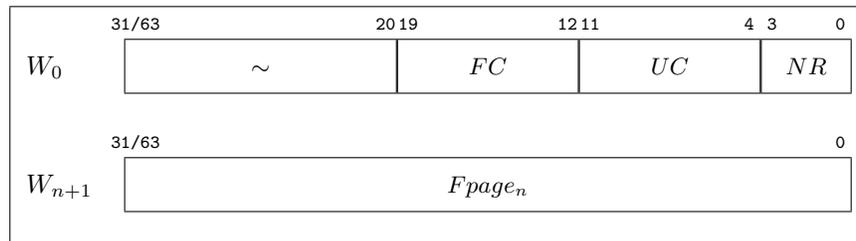


Figure 3.6: Unmap Input Parameters

#### NR (system-call number) field

The number of the system-call to call. See Section 3.1.2 for the value.

#### UC (unmap count) field

Unmap flexpages from (0) to ( $UC - 1$ ).

#### FC (flush count) field

Flush flexpages from ( $UC$ ) to ( $UC + FC - 1$ ).

#### Fpage (flexpage) fields

flexpage to unmap or flush.

#### Unmap Output Parameters

flexpage to unmap or flush.

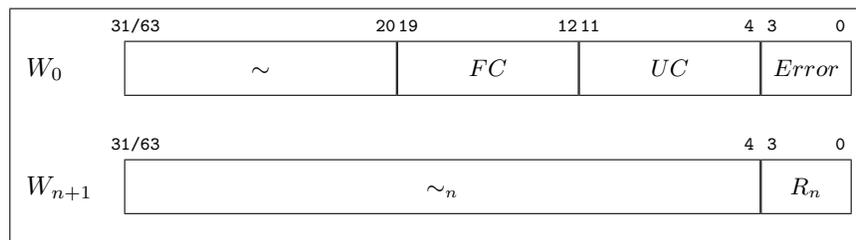


Figure 3.7: Unmap Output Parameters

#### Error field

The base error code from this system-call. See Section 3.1.2 for values. Some system-calls need more values than the 16, which are possible with this field. They have to use an extended error number to achieve this.

**UC (unmap count) field**

Unchanged unmap count value.

**FC (flush count) field**

Unchanged flush count value.

**R (reference bits) fields**

If the input parameter  $MR_n$  was a memory flexpage this returns the ORed values of the reference bits for all memory page capabilities in this flexpage. A bit in  $R$  is set if and only if after the last reset of reference bits a thread that directly or indirectly received a capability contained in this flexpage has performed a corresponding access to that page. The internal reference bits are reset after reading them.

**Reasoning:**

Note: It is also possible to implement reference bits for kernel objects and IO-ports. This is not included here, since there is currently no application for it.

---

**Permissions**

None.

### 3.2.4 create - create kernel objects

See Section 2.5 for a description of this system-call.

---

#### Create Input Parameters

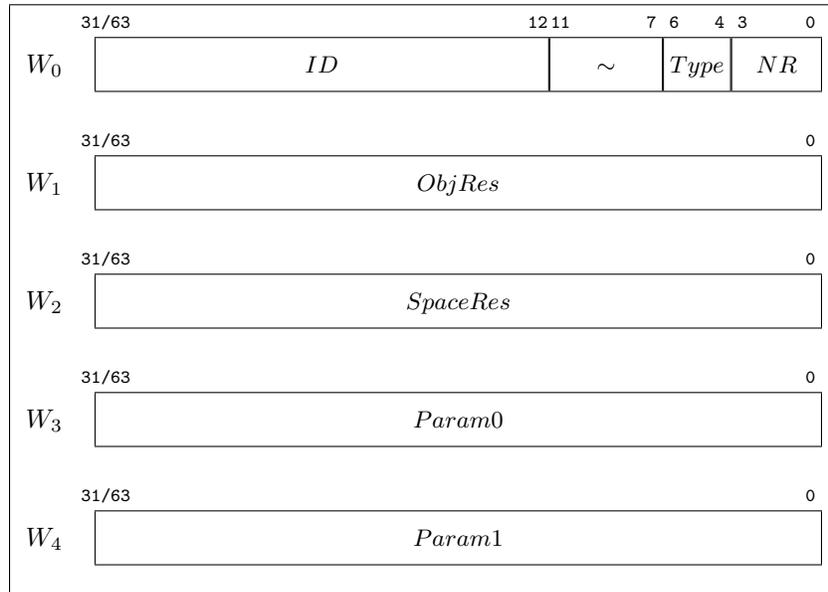


Figure 3.8: Create Input Parameters

#### NR (system-call number) field

The number of the system-call to call. See Section 3.1.2 for the value.

#### Type (object type) field

The type of the object to create.

1	Task
2	Thread
3	Endpoint
4	Kernel Memory
5	Reservation ( <i>reserved</i> )

#### ID (object capability id) field

The capability ID of the new object.

#### ObjRes (object resources) field

Resource flexpage to create the new object. The kernel-memory, this flexpage points to, is used allocate the object.

#### SpaceRes (space resources) field

Resources to insert the object in the capability space. The kernel-memory, this flexpage points to, is used to allocate capability-space tables.

#### Param0 (first parameter) field

Type dependent parameter. This is the memory region to convert while creating a kernel-memory object or the source flexpage to map while creating a task.

**Param1 (second parameter) field**

Type dependent parameter. This flexpage describes the destination region of a map while creating a task.

**Create Output Parameters**

Type dependent parameter. This flexpage describes the destination region of a map while creating a task.

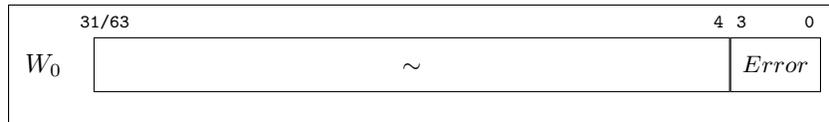


Figure 3.9: Create Output Parameters

**Error field**

The base error code from this system-call. See Section 3.1.2 for values. Some system-calls need more values than the 16, which are possible with this field. They have to use an extended error number to achieve this.

**Permissions****ObjRes**

The appropriate rights to create this type of object.

### 3.2.5 thread-control - read and write thread control register

See Section 2.2.2 for a description of this system-call.

---

#### Thread Control Input Parameters

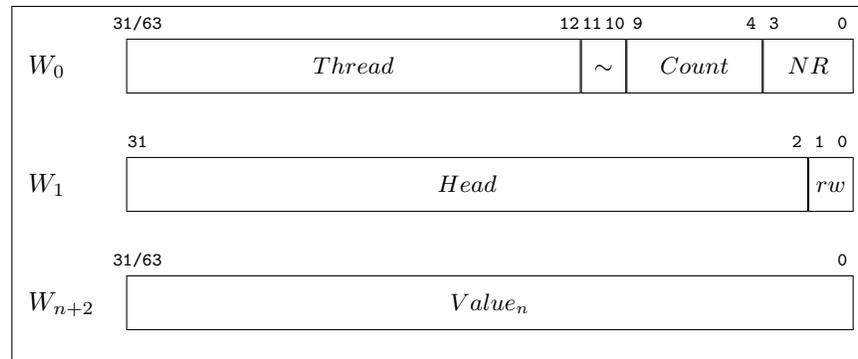


Figure 3.10: Thread Control Input Parameters

#### NR (system-call number) field

The number of the system-call to call. See Section 3.1.2 for the value.

#### Count (TCR count) field

Number of in- and output TCR values.

#### Thread (thread capability) field

The capability ID of the thread to control.

#### rw (flags) field

Decides whether the following 5 TCRs are read and/or written.

#### Head (TCR head) field

The head of this TCR list consists of 5 TCR indices each 6 bits width. A head in a TCR list is followed by 5 TCR values. The head is repeated every 6 parameter words to read/write up to 63 TCRs with one call.

#### Value (TCR values) fields

A TCR value to set if w=1.

---

#### Thread Control Output Parameters

A TCR value to set if w=1.

#### Error field

The base error code from this system-call. See Section 3.1.2 for values. Some system-calls need more values than the 16, which are possible with this field. They have to use an extended error number to achieve this.

#### Count (TCR count) field

Unmodified number of in- and output TCR values.

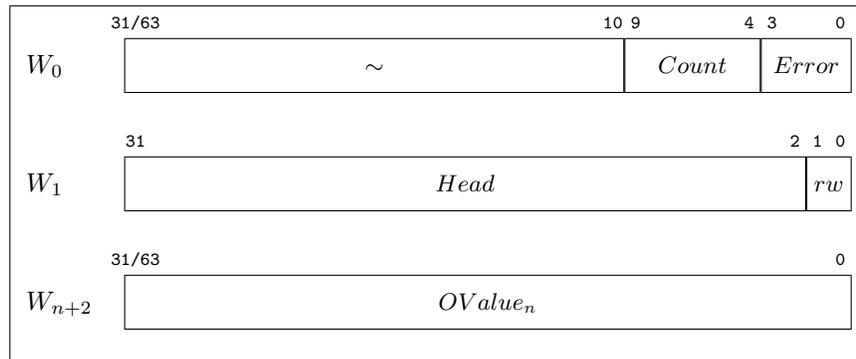


Figure 3.11: Thread Control Output Parameters

**rw (flags) field**

Unmodified flags.

**Head (TCR head) field**

Unmodified head.

**OValue (output TCR values) fields**

The value of a TCR if r=1.

**Permissions****Thread**

read (r) - to read TCRs

write (w) - to write TCRs

## Chapter 4

# Application Binary Interface

This ABI chapter describes the mapping from parameter words to registers and memory for different architectures and kernel entry methods.

### 4.1 x86

#### 4.1.1 `sysenter`

The kernel entry is done via `sysenter`. The parameter word mapping is:

EAX	$W_0$
EBX	$W_1$
ESI	$W_2$
EDI	$W_3$
EBP	$W_4$
ECX	<i>user ESP</i>
EDX	<i>user EIP</i>
ESP	<i>kernel ESP</i>
ECX +0x00	$W_5$
ECX +0x04	$W_6$
ECX +...	...

#### 4.1.2 `int 0x40`

The kernel entry is done via `int 0x40`. The parameter word mapping is unoptimized and the same as in the `sysenter` mapping, with the exception that EDX is unused and ECX points to a user defined buffer and not to the user stack pointer.



# Bibliography

- [DLSU04] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: <http://14hq.org/docs/manuals/>.
- [Kau05] Bernhard Kauer. L4.sec Implementation - Kernel Memory Management. Master's thesis, TU Dresden, May 2005.
- [Lie95] J. Liedtke. On  $\mu$ -kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.
- [Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.
- [Lie99] J. Liedtke. L4 nucleus version x reference manual (x86). Technical report, September 1999.