L4.sec Implementation
Kernel Memory Management

Bernhard Kauer

kauer@os.inf.tu-dresden.de

May 19, 2005

Contents

List of Figures

Chapter 1

Introduction

L4 does not have a security model and hence needs one.

Hermann Härtig

Microkernels aim to build flexible, small and modular systems. The first generation of microkernels, like Mach [ABB+86], with its feature richness and size, did not achieve this goal. Mainly the poor performance of Mach's IPC was the reason for this. This performance gap was closed by a radically new design [Lie93]. The microkernel provides now only the minimal functionality, needed to build a whole system on top of it. This approach increases the IPC performance by an order of magnitude. L4 was the firs, of the so called second generation of microkernels, which use these design techniques.

DROPS, the Dresden Real-Time Operating System[1], needs a real-time capable microkernel. The first L4 implementation was not suitable for real-time issues, therefore the L4 interface was reimplemented in Fiasco. Fiasco shows that real-time and performance can be combined. It closes the real-time gap by using better synchronization techniques [Hoh02].

Microkernel based systems are particularly suitable for security critical applications. They promise a small trusted computing base, due to modularity and small code size. Current L4 implementations have different security problems. Neither the kernel-resource management nor IPC control are implemented in a fast and reliable way. L4.sec tries to close this security gap.

This thesis is the first that evaluates, refines and implements the L4.sec ideas [PV05]. It is focused on the new kernel-memory management. The L4.sec ideas influence the whole kernel implementation and system design. Therefore they must be seen in the context of the overall system.

It is obvious that the result cannot be a complete implementation of L4.sec, due to the time-frame of this work. But this work could give an insight into the implementation problems and it should show, whether there are fundamental issues in the L4.sec ideas, which cannot be solved.

Document Layout

This thesis is organized as follows. In the next chapter I provide the fundamentals of this work. This includes an introduction into the new concepts L4.sec adds to L4. In Chapter 3 I evaluate the consequences of the new kernel-memory management to the implementation of the microkernel. Chapter 4 shows a new system-call binding, parts of L4.sec that must not be implemented and other implementation details. In Chapter 5 I give an estimation of the Short-IPC performance overhead, which is introduced with L4.sec and the code complexity of the current implementation.

[1] A research project at the TU Dresden with the aim to support applications with *Quality of Service* requirements. See also the Glossary A.1.

Additionally I propose a user-level communication protocol and describe scenarios how the new kernel-memory management could be used. The last Chapter 6 closes the thesis with a summary and a suggestion of further work areas.

Acknowledgement

First, I would like to thank Prof. Hermann Härtig for the opportunity to work on this project in the Operating Systems Group at the TU Dresden and for the time to discuss problems with him. My special thanks go to my supervisor Marcus Völp for the long clarifying design discussion and the time he sacrifices to teach me how to write a theses. Dietrich Clauss deserves many thanks to introducing me into FIASCO coding. I would also like to thank to everyone in the group who discuss ideas or give hints, which helps me to enhance this work. Thanks goes to Johannes Kauer, Franziska Gräbe, Marcus Völp and Dietrich Clauss for there valuable feedback while proofreading the thesis.

Finally, many thanks goes to my family for the possibility to study in Dresden and for their constant support all the time as well as to my friends for their encouragement. I cannot enumerate them all and give therefore only an abbreviation here:

$$JFMJBFAM^{+}WHFJLKSG.$$

Chapter 2

Fundamentals

In this chapter I briefly describe the L4 model and the new concepts introduced with L4.sec. This description of L4 could not give a complete view. Please take a look for a deeper insight into L4 at the reference manuals [DLSU04, PV05] and the websites [TUD, HQL]. If you are familiar with L4 you can skip the next section and continue with section 2.2.

2.1 L4

The determining criterion used is functionality, not performance.

Jochen Liedtke

2.1.1 What is L4?

L4 is a microkernel interface of the second generation initially developed by Jochen Liedtke [Lie95]. He also wrote the first implementation for x86 in assembler. Since that time the L4 interface is implemented several times [TUD] in high level languages like C or C++ and ported to different architectures (e.g., Alpha, ARM, MIPS).

The philosophy of L4, to provide only *minimal concepts* and that *functionality, not performance* should be the criterion to decide what is implemented in the kernel [Lie95], is one of the reasons of the success of L4.

The interface was enhanced from the original L4.V2 [Lie96], to the L4.X0 [Lie99] and finally to the L4.X2 [DLSU04] API. L4.sec is the latest L4 interface.

Fiasco

FIASCO [Hoh98] is an implementation of L4 in C++ and assembler written at the TU Dresden by Michael Hohmuth and others. It is fully preemptible and it has therefore excellent realtime properties. Until now FIASCO supports the L4.V2, L4.X0 and L4.X2 API's and it runs on native x86, ARM [War03] and as user-mode program on Linux, which is called FIASCOUX [Ste02].

Only parts of the Fiasco code could be reused (see Section 5.2.3), but even so it was the base for implementation of this work.

FIASCO Portings

The former portings of FIASCO to other architectures, like IA64 [War02], FIASCOUX [Ste02], ARM [War03], or to other API's like L4.X2 [Cla04] are not directly comparable to this work for the following reasons. First, they used a fixed L4 specification, for which other implementations existed. Second, they did not change the kernel memory management in such an extensive way and could therefore reuse much more code. However, they modularize most parts of Fiasco, which helps this work to reuse basic functionality of Fiasco.

2.1.2 Minimal Basic Concepts

L4 provides only a minimal set of abstractions or basic concepts. These concepts are:

Virtual Address Spaces
The microkernel protects virtual address spaces from each other with the help of the *hardware paging mechanism.* Pages can be `map`'ed from one address space to another. This mapping can be recursively revoked by `unmap`. A task is an entity that consists of an address space and of threads running in them. Initially all available physical memory is available to the first task, which is called σ_0.

Threads
Threads are the activities in the L4 system running in tasks. They execute user-level code and can communicate with each other through IPC.

Inter-Process-Communication (IPC).
IPC in L4 is synchronous and unbuffered. It is used for message transfer, mapping pages and synchronization of different threads. IPC can be used within and across tasks.

In previous papers unique identifier for threads are also mentioned as a basic concept for L4. This is not important anymore, because in L4.sec threads and other objects have only local names (2.2.1).

2.2 L4.sec - New Concepts

Any problem in computer science can be solved with another layer of indirection.

David Wheeler

The main reason for further development of L4, like the name of L4.sec suggest, was the lack of security. There was no fast way to implement information flow control and no way to manage the kernel memory a task consumes.

To solve the problems of previous versions L4.sec introduces some new concepts to be able to implement efficiently fine-grained access control (capabilities), multi-threaded servers (endpoints) and kernel-memory usage control (kernel-memory objects).

2.2.1 Capabilities

The most important improvements to L4 are the introduction of capabilities. In short, capabilities are references to kernel objects with permissions attached to them. Local names in a task local capability space translate to capabilities.

Kernel Objects

There are two types of kernel objects: named first class objects and unnamed second class objects. Named objects are threads, tasks, endpoints[1], kernel-memory objects [2] and CPU reservations[3]. Unnamed objects are mapping-nodes, page and capability tables and a couple of static kernel objects, which are only created initially at startup[4].

[1] See section 2.2.2.
[2] See section 2.2.3
[3] The scheduling model based on reservations is preliminary and not part of this work.
[4] e.g. the idle thread, IRQ threads, boot time allocators, ...

Capabilities point only to named objects. Unnamed objects are created indirectly and have no user-visible name.

Local Names

Capability ID's are task local and build an own namespace, the capability namespace, which provides a mapping between numbers and capabilities. Capability namespaces can be populated by

- creating new named objects through the `create` system-call, or by
- mapping the capabilities from one capability space to another.

Extending the recursive construction of namespaces through `map` from the memory space to the capability space also means that the corresponding `unmap` system-call can be used in the same way. This means the object is destroyed after the removal of the last capability[5].

A task in L4.sec has beside the memory namespace (virtual address space) and IO namespace [Stö02], also a capability namespace. Since every thread can create new objects in its task[6], there is no global view over all kernel objects. This is different to the memory namespaces where σ_0 knows all physical pages. However σ_0 in L4.sec knows the resources for all kernel objects.

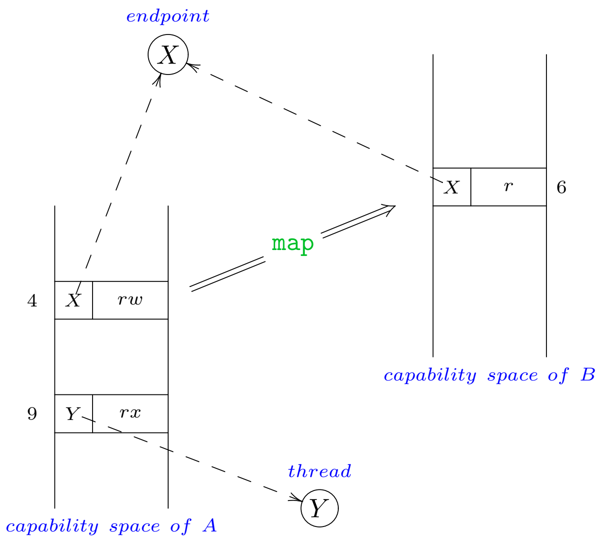

Figure 2.1: Capability example

In Figure 2.1 a task A has two capabilities in its capability space, located at the capability ID “4” and “9”. The first points to the endpoint X and the second to the thread Y. The task A has mapped the endpoint capability without the write permission to the capability space of B. The task B can refer to X with its own capability ID “6”.

Permissions

A capability contains 4 bits used as permissions. This allows to implement a simple rights scheme adopted from virtual page permissions. The semantics of the 4 bits depends on the type of object the capability points to. For example, write permission on a thread allows to modify thread state through `ex_regs` or `thread_ctrl`. Write permissions on an endpoint allows to send to them. See table 2.2 for details[7].

[5]This is the root capability in the capability space of the creating task, because capabilities can only be `map`'ed.
[6]Provided that the task has enough kernel memory.
[7]Note that it includes the changes of this work.

Object	Name	Description
Thread		
	read	thread state is readable
	write	thread state is writeable
	schedulable	reservation could be attached
Endpoint		
	recv	messages could be received
	send	messages could be send
	map	mappings could be transfered
Task		
	bind	threads could be bound to this task
Reservation		
	read	reservation state is readable
	write	reservation state is writeable
KernelMemory		
		to be defined

Figure 2.2: Capability permissions of different objects

2.2.2 Endpoints

To be able to send to one thread in a group of threads endpoints are introduced in L4.sec. Beside this L4.sec replaces the Clans & Chiefs model with a faster one.

What are Endpoints?

Endpoints could be seen as an abstraction for communication channels and they are used as rendezvous points for IPC. Instead of sending directly to a given thread or receiving directly from a thread, the IPC is performed over an endpoint.

An endpoint is a named kernel object which could be created and mapped into a capability namespace. It could be used from many threads across task boundaries and therefore it is used as unidirectional channel. For example in Fig. 2.3 a group of client threads are sent to an endpoint where a group of server threads are wait for the requests of clients. If a sender arrives at the endpoint the kernel selects one of the waiting receivers and performs the IPC between them. If no receiver is found the sender waits at the endpoint for them.

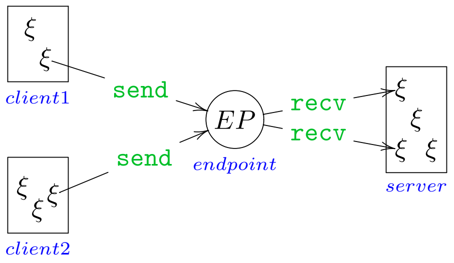

Figure 2.3: An endpoint example

An endpoint hides the thread structure of the server. It allows to dispatch incoming messages to worker threads without the need of a dispatcher thread.

The send permission of the endpoint capability is checked when sending and the receive permission when receiving from the endpoint. The map permission is used to restrict the propagation of capabilities and pages. If it is missing mapping is not allowed over the endpoint.

Sender Identification

Since many clients could send messages through an endpoint to a single server, a mechanism to distinguish different senders is needed. Therefore L4.sec introduces the sender ID. The sender ID is a bit string returned by a `receive`. The prefix of it, known as the badge, is part of the capability of the sending endpoint. The suffix can be specified by the sender at every IPC.

The badge cannot directly be modified. Instead, an additional bit string can be appended. This addition can be done during the mapping of an endpoint capability. This restriction allows to protect the first bits of the sender ID.

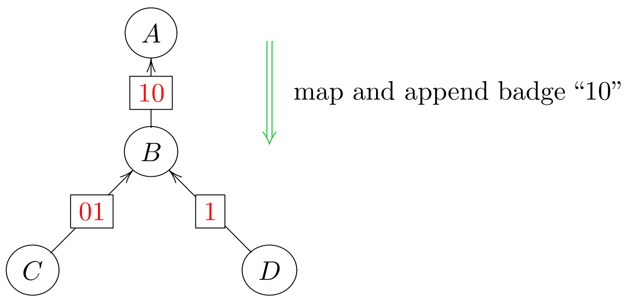

Figure 2.4: Badge example. *A* could send with every sender ID, *B* could send only with sender ID's starting with the bits "10", *C* could send only with "1001", and *D* could send only with "101".

The kernel does not associate any semantics with the sender ID. It is up to the user-level servers to define their usage. For example, parts of the sender ID could be used as session identification and other parts could be used to distinguish different server functions.

Replacement for Clans & Chiefs

Clans & Chiefs [Lie96, Lie92] are a security concept initially developed by Jochen Liedtke for L3, the predecessor of L4. A clan (denoted as an rectangle in figure 2.5) is a set of tasks (denoted as circles) headed by a chief task. If a message tries to cross a clan boundary, the message is redirected to the chief of the clan. The chief could use any security policy to decide whether the message should be forwarded to the destination or not.

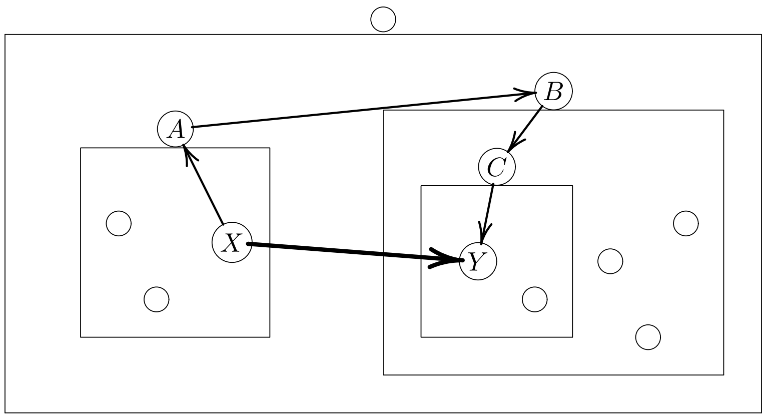

Figure 2.5: Clans & Chiefs are not fast enough. Every message between X and Y must be sent through A,B and C in this scenario.

In most L4 implementations Clans & Chiefs are not implemented because they are simply not fast enough. Crossing a single clan boundary for example doubles the time for the whole message transfer, since 2 IPC's[8] are needed instead of one direct IPC. This overhead must be payed for *every* message crossing a clan boundary, thus Clans & Chiefs are in practice unusable[9].

Beside the performance limitation the Clans & Chiefs concept has also a fundamental problem: it changes the IPC semantic [JTG+00]. If IPC at the sender site returns with no error, it does not mean that the intended receiver got the message. This only mean that a chief received it, which in fact makes IPC asynchronous.

IPC between tasks could be controlled in L4.sec by the presence or absence of a local name for an endpoint. If two tasks do not have capabilities (with the appropriate permissions) pointing to the same endpoint they cannot communicate which each other. Since initially only the creator of a task has a endpoint to the task, it can control which other endpoints are mapped into and out of this task. This role of the task creator is similar to that of the chief of a clan.

Capabilities introduce an initial overhead because establishing an endpoint between two tasks is expensive, but it must be paid only once. There is also an overhead for every message due to the indirection of the local names and the check for permissions. These costs of capabilities should be much smaller then the forwarding costs of Clans & Chiefs.

IPC Redirection

The idea of IPC redirection is presented in [JEL+99]. The kernel maintains a redirection function that decides for every $(src, dest)$ pair of thread ID's, the new destination of the IPC. The IPC is redirected to this new destination. Because global thread ID's build a sparse namespace, they use an inverted page table (hash table) for the lookup.

In contrast to capabilities they use global thread identifiers instead of local names. Capabilities avoid the problems of hash collisions. The capability space is dense, so a simple array lookup can be used, which should decrease the average lookup time.

2.2.3 Kernel-Memory Objects

Kernel-memory objects are a solution to manage and limit kernel memory from the user-level.

What is Kernel Memory?

Kernel memory is the memory the microkernel needs itself to provide its service. Its integrity and availability is protected by the kernel and it is limited at least by the available physical memory. Kernel memory could be for example memory for kernel code, for pagetables of a task, for thread control blocks (TCBs), or mapping nodes, which store `map` dependencies.

There is no way to recover from situations where not enough kernel memory exists. Blindly destroying tasks, like Linux it does, is not a choice. Thats why the *consumption* of kernel memory must be controlled to build a secure system, where not all tasks have to be trust each other.

In summary kernel memory is a limited resource and it must be possible to restrict its usage.

Current Handling

In previous L4 versions a fixed quantum[10] of physical memory was reserved at startup for kernel memory. If it is exhausted the system is stopped[11].

In L4.X2 only a a privileged task can create threads and tasks. Thus the main focus of kernel memory management in L4 is on mapping nodes and the corresponding pagetables. The reason

[8] From the source to the chief and from the chief to the destination.

[9] Jochen Liedtke's argument "crossing clan boundaries occurs seldom enough in practice" does not hold, because many clans must be crossed - in the extreme case one per involved task - to implement fine-grained access control.

[10] usually 10 percent

[11] Asking a privileged task, like σ_0 or σ_1, for more kernel memory was never used, at least not in Fiasco

for this is, that mapping is bound to IPC, an unprivileged system-call, which every thread could execute.

By mapping a single page multiple times in its own memory space every task could occupy[12] kernel memory. Therefore mapping into the own address space is forbidden in L4.X2, but this attack could be easily launched with another cooperating address space if they could communicate with each other.

Bounding the restriction of kernel memory for mapping nodes to IPC control is inflexible, since every message which could contain a mapping must be monitored. Also it is hard to monitor the destruction of mapping nodes resulting from `unmap` calls. Additionally the current approach forbids combining different untrusted servers in one address space and the implementation of multi address-space servers.

User-level Management of Kernel Memory

In [HE03] parts of the kernel memory could be exported to user-level. If an operation in the kernel needs kernel memory, a fault is send to a *kpager*. The *kpager* maps a page to the kernel and could later unmap the page. On unmapping a kernel page, the kernel memory on this page is converted to a secure external representation and given back to the *kpager*. The external representation could be used later to reestablish kernel objects that were previously allocated in this page. This is contrary to our model where the objects are simply destroyed, if a kernel-memory object is unmapped.

This model could be seen as a memory mapped interface, where a *kpager* could call many system-calls in one step on behalf of another thread. It allows to page kernel internal data, but it has the disadvantage that dynamic data like running- or sender-queues and security critical information like scheduling parameters could not be exported.

I have to conclude that the kernel memory problem was never really solved in L4.

Quota Systems

A simple quota system (*"Task X could use 4 pages of kernel memory"*) could restrict the kernel memory usage. Initially σ_0 or another privileged task has all kernel memory and can give parts of it away to other tasks.

The main disadvantage of quotas is that they only allow to restrict but not to manage kernel memory at user-level. The placement of new objects in kernel memory of a task cannot be controlled. Therefore the kernel cannot determine which objects should be destroyed if the quota of a thread is reduced below the used amount. Beside this, implementing cache coloring of kernel memory on top of the microkernel is not possible. Also a simple quota system does not allow to distinguish between different kernel memory pools with different priorities or replacement strategies. A server for example does not want to lose the main thread but could perhaps tolerate that worker threads or client mappings get destroyed if there is not enough kernel memory available.

In summary a quota based solution cannot handle overload situations and the controlled reduction of the used kernel memory.

The Idea

The idea behind kernel memory management in L4.sec is that ordinary user memory can be given to the kernel and can be reclaimed at anytime. L4.sec introduces therefore a new named object: the Kernel-memory object that represents memory to be used by the kernel as backing store for all dynamically created kernel objects.

Kernel-memory objects are created by *converting* a flexpage[13] from a user memory-space into a kernel page in the kernel memory-space. Afterwards the flexpage cannot be read or written from user-level anymore. The permissions are *preempted*. Threads that perform an system-call,

[12] In FIASCO up to 4 MB for $3 * 2^{18}$ mapping nodes and up to 3 MB for corresponding page-tables

[13] See the glossary A.1 for a description.

which needs kernel memory, can provide it by a capability pointing to a kernel-memory object. For example to create new objects (threads, endpoints, . . .) or to map from one namespace into another (mapping nodes and pagetables) a thread specifies a kernel-memory object in which the kernel allocates the TCB, an endpoint structure, the mapping nodes, or the page tables respectively.

A kernel-memory object is destroyed if the last capability pointing to it is `unmap`'ed or if the flexpage in the memory space of the creating task, is unmapped. Destroying a kernel-memory object *reconverts* the flexpage and deletes all objects allocated within it. The preempted permissions of the flexpage in the user memory-space are restored.

2.2.4 Summary

L4.sec promises to solve the security problems of previous L4 versions. It introduces some simple but powerful new concepts. Instead of passing the security critical functions to some privileged servers, it extends the hierarchic model of user memory management to kernel memory. In the same way IPC control is not bound to a single entity but can be distributed over the whole system. This work shows the feasibility and the consequences of the new concepts.

Chapter 3

Design

Kernel resources are handled in L4.sec quite different than in prior L4 versions. Mainly the *convert* of user memory into kernel memory and back, which is done by creating and destructing kernel-memory objects, has a deep impact on kernel design and implementation. In the following chapter I discuss the consequences of this fundamental change.

3.1 Dependencies

New kernel-memory handling is needed when creating new objects, when mapping from one space into another and when deleting kernel objects during an `unmap` system-call. The first two operations affect only a single object. The last operation - the deletion - affects a whole tree of objects, because deleting an object requires deleting all dependent objects. This requirement is obvious, since it assures that every object can be deleted and that no non-functional object remains.

There are 3 types of dependencies an object can have in L4.sec:

1. **resource dependency** - Who provides the resources with which the object is built?
2. **functional dependency** - What needs this object to work?
3. **map dependency** - Where is it mapped?

3.1.1 Examples

In the following, I give two examples for a better understanding of these dependencies.

Task Dependencies

Figure 3.1 shows the dependencies of a task object. A kernel-memory object provides the memory for the task-control block (resource dependency). The capability space (CS) and memory space (MS) are examples for functional dependencies. Without them, a task cannot exist and vice versa. These two-way dependencies are cycles in the graph.

Because a task is a named object and is therefore always mapped into a capability space, there exist also a map dependency. Here is also a cycle between the root mapping-node and the task object, because they can both only live together. This is not the case for child mapping nodes, because the dependency between them and the named object is divided into a dependency to the root mapping node and the dependency from the root to the object.

The dependency between a task and the threads, which run in them, can be called as a weak functional dependency. A thread needs a task in L4.sec to be runnable. But it is only stopped and not destroyed if the task gets deleted.

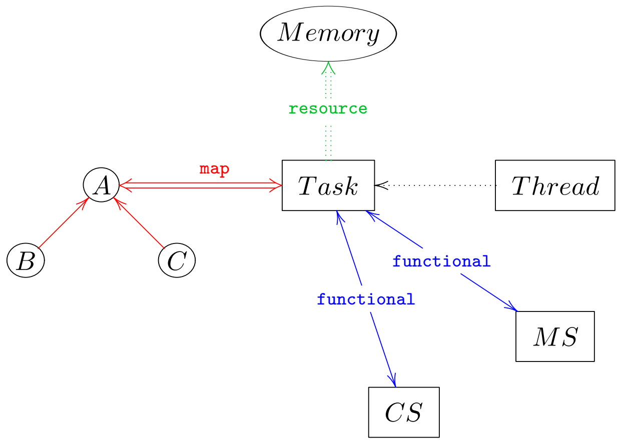

Figure 3.1: Dependencies for a task

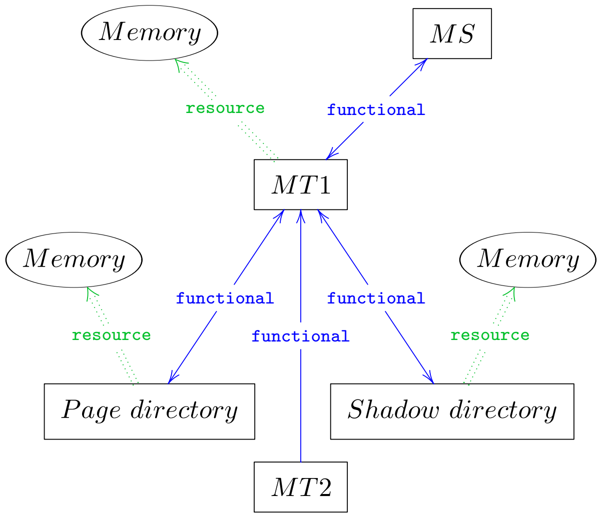

Figure 3.2: Dependencies for a first-level memory table (MT1)

Memory Table Dependency

Figure 3.2 is another example for the three mentioned dependencies. It shows the dependencies for a first-level memory table (MT1). This is an object which manages a page directory and a shadow directory.

The memory table is an unnamed object. Therefore, there is no map dependency since no mapping nodes point to it. MT1 is part of a memory space and can itself point to many second-level memory tables[1] (MT2). The former is a two-way functional dependency. The later is not, because memory spaces always have a first-level memory table but a MT1 needs not to have a MT2. Other functional dependencies point to the page directory and shadow directory. These are two-way dependencies, because both cannot exist without the memory table and the MT1 needs them too.

The resource dependencies are the kernel-memory objects in this example, which provide the memory for the MT1 itself and for the page and shadow directory.

[1]assuming at least a two-level page table here

3.1.2 Avoiding Cycles

The dependencies of all objects build a graph. Because a deletion needs to traverse the graph, it should be cycle-free. Cycles in these graph could lead to deadlocks if locking is used and additional resources of time and memory are needed to break them. Therefore I conclude:

- **The graph has to be cycle free.**

As shown in the examples, the resource dependencies have no cycles. This is because to create an object, the object resources must already be available. The functional and the map dependency can be two-way dependencies, which are in fact small cycles. How can these cycles be avoided?

Binding

There is only a weak functional dependency between a task and a thread running in them, as the task example shows. These dependencies come from the fact that L4.sec allows binding threads to a task. If this is not the case and the task is destroyed when the last thread dies, like in L4.X2, this would be a two-way functional dependency. Therefore I conclude:

- **Explicit binding avoids cycles.**

Collocation

Explicit binding is only possible with first-class objects, which have a name so that they can be individually addressed. Since an object get its name from a root mapping node, binding is not suitable to solve the cycle in a map dependency.

This map dependency has the problem that someone could do an `unmap` of the root-mapping node and another one could `unmap` the kernel-memory object in which the named object is stored. Both operations have to delete two objects: the root-mapping and the named object itself.

The cycle between the two objects can be avoided if they are only a single object. If the named object contains the root mapping-node the cycle is gone. This collocation could be implemented using class inheritance[2]. In summary I conclude:

- **Collocation avoids cycles.**

Redirection

There are cases where collocation is not possible. For example a page directory, like it is shown in the memory table example, cannot be collocated with the first-level memory table, since it needs a whole page itself.

That a MT1 needs a page directory and vice versa is a problem if the first is deleted coming from the memory space and the later coming from the kernel-memory object, which holds the page-table. Another problem with page tables is, that there is not enough space to store the pointer to the memory table.

This leads to the idea that the kernel-memory object could store the pointer to the memory table. If the kernel-memory object is deleted, it can now redirect the delete sequence directly to the memory-table. The page directory is now only deleted from the MT1. The kernel-memory object never destroys them anymore. Therefore I conclude:

- **Redirecting a dependency avoids cycles.**

[2]The named object could be a derived class of the root mapping-node.

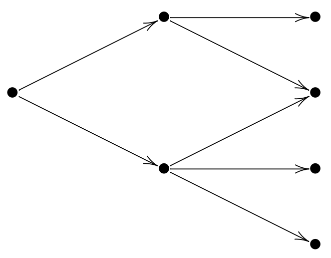

Figure 3.3: A directed dependency graph

Summary

In summary, cycles in the dependencies can be avoided by:

- explicit binding,
- collocation through class inheritance or
- redirecting dependencies.

Which of them can be chosen for a single case, depends on the involved objects. The current implementation shows that these three ways are sufficient to avoid all dependency cycles.

3.1.3 Tree Walk

The dependency graph is directed if there are no cycles. Figure 3.3 shows such a directed graph. Unmapping a single object means to start at a particular node in the graph and delete all objects which can be reached through it. If a depth-first deletion is used, this walk is the same as walking through a tree. It does not matter that an object can be reached through many parents. It is visited only once, since it is deleted and all dependencies are removed after that[3]. Therefore I conclude:

- **The delete sequence is a tree walk.**

Iterating or walking through a tree is a commonly known problem in computer science. But what are the requirements this operation must have?

Fixed Memory Usage

The kernel gets additional memory only from applications. If a task wants to unmap kernel objects, because it is out of kernel memory, it cannot provide additional memory for this operation. Therefore the deletion of objects must occupy only a fixed amount of kernel memory, independently of how many objects have to be deleted. In summary:

- **Only a fixed amount of kernel memory can be used.**

Recursive vs. Iterative

The simplest way to walk through trees is recursive. To implement a recursive walk, only the next children have to be provided, the return path is stored on the stack. Since the depth of the object trees only depends on the available memory, recursive deletion is not applicable, without the waste of having a very big stack. Therefore an iterative solution have to be used.

For the iterative tree walk a back pointer for every node is needed, like for a recursive one, otherwise the return path cannot be found. Because the back pointers cannot be stored on the stack, they have to be stored in every node itself. I conclude:

- **It must be an iterative tree walk.**

[3]Removing all dependencies is a locking problem, since the dependency tree has to be consistent at any time. Therefore this must be done atomically with respect to other graph traversing operations.

Abort and Preemptible

Unmapping a kernel-object tree is a long-running operation, since the number of objects that are deleted is only limited by the available kernel memory. For application programming it is a plus, if long running operations can be aborted. Beside this, there are cases where the kernel has to abort an `unmap`. For example, the deletion of the own thread is not allowed and it results in an abort of the operation[4].

A long-running operation can be a problem for the interrupt latency, which is a key factor for good real-time properties of a system. The `unmap` operation can therefore not be done in an atomic step. Instead, it must be possible to interrupt the operation and continue the work later. So I conclude that beside the possibility to abort an `unmap`, the operation must be preemptible.

The consequence of an abortable and preemptible `unmap` is that the object tree must be consistent at every time the operation is interrupted. Therefore the deletion cannot be executed in preorder, but it has to be done in postorder, as shown in [Voe02]. In summary:

- **The `unmap` operation must be abortable and preemptible.**

Summary

The deletion sequence has to

- use a fixed amount of kernel memory,
- must be an iterative tree walk, which is
- abortable and preemptible.

3.2 Locking

One consequence of user manageable kernel memory is, that kernel memory is no longer type-stable. The user can for example create a thread and reuse the same memory after destroying the thread to create a task or an endpoint. The kernel cannot assume with L4.sec that a particular virtual or physical page holds the same type of object anytime.

This consequence changes the way locking has to be implemented. There is no proof that fine granular locking, which is a base for good realtime properties as Fiasco shows, can be implemented in L4.sec.

First I describe why lock-free synchronization cannot be used anymore and that lookup and locking must be done atomically. After that I outline a locking scheme for L4.sec. Since this scheme is not implemented until now, this can only be a rough design.

3.2.1 Lock-Free Synchronization

In [Hoh02] Michael Hohmuth writes:

> *Preconditions for using lock-free synchronization are that primitives for atomic memory modifications are available and data is stored in type-stable memory.*

The first precondition is a hardware property and is therefore independent from L4.sec.

The second precondition does not hold with the new kernel memory management. Therefore we have to conclude that lock-free synchronization cannot be used anymore to synchronize object usage and destruction.

However if another lock assures that the object is not deleted and therefore does not change its type, lock-free synchronization could be used on top of it.

[4]See section 4.2.3

```
Thread *dst = lookup (regs->dst());

// do something

Lock_guard <Thread_lock> guard (dst->thread_lock());

// do more
```

Figure 3.4: Previous code does thread lookup and locking not atomically.

3.2.2 Atomic Lookup and Locking

Since the type of memory a pointer references can change, the usage of an object has to be locked against deletion. This means *lookup and locking must be atomically*, a property which previous Fiasco code is missing. I show this in the following for locking threads.

Thread Lookup

The lookup of a thread in previous L4 versions was simply calculating a thread pointer from the thread ID. Since thread ID's are specially designed for this case only arithmetic operations have to be executed[5].

Old Thread Locking

Figure 3.4 shows a typical usage of thread lookup and locking in Fiasco. In L4.sec the thread pointer could be invalidated until the lock is grabbed, since both operations are not done in an atomic step. Even the thread pointer is sometimes used between lookup and locking. This is only correct with the assumption that the type of the memory never changes. Because the assumption does not hold with L4.sec, this code cannot be reused.

3.2.3 New Locking Scheme

A locking scheme for L4.sec have to protect in-kernel data structures mainly named objects, mapping nodes and perhaps other unnamed objects.

Generation Numbers?

One idea to detect type changes in type-unstable memory was the usage of generation numbers. They could be used to track the creation and deletion of objects at page-table, page and object level. Due to its complexity and overhead (they have to be checked at every object usage), it seems that they are not practicable and therefore they are not on further focus. Instead the concurrent usage and deletion should be serialized through locks.

Locking of Named Objects

Named objects, like threads, tasks, kernel-memory objects or endpoints, are referenced by the user through capability ID's. These capability ID's are used to lookup the corresponding object in the capability space of its task. Fig. 3.5 shows how object lookup and locking, provided by a `get_obj` function, can be used.

Every named object has a helping lock, which is set during the lookup. Because these lookup must be protected with a global lock[6] from concurrent deletions, the lookup and locking is done atomically. This operation can be implemented as a lookup and a try-lock. If the lookup failed or

[5]Mainly *anding* a mask, *shifting* it and *adding* the offset of the TCB area.

[6]For example with the cpu-lock on uniprocessor or with a spin-lock on multiprocessor machines.

the locking was successful, the operation returns. Otherwise the thread helps the lock owner by switching to it. The lookup-lock operation is restarted if the thread is activated again.

The pseudo code in Figure 3.6 shows how `get_obj` can be implemented. A lock-guard is given into the lookup function and initialized if the lookup was successful. Only a *try_set*-function, which tries to get the lock and allows to set a guard in a called function and not only in the constructor of the guard, have to be added to the lock-guard implementation.

```
Lock_guard guard;
Thread *dst;

if (get_obj (block->dst(), &dst, &guard))
        // error
else
        // use dst
```

Figure 3.5: Named Object Locking

Locking of Unnamed Objects

Most operations on unnamed objects, like page- or capability table lookups are so fast, that a fine granular locking seems unnecessary. One exception of this is the deletion of objects. Unmapping[7] is a long running operation, which have to be synchronized with the usage of the objects.

For example the deletion of a mapping node has to wait until a `map` working on it has finished and vice versa. Locking the single mapping node and not a whole mapping tree, allows to give independent subtrees to tasks which do not trust each other[8].

The deletion of child objects like name-spaces in a task or capability tables can be synchronized either with the parent lock, if they are collocated or with an object lock to avoid that a concurrent unmap can happen from the kernel-memory object.

Helping locks should be used for named and unnamed objects locks. Global accessed data like the running queue or the timeout list should be protected with global locks like kernel or processor locks.

[7]`unmap` is the system-call to delete objects.

[8]This is the base for changing the root in a monitor scenario mentioned in section 3.3.3.

```
bool
get_obj (Address id, Object *obj, Lock_guard *obj_guard)
{
  for (;;) {
    Cpu_lock_guard guard;
    *obj = lookup(id);
    if (!obj)
      return false;
    else
        if (obj_guard->try_set(obj->lock()))
          return true;
        else
          obj->lock()->owner->switch_to();
  }
}
```

Figure 3.6: Get a Locked Object

Deletion vs. Running Operations

As said before, running operations on objects have to be finished before the objects can be deleted. This is easy if these operations hold a lock on the objects they use. This is mostly the case, but not for a running Long-IPC, where a lock to the partner is not hold while copying a message[9].

This case can be handled by introducing a less privileged usage-lock, that shows that an object is used[10]. A deleting thread, which holds the object lock, signals the abort to all "users" and use helping to ensure that all running operations are canceled before deleting. To avoid that new threads grab the usage-lock while deleting is in progress, the object lock has to be held to get them. Using these two locks is deadlock-free if the object lock is not requested while holding a less privileged usage-lock.

3.2.4 Summary

In this section I presented a rough design for a locking scheme for L4.sec. Since locking and realtime is not on the main focus of this work, these ideas are not implemented until now. Therefore a proof, whether the scheme works in any cases and has the needed properties, cannot be given.

But this approach supports the thesis that an L4.sec implementation can have the same real-time properties as Fiasco. Or in other words, that security, which comes with L4.sec, does not necessarily influence real-time.

3.3 Resource Dependencies

In previous L4 versions there exist only a map-dependency if a page was mapped from one task to another. The kernel stores the information, which map-dependencies exist, in mapping nodes. Since in L4.sec these mapping nodes are allocated in kernel-memory-objects, exists an additional dependency: the resource dependency. If the kernel-memory-object, that was used to allocate the mapping node, is deleted, the mapping node and with it the mapping is also deleted. This has the effect that not only the mapper (the source) and the mappee (the destination) can destroy a mapping but also a third party that gives the resources needed for this mapping.

Three-way dependencies need more design and implementation effort as dependencies between two parties. Thus I claim:

- **Three-way dependencies should be avoided.**

3.3.1 Map

A `map` can be seen as an operation between two tasks: the mapper and the mappee. But who has to pay for the resources needed to establish a mapping between them? To answer this question we have to distinguish the memory needed for the name-spaces (page- and capability-tables) and for the mapping nodes.

The resources of the name-space have to be allocated by the mappee, because a page-table can be used for many mappings, which can come from different mappers. Nobody wants that a untrusted server can remove trusted mappings, by revoking the kernel memory for the corresponding pagetables.

The mapping nodes are allocated by the mapper, to avoid a special case for the initial mappings into a task. The initial mappings cannot be received, if memory is needed for the mapping nodes to get them.

For the following I assume that the mapper allocates the mapping nodes, like it is defined in the L4.sec specification. If someone shows that a `copy` is really needed, this assumption have to be reconsidered.

[9]Because the copy could trigger pagefaults, which put the thread into a sleeping state until the pagefault is resolved. To avoid dead locks, sleeping threads must not hold any lock.

[10]This model is similar to a reader-writer lock scheme. The writer lock is the object lock, which can be hold only by a single thread. The usage-lock is a less privileged reader lock and can be used by many threads.

3.3.2 Copy vs. Grant

The copy of space entries was introduced in L4.sec as replacement for grant to allow flat mapping hierarchies and for the building of high-bandwidth one-way channels.

In the following I describe why grant is not used any longer and that copy has problems with the resource dependency.

Why not Using Grant?

The usage of grant was a fast way in L4.V2 to change the owner of a mapping. In fact, grant was rarely used.

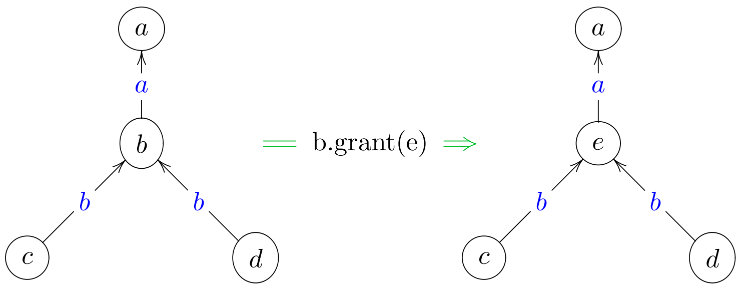

Figure 3.7: Grant an object

Granting is replaced by copy to avoid some problems using grant. For example granting a hole in a super-page is not possible since it needs resources for the page-table and mapping nodes. Another problem with grant is that it changes map dependencies[11] but not resource dependencies[12] like it is shown in figure 3.7. Granting a page from "b" to "e" changes the mapping from "b" to "e" but the resources of the child mappings are remain by "b".

Copy an Object

A copy duplicates the mapping-node for a given object (page, super-page, capability). For the mapping hierarchy it is the same as if the parent of the mapping-node calls a map. But copy has the same problem with resource dependencies like grant. The resources for the needed mapping nodes are given by the caller of the system-call. Figure 3.8 shows that for a mapping from "a" to "e" the caller of the copy "b" has given the resources.

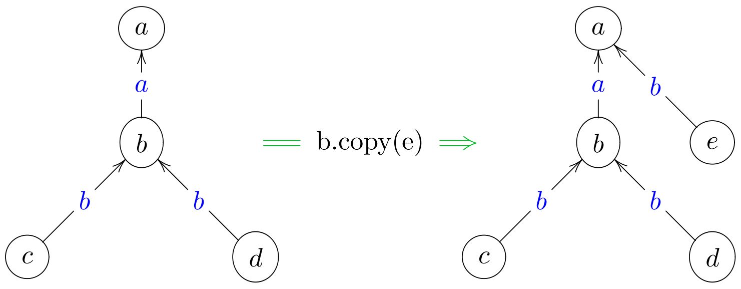

Figure 3.8: Copy an object

The copy of a root node has to be forbidden because it can lead to cycles in the dependency-tree[13]. This happens for example if the creating task moves, through a copy and unmap, the root node of a child task into the child.

[11]Who has mapped the page?
[12]Who gives the resources for the mapping-nodes?
[13]In fact the tree degenerate to a directed graph.

Summary

The new introduced `copy` has problems with resource dependencies. Either the decision that the mapper has to allocate the mapping nodes has to be changed, or `copy` has to be discarded. In the following I look at the second option.

3.3.3 Copy Obsolete?

Is `copy` really needed or is it obsolete and should be marked as deprecated? To answer this question we have to consider the use cases of `copy`:

- flat mapping hierarchies
- limited memory space
- high bandwidth one-way channels

Flat Mapping Hierarchies

The `copy`-function can be used to build flat mapping hierarchies. A pager[14], that does not want to `unmap` the mappings later, copies the mapping to the destination and `unmap` them only in the own address space.

The advantages of flat mapping hierarchies are obvious. The `copy` needs less resources than a full `map` and the corresponding `unmap` is faster because of a smaller mapping tree.

The whole `copy` functionality can be build in user-level if the pager of the thread does the `map` for them. The pager is trustworthy and can easily provide an interface to map to another space. So a `copy` is not really needed for flat mapping hierarchies.

Limited Memory Space

The size of the memory space is limited by the hardware. If a pager does not want to unmap the pages from a client it can `copy` them to the client and unmap them in the own address space. This avoids the pollution of the memory space of the pager.

Instead of `copy` the pager can use another memory space of a cooperating task to extend the number of map-able pages. Furthermore it is possible to build bridges between capability space and memory space. If pages can be mapped into a capability space and after that mapped back into another memory space, the limitation of the memory space is gone.

One-way Channels

To build security systems, one-way communication channels[15] are needed.

Suppose the following scenario: A one-way channel should be established between two tasks: from X to Y. Simply using an Endpoint between X and Y is not possible, because Endpoints allow synchronous IPC. The information whether Y is ready to receive can be used to back-channel information to X. To decouple the send and receive phase of the message transfer, a monitor M is inserted between the two partners. So M intercepts every message sent from X to Y (see figure 3.9).
The monitor provides asynchronous communication between X and Y. To handle overload situations, the message transfer is unreliable: messages could be lost.

[14]A task that provides the memory for another one.

[15]The information flow is unidirectional. The Bell-LaPadula model uses therefor the *-Property rule (also known as the Confinement Property.), which forces that information cannot flow from higher security levels to lower one.

$$\begin{tikzcd}
X \arrow[dr, Rightarrow, "ipc"] \arrow[dd, leftrightarrow, "pager"] & & Y \arrow[dd, leftrightarrow, "pager"] \\
& M \arrow[ur, Rightarrow, "ipc"] & \\
P_X & & P_Y
\end{tikzcd}$$

Figure 3.9: One-way channels using a monitor

High Bandwidth

To provide a high bandwidth one-way channel, a shared memory page between X and Y can be used. To establish this channel, X maps a page to M which maps the page read-only to Y. After this, X can write to the page and Y can only read from it.

If the monitor does not want to unmap the page, for example on policy changes, the second `map` is not needed. Instead, a `copy` can be used. The `copy` avoids the pollution of the memory space of M because M can unmap the first mapping. It cannot avoid the problem that M needs kernel-memory for the mapping nodes, to do the `map` or `copy` to Y. The requests and the used kernel-memory provide a back-channel, how many pages Y is receiving. Therefore, the kernel-memory cannot be requested from X or the Pager of X (P_X) and has to come from Y or the Pager of Y (P_Y).

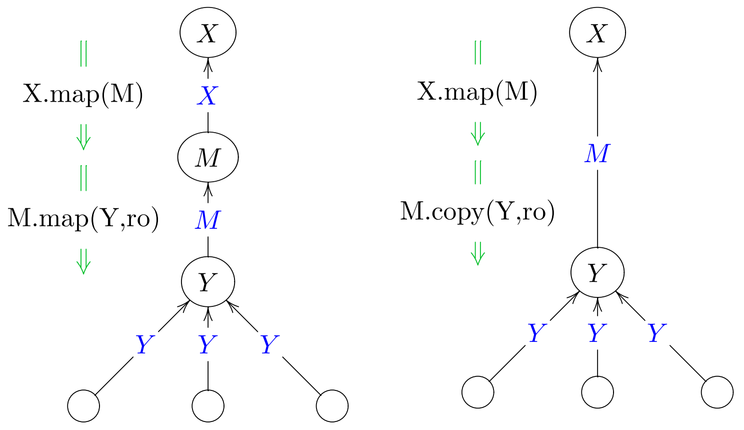

Figure 3.10: Mapping trees for high bandwidth one-way channels

Timing of `unmap`

The main problem of using shared memory as one-way channel is the timing of the `unmap` system-call. The time `unmap` needs depends on the number of child mappings in the mapping tree. If Y can influence this number, it could back-channel information to X. Y can influence the number in 4 ways:

- map the received page
- unmap the page

- accept a smaller flexpage that leads to a split of a super-page
- accept or not accept the page at receiving

To use a shared page anyway the monitor have to forbid the map and unmap of the page through Y and the mapping of super-pages through X. It is unclear how the accept-problem could be avoided.

Change the Root

The timing problem of `unmap` results from the map-dependency between X and Y. If the root of the mapping is not X but M the two X and Y can get independent branches of the mapping tree.

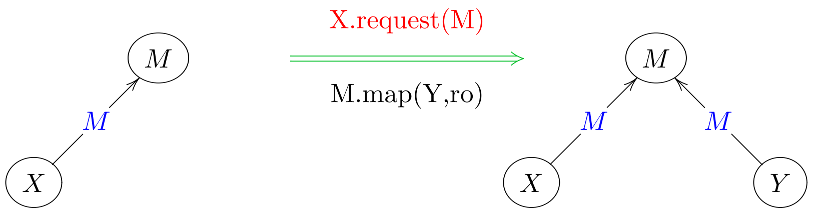

Figure 3.11: Change the root

Like shown in Fig. 3.11 X asks the monitor M to map a page from X read-only to Y. If X want to revoke the shared page from Y it asks M to do a unmap of it. With the changed root of the mapping tree a `copy` is unnecessary.

Conclusion

A `copy` is not needed for flat mapping hierarchies. The limited memory space, which `copy` improves, is not really a problem. High bandwidth one-way channels can be build without `copy`. Due to timing problems it is advised not to building them with `copy` anyway. Instead it can be build solely with `map` if the root of the mapping is changed.

The `copy` system-call can be discarded, because there is no scenario known that could only be build with it.

3.3.4 Summary

Resource dependencies are uncritical with `map`, independent whether the mapper or the mappee gives the resources for the mapping nodes. With the decision that the mapper must give the memory, resource dependencies are a problem with `copy`, since a third party is involved. A closer look at `copy` showed that it is not needed and could be removed from the specification. Without it the problem of resource dependencies is resolved.

3.4 Unpinned UTCB Memory

UTCB's are for some strange reasons API objects in X.2.

Uwe Dannowski

Up to now the kernel can pin pages of kernel-memory also if they are visible in user memory-spaces. Because user-level can revoke kernel-memory in L4.sec, a simple pinning is not possible anymore.

3.4.1 What are UTCB's?

UTCB's[16] are pinned kernel memory in the memory-space of a task. They are used for parameter transfer between kernel and user-level and have the advantage that no pagefault can happen by accessing them.

A fundamental disadvantage of UTCB's is that they restrict the memory layout of a task. Since UTCB's are a must for every thread in L4.X2, it is impossible that a user could use the whole address-space.

There are two possibilities to go with unpinned UTCB's. Either all operations using them have to be aborted or UTCB's are not used at all.

3.4.2 Implement UTCB's

How can UTCB's be integrated into L4.sec? Now I mention some additional disadvantages of UTCB'sin L4.X2. After that I present a rough design for UTCB's in L4.sec.

Disadvantages of UTCB's in L4.X2

UTCB's in L4.X2 have some other disadvantages:

- cannot be moved or unmapped
- limit the number of threads in the task
- resources must be given at task creation

These things are specific to L4.X2 and not inherent to UTCB's.

UTCB's in L4.sec

A new named kernel object, called UTCB object, could be implemented. Such an object manages a page of kernel-memory which is writable to the user. A thread could be bound to an UTCB object and not to a task anymore[17]. Threads are only runnable if they have a valid UTCB object, which must be attached to a task. If a UTCB object is destroyed, unbound from a thread or detached from a task, all threads using them are stopped and all running operations are aborted.

It seems that these rough design could be build in L4.sec. It avoids the disadvantages of the L4.X2 implementation, since UTCB's could be attached to and detached from a task and threads could be freely bound to them.

Due to the fact that the implementation of UTCB's still needs a lot of code, the second option, to waiving of UTCB's at all, is considered in the following.

3.4.3 Do we Need UTCB's?

Motivation

Using UTCB's needs a lot of kernel code and has the inherent problem that it pollutes the memory address-space of the tasks. So what is lost in a system without them? To answer this question have a look at the three reason why UTCB's are used:

- LIPC
- simpler kernel code
- optimizations

The abstraction of virtual registers and the possibility to map them to hardware registers are not bound to UTCB's. As W5 (see section 4.3) show, they can be used with other virtual registers as well.

[16]The user-level part of a thread control block.

[17]To prevent that a thread is bound to a UTCB object that is not in its task.

LIPC - Local IPC

UTCB's were introduced to build fast intra-task IPC in L4, known as LIPC, which is based on the idea of lazy process switching [LW01]. Due to different reasons, LIPC is not allowed for several cases. For example finite timeouts, receive- and send-only LIPC are not possible and normal IPC must be used instead.

The two available LIPC implementations for L4 show that LIPC decrease normal IPC performance by introducing an overhead of 14-30% [Wen02]. It is only faster if more than 5 LIPC are done between two normal IPC calls [Reu04]. This overhead results mainly from the design of LIPC. Since the kernel state is synchronized lazily, it must be checked at *every* Kernel entry whether this must be done or not. Additionally the synchronization is not as cheap as it was expected.

Endpoints can be used in L4.sec to send messages directly to worker threads, so LIPC will not be used for simple local message dispatching. Furthermore migrate-able user-level threads [Cla05] cannot use LIPC, but need another local communication mechanism.

In summary, the further development of L4.sec will not focus on LIPC.

Simpler Kernel Code

In principle there are two ways to handle pagefaults which could occur if the kernel accesses user-memory. First, they can be avoided by using pinned kernel memory, like UTCB's. And second, a transparent pagefault IPC can be send to the pager. Normally sending a pagefault IPC does not increase the kernel code complexity. But there are 2 exceptions: pagefaults in a Long-IPC[18] and *delayed preemption's*.

The implementation of *delayed preemption's* [DLSU04, Hoh02] is easy with UTCB's. *Delayed preemption's* requires two shared bits which are read and written in the timer-interrupt handler from the kernel and at start and end of a critical section from user-level. Since no pagefaults are triggered by using UTCB's the interrupt handler has not to take care of them. Without the possibility to store the two shared bits in pinned memory, the timer-interrupt handler need to know if a pagefault happened, which requires a special case in the pagefault handler. If such a pagefault occurred, the timer interrupt could handle this as a fault of the task and send a time-slice overrun message. In summary it seems to be that *delayed preemption* and likely other cases could be build without pinned kernel memory, but with an increase of the kernel code complexity.

UTCB's itself require a lot of code, that could be saved if they are not used. The implementation of a special case in the pagefault handler[19] seems to be less complicated than the introduction of a UTCB-object and of system-calls to manage them.

So kernel code seems to be simpler if UTCB's are omitted and the special cases are implemented.

Optimizations

UTCB's are also used as an optimization. For example system-call parameters are transfered through UTCB's, without the normally needed copy-in/out of the parameters.

Removing UTCB's could result in a performance bottleneck of the now needed copy-in/out of system-call parameters. To avoid this a kernel shared-memory object can be introduced. This is a kernel-memory object that can be mapped into the memory-space of many tasks. In contrast to UTCB's these objects could be optional, independent of threads, mappable and an unlimited number could be used in a task. In summary they are solely used as optimization.

Summary

As shown above LIPC is not part of L4.sec. The kernel code complexity decrease without UTCB's. For optimization purposes they could be replaced by a kernel shared-memory object. In summary a system without UTCB's can be build.

[18] A IPC that transfers strings between two address spaces. This is independent from UTCB's.

[19] Sending a time-slice overrun message must be implemented independently of UTCB's.

3.4.4 Conclusion

Since UTCB's give no benefit anymore and the kernel should not depend on optimizations, I propose that *UTCB's should be considered experimental in the L4.sec specification.* If this results in a performance bottleneck, the above described *kernel shared-memory objects should be introduced.*

3.5 Summary

In this chapter I discussed the consequences of the new kernel-memory management in L4.sec to the implementation of the kernel. At the beginning I discussed the effects of dependencies to the deletion process of kernel objects. After that I presented a first rough design for a locking scheme for L4.sec. By taking a closer look at the resource dependencies I found that `copy` is unneeded. At the end I described why UTCB's are not a must anymore.

Chapter 4

Implementation

Defining the design of the new implementation was not done in a whole at the beginning, since nobody had a full survey of the consequences resulting from the changes of L4.sec. Instead many small cycles of design, implementation and test were used.

First I describe some implementation details. After that I close some holes of the specification. At the end I present a new system-call binding for L4.sec.

4.1 Implementation State

4.1.1 Changes

The new concepts of L4.sec cause that new code has to be implemented. Additionally different changes of L4.sec prevent to reuse large parts of old FIASCO code. The main changes are:

The memory layout is modified.
A TCB area, where the thread control blocks can be easily found by their global name, is not useful anymore. A one-to-one mapping of physical pages is not used any longer. Since this limits physical memory that can be used for kernel memory[1]. Instead a region is needed, where converted pages can be made visible to the kernel.

A task is more than a page directory.
Previous L4 implementations use only the page directory as task object. Task local values are stored in invalid pagetable entries. Since a task object in L4.sec is a derived class where the storage of values cannot be controlled, this optimization can not be used anymore.

Thread locking and the system-calls differ.
As written in section 3.2, threads in L4.sec cannot be locked by writing to a memory location. Additional system-calls have to be rewritten, as the creation of threads and tasks, or IPC is handled in a different way than before.

Mapping database requirements changed.
Beside the fact that the current mapping database does not support the capability and IO space, single mapping nodes have to be allocated in task local memory. Storing a mapping tree in an array is no longer possible. Thus the mapping database has to be rewritten.

In the following I look at the mapping database implementation. After that I describe the kernel-memory object and other implementation details.

[1]The whole physical memory cannot be mapped within such a region.

4.1.2 MDB: The Mapping Database

What is a MDB?

The mapping database records map-dependencies to be able to execute the recursive `unmap`. The dependency, from where an object was mapped to another space, is stored in so-called mapping nodes.

The MDB was used before only for memory pages and IO-ports. Its usage is extended in L4.sec to capabilities as well.

While implementing the MDB and the `map` and `unmap` operations the following questions had to be answered:

1. Can we reuse the current implementation?
2. How to support different spaces and object types?
3. Can we handle that a thread want to unmap itself?

Current Implementation: Inapplicable

The current Fiasco mapping database [Gru98] is optimized for a minimal cache and TLB footprint. Only 5 bytes (or 8 bytes for the L4.X2 implementation [Cla04]) are needed for one mapping node. The mapping nodes of a single mapping-tree are stored in an dynamically allocated array of kernel memory. Because mapping nodes are unnamed objects in L4.sec and cannot be allocated in an array but must be allocated separately in different kernel memory objects[2], the current implementation cannot be used.

Different Spaces and Object Types

The `map` and `unmap` operations work on the MDB and on different name-spaces. To simplify the implementation a common interface was introduced into L4.sec. This *Space_Xe* interface hides the internal structure of the space, for example the page-table in a memory space. It promise an easier porting of the space implementation to other architectures.

The current space implementation uses class inheritance and virtual functions to implement the *Space_Xe* interface. As an optimization a template could be used. The template avoid the indirect calls of the space interface at the cost of increasing the code size.

Unmap operates, beside on the name-spaces, on different object types. It has to delete for example threads, mapping nodes, page-tables and kernel-memory objects. A large *switch* to distinguish all cases in the `unmap` code is large and not easy to manage. Instead of this error prune and bad style of type dependent code, all objects could share a common interface called *Kpage_allocable*. This interface is given in Fig. 4.1.

Using virtual functions for this interface simplifies the implementation. Compared to this should the overhead for storing the VMT (virtual method table) in every object and the overhead of the indirect call be negligible.

Using these two interfaces distributes the code for the `map` and `unmap` operation where the special knowledge is: at the spaces, objects and the mapping database.

Corner Case: Self-Unmap

One corner case was to handle the self-unmap of a thread. As defined in Section 4.2.3 the unmap has to be aborted and the thread has to be stopped in this case.

The deletion of the current thread is detected in the *predelete*-method of the thread. This information is propagated to the calling `unmap`-loop through the return value of the method, the loop is aborted and the locked nodes are unlocked. If the `unmap` returns such an error, the thread is stopped.

[2]to be able to destroy them if a kernel page is reconverted

```
class Kpage_allocable
{
public:
  virtual size_t size();
  virtual Kpage_allocable * next_subobject (Kpage_allocable *);
  virtual void delete_subobject (Kpage_allocable *);
  virtual bool predelete();
}
```

Figure 4.1: The Kpage_allocable interface is implemented by every object in L4.sec.

Summary

The old mapping database implementation could not be reused. The new implementation use clean interfaces for the different objects and name-spaces, which decreases the implementation effort a lot. The self-unmap problem is solved. As discussed in Section 5.2.3, the new implementation is of similar complexity then the old implementation.

4.1.3 Kernel-Memory Objects

A kernel-memory object is created in the memory space when converting a page. The destruction of them reconverts this page. Because both operations heavily depend on the memory space, they are implemented in the context of the memory space class. In the following I mention three issues according to this implementation.

Convert

At the first view it seems to be that only the system-bit in the page-table entry has to be set for converted pages to preempt the permissions. If it is set, the page is only accessible from kernel-mode and user-mode access results in a pagefault. A closer look shows that this leads to a security problem[3] if memory transfers are done in the kernel, like it happens on string messages. Memory references given to the kernel from user-level are usually, for performance reasons, only checked against the memory layout, but not on a single page level. Therefore page-table entries for converted pages have to be marked as *invalid*.

This invalidating of page-table entries, which are converted, have to be done for the whole mapping tree. Every superpage that contains the converted page and every splitting with the same physical address as the converted page, has to be made inaccessible. To track, which page is converted, a bit-field for every physical page is used. Splittings out of an inaccessible superpage use this bit-field to decide, whether they could be accessible or not.

Reconvert

A reconvert restores the preempted permissions. Therefore it has also to iterate over the whole mapping tree. To decide whether a superpage contains no converted page anymore and can therefore be made accessible, the number of converted pages is counted for every superpage. This optimization avoids to check 1024 bits in the bit-field.

Convertible Superpages

Superpages need not to be convertible because someone could split them by mapping smaller parts to another location. This is not true for sigma0. There is no place in the memory space where it could map smaller pages. Giving initial some smaller pages to sigma0 is not an option here, since

[3]The kernel memory can be read and written through user-level.

it limits the memory layout of the whole system[4] and the design of sigma0[5]. Thus superpages have to be convertible.

In contrast to a converted page, superpages can have more than one kernel-memory object using them. It must be ensured that all these objects can be reached while unmapping the superpage. Using a doubly linked list[6] should be sufficient. The list of the kmem mapping nodes can be reused for this, if we take care to update the object pointer of the superpage mapping node at removal.

Summary

The implementation of the kernel-memory object was straight forward. The only surprise was that a time intensive iteration over the whole mapping tree is needed. Creating a kernel-memory object is therefore not as fast as the creation of other objects or the mapping of pages and capabilities.

4.1.4 IPC

Implementing a full featured L4 IPC is a complex part of an L4 system, because it integrates synchronization, message transfer and mapping into a single system-call. To provide basic functionality to user-level and to check the mapping database implementation, I implemented only Short-IPC.

This Short-IPC implementation for L4.sec allows to transfer single words and map items. It is possible to abort a waiting sender or receiver before a rendezvous happens through `ex_regs`. The IPC timeout is currently ignored. Short-IPC's can be stacked, which is used to send the kernel generated pagefault IPC's in a system-call.

Splitting a thread into a sender and receiver part, like in Fiasco, is not implemented. This splitting was used to send preemption and activation IPC independent from a normal IPC, but it is not needed until now.

4.1.5 Testing

The implementation is more than a prototype or an experiment. It is the base for a new working kernel. This implies the need for a higher code quality. This was achieved by using 2 types of automatic tests: unit-tests and kernel-tests.

Unit-test

Unit-tests can check whether small units of code, like classes, work as expected. Fiasco have a unit-test framework which compile tests together with corresponding kernel classes and compare the output of the running test to a stored one. These framework was used up to now only to test the mapping database. This work uses unit-tests to check the mapping database, spaces, allocators and for simple system-call tests.

Kernel-test

This work introduce kernel-tests into Fiasco. Based on the unit-test framework, these tests execute at L4 user-level. They can be used to check whole kernel functionality and not only some small parts of the code.

Testing code, like IPC, where threads execute in parallel, is not easy. To test the behavior of the kernel in these cases a user-level scheduler is used. It is running at a high priority and use *switch_to* to select the next running thread according to a schedule list. This serialize the threads in a given order and simplifies the tests.

[4]These pages cannot be used as superpages or for other task
[5]It can use only this memory, not more
[6]they must be able to remove itself

Summary

Testing was a base to increase the code quality. Many implementation bugs were found, because a test does not give the expected output. Testing should be a must in further kernel development.

4.1.6 Summary

The implementation of L4.sec is incomplete yet. This result of the work was expected, due to the limited time-frame. But what is the current state of the implementation? Or what is working?

The implementation allows to:

- send a message through an endpoint,
- page a task,
- create kernel-memory objects by converting a page,
- create tasks, threads, endpoints and additional kernel-memory objects within a kernel-memory object,
- mapping capabilities and
- unmapping and destructing of kernel objects.

Since this is working at user-level, the current implementation is ready to run simple applications[7].

4.2 Simplify L4.sec

Make everything as simple as possible, but not simpler.

Albert Einstein

The current L4.sec specification is in most cases an extension to previous specifications and quite incomplete. Closing the holes and simplifying some parts makes the implementation in many cases easier.

4.2.1 The Initial Task σ_0

In the following I describe the changed behavior and the initial resources of σ_0.

Root-Task

Must the kernel start the root-task, like this was done before? Cannot σ_0 start it? The code to start the root-task is nearly the same whether it is started by the kernel or not. The advantage of starting it by σ_0 is that functionality which can be performed in user-level is moved out of the kernel. Beside this, it supports systems that want to use only one task or that want to have multiple root-tasks[8].

Therefore σ_0 start the root-task although this mix the paging job of σ_0 with a startup job.

[7]Complex applications need perhaps Long-IPC, IRQ's or other missing features.

[8]e.g. implementing a static resource splitting at σ_0

Initial Resources

Which resources need σ_0 initially to start a whole system?

It is indisputable that σ_0 gets the whole physical memory from the kernel like this was done before. To support memory-mapped IO[9] of physical devices σ_0 can now map the whole 4 GB of the memory space to any other task[10]. This removes the 1 GB shift in the σ_0-protocol implemented in Fiasco.

The number of initial objects in the capability space of σ_0 can be minimized. If σ_0 has a kernel-memory-object it can create additional kernel-memory objects. These objects can be used to create for example threads, tasks and endpoints. So the σ_0 capability-space needs initially only 3 entries: a kernel-memory-object to create other objects, the own task to bind threads to it and the own thread to call an `ex_regs` or a `thread_ctrl`. In the future these number may be extended by interrupts and scheduling reservations.

Changed σ_0 Behavior

The initial task σ_0 can give out memory, that is not used by the kernel or itself, more than once. The previous limitation is not needed since initially only a task started by σ_0 can send and receive from it ([PV05] section 12.6).

The kernel-σ_0-protocol is not needed anymore since the kernel will never ask σ_0 for memory.

Summary

From the kernel point-of-view σ_0 is an ordinary task, with the exception that it can map the whole 4 GB memory-address-space to other tasks. The initial resources of σ_0 that must be created by the kernel are minimal. The kernel need not to know anything about other tasks, like a root-task or σ_1[11].

4.2.2 The Kernel-Interface Page

The Kernel-Interface Page (KIP) contains API and kernel-version data, system descriptors and the clock. The KIP is part of every memory address space in L4.X2. Due to some disadvantages the kernel should not enforce the mapping of the KIP in every memory address space.

System-Call Links

In L4.X2 also system-call links reside in the KIP. The KIP system-call links point to the corresponding kernel-entry code in the KIP. This allows to exchange the kernel-entry code, without the need to recompile applications. Using KIP system-calls are enforced in L4.X2 by making the KIP as micro-kernel object visible in every task. The location where the KIP is mapped must be specified on task creation and cannot be changed during lifetime of the task.

Disadvantages

Enforcing the usages of the KIP in every task through the micro-kernel is not a good choice. It prohibits virtualizing of system-calls and of the clock through user-level. It needs additional resources at task creation and limits the memory layout with an additional page and a corresponding page-table[12]. Beside this, it increases the kernel code size and the policy is implemented in the kernel instead of user-level.

[9] aka adapter memory
[10] implementable with an additional mapping memory-space for sigma0
[11] A task that allocates TCB's to implement persistence [HE03]. Never implemented.
[12] Which prohibits the usage of a superpage in this region.

Summary

In summary L4 should not enforce but could encourage the usage of KIP system-calls. Therefore the KIP contains KIP system-calls. But it is a normal page, must not be handled in a special way and can be mapped to any location in any task. Every pager could offer a special protocol to provide the KIP or simply use an agreement with the clients where the KIP should be mapped. This reduces kernel code complexity since the kernel must not map the KIP and offer a special system-call for this.

4.2.3 Double-faults

There are cases where the kernel cannot recover from a situation a user thread is responsible for. These cases are called double-faults in the following. The specification cannot avoid that such double-faults happen, so the behavior of the kernel needs to be defined in these situations.

The Cases

There are 3 cases known where a double-fault can happen:

- a pagefault occurred and no pagefault endpoint is available
- an exception is thrown and no exception handler is registered
- the thread tries to unmap itself.

If there are more cases the kernel cannot recover from, they should be handled in the same way.

Stop the Thread

Previous L4 specifications do not define what should happen if the pager of a thread does not exist. Notify another thread is impossible since it can also be not existent. Looping in the kernel until a double-fault is solved is not a choice. This exhaust cpu-time and forces a thread that want to `ex_regs` the faulter to a higher priority. L4.X2 defines that a thread is stopped if an exception occurred but no exception handler is installed. It can be resumed with `ex_regs` later. Killing the faulting thread is also possible but it prevents that another thread can inspect the faulting one.

Stopping the thread seems to be the best choice. So all double-faults stop the faulting thread, which can be inspected and resumed with `ex_regs`.

The instruction pointer points to the next instruction with the exception of the no-pager case where it points to the faulting instruction[13]. If a double-fault happen that aborts a system-call, for example a self-unmap, the error-code is set to Cancel and the other output parameters are unspecified[14].

Extension

It is possible that someone want to be notified if a double-fault happen. Therefore an endpoint to a double-fault handler could be introduced. Because this endpoint should always be available it is collocated with the task and mapped to the task creator. Until now it is unclear whether this experimental feature is really needed.

4.2.4 A Unified Flexpage

An additional name-space, the capability space, is introduced with L4.sec. This leads to the question, whether this name-space can be integrated in the flexpage-format, like it was done with the IO name-space before.

In the following I describe an flexpage-format that unifies 4 name-spaces in one encoding.

[13]to be able to easily restart after setting a pager
[14]if the stopped thread is resumed

Motivation

Flexpages describe an aligned region of a name-space, starting at a base b with a size of 2^s and include permissions[15] p for this region. They are used e.g. as parameters for `map` and `unmap` and for regions of resources in `create`.

With an unified format for all name-spaces (memory, IO, and capability space), no extra parameters are needed to decide which name-space is used. Furthermore it allows to share common code in the kernel and user-level.

Requirements

The new flexpage format should not miss any feature of the format described by the L4.X2 specification. These are:

- 32 bits wide
- 1024-byte pages[16]
- 3 bits of permissions
- useable with 64-bit architectures

Beside these some new properties are needed:

- large space for a capability space
- additional permission bit[17]

Idea

The missing bit of the permissions is obtained simply because L4.X2 reserved the fourth bit of a flexpage.

Encoding a large capability name-space into a flexpage is not easily possible. Using a special size, like it was done with the IO-flexpage in L4.X2 is not useful here, because the 22 bits of the base are lead only to 2^{17} entries[18].

The base of a flexpage is aligned to the size. This leads to the idea that the lower bits of the base can be used on larger sizes for encoding the flexpage-type.

Format

The two lower bits of the base are encoded as type t of the flexpage if the size is at least 12. Otherwise the type is assumed to be 0.

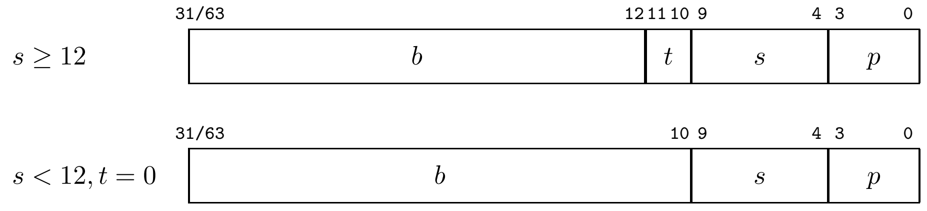

Figure 4.2: The new unified flexpage format

[15]In fact they are used as mask to limit the permissions by `map` and `unmap`.
[16]present e.g on ARM or Itanium
[17]memory-pages are convertible
[18]with a sub-size s' of 5 bits

This gives 2^{22} entries for the memory-space ($t = 0$) and 3 name-spaces with 2^{20} entries at 32-bit architectures. At 64-bit architectures the additional 32-bit are added to the base.

The capability space is encoded with $t = 1$. Over one million entries should be enough for it. The IO space is encoded with $t = 2$ and is on x86 with the 2^{16} IO-ports not fully used. The last type $t = 3$ is reserved and could be used in the future e.g. for an independent endpoint name-space.

The *nil*-flexpage, describing an empty flexpage, is encoded type independently as 0.

The *complete*-flexpage over a whole name-space, is encoded with *s==32* at 32-bit architectures and *s==1* at 64-bit architectures. This asymmetry is only an optimization for 32-bit architectures.

Conclusion

This new flexpage format fulfill the requirements of L4.sec. Beside this it has a simpler encoding for the IO name-space and it is extensible with a third name-space with 2^{20} entries. Therefore it is used in this work and should be integrated in the L4.sec specification.

4.2.5 Summary

The only task, which must be handled by the kernel in a special way, is σ_0. The KIP can be an ordinary page and the different name-spaces can be encoded in a single flexpage.

4.3 W5 - A Binding for L4.sec

If a system-call is slow, nobody will use it.

Andrew S. Tanenbaum

4.3.1 Motivation

The current L4.sec specification defines a binding based on the old L4.X2 specification. It merges some changes resulting from new requirements of endpoints and capability spaces. Because the specification was work in progress, there was no time to look at the user-level interface, the system-calls, in a global way.

In the following I give a binding for L4.sec that mainly uses 5 words[19] as parameters to the system-calls. It incorporates new definitions[20] and experiences[21] to get an optimized binding.

Why Another Binding?

To execute user-level code, which are mainly kernel tests in this work, a system-call binding is needed. Since this work and other contributions causes many changes of the specification[22], the old binding cannot be used any longer. Either the old binding is changed fundamentally or a new binding have to be defined. I decided to take the last option.

Why W5?

Transferring 5 words in registers between user-level and kernel is the least common dominator of all known kernel entry methods for x86. Since x86 is one of the hardware architectures with very few hardware registers, this can be generalized: 5 words can be transfered in the fastest way on every machine. Therefore this binding assumes that *at least 5 words can be transfered in registers.*

[19]This results in the name of this binding: W5.

[20]e.g. flexpage format and sizes

[21]design and implementation of memory management, map, unmap

[22]This work contributes unified flexpages, no UTCB, no SpaceCtrl and that the KIP is mapable. Others found that 2 capabilities for pager endpoints are needed, and that a receive window for every space is useful.

Interleaves

The format of the KIP, Map-items and the different protocols are out of the scope of this work. They could be easily adopted to W5 in the future.

4.3.2 Requirements

Before defining a new binding it should be clear what properties are required:

Efficiency
Calling the kernel through a binding is overhead. This overhead should be minimized.

Portability
Porting kernel and user-level to a new architecture should be easy. Especially it must be ready for 64 bit.

Simplicity and Extensibility
Only a few concepts should be used to build the binding. This helps to implement the binding and to add new system-calls.

Since no user-level software exist, backward compatibility is not required. This does not mean that nothing can be reused or that every part has to be reinvented.

4.3.3 Virtual Registers

The L4.V2 specification was initially developed solely for x86, which make ports difficult. Therefore introduce L4.X2 virtual register as base for platform independence. These virtual registers, namely Buffer Register (BR), Message Register (MR) and Thread Control Register (TCR), are primarily used for message transfer. The system-call parameters are not defined in L4.X2 as virtual registers. Thus the mapping from them to hardware registers have to be defined for every system-call.

W5 avoid this by using 2 types of virtual registers: Parameter Words (W) and Thread Control Register (TCR).

Parameter Words

Parameter words are an abstraction of input and output parameters of a system-call. They are named $W0, W1, ..., Wn$. 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. Such a mapping using sysenter/sysexit as kernel entry method is given in section A.2.2.

Thread-Control-Register

Thread Control Register 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.

A mapping from TCR number to name is given in the appendix section A.2.3.

Porting

The usage of Virtual Registers makes the porting of the binding easy:

32-bit architectures
Porting the binding to other 32 bit architectures, like ARM, should be easy. Only the mapping from parameter words to hardware registers and memory locations for a kernel entry method must be defined.

64-bit architectures

Registers, pointers and flexpages are 64 bit. Extending capability ID's from 20 to 52 bit simplifies the porting. This is not required, since a million capability entries should be enough. If the additional 32 bits in a parameter word are unused, they could be ignored.

4.3.4 Example: Short-IPC

In L4.V2 and L4.X0 register only IPC, named Short-IPC, can be used. The main advantage of Short-IPC is that it does not transfer strings. Therefore no pagefaults can occur on message transfer, which speed up and simplifies the IPC code.

With W5 a binding for Short-IPC can be given that transfers at least 3 words in registers.

System-Call Multiplexing

The *sysenter* instruction, in contrast to *int/iret*, does not distinguish between different system-calls. Because the fastest kernel entry method should be used for every system-call, the calls have to be multiplexed. Therefore the 4 lower bits of the first parameter word $W0$ are always used as system-call number in the input. These bits are reused as base error number in the output.

Using 4 bits in the parameters for the system-call number and adding a conditional branch in the critical path[23] seems to be low-cost, compared with the speedup of every non-IPC system-call and the simplified code.

A mapping from system-call number to the name of the system-call is given in the appendix section A.2.3.

Timeouts

In the past a send and receive timeout was given for every IPC. Mostly the two combinations $(never, never)$ for a client call and $(zero, never)$ for a server reply are used. Other combinations or finite timeouts are rarely[24]. In fact a thread has a small working set of timeouts. This lead to the idea to save system-call parameters by referencing timeouts indirectly.

Instead of giving the whole 32 bit timeout value with every IPC, the timeouts are stored in thread control registers. Only an offset of 2 bits into these registers is transfered. The first two values of this timeout number (0 and 1) stand for $(never, never)$ and $(zero, never)$. The other 2 values[25] can be freely managed by user-level. Reading and writing them can be done through *thread_control*.

The Binding

The W5 binding for Short-IPC is shown in the Appendix A.2.1. It consists of a header, the first two parameter words which describe the message, and a body, the remaining parameter words, which is the message itself. All parameters of this definition reside in parameter words, which simplify the porting of the binding.

A Short-IPC can transfer up to 63 words and 3 typed items. Because it must not trigger pagefaults, string items cannot be transfered.

The sender ID consists of the badge, which is stored in the capability of the send endpoint, and a part that could normally be given with every IPC. The current binding do not allow that the last part can be directly given, instead it is read from a TCR. If the sender ID is used frequently in the future, this decision should be revised.

[23] In current L4 versions only the IPC path uses *sysenter*.

[24] A zero receive timeout is used for IRQ attaching and finite timeouts mostly for sleep and in hardware drivers.

[25] Previous versions of W5 use 3 bits and have therefore 6 timeouts for user-level. It seemed that this number was to large.

Summary

With an overhead of 2 words for the header, this Short-IPC binding is the first one I know, that can transfer, also with *sysenter*, at least 3 words solely in hardware registers. This can be the base for fast interprocess communication in L4.sec. The low overhead mainly results from bit packing and from small source and destination ID's[26]. Beside this it is defined platform independently with the help of parameter words.

4.3.5 Conclusion

By extending the usage of virtual registers from message words to all system-call parameters, this binding promise easier portings. Additionally it decouples the definition and usage of the system-calls from the hardware registers and the kernel entry method.

W5 allows to use the fastest kernel entry method for every system-call and it allows to transfer many parameters in hardware registers. This should minimize the overhead of the system-call binding.

Furthermore it minimize the hardware dependent part of a system-call binding. All system-calls can share the same assembler stub, which makes the maintenance and the addition of system calls very easy.

In summary W5 is portable, efficient and simple enough to be extensible. The main reason for this is that **system call parameters are also stored in virtual registers**.

4.4 Summary

In this chapter I presented the current state of the L4.sec implementation, mentioned some simplifications and present a new system-call binding. I belief that three quarter of the source-code is present until now. From this a half are old code that is reused nearly unmodified from current FIASCO.

[26]Due to the local names of capabilities the version field of the previously used thread ID can be omitted.

Chapter 5

Evaluation

In this chapter I look at properties of an L4.sec system. First I discuss the performance overhead introduced with L4.sec in the IPC. Second I show that L4.sec does not necessarily increase the code complexity of the implementation. And third I propose how parts of user-land could be implemented on top of an L4.sec microkernel.

5.1 IPC Performance

IPC performance is the Master.

Jochen Liedtke

5.1.1 Introduction

In the past the performance of an L4 system was determined through the performance of IPC. Therefore heavy work was done to optimize Short-IPC, which is used in most cases for communication. This results in hand-optimized (assembler) shortcuts [Pet02].

It is obvious that functionality, as it IPC control is, degrades performance. But it is unclear, whether the assumption, the whole performance of the system is mainly influenced by the performance of a Short-IPC, could be mapped to an L4.sec system. Because the system design of a capability based system can be quite different from a current L4 system. Furthermore has nobody build user-land server[1] on top of L4.sec yet. However I try to estimate the additional overhead of a Short-IPC in L4.sec.

Why can this overhead not be measured today? First FIASCO for L4.sec is based on FIASCOUX and run therefore on a Linux system, but not on real hardware until now. Second there are many things missing in the implementation[2] as that the timings of IPC's in L4.sec and previous L4 versions are comparable.

So I can only estimate the performance effect of capabilities and endpoints. First I take a look at the performance estimates to implement IPC control given in previous papers. After that I describe the properties of the IPC in L4.sec. Finally a performance estimation is given based on our current knowledge of the system.

[1]In this work only kernel tests, which showed that particular functions of the kernel work correctly, are implemented.

[2]Mainly locking, Long-IPC, badges, deleteable capability tables and any kind of optimization. Of course an assembler shortcut does not exist.

5.1.2 Previous Work

Ports indirection

In [Lie93] Jochen Liedtke discuss how expensive the introduction of Mach like ports into the L3[3] IPC is. He does not consider the buffering feature of Mach, but he add indirection and port rights. As an optimization he let illegal entries pointing to an unmapped page. This moves the check, whether a port access is allowed or not, to the pagefault handler.

A best case estimation (without cache, but with TLB effects) gives an overhead of 29 cycles or 12% of 250 cycles[4]. A test implementation give the same results. Therefore Liedtke conclude that, in principle, port-based IPC can be implemented efficiently.

IPC redirection

The IPC redirection mechanism mentioned in Section 2.2.2 should results in an overhead of 3 cache lines or less than 50% of the register-only IPC cost. They expect that an optimized implementation gives an overhead of 20% for register-based IPC. For memory based IPC they assume that the overhead is negligible.

Summary

Both cases[5] introduce a table based indirection in every IPC. This indirection is comparable to L4.sec IPC, where an capability space is used to point to endpoints. Thus we assume that the overhead of capabilities and endpoints should be in an order of magnitude of 12-20%

5.1.3 L4.sec Short-IPC

Properties

Short-IPC in L4.sec has mainly the same properties like Short-IPC in previous L4 versions. The difference are in the partner selection and in the binding.

For the selection of the IPC partner the TCB of the corresponding sender or receiver has to be found. This needs in L4.V2 only *anding* a bit-mask to the thread ID and *adding* an offset to it[6]. In L4.sec it is a bit more complicated, because 2 indirections, capabilities and endpoints, are used. Additionally capability space and endpoints are usually allocated on own kernel pages independently from the threads. These things results in a larger cache and TLB working set.

Beside this disadvantages, which make L4.sec IPC slower, also properties exists, which make the IPC faster. The small binding overhead of W5, which needs for a Short-IPC only 2 words as header, transfers much more words solely in hardware registers. Thus the register payload of a Short-IPC can be up to 3 words with *sysenter*, 4 words with *syscall* or even 5 words with *int*. Also the collocation of the capability space with the task and the storage of the first 32 capabilities therein[7] are optimizations which make the IPC faster.

Overhead to Find a Receiver

After the general overview of the changes in L4.sec IPC, I give an estimation of the overhead to find a receiver in L4.sec. The other way around, to find a waiting sender, should be similar. Because cache and TLB misses dominate the IPC performance I focus on them and ignore the overhead of additional arithmetic instructions.

As shown in Figure 5.1 there are 4 steps until the sender found a receiver:

[3]L3, as the predecessor of L4, has the same IPC design.

[4]The work is based on a 486-DX50

[5]Clans & Chiefs are not considered here. See Section 2.2.2 why they are not fast enough.

[6]Due to the memory layout of the TCB's, which are allocated in an array. FIASCO uses larger kernel stacks and needs therefore an additional *shift* instruction.

[7]To remove the capability tables for the first capabilities, which avoids a special case at task creation, since no tables are needed for the first `map`.

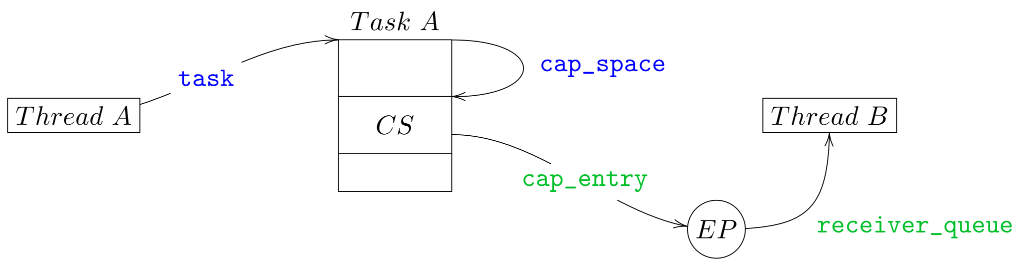

Figure 5.1: IPC partner lookup in L4.sec

Name	Cache and TLB misses
Lookup in the capability space	2
First capabilities in the task	-1
Dequeue partner from endpoint	3
Reuse partner	-1
Lazy queueing	-1
Summary	2

Figure 5.2: Overhead for finding a receiver by a sender.

1. **lookup the task pointer in the TCB**
   The TCB is used before and the value could be packed into a already used cache-line. Therefore needs this step no overhead.

2. **calculate the capability space pointer**
   The capability space is collocated with the task, therefore only a constant offset has to be added to the task pointer.

3. **lookup capability entry, get endpoint and check permissions.**
   A lookup in the capability space, which results in a cache and TLB miss, because this memory was not accessed before. Without the usage of the first capabilities, a additional cache and TLB miss occur for the capability table.

4. **dequeue receiver in the endpoint**
   The receiver thread has to be dequeued from the receiver queue (a double linked list), which head is stored in the endpoint. Thus the endpoint, the receiver and the next thread in the queue have to be accessed which results in 3 cache and 3 TLB misses. Because the receiver thread is used immediately one of the 3 can not be counted here. Additionally lazy queuing avoids that the next thread has to be accessed and it saves another cache and TLB miss[8].

The positive effect of the new binding cannot be estimated until now, but it should speedup the IPC's that are now fit into registers and have to be transfered with a memory copy before. The additional needed instructions and instruction cache lines are not considered here. Also special optimizations, like caching endpoints in the TCB, thread local capability entries or allocation of threads and endpoints in the same page, have not influenced the estimation.

In summary at least 2 additional cache lines and TLB entries to find a receiver are needed (see Figure 5.2). The receiver needs the same overhead to enqueue into the receiver list. Therefore we can conclude that, **L4.sec needs 4 additional data cache and TLB misses for a Short-IPC**.

5.1.4 Summary

An estimation gives an overhead of 4 additional cache and TLB misses, which is comparable to the IPC-redirection overhead. Whether this meets the 12-20% assumption and how it influences the

[8]Lazily updating the receiver queue is useful when a server thread mostly receives from a single endpoint and replies with zero timeout to its clients.

whole performance of the Short-IPC is unclear. Only an assembler implementation and a system using it, can decide, how fast the IPC in a secure microkernel can be.

5.2 Code Complexity

Adding more code adds more bugs.

Tanenbaums's first law of software

5.2.1 Motivation

The code complexity is one of reasons why software projects fail. L4.sec adds security and therefore functionality to the microkernel. These functionality should not increase the complexity of the kernel for the following reasons:

benefits from new implementation
A new implementation can avoid old design mistakes and learn from the former implementation. For example by providing cleaner interfaces in the kernel.

no backward compatible needed
There is no compatibility mode, which emulates old behavior and bugs, anymore.

less optimizations applicable
Optimizations, like that a task is only a page directory, can not be used in L4.sec. This decreases code complexity, but also performance.

simpler system-call binding
Due to new ideas the binding is much simpler and platform independent.

In summary I belief that L4.sec does not increase the code complexity. To support these thesis I measured the current implementation.

5.2.2 Measurement

It is widely accepted that the complexity of software depends on the lines of source code they consists of. But what a source code line is and what not is disputed.

For this work I use the tool *sloccount* [SLO] developed by David A. Wheeler. It counts physical source lines of code (SLOC).

What is a SLOC?

David Wheeler defines physical SLOC as follows[9]:

> *A physical source line of code is a line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character." Comment delimiters (characters other than newlines starting and ending a comment) were considered comment characters. Data lines only including whitespace (e.g., lines with only tabs and spaces in multiline strings) were not included.*

If I refer to lines of code in the following I mean SLOC's.

[9]See http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

name	L4.V2	L4.sec	difference
kernel	17805	14421	−3384
jdb	14656	3950	−9626
unit tests	400	1762	+1362
kernel memory and allocators	1521	972	−549
mapping database and spaces	1994	2614	+620
thread and syscalls	4347	2265	−2082
api	1067	889	−178
reused code	7287	6985	−302
others	1589	696	−893

Figure 5.3: Number of SLOC in FiascoUX for the L4.V2 and L4.sec version.

What Configuration is Measured?

Because the L4.sec implementation is based on FiascoUX, its source is compared against similar FiascoUX configurations. The build process use the default configuration and no experimental features are enabled. The code was measured in the same form as it is given to the compiler[10]. C preprocessor macros are used only sparsely in Fiasco. Therefore I did not check the effect of running the C preprocessor over the files. The assembler code is not counted here. It consists of an IPC shortcut, which is not written for L4.sec yet, and kernel entry code, which is quite similar to previous versions.

5.2.3 Results

First I compare the whole code of different API's. After that I measure different parts of them and identify reused lines.

Whole Code

The results of measuring the lines of code of FiascoUX in the L4.sec and L4.V2 version is given in Figure 5.3. The kernel debugger jdb is counted extra, because only a very small part (around a quarter) is adopted to L4.sec until now.

The L4.X2 version of FiascoUX have nearly the same number SLOC as the L4.V2 version[11]. Therefore only the L4.V2 version is used in this measurement.

The current state of the L4.sec implementation needs ca. 14400 SLOC. The L4.V2 version need about 3400 or nearly a quarter more. The main reason for this difference is that the L4.sec version is missing functionality. For example:

- locking,
- Long-IPC,
- IO-space and interrupts,
- FPU handling and
- schedule system-call

are not implemented until now. Implementing the missing features should result in a similar number of SLOC for the L4.sec implementation.

[10]This means the tool run in a build directory, after the tool *preprocess* has removed unused parts of the code.
[11]This measurement was done on the main-branch and not on the newly created V4-branch.

Smaller Parts

After counting all kernel code, I inspect only parts of the code. The results are given at the bottom of Figure 5.3. What are the reasons for the differences between L4.V2 and L4.sec?

The kernel memory and allocators are surprisingly smaller in L4.sec. The reasons for this could be:

- the L4.sec version allocates memory only from a kernel memory object[12]
- simpler allocators - there is no slab allocator implemented

The lines of code for the mapping database and spaces are comparable. The plus of 600 SLOC result from the capability space, which is not part of L4.V2.

The thread class and system-calls benefit from the simpler system-call interface. This does not fully explain that they need only 2000 SLOC or approximately the half. It is obvious that the previous mentioned missing functionality, for example Long-IPC, has also a huge influence.

The abi part of L4.sec misses a couple of interfaces, which are introduced into previous versions to unify different API's in one source tree.

Reused and Other Code

A half of the L4.sec code of this work is reused from previous FIASCO. Or in other words almost 40 percent of the FIASCO code is part of the L4.sec implementation. This small amount shows that the L4.sec implementation is more than another API.

Reused parts are platform and processor dependent parts, like:

- UX emulation,
- CPU dependent code and
- low level page-table handling.

Generic parts are also reused. This are for example:

- context, running queue and scheduling context
- timer and pic
- loader and console output

The differences in the numbers of SLOC for this results from small changes and deletions.

The code mentioned under the term “others” in the closing line of Figure 5.3, are either missing functionality, like IRQ and FPU handling, or generic code, like a free list or basic classes.

[12]This does not work for the boot-time, where the Kmem_allocator is used.

5.3 User-level

After looking at kernel properties in the previous sections, I give two user-level examples using L4.sec. First I describe a session based communication protocol for user-level servers and after that I look how the kernel-memory management could be used in ordinary applications.

5.3.1 A Session Based Communication Protocol

The usage of local names to specify communication partners and the availability of endpoints in L4.sec result in a different design of the user-level servers. While implementing and evaluating the L4.sec design, there was an idea of a session based communication protocol, which uses different sender ID's to identify the clients. I present it here, because it shows how L4.sec could be used. Beside this it is a good starting point for further work, which has to check, whether all authentication problems of L4 are gone.

The Details

1. **The server register itself by the name-server.**
   The server map to the name-server a capability to a self-created endpoint with a special badge, called the session-open badge. This badge is later used to distinguish session open requests from other client messages.

2. **The client asks the name-server for a server endpoint.**
   The name-server maps the endpoint to the client, it has from the server. Appending an additional badge is not needed here. The client has now a capability pointing to the server endpoint with the session-open badge.

3. **The client opens a new session by sending a client-endpoint to the server.**
   This session open message has the session-open badge as sender ID, so the server can distinguish it from normal requests. The client endpoint is mapped into a capability ID, the server thread specified in its capability receive window.

4. **The server maps its server endpoint back to the client.**
   The server can freely decide which client-badge it uses. For simplicity it could be the capability ID of the return endpoint. Since a selective unmap is not available, the server has to map the endpoint in its own capability space and than to the client. Otherwise closing a single session does not work.

The server can now receive client requests at the server endpoint with the client-badge and send a response through the client endpoint.

The session can be closed by the client if it unmaps the client endpoint from the server and the server endpoint from its own capability space. The server can use a LRU (Last Recently Used) close policy for client sessions and unmap the client capability and the corresponding copy of the server endpoint capability. It is also possible to use a explicit session-close command called by trusted clients[13].

Authentication

The loader, which created the server, could assure that a unique sender ID[14] is used for the communication with the name-server.

Additionally the server could use a challenge-response protocol or simply use a password to authenticate a client. Authentication is done at opening a session. After that, the server can trust that the kernel protects the sender ID's of the clients.

[13]Untrusted clients will never use it.

[14]For example a task ID or the hash of the started program, ...

Kernel-Memory Usage

What about the kernel memory in this protocol?

The servers need memory to map the endpoints to the clients. Either the servers have a fixed amount of kernel-memory and can therefore only handle a limited number of clients or the clients give kernel memory in the initial message to the servers.

Summary

This protocol can be used where the overhead of opening a session does not matter. It has the advantage that every server can manage the badges of its clients on its own. A system wide policy can be enforced through the name-server and a server wide policy through the servers itself.

This protocol can be used for single- and multi-threaded servers, because a server receives only from a single endpoint.

5.3.2 Kernel-Memory Management

How can an ordinary user-level application use the kernel-memory management of L4.sec?

Kernel memory is used in three operations:

- `create` new objects
- `map` objects
- `unmap` objects

In the following I describe three scenarios, which describe how applications could manage kernel-memory needed for these operations. This should give a deeper insight to L4.sec from the user-level point of view.

Simplest Way: Ignore it

The simplest way to use the new kernel memory management is not to use it at all in an application.

The objects could be created and destroyed by the pager. If an operation needs kernel memory, a fault could be send to the pager from the kernel. If the kernel does not send faults in this case[15] and return a error code instead, the fault could be send manually. The operations is simply retried, after the pager has answered the fault. All operations, including the retry, can be encapsulated in a library.

This solution does not give the new kernel-memory management to the application, but it has the advantage that it could be solely implemented in a library, which can be easily linked to an application.

Client Provided Resources

Another way to use the kernel-resource management is that the clients of a server have to provide the resources.

If a client opens a session, it has to map a kernel-memory object to the server. The server uses this kernel-memory object whenever it provides a service for the client, for example when asking another server or when mapping something back to the client. If a fault happen, because a client does not provide enough kernel memory or because it has revoked them, the server could simply close the session and free all session based data.

This solution has the advantage that it lets every client pay for the used service. In contrast to the first solution, every application has the benefit and perhaps overhead of the resource management.

[15]It is undecided until now, at which cases the kernel has to send faults.

Resource Manager

The kernel-resources could also be managed by a resource manager in or outside the task.

The resource manager requests the kernel-memory for the server once at startup from a global policy manager, or the loader provides this fixed amount of kernel resources. After that the server has to work with this amount. The resource manager can use any policy to subdivide the resources and assign them to client sessions.

The resource manager creates the objects and does bookkeeping of the kernel resources needed by every client. This allows to destroy client mappings or worker threads if there is a temporary shortage of resources. Worker threads in this multi-threaded server are created on demand by the resource manager in overload situations. Clients can also provide additional resources for the server, if they want to do extravagant operations or want to have real-time guaranties.

This solution can be extended until every resource is managed on-demand, because the resource manager can have a deep knowledge about the server it manages. Stacking different resource managers, to handle different subsystems, each with its own resource manager, is also possible.

Summary

The three scenarios can give only an insight in the power of the kernel-resource management of L4.sec. The possibility to implement fine granular policies and to integrate other kernel-resources, like scheduling time or power management, results in many degrees of freedom for the user-level design.

Chapter 6

Summary

In this thesis I implemented most parts of L4.sec. This implementation does not reveal fundamental issues in L4.sec that cannot be solved. The current state of the implementation is given in Section 4.1.6. I refine the L4.sec specification in Section 4.2 and propose W5, a new portable, efficient and simple system-call binding for L4.sec in Section 4.3.

The consequences of the kernel-memory management to the kernel implementation are evaluated in the Chapter 3. I discussed the effects of dependencies to the deletion process in Section 3.1 and give a rough design of a locking scheme for L4.sec. Furthermore I found that `copy` is unneeded and UTCB's are not a must anymore.

An estimation of the overhead introduced by L4.sec for a Short-IPC is given in Section 5.1. It shows that capabilities and endpoints are not for free. The real impact to the IPC performance remain open, which is a field for further research. In Section 5.2, I compared the code complexity of the implementation with a L4.V2 FiascoUX implementation. This evaluation show that a L4.sec implementation has a similar code complexity as previous L4 versions.

6.1 Further Work

There are many issues not solved until now. On the one hand design work is needed to answer different unsolved questions. On the other hand the implementation of L4.sec should be finished.

6.1.1 Design

In the last discussions about L4 and L4.sec the following questions appear:

- What are the application scenarios or how to build user-level on top of L4.sec?
- Are all authorization problems of L4 solved with L4.sec?
- In which cases a capability-fault must be raised or an error could be returned?
- What is the right IPC binding?
- How to handle interrupts?
- Does reservation based scheduling work?
- Are IPC-timeouts necessary or is a sleep enough?
- Can badges be a mask and not a bit-string?

These open questions could be a good starting point for the further development of L4.sec.

6.1.2 Implementation

There are many open implementation issues, for example:

- native ia32,
- performance measurement and optimizations,
- Long-IPC,
- IO-space and
- kernel debugger.

They should be closed in the next time. A stable and complete implementation is a precondition for application development on top of L4.sec.

Appendix A

Appendix

A.1 Glossary

ABI Application Binary Interface

API Application Programming Interface

Capability Capabilities are references to kernel objects with permissions. Capabilities are stored in capability spaces.

Capability space Translate task-local names into capabilites. `map` can be used to transfer capabilities between spaces.

DROPS The Dresden Realtime OPerating System Project is a research project at the TU Dresden aiming at the support of applications with Quality of Service requirements.

Flexpage An aligned region of a name-space with a size of a power of 2.

IPC Inter Process Communication. Transfer messages between tasks.

Kernel memory-space The part of the virtual memory address-space the kernel uses on its own. This is usually the memory above 3 gigabyte on x86.

Long-IPC Full featured IPC. Allows to transfer words, mappings and strings. Since memory transfers are included, pagefaults could occure.

MT1 First-level memory table. A unnamed object that consists of a page directory and a shadow directory.

Named object Also known as first class object. A kernel object that has a user-visible name in a capability space. These are for example threads, tasks, kernel-memory objects and endpoints.

Shadow directory Shadow tables and directories translates a virtual address into the address of the corresponding mapping node.

Short-IPC Hardware register based IPC that transfers 2-3 words. It is very fast and in contrast to Long-IPC no pagefaults could occure.

TCB Thread Control Block. Kernel data structure for every thread that contains state of the thread (register, ipc state, task pointer,. . .) and its own kernel stack.

Unnamed object Also known as second class object. A kernel object that has no user-visible name. These are for example mapping nodes or page tables.

A.2 W5 binding

A.2.1 Short-IPC

Short-IPC Input Parameters

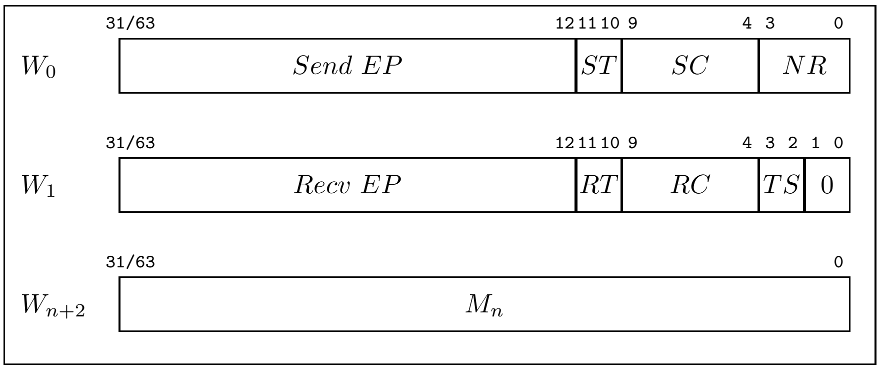

Figure A.1: Input parameters for Short-IPC.

The input parameter words for Short-IPC are given in Fig. A.1. The parameters have the following meaning:

NR (System call number) field
The number of the system call to call. This is 0 for Short-IPC. See Section A.2.3 for other values.

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 after message transfer.

Send EP (send endpoint capability) field
The capability ID of the endpoint the message is send to.

TS (timeout number) field
Timeout is either a $(never, never)$ 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

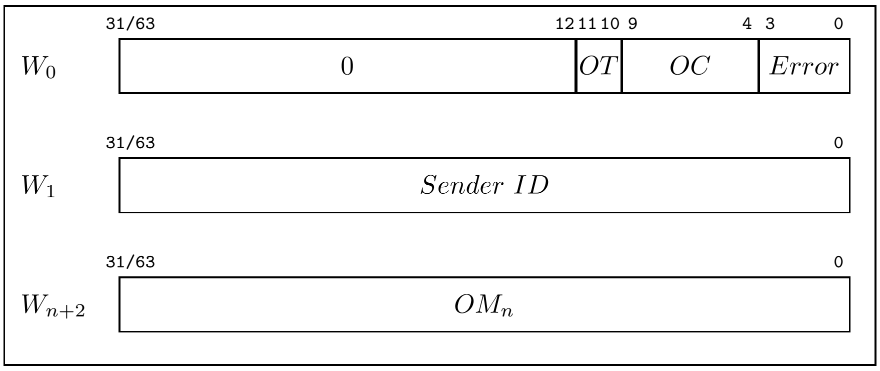

Figure A.2: Output parameters for Short-IPC.

The output parameter words for Short-IPC are given in Fig. A.2. The parameters mainly reuse the bit positions of the input. The parameters have the following meaning:

Error field
The base error code from this system-call. See Section A.2.3 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 SenderID TCR appended.

OM (received message word) fields
The received message is stored in OC parameter words.

A.2.2 Virtual Register Mapping

In this section a virtual register mapping is given for x86 *sysenter/sysexit*. A mapping for other kernel entry methods like software trap *(int,iret)* or AMD's syscall extension *(syscall,sysreturn)* could be easily specified.

Sysenter

The *sysenter/sysexit* instructions are used for fast kernel entry and exit. They are available since the Intel Pentium II. In AMD K7 and newer CPUs a similar instruction pair called *syscall/sysreturn* is available. The difference is mainly that *syscall/sysreturn* does not switch the stack pointer.

The caller of *sysexit* has to store the stack and instruction pointer in the two user registers ECX and EDX. These two values have to be given in the kernel at *sysenter* and occupy two registers. The kernel or user stack pointer is loaded into ESP and therefore ESP could not be used. So only 5 general purpose registers are available to transfer parameters from and to the kernel.

EAX	W0
EBX	W1
ESI	W2
EDI	W3
EBP	W4
ECX	user ESP
EDX	user EIP
ESP	kernel ESP
$ECX + 0x00$	W5
$ECX + 0x04$	W6
$ECX + ...$	...

Description: The additional words are reside on the user stack.
Design notes: They are only present if the syscall specifies that.
The syscall knows how many words are used.

Int 0x40

For the implemented kernel entry method (int 0x40) the same register mapping as for sysenter is used. The first five parameter words reside in EAX, EBX, ESI, EDI and EBP and the pointer to the following words in ECX.

A.2.3 Numbers

Syscall Number

Numbers:

0	ShortIPC
1	LongIPC
2	Unmap
3	Create
4	ThreadCtrl
5	ExRegs
6	Schedule
8	Debug

Design notes: The numbers are arbitrarily chosen.
Only ShortIPC could benefit from 0 and get a little bit simpler test.
Changes: 16 numbers are available now.

Bibliography

[ABB$^+$86] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*, pages 93–113, Atlanta, GA, June 1986.

[Cla04] Dietrich Clauss. Implementation of the L4 Version x.2 ABI in the Fiasco Microkernel. Study thesis, TU Dresden, 2004.

[Cla05] Dietrich Clauss. Investigation of Mechanisms to Support User-Level Thread Packages on Top of the L4-Fiasco Microkernel. Master's thesis, TU Dresden, February 2005.

[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://l4hq.org/docs/manuals/`.

[Gru98] Lukas Gruetzmacher. Entwurf und Implementierung einer Mappingdatenbank fuer L4. Study thesis, TU Dresden, October 1998.

[HE03] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, September 2003.

[Hoh98] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`.

[Hoh02] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems.* PhD thesis, TU Dresden, Fakultät Informatik, September 2002.

[HQL] L4 headquarters website. URL: `http://l4hq.org/`.

[JEL$^+$99] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *7th Workshop on Hot Topics in Operating Systems (HotOS)*, Rio Rico, Arizona, March 1999.

[JTG$^+$00] Trent Jaeger, Jonathon E. Tidswell, Alain Gefflaut, Yoonho Park, Kevin J. Elphinstone, and Jochen Liedtke. Synchronous ipc over transparent monitors. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 189–194. ACM Press, 2000.

[Lie92] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[Lie93] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[Lie95] J. Liedtke. On μ-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.

[LW01] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems*, Schloß Elmau, Oberbayern, Germany, May 20–23 2001.

[Pet02] Michael Peter. Leistungsanalyse und Optimierung des L4Linux-Systems. Master's thesis, TU Dresden, June 2002.

[PV05] Michael Peter and Marcus Völp. *L4-Sec Kernel Reference Manual*. TU Dresden, 0.1.9 edition, feb 2005.

[Reu04] Rene Reusner. Implementierung von Local IPC auf L4/Fiasco. Study thesis, TU Dresden, Germany, November 2004.

[SLO] Sloccount website. URL: `http://www.dwheeler.com/sloccount/`.

[Ste02] Udo Steinberg. Fiasco μ-Kernel User-Mode Port. Study thesis, TU Dresden, Germany, December 2002.

[Stö02] Jan Stöß. I/o-flexpages on the x86-architecture. Study thesis, System Architecture Group, University of Karlsruhe, Germany, May 31 2002.

[TUD] L4 website. URL: `http://os.inf.tu-dresden.de/L4/`.

[Voe02] Marcus Voelp. Design and implementation of the recursive virtual address space model for small scale multiprocessor systems. Master's thesis, System Architecture Group, University of Karlsruhe, Germany, September 2002.

[War02] Alexander Warg. Portierung von Fiasco auf IA-64. Study thesis, TU Dresden, Germany, May 2002.

[War03] Alexander Warg. Software Structure and Portability of the Fiasco Microkernel. Master's thesis, TU Dresden, July 2003.

[Wen02] Horst Wenske. Design and Implementation of Fast Local IPC for the L4 Microkernel. Study thesis, System Architecture Group, University of Karlsruhe, Germany, July 2002.