Implementation of the L4 Version x.2 ABI in the FIASCO Microkernel

Dietrich Clauß, TU Dresden

April 14, 2004

Contents

1 Introduction **3**

2 Fundamentals and Related Work **4**
2.1 The L4 Microkernel Interface 4
2.2 The Fiasco Microkernel 4
2.3 DROPS 5
2.4 The L4v4 ABI 5
2.4.1 Threads and Tasks 5
2.4.2 Thread IDs 5
2.4.3 Kernel Interface Page 6
2.4.4 UTCBs 6
2.4.5 System Calls 6
2.4.6 Flexpages 8
2.4.7 IPC 9
2.4.8 Protocols 9

3 Design **11**
3.1 Generic Design Goals 11
3.2 Data Types 11
3.2.1 Thread IDs 12
3.2.2 Flexpages 12
3.2.3 Timeouts 12
3.3 Threads and Tasks 13
3.3.1 Thread Data 13
3.3.2 Task Data 13
3.3.3 Thread Creation 13
3.4 UTCBs 14
3.4.1 User UTCB Access 14
3.4.2 Kernel UTCB Access 14
3.5 System Calls 14
3.5.1 System-Call Parameters 15
3.6 IPC 15
3.6.1 The IPC System Call 15
3.6.2 The Synchronization Functions 16
3.6.3 The Data Copy Routines 16
3.6.4 IPC Error Codes 16
3.7 Flexpage Mapping 16

3.7.1 Mapping Database . . . 17
3.8 Kernel Debugger . . . 18
3.9 Implementation Process . . . 18
3.9.1 Main Approach . . . 19
3.9.2 Fiasco-UX . . . 19
3.9.3 Building and Testing . . . 19
3.9.4 Simultaneous Development . . . 19

4 Measurements **20**

5 Conclusions, Open Topics, and Future Work **22**
5.1 Future Work . . . 22
5.2 Open Topics . . . 22
5.2.1 Covert Channel in the UTCB Pointer Page . . . 22
5.2.2 Thread ID Data Type . . . 23
5.3 Summary . . . 23

Glossary **24**

References **24**

Chapter 1

Introduction

The research on microkernels and the operating systems built on top of them is still in rapid movement. Many questions of how to design the kernel interface are still open. The ABI[1] has to both fit the applications' needs and allow an efficiently working implementation in the kernel. Thus there are many ABI versions out there, and there are as just as many ideas for ABI modifications waiting to be implemented and tested.

Today's microkernels are mostly written in high-level programming languages. It was shown that the performance overhead of even object oriented design compared with assembly language can be minimized. High-level languages on their part allow increased maintainability and portability of the code.

This work shows that the so gained flexibility can also be used to implement highly varying ABIs in the same kernel, whereas a large part of the kernel code gets reused.

[1] Application Binary Interface

Chapter 2

Fundamentals and Related Work

2.1 The L4 Microkernel Interface

L4[1] [Lie95] is a microkernel interface initially designed by Jochen Liedtke and implemented in the L4-GMD kernel. After that, L4 has been ported to various architectures and rewritten in various kernels[2]. On the other hand, the experience with L4 kernels and applications led to new ideas of how to improve the L4 interface. That is the reason why there are many L4 ABI extensions and experimental versions available. The following ones are of interest for this work:

v2 The initial and the current stable version [Lie96].

x.0 An experimental version [Lie99] derived from `v2`, implemented in the Hazelnut kernel [Haz].

x.2 alias v4 The second experimental version [DLSU04], implemented in the Pistachio kernel. This aims to be the upcoming next stable version. There are many differences to `v2`, as shown in the following chapters. Indeed, there are also shortcomings in `x.2` that may cause that the next stable version is not based on `x.2` but on one of the future experimental versions. As the discussion what the next stable L4 ABI version will be is beyond the scope of this work, I anyway talk about `v4`.

2.2 The Fiasco Microkernel

Fiasco [Hoh98, Hoh02] is a microkernel written by Michael Hohmuth and others at the Dresden University of Technology. It implements the L4 ABI. Before the start of this work, Fiasco implemented the L4 ABI versions `v2` and `x.0`. The `v2` support is stable and feature-complete. The kernel was initially written for the IA-32 architecture. Later it had been ported to both IA-64 [War02] and ARM [War03]. There is also a user-mode port called Fiasco-UX [Ste02], developed by Udo Steinberg.

Fiasco is written in both C++ and assembly language, where the latter is used only for things that cannot be achieved with C++, or for optimization purposes. It also uses *Preprocess* [Hoh04], a C++ preprocessor written in Perl. Preprocess allows both C++ function declarations and definitions to be placed in a single file. A second key feature of Preprocess is a mechanism that supports explicit compile-time polymorphism without overhead [War03].

[1] See http://os.inf.tu-dresden.de/L4/ and http://www.l4ka.org/

[2] For an overview of the several L4 implementations, see http://os.inf.tu-dresden.de/L4/impl.html

Before this work the FIASCO source code was already structured into modules and submodules, but there was no borderline between ABI-specific and generic code.

2.3 DROPS

The *Dresden Real-Time Operating System (DROPS)*[3] is a research project aiming at the support of applications with Quality of Service requirements. It consists of various application programs and server processes running on top of the FIASCO microkernel. The DROPS applications can be divided into two parts, the real-time component and the time-sharing component. The probably most important member of the time-sharing component is L^4*Linux* [Hoh96], an L4 port of the Linux operating system. It is used to be able to run normal Linux applications inside the time-sharing component of DROPS.

2.4 The L4v4 ABI

This section describes the `v4` ABI and the differences to the other versions `v2` and `x.0` that are relevant when implementing `v4` in the FIASCO kernel. However, `v2` and `x.0` have only minor differences and were already implemented in FIASCO, so it mainly comes to the differences between the two main ABI revisions, `v2` and `v4`.

2.4.1 Threads and Tasks

A *task* in terms of L4 is an entity that consists of an address space and a set of *threads*. A thread is an activity executing in an address space. A thread is characterized by a set of registers, including at least an instruction pointer (*EIP*), a stack pointer (*ESP*), and state information. The state also includes the address space in which the thread currently executes.

Threads and tasks exist in both the `v2` and the `v4` ABI. In `v2` the maximum number of tasks is 2048, where each task owns a fixed number of 128 threads.

In `v4`, the kernel can define the maximum count of threads itself. The number of threads per task is not defined by the ABI. A task must own at least one thread (that does not have to be active). From this it follows that the maximum number of tasks supported is implicitly set by the maximum number of threads. There is no task ID visible to the user. The only way for the user to address a task is using the thread ID of a thread running in that task.

Task Deletion

In `v2`, a task can be deleted explicitly. In `v4` the kernel deletes a task implicitly as soon as it runs out of threads. That is the case when the user deletes the final thread of this task, or migrates the final thread to another address space. Therefore, in `v4` the kernel has to keep a *thread counter* for each task to keep track of how many threads the task owns (see Section 3.3.2).

2.4.2 Thread IDs

In each ABI there are both local and global thread IDs, but their meanings are completely different.

In `v2`, a global thread ID contains the task number and the (task-local) thread number. The former is also called the task ID, whereas the latter is the local thread ID.

[3] See http://drops.inf.tu-dresden.de/

v2 **ABI**	v4 **ABI**
task_new	space_control thread_control
ipc	ipc lipc
unmap ex_regs thread_switch thread_schedule	unmap ex_regs thread_switch thread_schedule
id_nearest	 clock kernel_interface processor_control memory_control

Table 2.1: System Calls in the Different ABI Versions.

In v4, each thread has two IDs, a global and a local one. The global ID contains a global thread number only (and a version number, but that is irrelevant here). The local ID contains a pointer to the thread's UTCB (see Section 2.4.4). It may only be used when addressing threads in the task's own address space.

v4 thread IDs don't reveal the task the thread is running in.

2.4.3 Kernel Interface Page

The *kernel interface page (KIP)* is a memory page accessible read-only by each thread. On task creation, the kernel maps the KIP into the newly created address space. The KIP is used by the kernel to propagate system-wide information (e.g., the processor frequency). v4 adds the feature of KIP system calls (see Section 2.4.5) and makes minor changes to the KIP layout.[4] In other respects, the semantics of the KIP remains unchanged.

2.4.4 UTCBs

A user level thread control block (*UTCB*) is a memory object mapped by the kernel for each thread. UTCBs are new in v4. They are used to ease kernel access of user data and vice versa. Another application that even requires a UTCB is intra-task communication without kernel entry (local IPC[5]).

To allow a thread to find its UTCB, the kernel has to provide the UTCB address of the current active thread. The v4 ABI defines that this value has to be stored in the memory location gs:0.

2.4.5 System Calls

Overview

Table 2.1 shows an overview of the differences of the system-call interface between the ABI versions.

[4]To learn the exact set of changes, see [Lie96, DLSU04] and the Fiasco source code. They are not relevant here. Relevant is whether or not changes occured and code had to be reimplemented.

[5]See [LW]

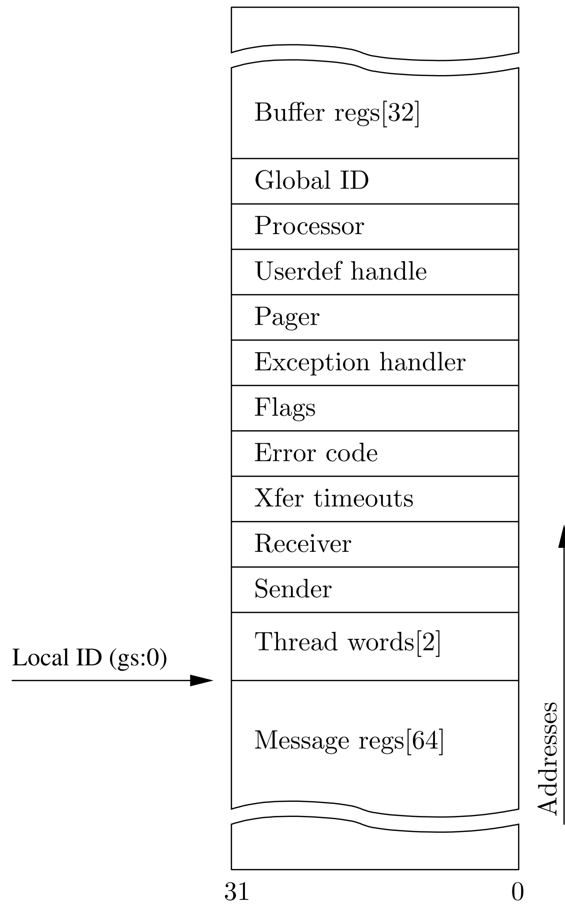


Figure 2.1: Layout of an Userlevel Thread Control Block (*UTCB*).

The `task_new` call has been replaced by two ones, `space_control` and `thread_control`. That has been reasonable, because `v4` allows creation of single threads, single spaces, and any correlation between threads and spaces. For a detailed description how these system calls work, see Section 2.4.8. The core functionality of the IPC call does not change, but in `v4` it makes use of the UTCBs, allowing more data to be transferred in a short IPC. The calls `unmap`, `thread_switch`, and `thread_schedule` remain unchanged. The `ex_regs` call now has a new parameter format and allows more registers to be changed.[6] There is no `id_nearest` system call any longer, because the Clans&Chiefs model [Lie91] has been canceled in `v4`. The `clock` system call delivers the system time. In fact, in `v2` there is a clock call, too, but it is not implemented as a system call. `kernel_interface` delivers the address where the KIP is mapped. The calls `processor_control` and `memory_control` have been added. They allow user tasks to control parameters of the hardware, for example, memory cacheability and processor frequency.

Privileged Task

The first task started in an L4 system is called the *root task*, or the *root server*. In `v4` the root task is also the one and only *privileged task*. That means that only threads running in that task can invoke a certain set of system calls, in particular, `thread_control`, `space_control`, `memory_control`, and `processor_control`. The idea behind that is to create a secure operating system by writing a root server that properly decides which threads may do critical operations. However, this concept is not yet complete, because there are still possibilities for nonprivileged threads to compromise the system. For instance, two threads can do large quantities of memory-map operations, until the kernel memory is exhausted. These problems will be coped with in future versions of the L4 ABI.

KIP System Calls

To invoke a system call, the user has to execute a machine instruction that makes the thread enter kernel mode. On the IA-32 architecture, the appropriate instructions are `int` and `sysenter`. In the `v2` ABI the user has to invoke the respective instruction directly. In `v4`, the kernel provides *system call entry stubs* on the KIP. An entry stub is a piece of code the user can access using a `call` instruction. The stub on its part enters the kernel using the appropriate machine instruction. This concept, called *KIP system calls*, has been added in `v4` to be able to change the way of how to enter the kernel without having to recompile all the userland applications. The drawback is that it adds another layer of indirection to the system calls, dropping their performance. These costs are considered to be tolerable, as the costs of entering the kernel are an order of magnitude higher as these of the call to the entry stub.

Fiasco/v2 already had an implementation of KIP system calls, so I only had to adapt it to match the `v4` specification.

System Call Parameters

In `v2` the system-call parameters are passed in registers only. In `v4`, UTCB variables are used in addition to registers. That causes differences in the call frames of the system calls.

2.4.6 Flexpages

A *flexpage* is a region of virtual address space. A flexpage is described by its base address and its size. The L4 ABI defines that the size of a flexpage must be a power of two, and the base

[6] For the exact set of changes, see [Lie96, DLSU04].

address must be aligned to the size. Flexpages can be *mapped* or *granted* from one address space to another.[7]

2.4.7 IPC

The general flow of an IPC operation is the same in all ABI versions. On the other hand, there are two major differences.

First, `v4` adds the possibility to send to one thread, whilst receiving from another, in a sole IPC call. That means that there may be three partners involved in a single IPC operation.

The second major difference is in the message format and the way how to get data from the sender and how to transfer it to the receiver. In `v4`, for this purpose the message registers in the UTCBs are used. They can contain up to 64 words to transfer, or 32 typed items, or a mix of them. A typed item can be a flexpage to map or grant, or a string item. If string items are to be transmitted, the call becomes a long IPC. On the receiver's side, the memory areas where to accept long IPC data are provided as flexpages in the buffer register UTCB variables.

2.4.8 Protocols

Thread-Creation Protocol

In `v2`, the user cannot create only a single thread, but create a whole task using the system call `task_new`. That implicitly conceptually creates all 128 threads that may run in that task, and makes thread 0 run.

`v4` has a more sophisticated scheme of thread creation. There are two cases to distinguish: whether a thread is to be created in an existing address space, or in a new space.

Thread creation in an existing space: In that case, only one system call `thread_control` is needed. This call sets both the global and local thread ID, the space, the scheduler, and the pager. If the caller omits the pager parameter, the kernel creates an *inactive* thread. Inactive threads can later be activated, using another `thread_control` call or an `ex_regs` call to set the pager.

Thread creation in a new space: To create a new address space, the following three steps have to be performed.

1. `Thread_control` call
   - creates both thread and address space
   - sets both global and local ID of the thread
2. `Space_control` call
   - sets both KIP and UTCB area for the space
   - maps the KIP
3. `Thread_control` or `Ex_regs` call
   - sets the pager for the new thread
   - activates the thread

A newly created or activated thread does not run yet, but it waits for a *startup message* from its pager. That message contains the start values for EIP and ESP (Figure 2.2). When the pager sends such a message, the kernel sets the thread's EIP and ESP accordingly and activates the thread.

[7] For a full description of the address space concept of L4 see [Lie96].

stack pointer	MR_2
instruction pointer	MR_1
0x00000002	MR_0
31 0	

Figure 2.2: Layout of a Thread Startup Message.

Pagefault Protocol

When a pagefault occurs, the kernel sends a message to the faulting thread's pager. This message contains the the fault address and an error code. In `v4` the message has another layout, and the IPC send operation uses other registers. Except for that the behavior remains unchanged.

Chapter 3

Design

3.1 Generic Design Goals

The main goal of the Fiasco/v4 project was to extend Fiasco to support the `v4` interface, whereas the efficiency and stability of the `v2` support should be preserved. Code duplications should be avoided to hold the code maintainability on a high level.

Fiasco already had a powerful configure and build system, which was developed to be able to port the microkernel to different architectures and implement compile-time features like *small address spaces* and *I/O port protection*. That system could perfectly be reused to add support for the new ABI. For further documentation regarding the Fiasco build system, see [Hoh98, Hoh02].

The major design decisions were to define where the interfaces between ABI-specific code and generic kernel code should be. Trying to hold the amount of ABI-specific code as small as possible may lead to performance penalties. On the other hand, having too much ABI-specific code will cause code duplications and thus decreases the maintainability. That decision had to be made for each part of the kernel that is affected by the ABI change.

In this chapter, I will expound all these issues and present the solution used in the current Fiasco/v4 implementation.

3.2 Data Types

There are data types defined in the ABI that are used for kernel internal purposes, too. When the ABI changes such a type, we have to do something with the kernel type. There are two possible solutions:

- Use different data types for ABI and kernel, or do not use that type inside the kernel at all. The main advantage of this method is the ability to run more than one ABI at the same time.

- Make two interfaces for the type, a generic one for kernel use and a ABI-specific one for ABI use. The main advantages of this method are:

  efficiency: As only the interface differs, the kernel does not need to convert between ABI and kernel types.

  simplicity: The kernel code uses the data types as usual, however, it may use the generic interface only.

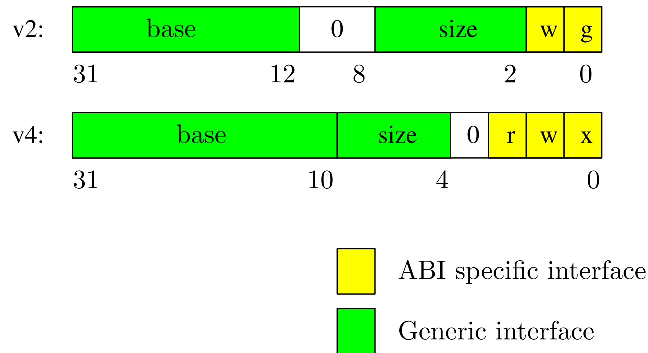


Figure 3.1: Layout of a Flexpage Descriptor in each ABI.

As simultaneous support for different ABIs is not required, I decided to use variant two for all affected data types. The types we are talking about are `L4_uid`, `L4_fpage`, and `L4_timeout`.

3.2.1 Thread IDs

The layout of the type `L4_uid` has changed a lot. In `v2`, an UID consists of a task and a local thread number, and both numbers together form the global ID. In `v4`, each thread has a local and a global ID, which are at first independent of each other. Furthermore, none of them says to which task the thread belongs.

I decided to reuse the existing `L4_uid` interface for both the local and the global ID. I only had to add one method, `is_local()`, which says, if the ID is a local or a global one. Starting from that the kernel has to decide, which access methods may be used. There are situations where the kernel has to convert a local ID to a global one or vice-versa. These translations need data from the space the thread is running in, and its TCB, thus they cannot be provided by the data type itself.

In addition to that, the `v4` ABI defines constants for the `ANY`, `ANY_LOCAL`, and `NIL` ID, whereas in `v2` there is a `NIL` ID, too, but there is also an `INVALID` ID, which differs from `NIL`. As there is no distinction between `INVALID` and `NIL` any longer, all occurences of the `NIL` and `INVALID` thread IDs in ABI-independent code had to be revised and changed accordingly.

3.2.2 Flexpages

Figure 3.1 shows the layout of a flexpage in each ABI. The generic interface comprises access methods for the base address and the size, whereas the ABI interface handles the access right bits and the *grant* bit. The `grant` bit does not belong to the flexpage in the strict sense. It is only needed when a map or grant operation is requested. In `v2` it is then coded into the flexpage descriptor, whereas in `v4` it is given as an extra parameter of the map/grant item.

3.2.3 Timeouts

The timeout data type has exactly the same interface for all ABIs. Only the binary representation differs. That leads to an ABI-specific implementation of the timeout representation only. The timeout-related kernel code remains unchanged.

3.3 Threads and Tasks

3.3.1 Thread Data

Threads in `v4` have the same TCB variables plus two new ones:

Local ID: the local ID of the thread, that means, a pointer to the thread's UTCB. This pointer is only valid in the user's virtual address space.

UTCB pointer: a pointer to the UTCB for kernel use. The kernel needs to access the UTCBs without looking in the user's page directory or even the user's address space, so this extra pointer is needed. For a more detailed description, see Section 3.4.

These new parameters are simply added to the TCB.

3.3.2 Task Data

The only data the kernel stores for each task is the task's page directory. As not the whole four gigabytes of the virtual address space is accessible by the user, there are page directory entries that are never accessed. The kernel uses them to store various per-task data. The parameters to store per task vary between the ABIs.

In `v2`, the following parameters are stored in unused page-directory entries:

- task number
- chief ID
- host process ID (Fiasco-UX only)

In `v4`, there are neither task numbers nor chief IDs. Instead of these, there are a UTCB area and a thread counter. So the respective entries simply get reused. The new entries have the following meaning:

UTCB area: the region where the UTCBs for the threads of this task may resist. It is stored in the format of a flexpage.

thread counter: the count of threads running in this task. The kernel needs this counter to keep track of how many threads run in this task. If a task gets empty, the kernel has to delete it implicitly.

3.3.3 Thread Creation

As shown in the former chapter, the scheme how the user makes the kernel create a thread heavily differs between the ABIs. On the other side, as much as possible of the thread creation code in the kernel should stay generic.

In case of the thread constructor, approximately half of the code is ABI dependent. Further, the parameter format differs. So I decided to make an ABI-specific constructor and factor out the common code into generic utility functions.

For the task constructor, the situation is quite similar, so it was reasonable to create a new constructor, too. Furthermore, an initialization function to initialize the UTCBs has been added. This cannot be done by the constructor, because at task creation the kernel does not know the UTCB related parameters yet.

3.4 UTCBs

On task creation, the kernel maps memory into the user-specified UTCB area of the new task. The threads residing in this task have access to that area. The whole area must be mapped at task creation, because we do not want to care about page-faults in the UTCB area for performance reasons.

The UTCBs must be accessible by both the kernel and the owner thread.

3.4.1 User UTCB Access

As each thread has mapped its UTCB in its address space, it simply accesses it using normal memory operations. However, the thread needs to know the address of its UTCB. The ABI defines that the kernel has to provide the current thread's UTCB address at the memory location `gs:0`.

The UTCB Pointer Page

To be able to provide a value at the memory location `gs:0`, Fiasco/v4 maps a page of memory into a formerly unused area of the user address space[1] and sets up a segment descriptor for the `gs` register to point at the beginning of this page. On a thread switch, the kernel puts the UTCB address for the new active thread there.

The page has to be user writable, too, if LIPC should be possible. As the user can also address the UTCB pointer page using the data segment, it potentially has read-write access to the whole page. So the page becomes a covert channel between all threads of the system. That is a known issue and can be solved by limiting the data segment to not cover the address of the UTCB pointer page. As that requirement was not relevant for running just test applications, I did not yet implement the solution. It will be implemented in a future version of Fiasco.

3.4.2 Kernel UTCB Access

The kernel has the whole physical memory mapped in its page directory. This mapping is used to access the UTCBs. On thread creation, the kernel determines the kernel-virtual address of the thread's UTCB and stores it in a TCB variable. As UTCBs cannot move, this calculation has to be done only once for each thread. An assumption for this method to work is that UTCBs must not cross page boundaries. That is the case because the kernel sets the UTCB alignment and size parameters accordingly. These parameters are visible to the user in the KIP. If a thread wants to create another one with a wrongly aligned UTCB, the corresponding `Thread_control` system call will simply fail.

3.5 System Calls

As the system-call interface changed a lot, most of the system calls had to be reimplemented. To enter the correct system call depending on the system-call number, the kernel uses an array of function pointers, whose n'th element contains the function pointer for system call n. This so-called *system-call dispatcher table* got ABI specific, too.

[1]On native IA32, an address above 3GB is used. On Fiasco-UX, the kernel uses a page from the end of the user-accessible virtual address space.

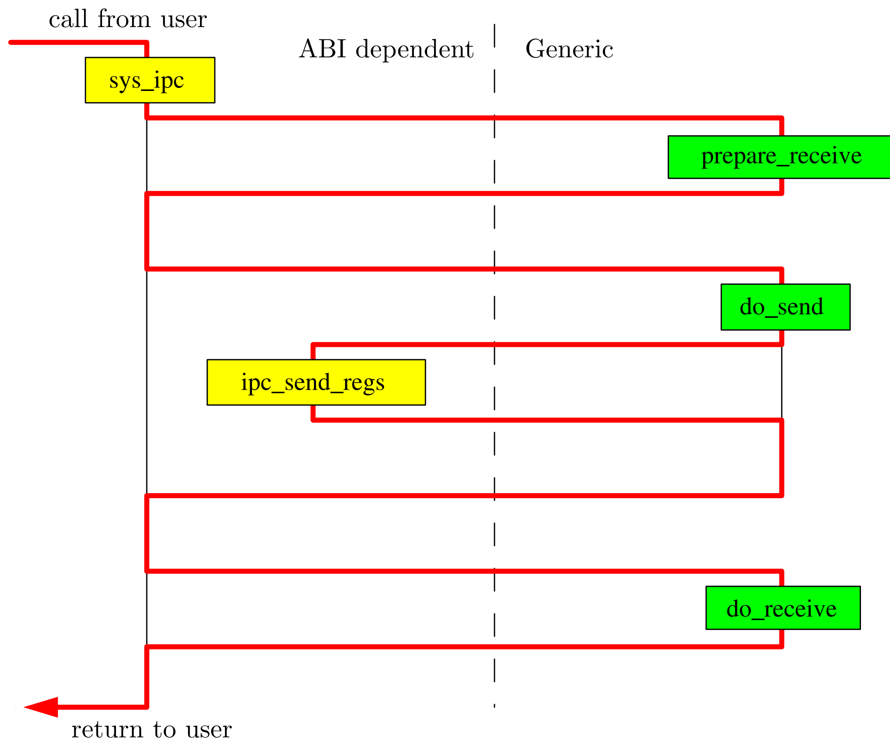


Figure 3.2: Application Flow of an IPC Operation.

3.5.1 System-Call Parameters

In addition to CPU registers, most system calls make use of UTCB variables to transfer parameters now. As the ABI makes no strong distinction between register and UTCB parameters, I decided to put both register and UTCB parameter evaluation in one and the same class. That class has access to both UTCB and register stack. If future ABI versions move parameters between UTCB and CPU registers, only the system-call frame code will have to be adapted.

3.6 IPC

One of the most important issues in designing Fiasco/v4 was dividing the IPC path into generic and ABI-specific parts. In general we can say that all the synchronization-specific things are generic, although the methods that actually copy data are ABI specific.

Fortunately, the IPC path was already split into several functions. I did not have to split up it anymore, but had to decide which functions stay generic and which ones need to be rewritten for `v4`. Figure 3.2 shows the flow of an IPC operation and the affected methods. All that methods belong to the class `Thread`.

3.6.1 The IPC System Call

The first method called when an IPC is invoked is the IPC system call. This method is ABI specific because of the new `v4` IPC call `reply_and_receive`, which involves three partners in

one IPC operation. To not drop the performance of `v2` IPC, I decided to duplicate the whole IPC system-call function. Possibly in future FIASCO versions the system call will be reunified.

3.6.2 The Synchronization Functions

The methods `prepare_receive`, `do_send`, and `do_receive` synchronize using TCB flags, locks, and scheduling functions. They have nothing to do with the actual IPC data, so they are not ABI specific. They only had to be adapted regarding return values and IPC error codes (see Section 3.6.4).

3.6.3 The Data Copy Routines

The actual copying of data from one thread to another is done by the methods `ipc_send_regs` and `do_send_long`. As the format of IPC messages changed a lot, these methods had to be completely rewritten.

However, `v4` long IPC is not implemented yet, so the corresponding `do_send_long` function is still empty.

3.6.4 IPC Error Codes

In case of IPC completion codes the situation is a bit more complicated than on the other data types. The reason is that ABI-specific and generic data are mixed in one and the same data type, `L4_msgdope`.

In `v4` there is a similar type called *message tag*, which contains the message length and some of the completion status bits. The actual error code is stored in a UTCB variable, `Error_code`. This variable additionally contains the number of bytes transferred successfully.

Commonly we can say that IPC-error-related data is ABI independent, whereas message related data differs between the ABIs. The binary representation of the error codes, however, is ABI dependent, too.

To not have to duplicate lots of code, there was no reasonable way to go without an own data type for the error code. That type contains all relevant error bits in an order that allows easy translation to the ABI-specific return values at the end of the IPC. Before the addition of `v4` support, FIASCO used a `L4_msgdope` as return value of most IPC-specific functions, whereas in most cases only the error-code part of the dope was relevant. So it took comparatively low effort to replace these `L4_msgdope`s with `Ipc_err` values. At the end of an IPC, the kernel calls an ABI-specific commit function, which puts the completion code into the right register(s).

3.7 Flexpage Mapping

Figure 3.3 shows the flow of a flexpage map operation. It is just the same in `v2` and `v4`, but there are three major differences:

1. layout of the flexpage type,
2. position of the `grant` bit,
3. behavior when the sizes of source and destination flexpage differ.

The layout difference has been covered using a ABI-specific interface for the data type, as shown in Section 3.2.

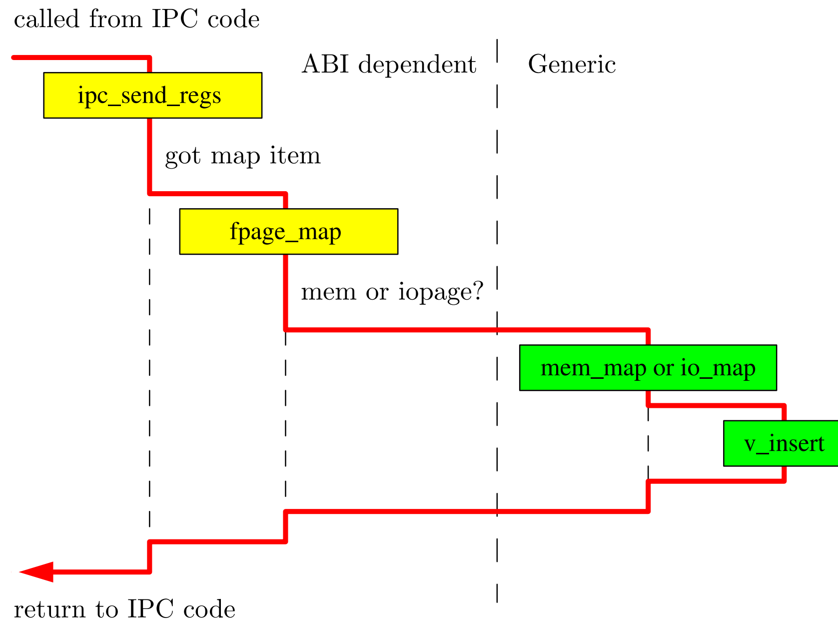


Figure 3.3: Application Flow of a Flexpage Map Operation.

The second and third difference, however, need specific functions. I decided to put that code into the function `fpage_map`. Thus now each ABI has its own `fpage_map` implementation, whereas the functions it calls are generic. Other solutions would have been

- duplicate the `mem_map` and `io_map` functions, too, or
- put the ABI separation directly into the IPC code.

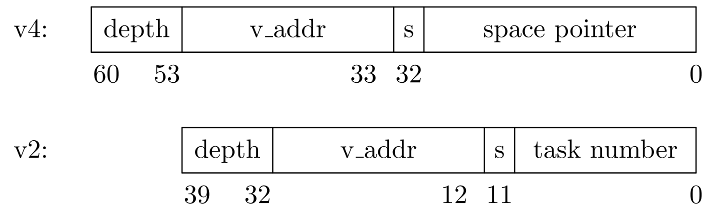
Variant 1 would result in too much code duplication and does not have other advantages. Variant 2 would have been possible, too, but I discarded it because it clutters up the IPC code.

3.7.1 Mapping Database

The *mapping database* is used by Fiasco to determine what mappings have to be revoked when a thread calls `flush`. The main function of the mapping database remains unchanged. Nevertheless, the implementation cannot be inherited without changes, because in `v4` there are no task numbers any longer. Up to now the mapping database used a task number to identify an address space. In `v4` the only way to identify a space is to use the pointer to its page directory (*space pointer*). To cope with the situation, three solutions were thinkable:

1. use space pointers in `v2`, too
2. generate task numbers for `v4`
3. make the mapdb entry ABI specific, using task numbers for `v2` and space pointers for `v4`.

Variant 1 would enlarge the mapping entries for `v2`, which was not desired. Number 2 has the advantage of small mapping entries in both ABIs, but there had to be included a whole task



depth depth in mapping tree

v_addr virtual address the mapping entry is for

s size (4M or 4K page)

space pointer / task number reference to the task the mapping entry is for

Figure 3.4: Layout of a Mapping Database Entry.

number management in the kernel for that purpose only. That was not desired, either. So I chose solution number 3, with the only disadvantage of a larger mapping entry in `v4` (Figure 3.4). That is considered to be not that bad, because the problem of kernel resource-management cannot be solved in saving some bits of mapping entries anyway.

3.8 Kernel Debugger

Fiasco has a built-in powerful low-level kernel debugger called JDB. From the strict point of view, the JDB does not actually belong to the microkernel. The kernel can also be built without the debugger.

The JDB can be used to set breakpoints and inspect or log almost everything that's going on in the kernel. That is realized by making the JDB class a friend of most of the kernel classes. However that causes that JDB highly relies on the layout of kernel-internal and ABI-specific data structures. That means that every modification in kernel code can break JDB. Solutions are adapting the JDB code, disabling the corresponding JDB feature, or build the kernel without JDB.

For the Fiasco/v4 project to be accomplished, doing without JDB was not really an option. Adapting the whole debugger code to `v4` was not possible, too. It would have been far too much effort, because the JDB is quite as much code as the rest of the kernel. So I implemented only the features that I needed to debug the `v4` kernel and the test applications. Fortunately, the JDB is composed of modules like the rest of the kernel is. That is the reason why it took relatively low effort to disable the modules that are not needed yet, and write `v4` versions for the others.

3.9 Implementation Process

This section will not be a whole documentation of the implementation process, it mentions only the major aspects that are of interest for future kernel and ABI development.

3.9.1 Main Approach

When I started with FIASCO/v4, there was already a v4 kernel available, *Pistachio* 0.1. The Pistachio project also contains a sigma0 server, a pingpong benchmark, and a boot loader. On the other side, there was FIASCO, implementing the v2 ABI. My approach was to take the Pistachio environment, replace the kernel by FIASCO and do the v4 implementation until the test program runs again.

3.9.2 Fiasco-UX

As the v4 implementation should work under FIASCO-UX as well as on native IA-32, I made use of FIASCO-UX for the most of the work. The UX kernel is considerably better suited for kernel development, because developers does not have to boot a test computer every time. Furthermore, the UX kernel provides a functioning `printf` from the very beginning, and there is the possibility to inspect the FIASCO-UX processes from outside, using the `/proc` filesystem on linux. For example, using this method the developer can look what memory a task has mapped.

On the other side, FIASCO-UX is not completely the same as native FIASCO. It took a lot of time to understand the differences and find out why something did not work on hardware, whereas it did fine on UX.

3.9.3 Building and Testing

The build system of FIASCO supports having an arbitrary count of build trees, where each one has its own kernel configuration. That was of great help when changes of generic kernel code had to be tested. However, as I did not have access to test boxes for all architectures FIASCO runs on, the test runs for these ports had to be done by the respective developers. Fortunately, the ABI-related code was not that enmeshed in the architecture dependent stuff, so that there was only a slight chance to accidentally break the ARM port.

3.9.4 Simultaneous Development

In former projects working on the FIASCO microkernel, it has been emphasized that it is a very difficult task to merge the development branch to the main branch when the project is finished. To cope with that, I kept my working copy up to date all the time and merged my changes as soon as possible. That method, of course, will not work if one makes major changes to the kernel that cannot be done piece by piece. For the FIASCO/v4 project, admittedly, it greatly served its purpose.

Chapter 4

Measurements

This chapter analyzes the performance of the Fiasco/v4 implementation. It is intended to be an overview only, as a detailed performance analysis is beyond the scope of this work. As test applications I used the pingpong benchmark programs that were already available for both `v2` and `v4`. The pingpong benchmark measures the round-trip time of an IPC message.

Table 4.1 shows the comparison of Fiasco/v4 and Pistachio. The performance penalty of Fiasco is considered to be due to its interruptible IPC path. The comparison of Fiasco/v4 and Fiasco/v2 is shown in Table 4.2. Fiasco/v2 provides an *IPC shortcut*. A shortcut is an optimized version of the IPC path the kernel uses if a particular set of boundary conditions is fulfilled.[1] Actually Fiasco/v2 provides two shortcuts, one written in C++, the other written in assembler. To get comparable values, I switched off the assembler shortcut. The `noshort` values are measured without any shortcut, whereas the other ones use the C++ shortcut.

If more than two dwords are to be transmitted in an IPC message, `v2` needs to apply a long IPC. Table 4.3 shows the benefit `v4` has in that case.

[1]For a detailed description where the shortcut gets applied, look at the Fiasco source code.

Machine	IPC type	Fiasco/v4		Pistachio	
		Cycles	Time	Cycles	Time
Intel Celeron 566 MHz	4 Dwords Intra-Task	1728	3.05 μs	402	0.71 μs
	20 Dwords Intra-Task	1837	3.25 μs	439	0.78 μs
	60 Dwords Intra-Task	2076	3.67 μs	573	1.01 μs
	4 Dwords Inter-Task	1891	3.34 μs	618	1.09 μs
	20 Dwords Inter-Task	1998	3.53 μs	654	1.16 μs
	60 Dwords Inter-Task	2239	3.96 μs	786	1.39 μs
AMD Duron 1.3 GHz	4 Dwords Intra-Task	1097	0.84 μs	418	0.32 μs
	20 Dwords Intra-Task	1206	0.93 μs	442	0.34 μs
	60 Dwords Intra-Task	1406	1.08 μs	494	0.38 μs
	4 Dwords Inter-Task	1063	0.82 μs	1055	0.81 μs
	20 Dwords Inter-Task	1190	0.92 μs	1092	0.84 μs
	60 Dwords Inter-Task	1386	1.07 μs	1145	0.88 μs

Table 4.1: IPC Performance of Fiasco/v4 and Pistachio.

Machine	Env.	IPC type	Fiasco/v4		Fiasco/v2		Fiasco/v2 noshort	
			Cycles	Time	Cycles	Time	Cycles	Time
Celeron 566	Native	Intra-Task	1728	3.05 μs	743	1.13 μs	2424	4.28 μs
		Inter-Task	1891	3.34 μs	1038	1.83 μs	2655	4.53 μs
Duron 1300	Fiasco-UX	Intra-Task	120000	92.3 μs	61000	46.9 μs	166000	128 μs
		Inter-Task	120000	92.3 μs	62000	47.7 μs	166000	128 μs

Table 4.2: IPC Performance of Fiasco/v4 and Fiasco/v2.

Machine	IPC type	Fiasco/v4		Fiasco/v2	
		cycles	time	cycles	time
Celeron 566	20 Dwords Inter-Task	2000	3.53 μs	4757	8.40 μs
	60 Dwords Inter-Task	2239	3.96 μs	7517	13.3 μs

Table 4.3: IPC Performance of Fiasco/v4 and Fiasco/v2 with `v2` Long IPC.

Chapter 5

Conclusions, Open Topics, and Future Work

The development of Fiasco/v4 has shown that it is possible to implement two highly varying ABIs in one and the same code base, whereas large parts of the code can be reused. The structuring of the code into modules and submodules is a great help when implementing new kernel features. On the other hand, in case of the `v4` implementation many kernel modules had to be changed, whereas most of the changes were only small ones. Thats why I had to dig into all parts of the kernel anyway. The question if it is better to hack a new kernel or to reuse an old one and implement the new features in its code base cannot finally be answered. It depends on the amount of changes requested. As long as the ABIs we want to implement are in some degree similar to the now used L4 interface, I consider the Fiasco code base to be a good starting point for future development.

5.1 Future Work

The following topics are considered to be dealt with next:

- complete JDB support for `v4`
- test more complex `L4v4` applications and complete the Fiasco `v4` support if neccessary
- implementation of `v4` long IPC
- implementation of `v4` exception IPC
- implementation of `v4` local IPC
- optimizations in the `v4` IPC path
- extend Fiasco to support the ABI versions currently in development

5.2 Open Topics

5.2.1 Covert Channel in the UTCB Pointer Page

The way of showing the user its UTCB address in the current implementation opens a covert channel between all user threads. The threads can access the UTCB pointer page also through

their data segment, where the limit of the GS segment does not apply. They have read/write access to the whole page. For now this does not matter, because there is no limitation for communication between threads anyway. The channel can be closed by shrinking the data segment for the user to avoid covering the physical address of the UTCB pointer page.

5.2.2 Thread ID Data Type

The use of one data type for both local and global thread IDs (see Section 3.2.1) was found to be way to error-prone. Reusing the `v2` interface of the `L4_uid` data type was a well-meant design decision, but a problem arose. Now we have a sole data type, but only a subset of the access functions work properly. The others silently fail or return wrong values. This invites trouble and has no advantages besides that. The solution will be using three types for thread IDs. One for local IDs, a second one for global IDs, and a third one, which detects with what type of ID we deal with.

5.3 Summary

With Fiasco/v4 we got an implementation of the `v4` ABI that is suitable to run simple test applications. It is still incomplete, so that it is impossible to run complex `v4` applications such as L^{4}Linux/v4, but achieving that is considered to be just a small step.

As the second achievement, the Fiasco code got split up into ABI specific and generic parts. So now it should be easier to use the Fiasco kernel in future L4 ABI development.

The Fiasco/v4 port has been merged into the main Fiasco source tree and is available via remote CVS at `http://os.inf.tu-dresden.de/fiasco/download.html`.

Glossary

ABI Application Binary Interface

API Application Program Interface

CPU Central Processing Unit

DROPS Dresden Realtime Operating System

dword Double-Word, a 32 bit value

ex_regs Exchange Registers

Fpage Flexpage, region of virtual address space

IA-32 Intel 32-bit Architecture, aka x86

ID Identifier

IPC Inter Process Communication (Message)

IRQ Interrupt Request

KIP Kernel Interface Page

LIPC Local IPC, intra task IPC without kernel entry

MapDB Mapping Database

NIL Not in List

regs Registers

UID Unique Identifier

UTCB Userlevel Thread Control Block

TCB Thread Control Block

QoS Quality of Service

v2 L4 ABI Version 2

v4 Upcoming L4 ABI Version 4, aka x.2

x.0 Experimental L4 ABI Version, based on v2

x.2 See v4

Bibliography

[DLSU04] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: http://l4hq.org/docs/manuals/.

[Haz] Hazelnut – performance evaluation. Available from URL: http://www.l4ka.org/projects/hazelnut/eval.asp.

[Hoh96] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/.

[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. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[Hoh04] M. Hohmuth. Preprocess – A Preprocessor for C and C++ Modules, 2004. Web site: http://os.inf.tu-dresden.de/~hohmuth/prj/preprocess.

[Lie91] J. Liedtke. Clans & chiefs, a new kernel level concept for operating systems. Arbeitspapiere der GMD No. 579, GMD — German National Research Center for Information Technology, Sankt Augustin, 1991.

[Lie95] J. Liedtke. Towards real μ-kernels. Arbeitspapiere der GMD No. 958, GMD — German National Research Center for Information Technology, Sankt Augustin, 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.

[LW] J. Liedtke and H. Wenske. Lazy process switching. Technical report.

[Ste02] U. Steinberg. Fiasco microkernel user-mode port, 2002.

[War02] A. Warg. Portierung von Fiasco auf IA-64, 2002.

[War03] A. Warg. Software structure and portability of the fiasco microkernel, 2003.