Министерство образования РФ

Хабаровский государственный технический университет

Кафедра «Автоматика и системотехника»
Специальность 071900 «Информационные системы»

«ДОПУСТИТЬ К ЗАЩИТЕ»

Зав. кафедрой ___________ Чье Ен Ун
подпись ФИО

«___» __________________ 2004 г.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту

на тему *Документирование среды программирования для операционной системы L4Ka*

Проект выполнил ____________ Чебиряк Ю. А.
подпись ФИО
«_____»____________2004 г.
дата

Руководитель проекта ____________ Шалобанов С. В.
подпись ФИО
«_____»____________2004 г.
дата

Нормоконтроль ____________ Студеникин Ю. Е.
подпись ФИО
«_____» ____________2004 г.
дата

Консультанты:

По основной части ____________ ____________
подпись ФИО
«_____» ____________ 2004 г.
дата

По экономической части ____________ ____________
подпись ФИО
«_____» ____________2004 г.
дата

По охране труда ____________ ____________
подпись ФИО
«_____» ____________ 2004 г.
дата

По ____________ ____________ ____________
подпись ФИО
«_____» ____________ 2004 г.
дата

Хабаровск-2004

Abstract

The final qualifying work contains the report with 122 pages in A4 format, including 29 figures, 6 tables, and 23 references.

OPERATING SYSTEM, KERNEL, MICROKERNEL, FLEX-PAGE, PAGER, SCHEDULING, THREAD, TASK, ADDRESS SPACE, MAPPING, GRANTING, INTER-PROCESS COMMUNICATION

There is a project at our chair that aims to verify a complete implementation of a microkernel OS. The goal of this thesis is to discover all the nasty details of the L4Ka::Hazelnut system calls, helping the developers of this new (to be verified) microkernel to lay down a clearer and simpler interface for the user application.

The results obtained in this thesis have showed that L4Ka::Hazelnut microkernel not only has negligible errors in implementation, but also vulnerable to malicious behavior of user-level tasks. Thus, even main principles of it fail. For example, there are some Denial of Service attacks possible via page fault mechanism provided by L4Ka kernel. Scheduling principle also vulnerable to Denial of Service attacks.

For all these reasons, our chair develops new microkernel operating system, which in the near future will be proven. Safety will be provided by correctness and liveness proof of processor VAMP (DLX processor with pipeline), programming language C0 (subset of C), compiler and operating system SOS (Simple Operating System). Last two to be written in C0, guarantee the reliability. All parts of a system will be proven in Isabelle (automated theorem proover).

Contents

Introduction **7**

1 Operating Systems **9**

1.1 Classification by Kernel Architecture 9

1.1.1 The OS without Kernel 9

1.1.2 OS with a Kernel 10

1.2 Monolithic Kernels 10

1.2.1 Critique of Monolithic Kernel Systems 11

1.3 Microkernels 12

2 L4Ka **14**

2.1 Design Philosophy 14

2.1.1 Flexible Model 14

2.2 Kernel Mechanisms 15

2.2.1 Address Space 16

2.2.2 Inter-process Communication 17

2.2.3 Threads 17

2.2.4 Address Space and Mapping (revisited) 18

2.2.5 Scheduling 19

2.2.6 UID 20

2.2.7 Resource Allocation 20

2.3 L4Ka::Hazelnut 20

2.3.1 Relevance to the Thesis 21

3 L4Ka Data Types **22**

3.1 Getting Started 22

3.2 Base Data Types 22

3.3 Unique Ids 23
3.4 Flex-pages 26
3.4.1 IO-Ports 30
3.5 Timeout 30
3.6 IPC Result Status 33
3.7 Schedule Parameter Word 36

4 L4Ka System Calls 38
4.1 Overview 38
4.2 Task Creation and Deletion 38
4.3 Thread Related System Calls 39
4.3.1 Thread Manipulation 39
4.3.2 Release CPU 40
4.3.3 Thread Scheduling 40
4.4 Revoking Mappings 42
4.5 C Interface 44
4.5.1 l4_myself 44
4.5.2 l4_fpage_unmap 44
4.5.3 l4_thread_switch 45
4.5.4 l4_thread_ex_regs 45
4.5.5 l4_thread_schedule 46
4.5.6 l4_task_new 47
4.5.7 l4_ipc_call 48
4.6 Examples 52
4.6.1 Thread Creation 52
4.6.2 Task Creation 54
4.6.3 Thread Scheduling 58

5 L4Ka IPC 68
5.1 Why Inter-Process Communication? 68
5.2 IPC Overview 69
5.3 Message Types 69
5.4 Sending/Receiving IPC Messages 70
5.5 L4Ka IPC Messages 71

5.5.1 Message Descriptors . . . 71
5.5.2 Short Messages . . . 75
5.5.3 Long Messages . . . 78
5.5.4 Message Header . . . 79
5.5.5 Message `mwords` . . . 80
5.5.6 String Dopes . . . 82
5.5.7 Generic Long IPC Message . . . 83
5.6 L4Ka IPC Message Summary . . . 83
5.6.1 Send/Receive Protocol . . . 86
5.6.2 Sending Message (Generic Algorithm) . . . 86
5.6.3 Receiving Message (Generic Algorithm) . . . 87
5.7 Examples . . . 87
5.7.1 Short IPC Messages between Threads . . . 87
5.7.2 Long IPC Messages between Threads . . . 91

6 L4ka Additional Information 97
6.1 Bootstrap . . . 97
6.2 Page Fault Handling . . . 97
6.3 The Main Pager σ_0 . . . 100
6.3.1 σ_0 Protocol . . . 100
6.4 Interrupt Handling . . . 101
6.5 Preemption . . . 102
6.6 Examples . . . 102
6.6.1 σ_0 Extension . . . 102
6.6.2 Interrupt Handling . . . 117

7 Conclusions and Future Work 120

Introduction

Most operating systems are either *microkernel-based* systems or *macrokernel* (or *monolithic kernel*) based systems. Early system designers did not consider the need to keep kernels small. But in current time it is essential. For example, dependable systems need to operate quick and reliable.

As a result, the kernel is reduced to a bare minimum set of primitives and abstractions without compromising on the variety of applications it may serve. Other functionality is implemented at the user level as external servers, giving rise to a client-server architecture in the operating system. Such an operating systems (with small kernel) are called *microkernel-based*. There are already exist examples of microkernel-based operating systems.

L4Ka is being one of them. L4Ka is designed with three primary goals in mind: minimality, generality and performance. A minimalist approach assists in the generality of the system, since few assumptions and policies are enforced by the kernel. One of the intentions of L4Ka is to be the nucleus of a wide range of full-featured operating systems.

At the time of writing, there is no microkernel that has been formally proven to be strictly adhering to its specification. And this is too complex to be done on monolithic kernels.

There is a project at our chair that aims to verify a complete implementation of a microkernel OS. The goal of this thesis is discover all the nasty details of the L4Ka::Hazelnut system calls, helping the developers of this new (to be verified) microkernel to lay down a clearer and simpler interface for the user application.

Below is a brief overview of chapter:

Chapter 1. Operating Systems. In this chapter a definition of the operating system is given. Classification of operating systems by the kernel architecture is presented.

Chapter 2. L4Ka. The design philosophy of L4Ka microkernel operating system is given. All the main mechanisms are described.

Chapter 3. L4Ka Data Types. In this chapter all the data types used by L4Ka to operate are described precisely.

Chapter 4. L4Ka System Calls. In this chapter system calls of L4Ka are described. Also C interface for them is presented.

Chapter 5. L4Ka IPC. The main mechanisms of L4Ka, Inter-process Communication, is described in details.

Chapter 6. L4Ka Additional Information. In this chapter some additional information about L4Ka is gathered.

1 Operating Systems

An operating system is a program, which drives the resources of the system and provides the interface for applications programs that use the system.

Managing the hardware resources is needed, because various programs compete for the attention of the central processing unit (CPU), demand memory, storage and input/output (I/O) bandwidth for their own purposes. The operating system makes sure that each application gets the necessary resources.

Providing a consistent application interface, is necessary to hide the complexity and the variety of the hardware to the software. A consistent application program interface (API) allows a software developer to write an application on one computer and have a high level of confidence that it will run on another computer of the same type, even if the amount of memory or the quantity of storage is different on the two machines.

1.1 Classification by Kernel Architecture

Operating Systems can be written so that most services are moved outside the OS core and implemented as processes. This OS core then becomes a lot smaller, and called "kernel". When this kernel only provides the basic services, such as basic memory management and multithreading, it is called a microkernel or even nanokernel for the super-small ones. To stress the difference between the Unix-type of OS, the Unix-like core is called a monolithic kernel. A monolithic kernel provides full process management, device drivers, file systems, network access etc.

1.1.1 The OS without Kernel

Not all operating systems have a "kernel" which is protected from user programs and which manages the hardware and the user programs. Some operating systems, the early ones, just provided some interface to the hardware programs could run on, but did

not protect themselves from these programs or did not offer to protect the programs from each other.

1.1.2 OS with a Kernel

This architecture evolved to an OS design with two rings: one ring running in system mode, and a ring running in user mode. The kernel has full control over the hardware and provides abstractions for the processes running in user mode. A process running in user mode cannot access the hardware, and must use the abstractions provided by the kernel. It can call certain services of the kernel by making "system calls" or kernel calls. The kernel only offers the basic services. All others are provided by programs running in user mode.

There are several ways to define the kernel of an operating system:

- **mandatory** — the kernel denotes the essential part of the operating system common to all other software [1];
- **fundamental** — the kernel provides the most fundamental features upon which the rest of the operating system relies on. The system cannot function if it is deprived of kernel services;
- **privileged** — the kernel runs in privileged mode, therefore it is able to access the hardware in every way possible. No other mechanisms control the kernel except itself;
- **irreplaceable** — the kernel is the static part of the system that is cannot be modified or replaced during run-time. There cannot be two competing versions of the kernel in the operating system at once.

Most operating systems are either *microkernel-based* systems or *macrokernel* (or *monolithic kernel*) based systems. The distinction lies in the design of the operating system, in particular how much functionality is to be offered by the kernel. This design decision invariably affects the size of the kernel, hence the naming. Traditionally kernels are monolithic in nature, and it remains the more popular approach today. The microkernel approach is newer but less commonly used.

1.2 Monolithic Kernels

Early system designers did not consider the need to keep kernels small. Later operating systems were based on older ones for code reuse and backwards compatibility, while

providing many new features and satisfying new requirements that older systems did not have. As a result, modern kernel have become much larger than earlier ones. Different aspects of functionality are built on and dependent on each other in a layered structure, which gives rise to the name of monolithic kernels.

Although monolithic approach is the older of the two, it is still used in most operating systems, an example is mainstream Linux. This is due to a variety of reasons. Both code and design of older operating systems are often recycled in its newer versions to save development time and ensure compatibility within the lineage. This is especially important for systems that are widely used. Another major reason is the difficulties encountered by microkernel research in its early stages. The idea of microkernel design was considered to be very appealing by the systems research community. However, first generation microkernels did not deliver on promises theoretically achievable by this approach [1]. To an extent the community has shunned the microkernel approach as a practical way of building operating systems, albeit prematurely. Without a better alternative, designers retain the old paradigm.

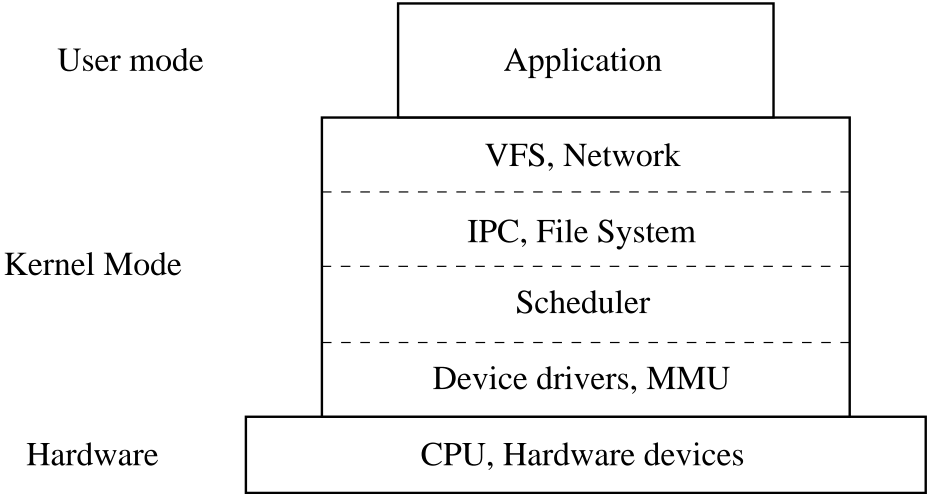

Figure 1.1: Monolithic Kernel Layout (adopted from [2])

1.2.1 Critique of Monolithic Kernel Systems

While monolithic kernel design is the prevalent approach today, it is not without its problems. Monolithic kernels are seldom modular, therefore it is difficult to separate individual kernel components that provide independent features. Rather, components are tightly coupled to each other. Strong coupling and co-dependencies are detrimental to system development and maintenance, making feature enhancements more difficult. As requirements for modern operating systems accumulate, maintenance consumes a dominant

proportion of development cost. In addition, the complexity of the kernel makes it practically impossible to mathematically prove that the system is secure for reliability-critical applications.

Monolithic kernels also lack flexibility from the constraints of the initial design. The more assumptions a kernel makes about the underlying hardware architecture, the harder it is for the kernel to be ported to a platform other than the initial one its is designed for. It is also more difficult for a kernel that is highly optimized for a particular usage to adapt to another purpose. This is an even worse problem for monolithic kernels, since a larger kernel invariably makes more assumptions about its purpose, as manifested in the data structures, algorithms and optimizations used within the system.

1.3 Microkernels

As an increasing number of requirements are demanded from the operating system, advances in software development technologies permitted new and novel system designs that are able to handle a range of different requirements. A natural progression form the singular, large and layered monolithic operating system design is to separate, segregate and modularize the system architecture.

As a result, the kernel is reduced to a bare minimum set of primitives and abstractions without compromising on the variety of applications it may serve. Other functionality is implemented at the user level as external servers, giving rise to a client-server architecture in the operating system.

Microkernel layout is depicted in Figure 1.2. Microkernel (μ-kernel) provides minimum set of primitives, other functionality is implemented at the user level. Several operating systems or services (e-mail server, ftp daemon, etc) can be placed on top.

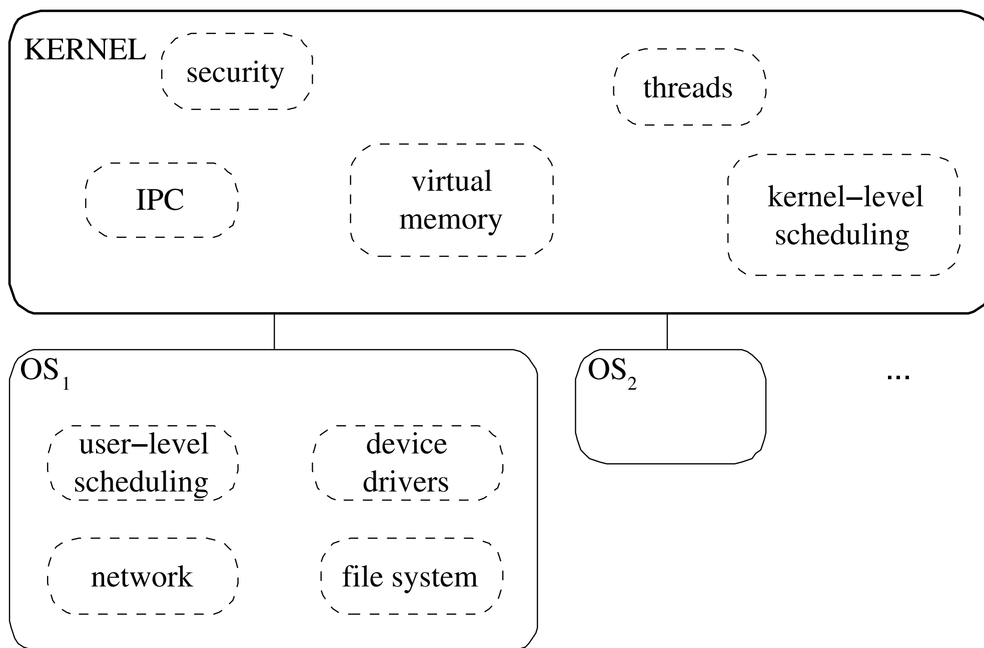

Figure 1.2: Microkernel Layout.

2 L4Ka

The L4 microkernel project was originally founded by Jochen Liedtke in the 1990's. It is currently headed by the L4Ka team based at the University of Karlsruhe, Germany. It is one of the most active microkernel projects at present, with a number of related and client projects — for example, Mungi [3] and Sawmill Prime [4]. L4 is formally specified by a platform-independent specification; implementations exist on many different platforms. The first available implementation of version 4 was developed by the L4Ka team in collaboration with the DiSy group at the University of New South Wales, codenamed Pistachio [5].

Currently, Pistachio runs on a wide variety of architectures, such as IA-32, IA-64, PowerPC-32, Alpha 21164/21264, MIPS, and StrongARM (see [5]). Recently, it has also been ported to the PowerPC-64 architecture.

This chapter describes the L4Ka microkernel system and its design. A brief summary of the mechanisms and abstractions provided by the L4Ka specification is presented in this chapter.

2.1 Design Philosophy

L4Ka is designed with three primary goals in mind: minimality, generality and performance [6]. These three criteria dictate the way L4Ka is designed and built. L4Ka avoids the problems of earlier microkernels, which were relatively large and bloated, and aims to include a minimal set of features able to derive the maximum set of functionality outside of the core kernel. A minimalist approach assists in the generality of the system, since few assumptions and policies are enforced by the kernel. One of the intentions of L4Ka is to be the nucleus of a wide range of full-featured operating systems.

2.1.1 Flexible Model

The minimal primitives provided by L4Ka were designed to support many different system architectures on top of it. OS personality uses system calls of a μ-kernel and

hence can abstract from specific of particular hardware. Thus, in order to run OS on another architecture it is only necessary to rewrite the back-end of a μ-kernel.

The general IPC mechanism may derive different types of communication. A natural way of utilizing L4Ka IPC is to implement a client-server protocol. A microkernel with efficient communication primitives makes a strong case for a multi-server operating system. When IPC is combined with mapping mechanisms, shared-memory communication is also possible across different processes.

The user has fine-grained control over system policies such as paging and interrupt handling. L4Ka imposes few restrictions on these mechanism and delegates control to the user via *pager*. Pager is a specific process, which handles page faults occurred in the *faulter* process. This delegation allows a lot of flexibility for the user to set specific policies at the operating system level.

Flexibility is also provided by data structures and system calls. As an input parameters for those system calls L4Ka uses special data types (hereinafter called *base data types*). Base data types of L4Ka can be used in order to express many different entities (e.g. UID which can stand for identifier of thread, task or interrupt) which makes kernel faster to operate, but complicates the usage. For what concerns system calls, there are only seven of them. For example, generic `l4_ipc_call` incorporates primitives for *Send*, *Receive from*, *Send to & Receive* (combined send and receive operation), *Open Receive* (receive from arbitrary client thread) operations, interrupt handling mechanism, and *Sleep* function.

2.2 Kernel Mechanisms

The L4Ka μ-kernel provides a small set of mechanisms:

- **thread/task management**;
- **inter-process communication (IPC)** — the fundamental mode of communication between threads;
- **address spaces and mapping** — a protected set of translations from virtual addresses to physical memory;
- **scheduling** — control over the order and duration of execution of runnable threads.

Other functionality traditionally found in the kernel, such as file systems, device drivers and networking, is built on top of these primitives on the user level. Advanced

services such as distribution, security policies and persistence, are by no means restricted by the simplicity of these mechanisms.

2.2.1 Address Space

An address space contains all the data (other than hardware registers) which are directly accessible by a thread. An address space is a set of translations from virtual to physical memory (which is partial in the sense that many such translations are undefined, making the corresponding virtual memory inaccessible). Address spaces in L4Ka can be recursively constructed: a thread can *map* or *grant* parts of its address space into another thread's address space.

Mapping means that mapper and mappee will share the region of the address space. Mappings can be revoked at any time, so the mapper retains full control. Alternatively, virtual address space can be *granted* to a different address space. In this case, the granter relinquishes all control over the data mapped by that part of the address space and no longer has a valid mapping for that address space region. The granter cannot revoke a grant. The grantee, in contrast, inherits full control over the granted address space (unless the grant was read-only, in which case write access is lost.)

Figure 2.1 illustrates the difference between mapping and granting operations: after grant operation page is no longer represented in the address space of the controller pager.

Figure 2.1: L4Ka Address Spaces and Mapping (adopted from [7])

Note that while a grant is irreversible, the granter, in general, has received the ad-

dress space (directly or indirectly) via mapping, and an address space at the beginning of the mapping chain can still revoke the mapping.

Mapping and granting are implemented as operations on page tables, without copying any actual data. Mapping and granting are achieved as a side effect of IPC operations and specified by the means of flex-pages. This is not accidental: for security reasons mapping requires an agreement between sender and receiver, and thus requires IPC anyway.

The concept of a *task* is essentially equivalent to that of an address space. In L4Ka, a *task is the set of threads sharing an address space*. Creating a new task produces an address space with one running thread. Strictly speaking, the number of tasks is a constant. There are two kinds of tasks: active and inactive ones. Creation of task means activation of an inactive one. Inactive tasks are essentially capabilities (task creation rights). This is important, as a thread can only create a task if it already owns the task ID to use.

2.2.2 Inter-process Communication

IPC is the fundamental mode of communication and synchronization in L4Ka, and is also used to transfer access to memory between address spaces. It is the basis for many other types of operations in the system, therefore it is essential that IPC is implemented efficiently. This is particularly crucial in μ-kernel-based systems, since the amount of communication required between components is higher compared to monolithic systems.

The L4Ka μ-kernel provides a total of seven system calls (IPC is one of them). Everything else must be built on top, implemented by server threads, which communicate with their clients via IPC. IPC is used to pass data by value (i.e., the L4Ka μ-kernel copies the data between two address spaces) or by reference (using mapping or granting). IPC is also used for synchronization (as it is blocking, so each successful IPC operation results in a rendez-vous), wakeup-calls (as timeouts can be specified), pager invocation (the L4Ka μ-kernel converts a page fault into an IPC to a user-level pager), and interrupt handling (the L4Ka μ-kernel converts an interrupt fault into an IPC to a user-level interrupt-handler from a pseudo-thread). Device control is registered via IPC (although actual device access is memory mapped).

2.2.3 Threads

A thread is the basic execution abstraction. A thread has an address space (shared with the other threads belonging to the same task), a UID, a register set (including an in-

struction pointer and a stack pointer), and a page fault handler (pager). IPC operations are addressed to threads (via their UIDs). Threads are extremely light-weight and cheap to create, destroy, start and stop. The lightweight thread concept, together with very fast IPC, is the key to the efficiency of L4Ka and OS personalities built on top.

Multiple threads can share the same address space. The combination of an address space and a set of threads is known as a task in L4Ka. A thread is able to manipulate the registers of other threads in its own address space/task. When a new task is created, the first thread is created automatically by L4Ka. This thread may then create other threads.

Pager Threads

Pagers are responsible for handling page faults generated by their subject threads. When a thread generates a page fault, an IPC containing the faulting memory and instruction address is sent to the pager. The pager then determines the memory region to grant to the faulting thread and sends a reply IPC back to the thread. The kernel intervenes during the reply to update the relevant data structures. The mechanism to transfer memory mappings across different address spaces is described in Section 2.2.1.

The elegance of this model is that paging policies are configurable by the user, so a variety of memory management policies can be implemented. In addition, policies can be easily changed on-the-fly by switching to a different pager threads. Pagers are also responsible for starting a newly created thread. It does so by sending an IPC containing its initial stack and instruction pointer. For more detailed picture of such interaction, see Figure 6.2.

Interrupt Handling

Interrupts are abstracted via IPC as well. A thread can ask L4Ka (again, via IPC) to become the handler of a given type of interrupt, handed out by L4Ka on a first come basis. When an interrupt of the given type occurs, it is transformed into an IPC message, which is then forwarded to the handling thread.

2.2.4 Address Space and Mapping (revisited)

The address space is empty upon creation, and it must gain access to memory by mapping from other address spaces. Mapping operations allow address spaces to share parts of it with other address spaces. The root address space, known as σ_0, is the very first address space and is a direct mapping onto the physical memory of the system. Sub-

sequently created address spaces gain access to physical memory by mapping directly or indirectly onto σ_0.

There are no restrictions on the depth of mapping. In other words, mappings can be recursively propagated down several address spaces. On each mapping, however, access rights can only be a subset of ones already owned by the mapper. The kernel maintains a hierarchical mapping database that contains the tree representing all mappings in the system. This is necessary since unmapping operations causes the recursive removal of all mappings under it. Memory regions are expressed in L4Ka as *flex-pages* (*fpages*), which contains the base address and size of the region. All flex-pages must have a size of a power of two.

Mappings operations can be a *grant*, where the mapper removes its own mapping from its parent and transfers it completely to the mappee. In effect, this reduces the number of unnecessary mappings across layers of address spaces.

Protocol of σ_0

σ_0 theoretically accepts requests for physical memory from any thread. In practice, root pagers are written in such a way that they request all physical memory in the system during initialization time, not lazily upon request. This ensures that no user thread may steal memory from the system. This also allows partitioning of the physical memory into discrete portions such that even multiple operating systems can be run on top of L4Ka concurrently on the same machine without interfering each other [8].

2.2.5 Scheduling

The L4Ka kernel supports scheduling at two levels. The kernel employs a preemptive, hard-priority round-robin thread scheduler, beyond the control of the user. At the user level, a thread's priority and its time slice may be controlled and allocated. Every thread has two time slice values associated with it: the current time slice and the total quantum, the sum of all time slices. The current time slice determines how long a thread may run before the next preemptive intervention from the kernel, and the total quantum is deducted every time the thread is scheduled. When a thread has exhausted its total quantum its scheduler will be notified.

Arbitrary scheduling strategies can be implemented on top by modifying the threads' priorities and time slices. These policies and related data are to be kept in the scheduler thread's memory. For more detailed information, see 4.3.3 and [9].

2.2.6 UID

A UID of a thread is that of its task plus the number of the thread within the task (sometimes called lthread or local thread number). The UID of a task consists of the task number, some fields describing its place in the task hierarchy, and a version number. Both, tasks and threads are limited (in L4Ka for x86 there are 255 tasks and within each task there are 64 threads). This means that tasks (or address spaces) and threads must be recycled. The L4Ka μ-kernel ensures uniqueness of task IDs by incrementing the version number whenever a task number is reused. Obviously, both thread and task numbers are insufficient for a real multi-user operating system. This means that an OS personality will in general need to map its own task and thread abstraction onto L4Ka's. How this is done is up to the OS personality, L4Ka only provides the tools.

2.2.7 Resource Allocation

As was mentioned above, the classical job of the OS is resource allocation. In a μ-kernel-based system, this is left to the OS personality, the L4Ka μ-kernel only provides the tools, and enforces security. L4Ka's view of resources is simple: each resource is allocated on a first-come-first-served basis. This is by no means a free-for-all: The servers implementing OS personalities are started at boot time by the L4Ka μ-kernel. As they are the first running tasks, they have the chance to allocate all resources to themselves before any "user" tasks exist. Initial servers must be contained in the L4Ka boot image, and must be identified as to be started up by L4Ka.

2.3 L4Ka::Hazelnut

L4Ka::Hazelnut was designed to be portable across 32bit platforms [10]. L4Ka separated general code like IPC, thread management, and scheduling from platform dependent code like pagetable management and exception handling.

The L4Ka::Hazelnut microkernel is currently available for the following processor families: IA32 (Pentium and higher), ARM (StrongARM SA110 and SA1100) [10].

2.3.1 Relevance to the Thesis

At the time of writing, there is no microkernel that has been formally proven to be strictly adhering to its specification. However, this is almost certainly has very large complexity on monolithic kernels.

There is a project at our chair that aims to verify a complete implementation of a microkernel OS. The goal of this thesis is to discover all the nasty details of the L4Ka::Hazelnut system calls, helping the developers of this new (to be verified) microkernel to lay down a clearer and simpler interface for the user application.

For testing purposes L4Ka::Hazelnut/x86 was used on a VMWare system.

3 L4Ka Data Types

In this chapter fundamental data structures used by L4Ka μ-kernel to operate are presented. They are provided as input parameters for the system calls or returned in registers. Almost all types are complex and have many fields, which are explained in details in this chapter.

3.1 Getting Started

At the home page of the L4 community can be found all the kernels, some L4 environments and L4 based operating systems[1]. Development of L4Ka is done by System Architecture Group in University of Karlsruhe.

After extracting source code kernel must be configured properly. In order to boot up L4Ka kernel and all built applications must be copied to boot image.

The main work of a user-level programmer is to write C code for the root task, implementing OS personality on top of L4Ka. It is located in `apps/root_task/main.c`. In order to use the system calls and data types the user-level programmer should include system files (usually "`l4/l4.h`", "`l4io.h`" and "`l4/helpers.h`" is enough). They are located in `apps/include` directory (and its subdirectories).

3.2 Base Data Types

Base data types declarations are concentrated in files "`l4.h`", "`types.h`" and "`ipc.h`". Also file "`helper.h`" can be helpful. It contains some wrappers for generic L4Ka system calls. Almost all definitions are taken from these files for x86 architecture. Base data types are shown in Listing 3.1.

[1]The home page of the L4 community is located at http://l4hq.org/. The L4Ka research project can be found at http://L4Ka.org/. L4Ka::Hazelnut kernel is freely downloadable from http://L4Ka.org/projects/hazelnut/download.php.

Listing 3.1: Base types declarations in “l4.h”

```
typedef unsigned long long  qword_t;
typedef unsigned int        dword_t;
typedef unsigned short      word_t;
typedef unsigned char       byte_t;
typedef signed long long    sqword_t;
typedef signed int          sdword_t;
typedef signed short        sword_t;
typedef signed char         sbyte_t;
typedef dword_t*            ptr_t;
```

Most frequently used types are dword_t (unsigned int) and ptr_t (pointer to unsigned integer). Since target architecture is 32-bit, almost all data structures are 32-bit aligned to increase performance. Hereinafter type dword_t is widely used for 32-bit parameters of functions in L4Ka.

3.3 Unique Ids

As any other multitasking operating system, L4Ka uses some data structures to identify the processes and applications. Unique ids serves as identifiers for threads, tasks and hardware interrupts.

Format

Each unique id is a 32-bit value, which is unique until the system is rebooted. An unique id is presented in Figure 3.1. This picture outlines certain fields, their size and position in a 32-bit word. Meaning of all the fields was mentioned above (see 2.2.6).

Table 3.1: Permitted Values for Fields of UID

Field	Quantity	Available values	Comments
Task	255	1-255	task 0 does not exist
Thread	64	0-63	
Version	1024	0-1023	

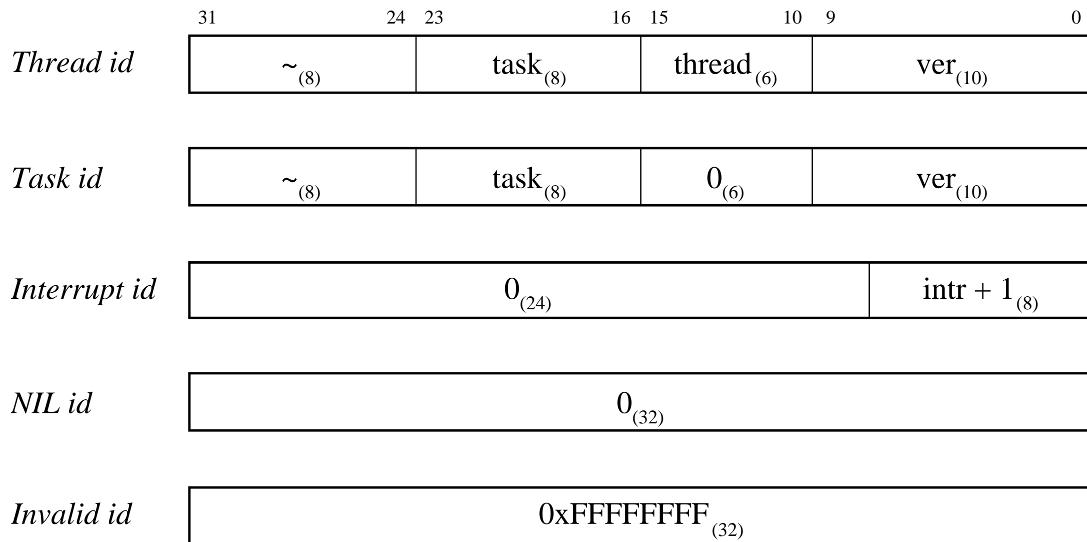

Figure 3.1: Unique Ids

x86 Implementation

Thread ids are defined in file “`l4.h`” and are presented in Listing 3.2. Note that chiefs and clans are not implemented in L4Ka, thus first field of unique ids no more meaningless.

Listing 3.2: Thread id implementation in “`l4.h`”

```
typedef union {
    struct {
        unsigned        version        : 10;
        unsigned        thread         : 6;
        unsigned        task           : 8;
        unsigned        chief          : 8;
    } id;
    dword_t raw;
} l4_threadid_t;
/*
 * Some well known thread id's.
 */
#define L4_KERNEL_ID    ((l4_threadid_t) { id : {0,1,0,0} })
#define L4_SIGMA0_ID    ((l4_threadid_t) { id : {1,0,2,4} })
#define L4_ROOT_TASK_ID ((l4_threadid_t) { id : {1,0,4,4} })
#define L4_INTERRUPT(x) ((l4_threadid_t) { raw : {x + 1 } })
#define L4_NIL_ID       ((l4_threadid_t) { raw :  0 })
#define L4_INVALID_ID   ((l4_threadid_t) { raw : ~0 })
#define l4_is_nil_id(id)        ((id).raw == L4_NIL_ID.raw)
#define l4_is_invalid_id(id)    ((id).raw == L4_INVALID_ID.raw)
```

Thread id is defined as a union `l4_thread_id_t`. Dword field `raw` allows access in a raw mode. A structure `id` contains all the fields as a bit vectors according to Figure 3.1.

Here are also some well known thread id's:

- **L4_SIGMA0_ID** specifies thread id of σ_0 pager;
- **L4_ROOT_TASK_ID** is task id of the *root task* (which starts after OS bootstrap and grabs all available memory from σ_0 pager);
- **L4_KERNEL_ID** is used to distinguish between the kernel and user-level threads (it is used inside the kernel and in σ_0, not necessary on user level);
- **L4_INTERRUPT** is a macros to define interrupt handler's thread id;
- **L4_NIL_ID** indicates, that the id of the task/thread is not valid, i.e. all entries are set to 0. It is used for example if the task creation fails (see Sections 4.2 and 4.5.6).

Since for interrupt handlers there is a special thread id format, it is declared in rather easier way in "`types.h`" as a structure `l4_intrid_struct_t`. For simplified access there is also union `l4_intrid_t`, which allows to use it in a raw mode through field `dw`. Implementation is shown in Listing 3.3.

Listing 3.3: Thread id for interrupt handlers in "`types.h`"

```
typedef struct {
  unsigned intr:8;
  unsigned char zero[3];
} l4_intrid_struct_t;

typedef union {
  dword_t dw;
  l4_intrid_struct_t id;
} l4_intrid_t;
```

Task id is defined in the same way as thread id (see Listing 3.4). The user must manually set `thread` field to zero value.

Listing 3.4: Task id implementation in "`types.h`"

```
typedef l4_threadid_t l4_taskid_t;
```

3.4 Flex-pages

Abstraction

Main abstraction of L4Ka operating system is contiguous regions of address space called flex-pages (fpages). Flex-pages are required for mapping and granting virtual memory. Flex-pages are specified by the mapper and received by the mappee as part of an IPC message. For each flex-page successfully received, the valid pages within that flex-page become a part of the receiver's address space (mapped or granted to it).

Definition

Flex-pages are regions of the virtual address space. A flex-page consists of all pages actually mapped in this region. A flex-page has the following properties:

- size is 2^s bytes. The smallest size allowed for any flex-page is the hardware page size. On the x86 processors, the smallest possible value for s is 12, therefore hardware pages are at least 4K;
- base address b of flex-page is aligned to 2^s (*base address* mod $2^s = 0$);
- contains all the *mapped* pages within the region described by the flex-page. That is, those pages which belong to the specified region *and* which have been mapped into the sender's virtual space.

Format

A flex-page is specified by providing the values b and s. The flex-page is then defined to be the region $[b \times 2^s, (b+1) \times 2^s]$. A flex-page with base address b and size 2^s is denoted by a 32-bit word as depicted in Figure 3.2.

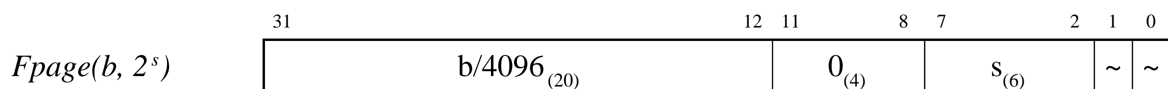

Figure 3.2: Flex-page Format

A flex-page, which covers complete user address space (with base address 0, size $2^{32} - K$, where K is the size of the kernel area) is denoted by $b = 0$, $s = 32$ (see Figure 3.3).

In addition, a hot-spot, h, is also required for the sender if the sender and receiver specify flex-pages of different sizes.

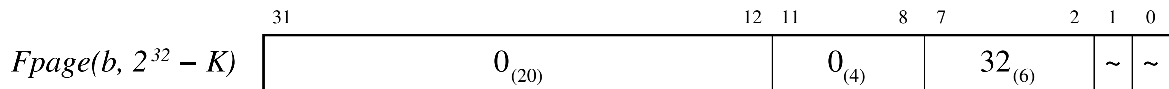

Figure 3.3: Complete User Address Space Format

Hot-spot

Hot-spot make sense only in the case of memory mappings. It is a 32-bit value followed by flex-page in IPC messages. Flex-page and hot-spot (followed by it) form a flex-page descriptor (see Section 5.5.5).

In the case, when sender and receiver specify flex-pages of different sizes, the hot-spot specification is used to determine how the mapping between the two different size flex-pages occurs.

If 2^s is the size of the larger, and 2^t the size of the smaller flex-pages, then the larger flex-page can be thought of as being tiled by 2^{s-t} flex-pages of the smaller size. One of these pages is uniquely identified as containing the *hot-spot* address (mod 2^s). This is the flex-page which will actually be mapped/granted.

Figure 3.4 gives two examples of mappings which involve different flex-page sizes. The figure also illustrates the fact that the receiver's flex-page allows the receiver to specify the window where mappings are permitted to occur (greater security). In first situation 1K page at address 5K is mapped into a 4K space at zero address. Hot-spot is used to determine into which 1K chunk of 4K page data have to be mapped. Hot-spot assigned to 2K (with some negligible offset). Thus, page is mapped at address 2K. Second illustration covers opposite situation: data to be mapped from 4K space into 1K. As before, the bigger page is tiled according to the size of the smaller one. And again hot-spot determines offset of chunk to be mapped.

A precise definition of the intuitive description above is now given for completeness. If the sender defines its flex-page as b, s, h and the receiver defines its flex-page as b', s' then the mappings for the three situations would be:

$s = s'$:	$b \times 2^s \longmapsto b' \times 2^s$,	hot-spot is not used;
$s' > s$:	$b \times 2^s \longmapsto b'_{[31:s']} h_{[s':s-1]} 0^s$,	the sender's flex-page is aligned around the hot-spot;
$s > s'$:	$b_{[31:s]} h_{[s-1:s']} 0^{s'} \longmapsto b' \times 2^{s'}$,	the receiver's flex-page is aligned around the hot-spot.

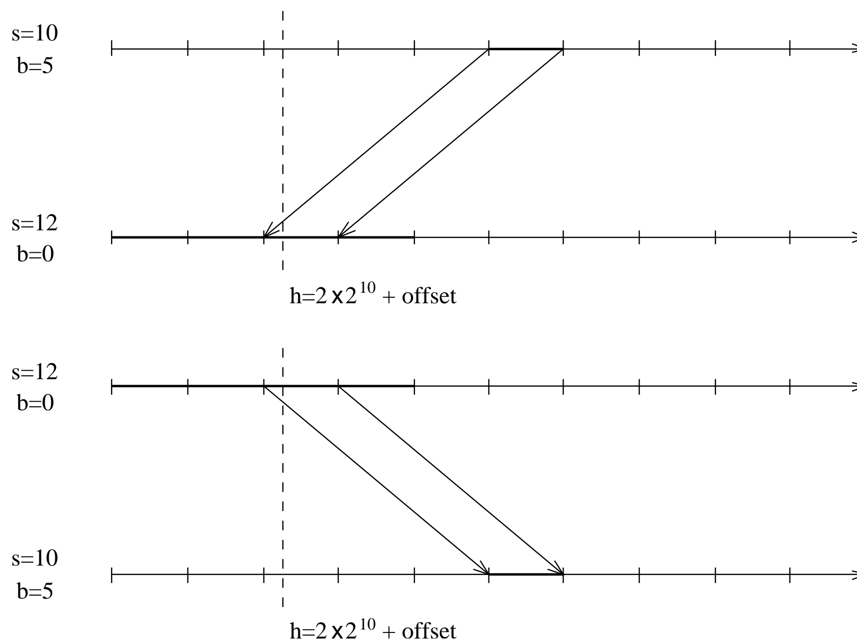

Figure 3.4: Flex-page Mapping Example Using Hot-spot

x86 Implementation

Flex-pages are implemented in L4Ka as a union `l4_fpage_t`. Listing 3.5 shows, that this union contains a structure `l4_fpage_struct_t` of bit vectors for each field, and integer `fpage` of type `dword_t` for a raw access.

Listing 3.5: Declarations concerning flex-pages in "types.h"

```
typedef struct {
  unsigned grant:1;
  unsigned write:1;
  unsigned size:6;
  unsigned zero:4;
  unsigned page:20;
} l4_fpage_struct_t;

typedef union {
    dword_t fpage;
    dword_t raw;
    l4_fpage_struct_t fp;
} l4_fpage_t;

#define L4_PAGESIZE               (0x1000)
#define L4_PAGEMASK               (~(L4_PAGESIZE - 1))
#define L4_LOG2_PAGESIZE          (12)
#define L4_SUPERPAGESIZE          (0x400000)
#define L4_SUPERPAGEMASK          (~(L4_SUPERPAGESIZE - 1))
```

```
#define L4_LOG2_SUPERPAGESIZE   (22)
#define L4_WHOLE_ADDRESS_SPACE  (32)
#define L4_FPAGE_RO      0
#define L4_FPAGE_RW      1
#define L4_FPAGE_MAP     0
#define L4_FPAGE_GRANT   1

L4_INLINE l4_fpage_t l4_fpage(unsigned long address, unsigned int size,
                                  unsigned char write, unsigned char grant)
{
  return ((l4_fpage_t){fp:{grant, write, size, 0,
                               (address & L4_PAGEMASK) >> 12 }});
}
```

There is one difference from definition in Figure 3.2: the least significant two bits are not meaningless in structure `l4_fpage_struct_t` and are defined as `grant` and `write` correspondingly. They will be explained and used later to determine operation on flex-page (*mapping*/*granting*) and permission of usage (*write*/*read-only*).

There are also some predefined constants in "`types.h`":

- **L4_PAGESIZE** is set to to 4Kbytes. This size of a flex-page corresponds to $s = 12$;
- **L4_LOG2_PAGESIZE** is then assigned to 12;
- **L4_PAGEMASK** is used to mask address. Let us consider an address a, which belongs to some 4K flex-page. Applying of ordinary **AND** operation with such a mask returns the starting address of this contiguous 4K flex-page;
- **L4_SUPERPAGESIZE** is set to 4Mbytes. This size of a flex-page corresponds to $s = 22$;
- **L4_LOG2_SUPERPAGESIZE** is then assigned to 22;
- **L4_WHOLE_ADDRESS_SPACE** is used to define *complete user address space*;
- **L4_FPAGE_RO = 0**, value for bit `write` for *read-only* permissions;
- **L4_FPAGE_RW = 1**, value for bit `write` in case of *write* and *read* permissions;
- **L4_FPAGE_MAP = 0**, value for bit `grant` to ensure *mapping* of this page;
- **L4_FPAGE_GRANT = 1**, value for bit `grant` in case of *granting* operation.

Function `l4_fpage` provides simple interface to form a flex-page from given parameters (it returns value of type `l4_fpage_t`).

3.4.1 IO-Ports

x86 IO-ports form a separate address space besides the conventional memory address space. Its size is 64K and its granularity is 16 bytes. However, IO-ports can only be mapped idempotent, i.e. physical port N is either mapped at the address N in the task's IO address space or it is not mapped.

L4Ka handles IO-ports like memory, i.e. as flex-pages. IO-flex-pages can be mapped, granted and unmapped like memory flex-pages. However, since IO-ports can only be mapped idempotent, always the complete IO space (64K) should be specified as receive flex-page.

An IO-flex-page of size 2^s ($4 \leq s \leq 16$) has a 2^s-aligned base address p , i.e. p mod $2^s = 0$. An flex-page with base port address p and size 2^s is denoted by a 32-bit word (see Figure 3.5).

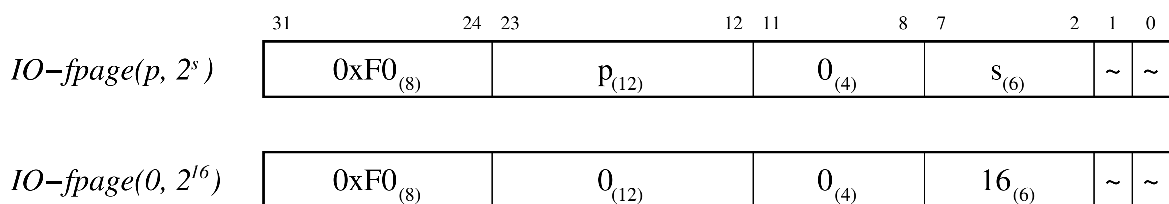

Figure 3.5: Format of Flex-page for IO-port

3.5 Timeout

Timeouts are used to control IPC operations. Each IPC operation specifies four timeout values. The complete quadruple is packed into one 32-bit word (see Figure 3.6).

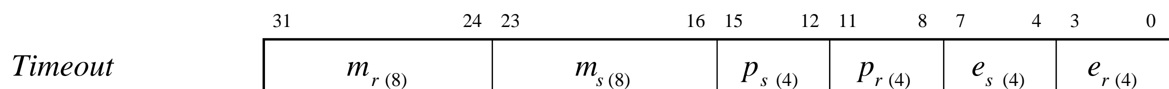

Figure 3.6: Timeout Format

The first two timeouts are with respect to the time *before message transfer starts*. These timeouts are no longer relevant once message transfer starts.

Receive timeout specifies how long to wait for an incoming message. The receive operation fails if the timeout period is exceeded before message transfer starts. The receive timeout is calculated as follows:

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

Send timeout specifies how long IPC should try to send a message. The send operation fails if the timeout period is exceeded before message transfer starts. The send timeout is calculated as follows:

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

The other timeout values are used if a page fault occurs during an IPC operation. A page fault is converted to an IPC to a pager by the kernel (see Section 6.2). The page fault timeouts are with respect to this IPC message.

Receive page fault timeout is used for both the send and receive timeouts of the page fault handling when a page fault occurs in the sender's address space during an IPC. This value is set by the receiver and is calculated as follows:

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

Send page fault timeout is used for both the send and receive timeouts of the page fault handling when a page fault occurs in the receiver's address space during an IPC. This value is set by the sender and is calculated as follows:

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

There are two special timeout values: ∞ and 0. An infinite value means no timeout (i.e. possibly indefinite blocking) and is specified by zero values of e or p. A zero timeout value represents non-blocking IPC and is specified by zero values of m (with $e > 0$). A maximum value for p ($p = 15$) means that the IPC will fail if a page fault occurs. Besides the special timeouts, periods from 1 μs up to approximately 19 hours can be specified. They are shown in Table 3.2.

Table 3.2: Approximate Timeout Ranges

e_s, e_r, p_s, p_r	send/receive timeout			pagefault timeout
0		∞		∞
1	256 s	...	19 h	256 s
2	64 s	...	4.5 h	64 s
3	16 s	...	71 m	16 s
4	4 s	...	17 m	4 s
5	1 s	...	4 m	1 s
6	262 ms	...	67 s	256 ms
7	65 ms	...	17 s	64 ms
8	16 ms	...	4 s	16 ms
9	4 ms	...	1 s	4 ms
10	1 ms	...	261 ms	1 ms
11	256 μs	...	65 ms	256 μs
12	64 μs	...	16 ms	64 μs
13	16 μs	...	4 ms	16 μs
14	4 μs	...	1 ms	4 μs
15	1 μs	...	255 μs	0
$m = 0, e > 0$		0		—

x86 Implementation

Timeouts are implemented as a `l4_timeout_struct_t` structure and a `l4_timeout_t` union with additional field for a raw access. These declarations are presented in Listing 3.6.

There are also some useful definitions:

- **L4_IPC_NEVER** sets timeout to infinity value (i.e. possibly indefinite blocking);
- **L4_IPC_TIMEOUT_NULL** sets timeout to zero value (non-blocking IPC);
- **L4_IPC_TIMEOUT** — function to fill in all timeout fields.

Listing 3.6: Timeout declaration in “l4.h”

```
typedef struct {
  unsigned rcv_exp:4;
  unsigned snd_exp:4;
  unsigned rcv_pfault:4;
  unsigned snd_pfault:4;
  unsigned snd_man:8;
  unsigned rcv_man:8;
} l4_timeout_struct_t;

typedef union {
  dword_t raw;
  l4_timeout_struct_t timeout;
} l4_timeout_t;

#define L4_IPC_NEVER     ((l4_timeout_t) { raw: 0})
#define L4_IPC_TIMEOUT_NULL ((l4_timeout_t) { timeout: {15, 15, 15, 15, 0, 0}})
#define L4_IPC_TIMEOUT(snd_man, snd_exp, rcv_man, rcv_exp, snd_pflt, rcv_pflt)\
                ( (l4_timeout_t) {timeout: { rcv_exp, snd_exp, rcv_pflt, \
                snd_pflt, snd_man, rcv_man } } )
```

3.6 IPC Result Status

The status of each IPC operation is returned in a message dope with the format depicted in Figure 3.7.

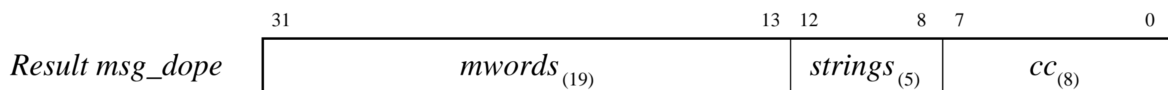

Figure 3.7: IPC Result Status in Message Dope

Figure 3.8: Condition Code of Result Status

Message dope describes received message. If no message was received, only *cc* is delivered. Otherwise, all fields are used:

a) **mwords** determines size *in dwords* of in-line data received (excluding register data);

b) **strings** determines number of strings received;

c) **cc** is *condition code* with format, presented in Figure 3.8. Meaning of these fields is following:

1) **d** — deceited message (redirected by a chief). Since clans and chiefs are not implemented in L4Ka::Hazelnut field is meaningless;
2) **m** — message includes mapping/granting;
3) **r** — message was redirected by the L4Ka μ-kernel;
4) **i** — message comes from an inner clan. Since clans and chief are not implemented in L4Ka::Hazelnut field is meaningless;
5) **ec** is an *error code* associated with the IPC.

For more details see Table 3.3.

Table 3.3: Condition Code Fields

Field	Value	Description
m	0	The received message does not contain flex-pages.
	1	The sender mapped or granted flex-pages. The sender's flex-page descriptors were also (besides mapping/granting) transferred as mwords.
r	0	The received message was directed to the actual recipient, not redirected.
	1	The received message was redirected by a kernel.
ec	0	There are no errors. The optional send operation was successful, and if a receive operation was also specified (*rcv descriptor* $\neq$ *nil*) a message was also received correctly.
	$\neq 0$	If IPC fails the *condition code* is in the range `0x10...0xF0`. If the send operation already failed, IPC is terminated without the potentially specified receive operation. s specifies whether the error occurred during the receive ($s = 0$) operation or during the send ($s = 1$) operation
	1	Non-existing destination or source.
	2+s	Timeout.
	4+s	Canceled by another thread (system call `l4_thread_ex_regs`).
	6 + s	Map failed due to a shortage of page tables.
	8 + s	Send pagefault timeout.
	$A + s$	Receive pagefault timeout.
	$C + s$	Aborted by another thread (system call `lthred_ex_regs`).

continued on the next page

Table 3.3: (continued)

Field	Value	Description
	$E+s$	Cut message. Potential reasons are (a) the recipient's mword buffer is too small; (b) the recipient does not accept enough parts; (c) at least one of the recipient's part buffers is too small.
	1...5	The according operation was terminated before a real message transfer started. No partner was directly involved.
	6...F	The according operation was terminated while a message transfer was running. The message transfer was aborted. The current partner (sender or receiver) was involved and got the corresponding error code. It is not defined which parts of the message are already transferred and which parts are not yet transferred.

L4Ka OS does not store dope word of the received message in the receive message buffer. System call for IPC `l4_ipc_call` returns result status via the last parameter (pointer to `l4_msgdope_t` structure, see Section 4.5.7).

x86 Implementation

The message dope is defined as a union `l4_msg_dope_t`. Dword field `raw` allows access in a raw mode. Structure `md` contains all the fields as a bit vectors according to Figure 3.7. These declarations can be found in file "`l4.h`" and are presented in Listing 3.7.

Listing 3.7: Message dope declaration in "`l4.h`"

```
typedef union
{
    struct {
        dword_t msg_deceited    :1;
        dword_t fpage_received  :1;
        dword_t msg_redirected  :1;
        dword_t src_inside      :1;
        dword_t error_code      :4;
        dword_t strings         :5;
        dword_t dwords          :19;
    } md;
    dword_t     raw;
} l4_msgdope_t;
```

```
/*
 * Some macros to make result checking easier
 */

#define L4_IPC_ERROR(x)          ((x).md.error_code)

/*
 * IPC results

#define L4_IPC_ENOT_EXISTENT            0x10
#define L4_IPC_RETIMEOUT                0x20
#define L4_IPC_SETIMEOUT                0x30
#define L4_IPC_RECANCELED               0x40
#define L4_IPC_SECANCELED               0x50
#define L4_IPC_REMAPFAILED              0x60
#define L4_IPC_SEMAPFAILED              0x70
#define L4_IPC_RESNDPFTO                0x80
#define L4_IPC_SERCVPFTO                0x90
#define L4_IPC_REABORTED                0xA0
#define L4_IPC_SEABORTED                0xB0
#define L4_IPC_REMSGCUT                 0xE0
#define L4_IPC_SEMSGCUT                 0xF0

 */
```

3.7 Schedule Parameter Word

There is only one parameter used for scheduling — `param word`. It contains all the necessary fields (see Figure 3.9). Note that here is only presented the format of this parameter, for a precise description of all the fields and restrictions on them please refer to Section 4.3.3.

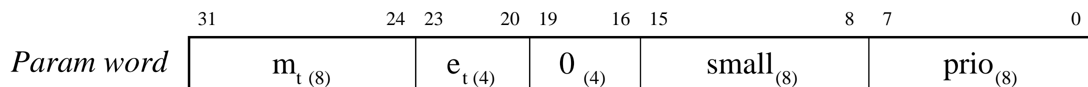

Figure 3.9: Schedule Parameter Word

Schedule param word fields:

- **prio** is a priority of the thread;
- **small** — Sets the *small address space number* for the addressed task. On Pentium, small address spaces from 1 to 127 currently available. A value of 0 or 255 in

this field does not change the current setting for the task. This field is currently ignored for 486 and PentiumPro (only effective for Pentium);

– $\boldsymbol{m_t}, \boldsymbol{m_e}$ defines time slice length of thread. The time slice quantum is encoded like a timeout: $4^{15-e_t} m_t \ \mu s$.

There is also special value for schedule parameter word — `0xFFFFFFFF`. It is called *invalid* and used to get current state of thread (thus, current priority and time slice length are not modified) when provided to a `l4_thread_schedule` system call (see Section 4.5.5).

x86 Implementation

Definition of the schedule parameter word is presented in Listing 3.8.

Listing 3.8: Schedule Parameter Word from "`types.h`"

```
typedef struct {
  unsigned prio:8;
  unsigned small:8;
  unsigned zero:4;
  unsigned time_exp:4;
  unsigned time_man:8;
} l4_sched_param_struct_t;

typedef union {
  dword_t sched_param;
  l4_sched_param_struct_t sp;
} l4_sched_param_t;

#define L4_INVALID_SCHED_PARAM ((l4_sched_param_t){sched_param:-1})
```

4 L4Ka System Calls

4.1 Overview

File "`apps/include/l4/x86/syscalls.h`" contains declarations of all system calls of L4Ka::Hazelnut. Almost all of them are described in this chapter:

- **l4_fpage_unmap** is used to unmap flex-page from all address spaces of invoker;
- **l4_myself** is used to obtain UID of the current thread;
- **l4_thread_ex_regs** is used to create threads. Also reads and writes some register values of a thread in the current task;
- **l4_thread_switch** is used to release the CPU for other ready threads;
- **l4_thread_schedule** is used by scheduler to define the priority and the time slice length of threads;
- **l4_task_new** is used to delete and/or create a task.

For a **l4_ipc_call** system call only C interface is presented. It is used for inter-process communication and synchronization. This generic system call provides several mechanisms for IPC (*Send*, *Receive from*, *Send to & Receive*, *Open Receive*), interrupt handling (see Section 6.4) and *Sleep* operation. Sufficient information concerning inter-process communication is presented in the next chapter.

Since clans and chiefs are not implemented in L4Ka::Hazelnut, system call **l4_nchief** (to get the next chief) is useless.

4.2 Task Creation and Deletion

A task can be in either an *active* or an *inactive* state. Creating an active task creates a new address space as well as the maximum number of threads for a task (64). Initially, all the threads of an active task except for one (called *thread 0*) are *inactive*. In contrast, an inactive task has no address space and no threads and thus consumes no resources.

The L4Ka μ-kernel only allows a certain number of tasks to be created (see Table 3.1). UIDs are assigned to tasks on a first-come-first-served basis with subsequent attempts to create a new task failing. Thus, the purpose of creating an inactive task is essentially reserving the right to create, later on, an active task. Deleting a task removes the address space and all associated threads of the task.

Task creation and deletion is done by using the `l4_task_new` system call. This system call first deletes a task (active or inactive) and creates a new one (active or inactive). If the task is created active, it gets the same task number (as provided in the *dest* parameter) but a different version number, hence producing a different ID. An active task is created by providing a valid pager id to the system call while a nil pager produces an inactive task (details of page fault handling and pager thread creation are in Section 6.2).

Note that there is no separate task deletion system call as such. To kill a task, simply create a new inactive task providing the id of the task to be killed to the `l4_task_new` system call (as the *dest* parameter).

Creating an *active* task requires the caller to supply:

- a pager (the pager for the new task's *thread 0*, as well as the default pager for all further threads);
- a start address and an initial stack pointer;
- a maximum controlled priority. For a detailed description of all the scheduling parameters, please refer to the Section 4.3.3.

Creating an *inactive* task (or deactivating an active one) requires the caller to specify only a null pager (which identifies the new task as being inactive).

At the end of this chapter, an example of task creation with a simple pager is presented.

4.3 Thread Related System Calls

4.3.1 Thread Manipulation

As it was mentioned above, an active task is created with a full set (64) of threads but with only one thread active (*thread 0*). A thread is activated by setting its instruction pointer (IP) and stack pointer (SP) to valid values. Once active, a thread cannot be deactivated (other than by deleting its task). To stop a thread from running it need to be blocked on an IPC, which will never succeed.

A thread's instruction and stack pointers, along with its exception handler and pager, are set by manipulating the thread's register values through the `l4_thread_ex_regs` system call. Providing the invalid value (-1) for any of these to this system call will retain the old values. `l4_thread_ex_regs` also gives back the old values of the instruction pointer, stack pointer, exception handler and pager. Thus, the call can also be used to check a thread's current context (by providing invalid values only). A thread's pending IPCs are cancelled and those in progress are aborted by this system call.

4.3.2 Release CPU

A thread can use the `l4_thread_switch` system call to voluntarily release the CPU. The releasing thread can specify a thread to which to donate its remaining time slice. If ready, the thread receiving the donation obtains the remaining time of the other thread on top of its own time slice. Alternatively, if the receiving thread is not ready or if the releasing thread does not specify a destination thread as part of the system call, the caller's remaining time slice is simply forfeited and normal scheduling takes place immediately.

4.3.3 Thread Scheduling

Thread scheduling in L4Ka is controlled by three parameters: *time slice length*, *thread priority* and *maximum controlled priority (MCP)*.

Time Slice Length

Each L4Ka thread has a time slice length associated with it. The time slice value can range from 0 to `MAX_TIMESLICE` and can vary between individual threads of a task. Each thread is scheduled for the time slice length currently associated with it. When a thread's time quantum expires, the scheduler selects the next runnable thread as described in the following section.

A time slice length of zero is valid. A thread with zero time slice is taken out of the ready queue and therefore never scheduled (until it is given a non-zero time slice length).

Note that a thread's time slice length is in no way determined by its priority. It is valid for threads of the same priority to have different time slice lengths. A thread initially gets the same time slice length as its parent and that value can only be changed via a `l4_thread_schedule` system call.

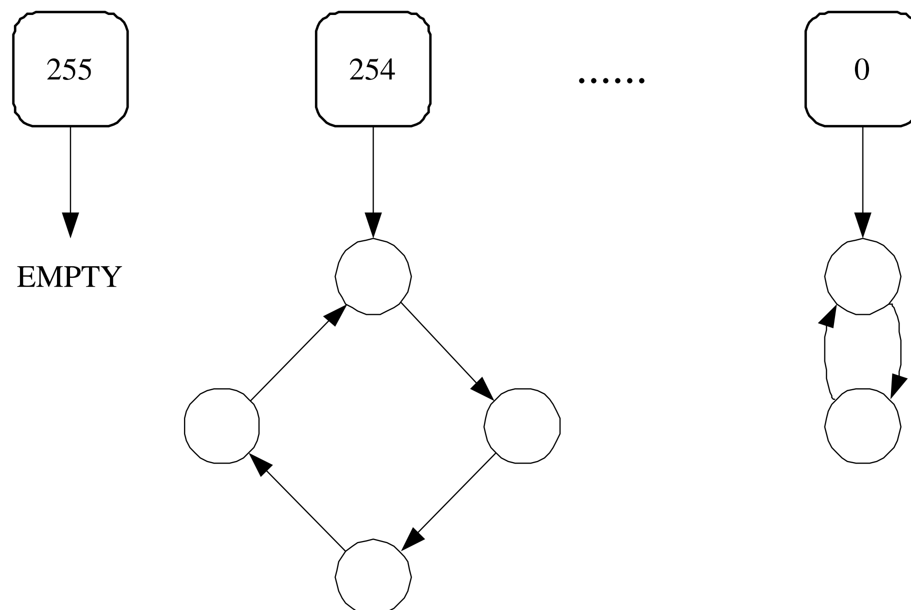

Figure 4.1: Example Ready Queue

Priority

The kernel defines 256 levels of priority in the range [255..0] with 255 being the highest priority and 0 the lowest. L4Ka's internal scheduler uses multiple-level round-robin queues such that there is a queue (possibly empty) associated with each priority level. All the queues taken together form the kernel's ready queue. Figure 4.1 shows an example of a ready queue in L4Ka. Each circle represents a thread in a particular round-robin queue.

Each thread has an associated priority at any given time. The priority determines which round-robin queue the thread belongs to in the kernel ready queue. Changing a thread's priority (via `l4_thread_schedule`) will change the queue it belongs to.

L4Ka priorities are absolute. On each scheduling event, the scheduler will always select the next thread to run from the head of the highest priority queue that is currently non-empty. For example, in Figure 4.1, the scheduler would take the thread at the head of the queue associated with priority 254.

Maximum Controlled Priority (MCP)

Unlike the time slice length and priority, the MCP is not thread based but rather task based. The MCP of a task is specified at creation time (see Section 4.2) and all threads in the task will share this MCP value.

Maximum controlled priority was meant for task hierarchy to be fair and secure, but

currently it is not used (see Section 6.5). So, if some task A creates another task B providing value of MCP, say MCP_{new}, to `l4_thread_schedule` system call following condition would guarantee fairness:

$$B_{MCP} \quad = \quad min(MCP_{new}, A_{MCP}) \ .$$

Another scenario: particular thread (of task A) tries to change scheduling parameters of another thread (of task B). Right conditions on which invoker can change priority of target thread to $prio_{new}$:

$$A_{MCP} \geq B_{MCP} \quad \wedge \quad A_{MCP} \geq prio_{new} \ .$$

Scheduling Parameter Inheritance

When an L4Ka task is created (by calling `l4_task_new`), only the MCP is specified but neither the time slice length nor priority is given. Similarly, creating an L4Ka thread (by calling `l4_thread_ex_regs`) does not explicitly require any of the scheduling parameters to be provided. Thus, there is implicit scheduling parameter inheritance rules defined for new tasks and threads. All the scheduling parameters, which are not provided in system calls are inherited from parent thread.

At boot time, σ_0 pager (see Section 6.3) starts up all the programs.σ_0 has maximum values for all its scheduling parameters and also calls `l4_task_new` (to create the initial servers) with the MCP parameter set to maximum. Hence, all initial servers will have maximum values for their MCP and thread priority. This behavior is sensible because initial servers should form the OS and thus should be given maximum privileges.

4.4 Revoking Mappings

Mappings can be recalled by using the `l4_fpage_unmap` system call. The invoker specifies a flex-page to be revoked from all address spaces into which the invoker mapped directly or indirectly. The unmapping can be done partially (revert to read-only) or completely (pages no longer part of the other address spaces). As part of the unmapping, the invoker can optionally elect to remove the pages from its own address space.

The `l4_fpage_unmap` system call takes two parameters:

a) **flex-page** defines flex-page to be unmapped. As with mapping and granting, the flex-page specifies memory region;

b) **map_mask** determines how the unmap is performed by indicating:

1) the *unmap operation* — set flex-page to read-only (`L4_FP_REMAP_PAGE`) or completely unmap flex-page (`L4_FP_FLUSH_PAGE`);
2) the *unmap extent* — apply the *unmap operation* to all other address spaces in which the flex-page has been mapped but not the original flex-page (`L4_FP_OTHER_SPACES`) or apply the *unmap operation* in every address space including the original
(`L4_FP_ALL_SPACES`).

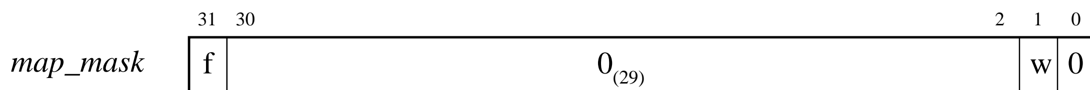

Figure 4.2: Format of the `map_mask`

Note that the *unmap operation* and *unmap extent* are orthogonal and so both should be specified (by combining the two attributes with a logical OR). Table 4.1 shows all the valid values for `map_mask`. Figure 4.2 illustrates format of the `map_mask` and Table 4.2 explains its fields. Listing 4.1 shows concrete values of `map_mask` constants for x86 architecture.

Table 4.1: Fields of the `map_mask`

Field	Value	Description
w	0	Flex-page will be partially unmapped. Already read/write mapped parts will be set to read only. Read only mapped parts are not affected.
	1	Flex-page will be completely unmapped.
f	0	Unmapping happens in all address spaces, into which pages of the specified flex-page have been mapped directly or indirectly. The *original* pages in the own task remain mapped.
	1	Unmapping happens in all address spaces, into which pages of the specified flex-page have been mapped directly or indirectly. Additionally, also the original pages in the own task are unmapped (flushing).

Table 4.2: Common map_masks

map_mask	Description
L4_FP_REMAP_PAGE | L4_FP_OTHER_SPACES	Map flex-page read-only in all other address spaces, in which the flex-page has been mapped
L4_FP_FLUSH_PAGE | L4_FP_OTHER_SPACES	Completely unmap the flex-page in all other address spaces, in which the flex-page has been mapped
L4_FP_REMAP_PAGE | L4_FP_ALL_SPACES	Map flex-page read-only in all address spaces
L4_FP_FLUSH_PAGE | L4_FP_ALL_SPACES	Completely unmap the flex-page in all address spaces

Listing 4.1: Map_masks from file apps/include/l4/x86/syscalls.h

```
#define L4_FP_REMAP_PAGE        0x00    /* Page is set to read only */
#define L4_FP_FLUSH_PAGE        0x02    /* Page is flushed completly */
#define L4_FP_OTHER_SPACES      0x00    /* Page is flushed in all other */
                                        /* address spaces */
#define L4_FP_ALL_SPACES        0x80000000U
                                        /* Page is flushed in own address */
                                        /* space too */
```

4.5 C Interface

This section contains concise reference book of all the L4Ka system calls.

4.5.1 l4_myself

```
L4_INLINE l4_threadid_t
        l4_myself(void);
```

The system call returns the UID of the current thread.

4.5.2 l4_fpage_unmap

```
L4_INLINE void
        l4_fpage_unmap(l4_fpage_t fpage,
                       dword_t map_mask);
```

The flex-page specified by *fpage* parameter is unmapped in all address spaces, into

which the invoker mapped it directly or indirectly. Parameter *map_mask* have effect according to Table 4.2.

4.5.3 l4_thread_switch

```
L4_INLINE void
        l4_thread_switch(l4_threadid_t destination);
```

The invoking thread releases the processor so that another ready thread can be processed.

If *destination* thread id is `L4_NIL_ID`, processing switches to an undefined ready thread, which is selected by the scheduler (it might be the invoking thread).

Otherwise if *destination* thread is ready, processing switches to this thread. If the *destination* thread is not ready, the system call operates as described for *destination* = `L4_NIL_ID`.

4.5.4 l4_thread_ex_regs

```
L4_INLINE void
        l4_thread_ex_regs(l4_threadid_t destination,
                          dword_t EIP,
                          dword_t ESP,
                          l4_threadid_t *preempter,
                          l4_threadid_t *pager,
                          dword_t *old_eflags,
                          dword_t *old_EIP,
                          dword_t *old_ESP);
```

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

It also creates threads. Conceptually, creating a task includes creating all of its threads. The kernel does neither allocate control blocks nor time slices etc. to them (except *thread 0*, of course). Setting stack and instruction pointer of such a thread to valid values then really generates the thread.

Note that this operation reads and writes the user-level registers (ESP, EIP and EFLAGS). However, any IPC operation is cancelled or aborted. If the IPC is either waiting to send a message or waiting to receive a message, i.e. a message transfer is not yet running, IPC is cancelled (*condition code* is `0x40` or `0x50`, see Section 3.6). If a message transfer is currently running, IPC is aborted (*condition code* is `0xC0` or `0xD0`).

ESP parameter defines new stack pointer for the thread. It must point into the user-

accessible part of the address space. Providing invalid value 0xFFFFFFFF will not affect stack pointer.

EIP parameter defines new instruction pointer for the thread. It must point into the user-accessible part of the address space. Providing invalid value 0xFFFFFFFF will not affect instruction pointer.

Preempter defines the internal preempter used by the thread (L4_NIL_ID is a valid id). Providing invalid value 0xFFFFFFFF will not modify the existing internal preempter id. Internal preempter is returned via this *preempter* parameter. This parameter is not used (see Section 6.5 for more details about preemption).

Pager defines the pager used by the thread. Providing invalid value 0xFFFFFFFF will not modify the existing pager id, which is then returned via this *pager* parameter.

Parameters *old ESP* and *old EIP* returns old values of stack pointer and instruction pointer of the thread correspondingly.

Parameter *old eflags* returns flags of the thread. For more details about these flags refer to [11].

4.5.5 l4_thread_schedule

```
L4_INLINE cpu_time_t
        l4_thread_schedule(l4_threadid_t dest,
                           l4_sched_param_t param,
                           l4_threadid_t *ext_preempter,
                           l4_threadid_t *partner,
                           l4_sched_param_t *old_param);
```

Value of type cpu_time_t is returned by l4_thread_schedule system call. This type is a synonym for long long defined in apps/include/l4/x86/types.h. In this case it has format presented in Figure 4.3. *T* is a CPU time (48-bit value) in microseconds, which has been consumed by the *dest* thread.

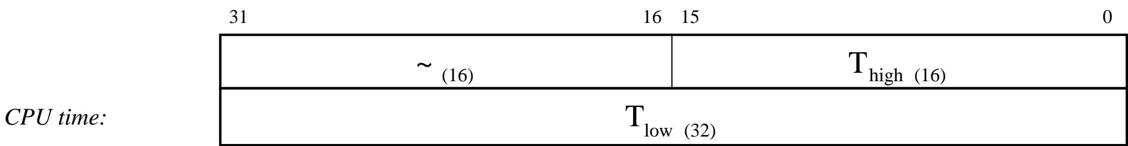

Figure 4.3: Format of the cpu_time_t

Parameter *param* defines schedule parameter word (priority and time slice length) for the *dest* thread in format described in Section 3.7. If the invalid value is provided,

scheduling parameters are not modified. Old schedule parameter word is returned via *old_param* parameter.

Parameter *ext_preemter* defines the external preempter for the destination thread (`L4_NIL_ID` is a valid id). Providing invalid value `0xFFFFFFFF` will not change the current external preempter id. External preempter is returned via *ext_preempter* parameter. See Section 6.5 for more details about preemption.

Parameter *partner* is only valid, if the thread is receiving, sending or waiting for IPC. *Partner* is then returns a partner of an active user-invoked IPC operation. An `L4_INVALID_ID` is returned if there is no specific partner, i.e. if the thread is in an open receive state.

4.5.6 l4_task_new

```
L4_INLINE l4_taskid_t
        l4_task_new(l4_taskid_t destination,
                    dword_t MCP,
                    dword_t ESP,
                    dword_t EIP,
                    l4_threadid_t pager);
```

This function deletes and/or creates a task (see Section 4.2). Deletion of a task means that the address space of the task and all threads of the task disappear.

Tasks may be created as *active* or *inactive*. For an active task, a new address space is created together with 64 threads. *Thread 0* is started, the other ones wait for a creation by `l4_thread_ex_regs`. An inactive task is empty. It occupies no resources, has no address space and no threads. Communication with inactive tasks is not possible. Loosely speaking, inactive tasks are not really existing, but represent only the right to create an active task.

Parameter *pager* determines whether new task is created as *active* (*pager* $\neq$ `L4_NIL_ID`, the specified pager is associated with *thread 0*) or as *inactive* (*pager* = `L4_NIL_ID`, no *thread 0* is created).

ESP parameter determines initial stack pointer for *thread 0* if the new task is created as an active one (it is ignored otherwise).

EIP parameter determines initial instruction pointer for *thread 0* if the new task is created as an active one (it is ignored otherwise).

MCP (maximum controlled priority) parameter defines the highest priority that can be ruled by the new task acting as a scheduler. See Section 6.5 for more details.

Return value of this system call is of type `l4_taskid_t` (which is synonym for

l4_threadid_t). The task creation is failed if this UID is L4_NIL_ID. Otherwise, task creation succeeded and if the new task is already active, this returned UID would have a new version number so that it differs from all task ids used earlier.

4.5.7 l4_ipc_call

This is the basic system call for inter-process communication and synchronization. It may be used for intra- and inter-address-space communication. All communication is synchronous and unbuffered: a message is transferred from the sender to the recipient if and only if the recipient has invoked a corresponding IPC operation. The sender blocks until this happens (or timeout expires).

IPC can be used to copy data as well as to *map* or *grant* flex-pages from the sender to the recipient. For the description of messages, see Chapter 5.

Messages of up to three dwords can be transferred solely via the registers and are thus specially optimized. If possible, short messages should therefore be reduced to 12-byte messages.

A single `l4_ipc_call` combines an optional send operation with an optional receive operation. Whether it includes a send respectively a receive is determined by the actual parameters. If the send or receive address is specified as *nil* (`0xFFFFFFFF`), the corresponding operation is skipped.

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

In contrast to other system calls, `l4_ipc_call` is declared in file `apps/include/l4/x86/ipc.h` and has several prototypes:

- `l4_ipc_call`,
- `l4_ipc_reply_and_wait`,
- `l4_ipc_send`,
- `l4_ipc_receive`,
- `l4_ipc_wait`.

Note that all these functions returns the result status of IPC via the last input parameter (**`l4_msgdope_t`** `*result`).

Prototype l4_ipc_call

```
L4_INLINE int
        l4_ipc_call(l4_threadid_t dest,
                    const void *snd_msg,
                    dword_t snd_dword0,
                    dword_t snd_dword1,
                    dword_t snd_dword2,
                    void *rcv_msg,
                    dword_t *rcv_dword0,
                    dword_t *rcv_dword1,
                    dword_t *rcv_dword2,
                    l4_timeout_t timeout,
                    l4_msgdope_t *result);
```

This prototype is usually used for blocking. *snd msg* is sent to *dest* and the invoker waits for a reply from *dest* thread. Messages from other sources are not accepted. Note that since the send/receive transition needs no time, the destination can reply with *snd timeout* = 0. This operation can also be used for a server with one dedicated client. It sends the reply to the client and waits for the client's next order.

Prototype l4_reply_and_wait (*Send to & Receive* operation)

```
L4_INLINE int
        l4_ipc_reply_and_wait(l4_threadid_t dest,
                              const void *snd_msg,
                              dword_t snd_dword0,
                              dword_t snd_dword1,
                              dword_t snd_dword2,
                              l4_threadid_t *src,
                              void *rcv_msg,
                              dword_t *rcv_dword0,
                              dword_t *rcv_dword1,
                              dword_t *rcv_dword2,
                              l4_timeout_t timeout,
                              l4_msgdope_t *result);
```

snd msg is sent to *dest* and the invoker waits for a reply from any source. This is the standard server operation: it sends a reply to the actual client and waits for the next order, which may come from a different client. For example, the main pager σ_0 uses this kind of `l4_ipc_call` prototype to serve requests (see Section 6.3). Thread id of the client is returned via *src* parameter.

Prototype l4_ipc_send (*Send* operation)

```
L4_INLINE int
        l4_ipc_send(l4_threadid_t dest,
                    const void *snd_msg,
                    dword_t snd_dword0,
                    dword_t snd_dword1,
                    dword_t snd_dword2,
                    l4_timeout_t timeout,
                    l4_msgdope_t *result);
```

snd msg is sent to dest. There is no receive phase included. The invoker continues working after send the message is received on the opposite side or timeout expires. This prototype is a wrapper for generic `l4_ipc_call` with **rcv msg* parameter set to "nil" value (`L4_IPC_NIL_DESCRIPTOR`).

Prototype l4_ipc_receive (*Receive From* operation)

```
L4_INLINE int
        l4_ipc_receive(l4_threadid_t src,
                       void *rcv_msg,
                       dword_t *rcv_dword0,
                       dword_t *rcv_dword1,
                       dword_t *rcv_dword2,
                       l4_timeout_t timeout,
                       l4_msgdope_t *result);
```

This operation includes no send phase. The invoker waits for a message from src. Messages from other sources are not accepted.

A hardware interrupt might be specified as source. Then provided UID must be set according to Figure 3.1. In order to associate interrupt with currently running thread zero *rcv timeout* value must be specified. After invoking this system call current thread is:

1) detached from its currently associated interrupt (if any);
2) associated with the specified interrupt provided that this one is free, i.e. not associated with another thread.

If the association succeeds, the *condition code* is `0x20` (*receive timeout*) and no interrupt is received. If an interrupt from the currently associated interrupt was pending, this one is delivered together with *condition code* of `0x00` (*ok*, see Table 3.3 for *condition code* values) and interrupt association is not modified. If the requested new interrupt is already associ-

ated to another thread or is not existing, *condition code* `0x10` is delivered and the interrupt association is not modified.

Getting rid of an associated interrupt without associating a new one is done by issuing a receive from NIL thread (`0x00`) with zero *rcv timeout*.

In order to receive interrupt thread should call `l4_ipc_receive` again. Note that interrupt messages come *only* from the interrupt, which is currently associated with this thread. The *src* parameter is only evaluated if *rcv timeout* is zero.

Prototype l4_ipc_wait (*Open Receive* operation)

```
L4_INLINE int
        l4_ipc_wait(l4_threadid_t *src,
                    void *rcv_msg,
                    dword_t *rcv_dword0,
                    dword_t *rcv_dword1,
                    dword_t *rcv_dword2,
                    l4_timeout_t timeout,
                    l4_msgdope_t *result);
```

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

Sleep operation

Sleep operation is a special kind of receive operation. It is presented in Listing 4.2. Since `L4_NIL_ID` is specified as source, no message can arrive and the IPC will be terminated after the time specified by the *rcv timeout* parameter is elapsed. Thus, function `l4_sleep` blocks thread for the time in μs given as input parameter.

Listing 4.2: Sleep operation

```
L4_INLINE
        void l4_sleep(dword_t us)
{
    dword_t dummy;
    l4_msgdope_t result;

    l4_ipc_receive(L4_NIL_ID,
                   0,
                   &dummy,
                   &dummy,
                   &dummy,
                   us ? (l4_timeout_t) {timeout : { best_exp(us), best_exp(us),
```

```
                    0, 0, best_mant(us), best_mant(us)}} : L4_IPC_NEVER,
                &result);
}
```

4.6 Examples

All presented examples and tests must be run as the root task. They use `urriy.h` file, which contains some useful constants and data types and can also be included if necessary. All examples use functions of kernel debugger *kdebug*:

– void **outchar** (char) — outputs a single ASCII character (given as a parameter) on the screen;

– void **outstring** (char *) — outputs null-terminated ASCII string (given as a parameter) on the screen;

– void **outhex32** (int) — outputs integer value (given as a parameter) on the screen in hexadecimal number system;

– void **outdec** (int) — outputs integer value (given as a parameter) on the screen in decimal number system;

– char **kd_inchar** () — kernel debugger *kdebug* stops the running of L4KA OS and inputs one character (returned by function);

– **enter_kdebug** (text) — kernel debugger *kdebug* stops the running of L4KA OS and outputs given text on the screen. User gets control over the kernel debugger *kdebug* and can use dump memory, trace IPC and page fault, etc.

These functions can be found in file “`kdebug.h`”. In order to use them kernel must be compiled with kernel debugger *kdebug* support. For more detailed information about kernel debugger *kdebug* for Hazelnut refer to [12].

4.6.1 Thread Creation

Root task creates a new thread using `l4_thread_ex_regs` system call. Another way is to use wrapper function `create_thread` from “`helpers.h`”.

Listing 4.3: Thread Creation

```
/***********************************************************
    Thread creation.
    Root task creates a thread, which then outputs its id on the screen.
```

```
****************************************************************/
#include "urriy.h"


void foo (void);

#define FOOSTACKSIZE 1024

    l4_threadid_t foo_tid; //thread id of target thread
    l4_threadid_t main_tid; //thread id of the root_task
    dword_t foostack[FOOSTACKSIZE]; //stack
    dword_t dummy;

//_______________FOO
void foo()
{
         //outstring(char *) -- outputs string on the screen
    outstring("\n foo: successfully started with thread id:");
         //outhex32(int ) -- outputs integer value on the screen in hex
    outhex32(l4_myself().raw);

    while(1){l4_sleep(1000000);outchar('+');}
}

//_____________________MAIN -- root task
int main(dword_t mb_magic, struct multiboot_info_t* mbi)
{
        //thread id of the root_task
    main_tid = l4_myself();
        //thread id of the pager of the root task
    l4_threadid_t my_pager = get_current_pager(l4_myself());

    foo_tid=main_tid;
    foo_tid.id.thread = 8;

    // foo thread creation using system call l4_thread_ex_regs
l4_thread_ex_regs(
        foo_tid,        //target thread id
        (dword_t) foo,  //IP
        (dword_t) &foostack[FOOSTACKSIZE - 1], //SP
        &my_pager,      //preempter thread id of target thread
        &my_pager,      //pager thread id of target handler
        &dummy,         //old flags -- don't care
        &dummy,         //old IP -- don't care
        &dummy);        //old SP -- don't care

    outstring("\n root_task: new thread is created with thread id ");
    outhex32(foo_tid.raw);
```

```
    while(1){l4_sleep(1000000);outchar('-');}
}
```

4.6.2 Task Creation

Creation of task is rather complicated and requires from user-level programmer knowledge of page fault handling (see Section 6.2) and principles of inter-process communication (see next chapter). In this example pager implements very simple policy — just mapping requested page to the faulter without any checks. On every request, pager asks higher-level pager (pager of itself) for this page. On success, pager delivers page to faulter via short IPC message.

Listing 4.4: Task Creation

```
/**********************************************************************
 * File path:     root_task/main.c
 * Description:   task creation
 *                new task is created by the root task
 *                  and notifies its start by writing on
 *                  the screen.
 *                pager: on page fault asks pager of itself for
 *                   this page, then delivers it via short IPC message
 *                   with mapping to the faulter.
 *                   Works only with 4k pages.
 *********************************************************************/
#include "urriy.h"

l4_threadid_t main_tid;
l4_threadid_t subtask_tid;

#define PAGERSTACKSIZE 4096
dword_t pager_stack[PAGERSTACKSIZE];

#define SUBTASKSTACKSIZE        4096
dword_t subtask_stack[SUBTASKSTACKSIZE];

void pager(void);
void subtask(void);
l4_threadid_t pager_tid;
l4_threadid_t dummy_tid;

dword_t dummy;

//________________________________SUBTASK
void subtask()
```

```
{
    outstring("\n New task is started with id: ");
    outhex32(l4_myself().raw);

        //kernel debugger input character
    kd_inchar();

        outstring(" ... nothing to do, exiting.");

while(1) {l4_sleep(1000000);outchar('.');}

}

//________________________________PAGER
void pager()
{
        l4_threadid_t client;
        l4_msgdope_t dope;
        dword_t dw0, dw1, dw2;
        dword_t map=2;
        dword_t fault_addr;
        dword_t addr;

        l4_threadid_t s0 = L4_SIGMA0_ID;
        l4_threadid_t h_pager = get_current_pager(l4_myself());

while(4)
 {
        outstring("\n pager: waiting IPC message from kernel ");
        l4_ipc_wait(&client, 0, &dw0, &dw1, &dw2, L4_IPC_NEVER, &dope);

        while(5)
        {
                //fault address without mask
        fault_addr = (dw0 & (~(dword_t) 3));
                //let pager notify higher level pager that writing access is required
        dw0 = fault_addr | 2;
        outstring("\n pager: handling a page fault at address ");
        outhex32(fault_addr);

                //asking page from higher level pager
        l4_ipc_call(h_pager,
                L4_IPC_SHORT_MSG,
                dw0,
                dw1,
                dw2,
                        //accepting page in whole address space
                (void *) l4_fpage(0, L4_WHOLE_ADDRESS_SPACE, 1, 0).raw,
                &addr, //address to be returned in the first dword
```

```
			&dummy,
			&dummy,
			L4_IPC_NEVER,
			&dope);

		if (L4_IPC_ERROR(dope))
			{
			outstring("\n pager: error while asking for page ");
			break;
			}

			outstring("\n pager: memory page is given by higher level pager. address ");
			outhex32(addr);

			//first dword contains flex-page describing the memory region
		dw0 &= L4_PAGEMASK;
			//second dword contains hot-spot which must be specified correct
		dw1 = dw0 | (L4_LOG2_PAGESIZE << 2) | (L4_IPC_SHORT_MAPMSG);



			//send IPC message, which contains a mapping of desired page
			// and then wait for the next page fault
		l4_ipc_reply_and_wait(client,
			(void *) L4_IPC_SHORT_MAPMSG,
			dw0,
			dw1,
			dw2,
			&client,
			L4_IPC_SHORT_MSG,
			&dw0,
			&dw1,
			&dw2,
			L4_IPC_NEVER,
			&dope);

		if (L4_IPC_ERROR(dope))
			{
			outstring("\n pager: error reply and wait ");
			break;
			}
		}
	}
}

//______________________MAIN
int main(dword_t mb_magic, struct multiboot_info_t* mbi)
{
	l4_threadid_t m_pager=get_current_pager(l4_myself());
	l4_threadid_t s0 = L4_SIGMA0_ID;
```

```
        main_tid = l4_myself();
        pager_tid = main_tid;
        pager_tid.id.thread = 1;

        //creating a pager thread
l4_thread_ex_regs
        (pager_tid,
        (dword_t) pager,
        (dword_t)&pager_stack[PAGERSTACKSIZE - 1],
        &m_pager,
        &m_pager,
        &dummy,
        &dummy,
        &dummy);

enter_kdebug("\n main: pager is created ");

        //creating a new task
        subtask_tid.raw = main_tid.raw;
        subtask_tid.id.task = 10;
        subtask_tid.id.thread = 0;

        //system call will return a thread id of new task
subtask_tid = l4_task_new
        (subtask_tid, //thread id of destination task
        255,    //MCP value to maximum possible priority
        (dword_t) &subtask_stack[SUBTASKSTACKSIZE - 1], //SP
        (dword_t) subtask,      //IP
        pager_tid       //thread id of pager
        );

        outstring("\n main: subtask will be created with id: ");
        outhex32(subtask_tid.raw);
        outchar('\n');
        outstring("\n main: subtask IP is ");
        outhex32((dword_t) subtask);
        outstring("\n main: subtask SP is ");
        outhex32((dword_t) &subtask_stack[SUBTASKSTACKSIZE - 1]);

        enter_kdebug("\n main: subtask is created ");

        while(1){l4_sleep(1000000);outchar('-');}
}
```

4.6.3 Thread Scheduling

This example utilizes scheduling mechanism using both the system call described above (`l4_thread_schedule`) and a wrapper function `l4_set_prio` from "`helpers.h`".

Listing 4.5: Thread Scheduling

```
/*********************************************************************
 * File path:     root_task/main.c
 * Description:   scheduling
 *                                0. root task has priority of 255, starts pager with
 *                                         priority of 255 and subtask with MCP value = 127
 *                                         and sets prio of subtask to 127
 *                                1. subtask tries to do something but is never
 *                                         scheduled if root task not idle
 *                                2. root task decreases prio of itself to 127 and
 *                                         now runs concurrently with subtask
 *                                3. subtask tries to increase prio of itself to 255
 *                                         and succeeds
 *                                4. subtask then decreases prio to 127 and again runs
 *                                         concurrently with the root task
 *                                5. subtask tries to decrease prio of the root task
 *                                         to 1 and succeeds.
 ********************************************************************/
#include "urriy.h"

l4_threadid_t main_tid;
l4_threadid_t subtask_tid;

#define PAGERSTACKSIZE 4096
dword_t pager_stack[PAGERSTACKSIZE];

#define SUBTASKSTACKSIZE        4096
dword_t subtask_stack[SUBTASKSTACKSIZE];

void pager(void);
void subtask(void);
l4_threadid_t pager_tid;
l4_threadid_t dummy_tid;

dword_t dummy;

//_____________________MAIN
int main(dword_t mb_magic, struct multiboot_info_t* mbi)
{
        l4_threadid_t my_pager=get_current_pager(l4_myself());
        l4_threadid_t s0 = L4_SIGMA0_ID;
```

```
        main_tid = l4_myself();
        pager_tid = main_tid;
        pager_tid.id.thread = 1;
        l4_sched_param_t param, old_param;

        outstring(" root task: started with prio = 255 and id = ");
        outhex32(main_tid.raw);

        //preparations for scheduling
        param.sp.small = 0;
        param.sp.zero = 4;
        param.sp.time_exp = 5;
        param.sp.time_man = 1;
        param.sp.prio = 255;
        dummy_tid = L4_INVALID_ID;
                        //scheduling using system call
                l4_thread_schedule(
                        main_tid,
                        param,
                        &dummy_tid,
                        &dummy_tid,
                        (l4_sched_param_t *) &old_param
                        );

        //creating a pager thread
l4_thread_ex_regs(
        pager_tid,
        (dword_t) pager,
        (dword_t)&pager_stack[PAGERSTACKSIZE - 1],
        &my_pager,
        &my_pager,
        &dummy,
        &dummy,
        &dummy);

        //preparations for scheduling
        param.sp.small = 0;
        param.sp.zero = 4;
        param.sp.time_exp = 5;
        param.sp.time_man = 1;
        param.sp.prio = 255;
        dummy_tid = L4_INVALID_ID;
                        //scheduling using system call
                l4_thread_schedule(
                        pager_tid,
                        param,
                        &dummy_tid,
                        &dummy_tid,
                        (l4_sched_param_t *) &old_param
```

```
                        );

enter_kdebug("\n main: pager is created with prio = 255 ");

                //creating a new task
        subtask_tid.raw = main_tid.raw;
        subtask_tid.id.task = 10;
        subtask_tid.id.thread = 0;

                //system call will return a thread id of new task
subtask_tid = l4_task_new(
        subtask_tid, //thread id of destination task
        127,    //MCP value to maximum possible priority
        (dword_t) &subtask_stack[SUBTASKSTACKSIZE - 1], //SP
        (dword_t) subtask,      //IP
        pager_tid       //thread id of pager
        );

                //preparations for scheduling
        param.sp.small = 0;
        param.sp.zero = 4;
        param.sp.time_exp = 5;
        param.sp.time_man = 1;
        param.sp.prio = 127;
        dummy_tid = L4_INVALID_ID;
                        //scheduling using system call
                l4_thread_schedule(
                subtask_tid,
                param,
                &dummy_tid,
                &dummy_tid,
                (l4_sched_param_t *) &old_param
                );

        outstring("\n main: subtask is created with prio = 127 and id = ");
        outhex32(subtask_tid.raw);

        outstring("\n root task: 50 iterations ");

        int k;
        for (int i=0; i<50; i++)
        {
        for (int j=0;j<1000000;j++) k=j-i;
        outchar('+');
        };

                //scheduling using wrapper from <helpers.h>
        l4_set_prio(main_tid, 127);
```

```
        outstring("\n root task: subtask is rescheduled, prio = 127 ");
        outstring("\n root task: everything is done. ");

        while(1){
                for (int j=0;j<1000000;j++) k=j+j;
        outchar(',');
                        }
}

//________________________________SUBTASK
void subtask()
{
        outstring("\n subtask: ... is started with id: ");
        outhex32(l4_myself().raw);

        outstring("\n subtask: 70 minuses ");

        int k;
        for (int i=0; i<70; i++)
        {
        for (int j=0;j<1000000;j++) k=j-i;
        outchar('-');
        };

enter_kdebug("\n subtask: trying to reschedule to prio = 255 ");

                //scheduling using wrapper from <helpers.h>
        l4_set_prio(subtask_tid, 255);

        outstring("\n subtask: rescheduled ");

        for (int i=0; i<150; i++)
        {
        for (int j=0;j<1000000;j++) k=j-i;
        outchar('=');
        };

enter_kdebug(" \n subtask: will run concurrently with root_task ");

                //scheduling using wrapper from <helpers.h>
        l4_set_prio(subtask_tid, 127);

        for (int i=0; i<150; i++)
        {
        for (int j=0;j<1000000;j++) k=j-i;
        outchar('=');
        };

enter_kdebug(" \n subtask: will try to decrease prio of the root_task ");
```

```
		//scheduling using wrapper from <helpers.h>
	l4_set_prio(main_tid, 1);

while(1)
		{
    for (int j=0;j<1000000;j++) k=j+j;
    outchar('.');
		}

}

//________________________________PAGER
void pager()
{
	l4_threadid_t client;
	l4_msgdope_t dope;
	dword_t dw0, dw1, dw2;
	dword_t map=2;
	dword_t fault_addr;
	dword_t addr;

	l4_threadid_t s0 = L4_SIGMA0_ID;
	l4_threadid_t h_pager = get_current_pager(l4_myself());

while(4)
 {
	outstring("\n pager: waiting IPC message from kernel ");
	l4_ipc_wait(&client, 0, &dw0, &dw1, &dw2, L4_IPC_NEVER, &dope);

	while(5)
	{
		//fault address without mask
	fault_addr = (dw0 & (~(dword_t) 3));
		//let pager notify higher level pager that writing access is required
	dw0 = fault_addr | 2;
	outstring("\n pager: handling a page fault at address ");
	outhex32(fault_addr);

		//asking page from higher level pager
	l4_ipc_call(h_pager,
		L4_IPC_SHORT_MSG,
		dw0,
		dw1,
		dw2,
			//accepting page in whole address space
		(void *) l4_fpage(0, L4_WHOLE_ADDRESS_SPACE, 1, 0).raw,
		&addr, //address to be returned in the first dword
		&dummy,
```

```
                &dummy,
                L4_IPC_NEVER,
                &dope);

        if (L4_IPC_ERROR(dope))
                {
                outstring("\n pager: error while asking for page ");
                break;
                }

                outstring("\n pager: memory page is given by higher level pager. address ");
                outhex32(addr);

                //first dword contains flex-page describing the memory region
        dw0 &= L4_PAGEMASK;
                //second dword contains hot-spot which must be specified correct
        dw1 = dw0 | (L4_LOG2_PAGESIZE << 2) | (L4_IPC_SHORT_MAPMSG);



                //send IPC message, which contains a mapping of desired page
                // and then wait for the next page fault
        l4_ipc_reply_and_wait(client,
                (void *) L4_IPC_SHORT_MAPMSG,
                dw0,
                dw1,
                dw2,
                &client,
                L4_IPC_SHORT_MSG,
                &dw0,
                &dw1,
                &dw2,
                L4_IPC_NEVER,
                &dope);

        if (L4_IPC_ERROR(dope))
                {
                outstring("\n pager: error reply and wait ");
                break;
                }
        }
 }
}
```

Urriy.h

File `urriy.h` is used by all the examples. It contains some useful constants.

Listing 4.6: urriy.h

```
#include <l4/l4.h>
#include <l4io.h>
#include <l4/helpers.h>

#include "ipc2.h"

#define L4_IPC_NIL_DESCRIPTOR   ((dword_t) -1)
#define L4_IPC_SHORT_MSG        ((dword_t) 0)
#define L4_IPC_SHORT_MAPMSG     ((dword_t) 2)
#define L4_IPC_SHORT_DECEITING  ((dword_t) 1)
#define L4_IPC_SHORT_RECEIVE_FROM       ((dword_t) 0)
#define L4_IPC_SHORT_OPEN_RECEIVE       ((dword_t) 1)
#define L4_IPC_RECEIVE_MAPMSG(address, size)  \
     ((dword_t)( ((address) & ~((dword_t)((1<<size)-1))) \
     | ((size) << 2) | 0x00000002) )

typedef struct header
{
    l4_fpage_t rcv_fpage;
    l4_msgdope_t size_dope;
    l4_msgdope_t snd_dope;
} header_t;

//______________WRITING into defined memory page of size 2^size
void writing(dword_t startaddr, int size, int debug, char fillchar)
{
 char *ptr;
 ptr    =(char *) startaddr;
 char c0=(char ) startaddr;
 char c1=(char ) (startaddr>>8);
 char c2=(char ) (startaddr>>16);
 char c3=(char ) (startaddr>>24);

int i;
        if (debug) enter_kdebug("\n just before writing to memory ");


 if (fillchar==0)
  {
    for (i=0; i<(1 << size); i++)
    {
        switch (i)
        {
        case 0: *ptr=(char ) c0; break;
        case 1: *ptr=(char ) c1; break;
        case 2: *ptr=(char ) c2; break;
        case 3: *ptr=(char ) c3; break;
        default:
```

```
        *ptr=(char )i;
        }
        ptr++;
    }
  }
  else
  {
   for (i=0; i<(1 << size); i++)
   {
   *ptr= fillchar;
    ptr++;
   }
 }

if (debug==2) enter_kdebug("\n just after writing to memory ");
}

//_________________FREEING simple unmap call
void freeing(dword_t addr, int size, int grant, int write, dword_t mask)
{
l4_fpage_t fp;
fp.fp.grant=grant;
fp.fp.write=write;
fp.fp.size=size;
fp.fp.zero=0;
fp.fp.page=addr>> 12;//we need only 20 bits of address

l4_fpage_unmap(fp, mask);

}
```

Ipc2.h

File `ipc2.h` is used by several examples. It contains `l4_ipc_call2` function — similar to initial `l4_ipc_call`, but without any masks.

Listing 4.7: ipc2.h

```
#ifndef __L4_X86_IPC2_H__
#define __L4_X86_IPC2_H__


#if !defined(CONFIG_L4_SYSENTEREXIT)
#define IPC_SYSENTER    "int    $0x30           \n\t"
#else
#define OLD_IPC_SYSENTER        \
"push   %%ecx                                   \n\t" \
```

```
"push    %%ebp                                              \n\t" \
"push    $0x23          /* linear_space_exec     */        \n\t" \
"push    $0f            /* offset ret_addr       */        \n\t" \
"mov     %%esp,%%ecx                                        \n\t" \
"sysenter               /* = db 0x0F,0x34        */        \n\t" \
"mov     %%ebp,%%edx                                        \n\t" \
"0:                                                         \n\t"
#define IPC_SYSENTER     \
"push    %%ecx                                              \n\t" \
"push    %%ebp                                              \n\t" \
"push    $0x1b          /* linear_space_exec     */        \n\t" \
"push    $0f            /* offset ret_addr       */        \n\t" \
"mov     %%esp,%%ecx                                        \n\t" \
"sysenter               /* = db 0x0F,0x34        */        \n\t" \
"mov     %%ebp,%%edx                                        \n\t" \
"0:                                                         \n\t"
#endif

 /*
 * Internal defines used to build IPC parameters for the L4 kernel
 */

#define L4_IPC_NIL_DESCRIPTOR   (-1)
#define L4_IPC_DECEIT           1
#define L4_IPC_OPEN_IPC         1



/*
 * Prototypes
 */

L4_INLINE int
l4_ipc_call2(l4_threadid_t dest,
            const void *snd_msg,
            dword_t snd_dword0, dword_t snd_dword1, dword_t snd_dword2,
            void *rcv_msg,
            dword_t *rcv_dword0, dword_t *rcv_dword1, dword_t *rcv_dword2,
            l4_timeout_t timeout, l4_msgdope_t *result);
/*
 * Implementation
 */

#define SCRATCH_MEMORY 1
#ifdef __pic__

#error no version X bindings with __pic__ enabled

#else /* __pic__ */
```

```
L4_INLINE int
l4_ipc_call2(l4_threadid_t dest,
            const void *snd_msg, dword_t snd_dword0, dword_t snd_dword1,
            dword_t snd_dword2, void *rcv_msg, dword_t *rcv_dword0,
            dword_t *rcv_dword1, dword_t *rcv_dword2,
            l4_timeout_t timeout, l4_msgdope_t *result)
{
    dword_t dw[3] = {snd_dword0, snd_dword1, snd_dword2};
    asm volatile(
        "pushl  %%ebp           \n\t"           /* save ebp, no memory
                                                   references ("m") after
                                                   this point */
        "movl   %%edi,%%ebp     \n\t"
        "movl     4(%%edx),%%ebx \n\t"
        "movl     8(%%edx),%%edi \n\t"
        "movl     (%%edx),%%edx  \n\t"
        IPC_SYSENTER
        "popl   %%ebp           \n\t"           /* restore ebp, no memory
                                                   references ("m") before
                                                   this point */
        :
        "=a" (*result),                         /* EAX, 0       */
        "=d" (*rcv_dword0),                     /* EDX, 1       */
        "=b" (*rcv_dword1),                     /* EBX, 2       */
        "=D" (*rcv_dword2)                        /* EDI, 3       */
        :
        "c" (timeout),                          /* ECX, 4       */
        /*Urriy*/
//      "D" (((int)rcv_msg) & (~L4_IPC_OPEN_IPC)),/* EDI, 5, rcv msg -> ebp */
        "D" ((int)rcv_msg),
        "S" (dest),                             /* ESI, 6, dest   */
        /*Urriy*/
//      "0" (((int)snd_msg) & (~L4_IPC_DECEIT)),/* EAX, 0       */
        "0" ((int)snd_msg),
        "1" (&dw[0])                            /* EDX, 1,      */
#ifdef SCRATCH_MEMORY
        : "memory"
#endif /* SCRATCH_MEMORY */
        );
    return L4_IPC_ERROR(*result);
}


#endif /* __pic__ */


#endif /* __L4_X86_IPC_H__ */
```

					ДП.991137.ПЗ	Лист
						67
Изм.	Лист	№ докум.	Подпись	Дата		

5 L4Ka IPC

5.1 Why Inter-Process Communication?

There are many scenarios, which serves as examples of this necessity. Assume for instance client-server model. Server, say Web-server, must provide service (web-page) to many clients. Client makes requests in arbitrary order at any time. In Linux OS solution would be to create a main process of a web-server and then make a copy of it using `fork()` for every new client. Then, looking at its PID (process ID) process can distinguish itself from a parent process and perform actions to provide service for a given client.

A Web-server application uses some amount of memory to operate and needs to communicate with other applications. For latter can stand some Database Engine, interpreter (say, Perl or CGI scripts interpreter) and some others. Former will require copy operation of memory of a parent process (in case of Linux OS). Assume that a child process of a Web-server needs to perform a SQL request to Database. Then using Perl interpreter construct a Web-page to be returned to a client.

Thus, it is necessary to organize efficient communication between all these applications. Efficient in terms of waiting time of a client to be served (to get the requested page).

Using L4Ka IPC on the kernel level all these operations can be done very efficient. First, web-server can be created as one task. Main thread of it wakes up other threads of a task on every new request. Parameters of this request can be sent by task in IPC message.

Communication with other tasks can be done using memory mappings — thus copy of memory does not takes place. Since all IPC operations atomic and unbuffered these scheme is rather efficient. At the time of writing L4Ka IPC claims to be the most efficient one [13].

Moreover, some QoS policies can be applied on top: if some client must be served move quickly (or even real-time), change of priorities and time slices using system call `l4_thread_schedule` affect the quality of service. The DROPS project (The Dresden Real-

Time Operating System Project, [14]) aims at supporting applications with Quality of Service requirements. L[4]Linux is used for servicing standard Linux applications. Specific real-time applications are served by a set of real-time components running on top of L4.

5.2 IPC Overview

Message passing is the basic IPC mechanism in L4Ka. It allows L4Ka threads to communicate via messages. All L4Ka IPC is synchronous and unbuffered. Unbuffered IPC reduces the amount of copying involved.

Synchronous IPC requires an agreement between both the sender and the receiver. The main implications of this agreement are that the receiver is expecting an IPC and provides the necessary buffers. If either the sender or receiver is not ready, then the other party must wait.

L4Ka IPC operation can have a send and a receive phase. L4Ka IPC uses a single system call instead of many. There are some advantages of this approach:

- **Combined send and receive** The implementation of the individual send and receive is very similar to the combined send and receive. It was meant to reduce cache footprint of the code and make applications more likely to be in cache.
- **Open receive** It is special IPC primitive for servers to communicate with a thread unknown *a priori*. IPC primitive *Send to & Receive* is atomic operation for optimization reasons. Typically used by servers to reply and wait for the next request (from anyone).
- **Atomicity** Other threads are blocked during IPC until it is their turn.
- **Flexibility** Many operations (IPC primitives), different message types and other features are enclosed in same IPC call.

5.3 Message Types

Data can be transferred in three ways using L4Ka IPC:

1) **By-register** In-line by-value data. A limited amount of such data is passed directly in registers.
2) **By-value** Arbitrary out-of-line buffers which are copied to the receiver. Transfer is done by strings and `mwords`.

3) **By-reference** Messages that map (grant) pages from sender to receiver. Contiguous regions of address space is called *flex-page* (see definition in section 3.4). Virtual memory can be *mapped* or *granted* (see definitions on page 16).

Messages via registers are called *short*, rather than by-value which are *long*. Mappings can be sent either using *short* or *long* IPC message.

5.4 Sending/Receiving IPC Messages

Before considering *how* to send or receive, it is necessary to decide *what* form of data is to be sent. Small amount of data (not more than 12 bytes) can be transferred by-register.

Otherwise (if data does not fit into three dwords), a decision must be made on whether to send data in-line (in `mwords`) or as strings. Both have the same effect of making a copy of the sender's data available to the receiver. The difference lies in efficiency and appropriateness.

In-line data need to be copied to a buffer first and must be aligned to dword size. Thus, the in-line option is best for small amounts of data and is useful when some kind of handshake is required (to define the protocol and parameters of further transfer: size of buffers, etc.).

Strings avoid extra copying and can be located anywhere in memory (no alignment necessary) but require buffers to be specified (via string dopes). Buffer specifications must also be consistent on both the sender and the receiver end. In particular, the receiver must specify and expect to receive the (at least) the same number and size strings that the sender sends.

When it is necessary to send even more data (than possible using strings), memory granting via flex-pages should be used (by-reference method). In order to use some memory regions simultaneously by several tasks memory mapping solves the problem.

As it was mentioned above, the generic system call `l4_ipc_call` provides a number of IPC operations. They are differentiated in the following ways:

- *Send* operation — send data to another thread;
- *Receive from* operation — receive data from particular thread;
- *Send to & Receive* operation — combined send and receive. Send to particular thread, receive from arbitrary;
- *Open Receive* — receiving from arbitrary thread.

In order to send or receive a message, certain parameters must be provided. Namely

- *dest/src thread id* — identifier of message destination/source thread respectively (format of the unique ids is described in section 3.3);
- *snd msg/rcv msg* — descriptor for send/receive part of IPC respectively (format of a message descriptor is described in section 5.5.1);
- In-line by-value data to be sent. Three dwords can be sent on x86 architecture and eight on MIPS R4k. These data can define mappings via flex-page descriptors (format of the flex-page descriptor is described in section 5.5.5);
- Pointers on dwords `ptr_t` for in-line by-value data to be received;
- *timeout* — timeout specification. It is used to ensure that a thread need not be blocked indefinitely (format of the timeout structure is described in section 3.5);
- Pointer to `l4_msgdope_t` structure. Variable serves to store the result status of the IPC operation (format of the result status of IPC is described in section 3.6).

5.5 L4Ka IPC Messages

5.5.1 Message Descriptors

A message descriptor is a pointer to the start of a message buffer or indication that the IPC is purely register based. There are two types of message descriptors: one for sending and one for receiving IPC.

The format of the *send message descriptor* is presented in Figure 5.1. A non-zero message descriptor address (*snd msg*) is interpreted as the start address of the sender's message buffer. A zero value indicates a purely register based IPC. Setting the m-bit indicates that the IPC includes mappings (i.e. flex-pages are present followed possibly by by-value data). A zero value for the m-bit indicates that the message contains only by-value data and no flex-pages.

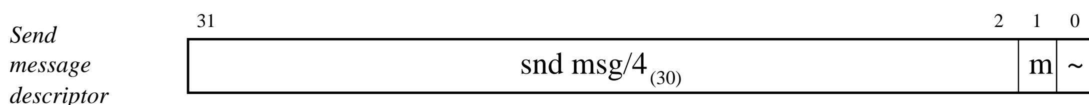

Figure 5.1: Format of the Send Message Descriptor

The format of the *receive message descriptor* is very similar and presented in Figure 5.2.

If the m-bit is not set, the message descriptor address (*rcv msg*) is interpreted as the start address of the receiver's message buffer, which may contain a receive flex-page, indicating the caller's willingness to receive mappings or grants. Setting the m-bit indicates that the caller is willing to accept flex-page mappings but no long message, and has supplied the receive flex-page directly as the *rcv msg* parameter (there is no message buffer in this case). Setting the o-bit will allow a receive from any sender (*Open Receive*) while a zero value for the o-bit allows receiving only from the specified sender (*Receive from*). Note that the message descriptor addresses have had their least significant two bits removed. These two bits are not needed as the message buffer must be word aligned.

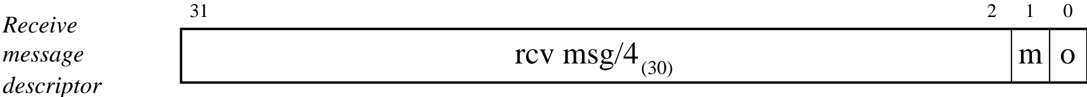

Figure 5.2: Format of the Receive Message Descriptor

More detailed information about all types of message descriptors is presented in Table 5.1. Most frequently used descriptors can be found in file "`urriy.h`" and are shown in Listing 5.1.

Listing 5.1: Widely used message descriptors in "`urriy.h`"

```
#define L4_IPC_NIL_DESCRIPTOR   ((void *) -1)
#define L4_IPC_SHORT_MSG        ((void *)0)
#define L4_IPC_SHORT_MAPMSG     ((void *)2)
#define L4_IPC_SHORT_DECEITING  ((void *)1)
#define L4_IPC_SHORT_RECEIVE_FROM       ((void *)0)
#define L4_IPC_SHORT_OPEN_RECEIVE       ((void *)1)
#define L4_IPC_RECEIVE_MAPMSG(address, size)  \
     ((void *) (dword_t)( ((address) & ~((dword_t)((1<<size)-1))) \
     | ((size) << 2) | 0x00000002)
```

Table 5.1: Message Descriptors

Type	Description
SEND MESSAGE DESCRIPTOR	
"nil"	31 … 0 0xFFFFFFFF (32) IPC does not include a send operation.

continued on the next page

Table 5.1: (continued)

Type	Description
"mem"	31 … 2 1 0 — [snd msg/4 (30) | m | ~] IPC includes sending a message to the destination specified by *dest id*. *snd msg* must point to a valid message. In case of x86 architecture the first two 32-bit words of the message are *not* taken from the message data structure but must be contained in registers EDX and EBX. For more detailed view of generic IPC message see Figure 5.11.
"reg"	31 … 2 1 0 — [0 (30) | m | ~] IPC includes a message to the destination specified by *dest id*. Message is purely register based. For x86 architecture message consists solely of the three 32-bit words in registers EDX, EBX and EDI.
$m = 0$	Value-copying send operation; the dwords or the message are simply copied to the recipient.
$m = 1$	Flex-page-mapping send operation. The dwords of the message to be sent are treated as 'send flex-pages'. The described flex-pages are mapped (respectively granted) into the recipient's address space. Mapping/granting stops when either the end of the dwords is reached or when an invalid flex-page descriptor is is found, in particular 0. The send flex-page descriptors and all potentially following words are also transferred by simple copy to the recipient. Thus, a message may contain some flex-pages and additional value parameters. The recipient can use the received flex-page descriptors to determine what has been mapped or granted into its address space, including location and access rights.
RECEIVE MESSAGE DESCRIPTOR	

continued on the next page

Table 5.1: (continued)

Type	Description
"nil"	31 … 0: 0xFFFFFFFF (32) IPC does not include a receive operation.
"mem"	31 … 2, 1, 0: rcv msg/4 (30) | 0 | o IPC includes receiving a message respectively waiting to receive a message. *rcv msg* must point to a valid message. For x86 architecture the first two 32-bit words of the received message are *not* stored in the message data structure but are returned in registers EDX, EBX and EDI. For more detailed view of generic IPC message see Figure 5.11.
"reg"	31 … 2, 1, 0: 0 (30) | 0 | o IPC includes receiving a message respectively waiting to receive a message. All data transferred via registers. On x86 architecture only messages up to three 32-bit words are accepted. The received message is returned in registers EDX, EBX and EDI. For more detailed view of generic IPC message see Figure 5.11.
"rmap"	31 … 2, 1, 0: rcv fpage (30) | 1 | o IPC includes receiving a message respectively waiting to receive a message. Message includes mapping. On x86 architecture only send-flex-page or up to three 32-bit words are accepted. The received message is returned in registers EDX, EBX and EDI. If a map message is received, *rcv fpage* describes the receive flex-page (instead of *message receive flex-page option* in a memory message buffer). Thus, flex-pages can also be received without a message buffer in memory.
$o = 0$	Only messages from the thread specified as *dest id* are accepted ("closed wait"). Any send operation from a different thread (or hardware interrupt) will be handled exactly as if the actual thread would be busy.

continued on the next page

Table 5.1: (continued)

Type	Description
$o = 1$	Messages from any thread will be accepted ("open wait"). If the actual thread is associated with a hardware interrupt, also messages from messages from this hardware interrupt can arrive.

5.5.2 Short Messages

Common Format

Short messages are passed directly in registers. Amount of transferred data is limited by architecture design. These amount of data goes through the registers even in case of a long message. Format of short message is shown in Figure 5.3.

The presence of flex-pages is indicated by setting the m-bit in the message descriptor. Processing of flex-pages starts at the beginning of the message and continues until an invalid flex-page is encountered. This last dword and the remainder of the in-line data is interpreted as by-value data.

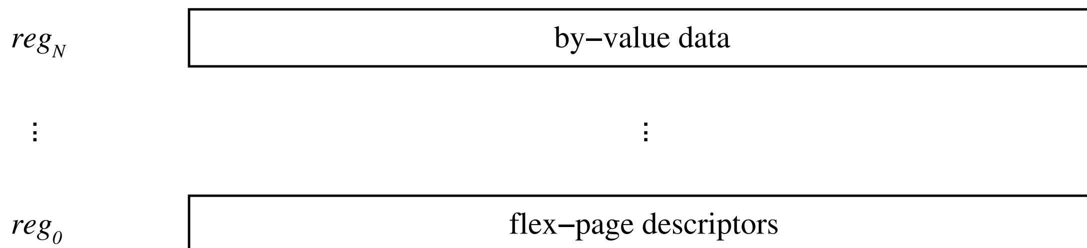

Figure 5.3: Format of the Short Message

Architecture: MIPS R4k

Every successful IPC operation will always copy at least eight dwords to the receiver. These eight dwords contain the first 64 bytes of a message's in-line data. The short message is transferred via eight registers (from s_0 to s_7, see Figure 5.4).

Architecture: x86

Register messages consist of up to 3 words of 32 bits. Upon sending, the message is located in the registers EDX, EBX, and EDI (see Figure 5.5). Upon receiving, the same

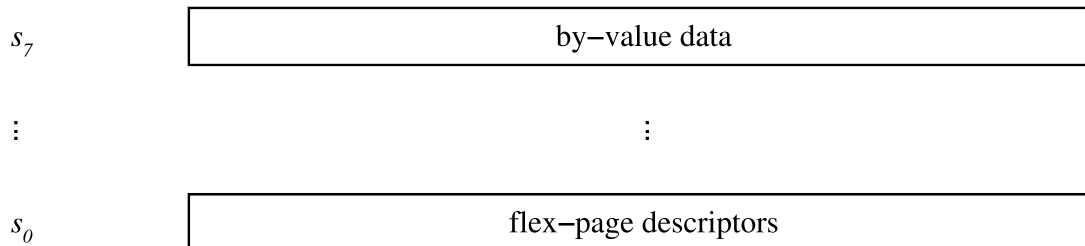

Figure 5.4: Format of the Short Message

registers serve as a buffer, i.e. the registers EDX, EBX, and EDI contain the received message (where send EDX is received EDX, etc.).

Since flex-page descriptor occupies two dwords, only one mapping can be sent via register within one short message. Format of flex-page descriptor described in section 5.5.5.

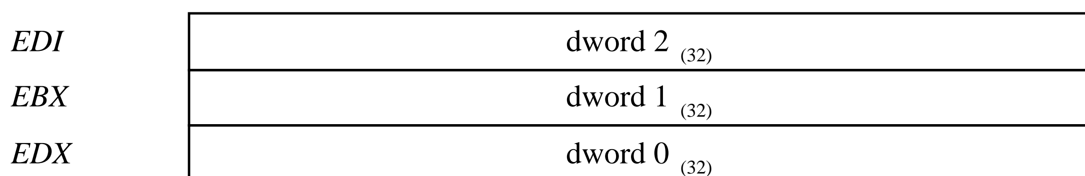

Figure 5.5: The Short Message via 3 Registers

Sending the Short IPC Message on x86

Note that since for sending and receiving operations L4Ka uses the same system call (`l4_ipc_call`), it contains both send and receive part (each of them is optional). We are now not interested in IPC messages like *Send to & Receive* but only in simple send operation.

According to requirements on page 71 it is necessary to set up following parameters in order to send short message:

- *dest id* must be set to destination thread's identifier;
- *snd msg* descriptor must be set either to `L4_IPC_SHORT_MAPMSG` (if message contains mapping/granting) or to `L4_IPC_SHORT_MSG` (otherwise);
- *rcv msg* descriptor defines the reply action. Since message does not contain receive part, *rcv msg* descriptor must be set to `L4_IPC_NIL_DESCRIPTOR`;
- three dwords to be sent. In the case of IPC with mapping on x86 architecture flex-page descriptor is stored in first (*snd base*) and second (*snd fpage*) dwords (see Section 5.5.5);

– three pointers on dwords `ptr_t` to be received. Since message does not contain receive part these fields are not significant;

– *timeout* must be provided as in any IPC;

– pointer to `l4_msgdope_t` structure where to store the result status of the IPC operation.

Simple example of send operation without mappings and reply presented in Listing 5.2.

Listing 5.2: Sending short IPC message without mapping (and receive part)

```
l4_threadid_t dest_id;        //here the destination id must be stored
dword_t dummy;
l4_msgdope_t result;          //message dope to store result status
//then goes IPC call
l4_ipc_call2(
    dest_id,                  //destination thread id
    L4_IPC_SHORT_MSG,         //send message descriptor
    1,                        //first dword to send
    2,                        //second
    3,                        //third
    L4_IPC_NIL_DESCRIPTOR,    //no receive operation
    &dummy,
    &dummy,
    &dummy,
    L4_IPC_NEVER,             //timeout never expires
    &result                   //result status of the IPC
    );
```

Receiving the Short IPC Message

According to requirements on page 71 it is necessary to set up following parameters in order to receive short message:

– *src id* must be set to source thread's identifier;

– *snd msg* descriptor must be set to `L4_IPC_NIL_DESCRIPTOR` (since IPC does not contain send part);

– *rcv msg* descriptor must be set either to `L4_IPC_SHORT_RECEIVE_FROM` (*Receive from* — to receive message only from source thread declared in *src id* field) or to `L4_IPC_SHORT_OPEN_RECEIVE` (*Open Receive* — to receive message from any source). In the case of the *Open Receive* operation invoker should also provide pointer to `l4_threadid_t` as src parameter to `l4_ipc_call`. Via this param-

eter UID of communicating thread will be returned. For more details about C interface, see Section 4.5;

- three dwords to be sent. Since there is no sending operation these fields are not significant;
- three pointers on dwords `ptr_t` to be received;
- *timeout* must be provided as in any IPC;
- pointer to `l4_msgdope_t` structure where to store the result status of the IPC operation.

Simple example of *Receive From* operation without mappings presented in Listing 5.3.

Listing 5.3: Receiving short IPC message without mapping from particular thread

```
l4_threadid_t src_id;            //here source id must be stored
dword_t dummy;
l4_msgdope_t result;             //message dope to store result status
dword_t dw0, dw1, dw2;           //dwords to store received data
//then goes IPC call
l4_ipc_call2(
    src_id,                      //source thread id
    L4_IPC_NIL_DESCRIPTOR,       //no send operation
    dummy,
    dummy,
    dummy,
    L4_IPC_SHORT_RECEIVE_FROM,   //receive from particular thread
    &dw0,                        //receive buffer for first dword
    &dw1,                        //               for second
    &dw2,                        //               for third
    L4_IPC_NEVER,                //timeout never expires
    &result                      //result status of the IPC
    );
```

5.5.3 Long Messages

The long part of the message is optional and its presence is indicated by the message descriptor (*snd msg/rcv msg*). If present, it is a dword-aligned memory buffer pointed to by a *message descriptor*. The buffer contains a three dword *message header*, followed by a number of `mwords` (the rest of the in-line data), followed by a number of string dopes. The number of `mwords` (in 32-bit dwords, excluding those copied in registers) and string dopes is specified in the *message header*.

The format of the long part of the message is depicted in Figure 5.6.

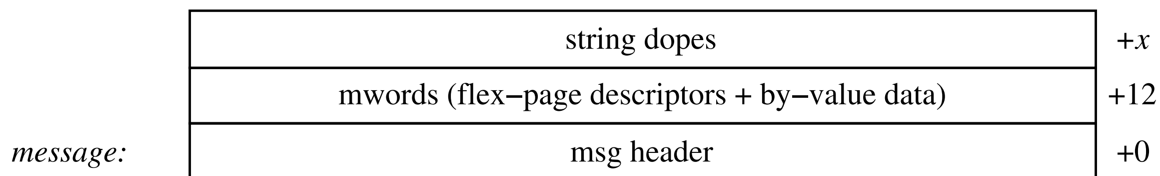

Figure 5.6: Format of the Long Message

The value of x is determined by the number of `mwords` in the message as specified in the *message size dope* of the *message header*.

5.5.4 Message Header

The *message header* describes the format of the long message. It is illustrated in Figure 5.7.

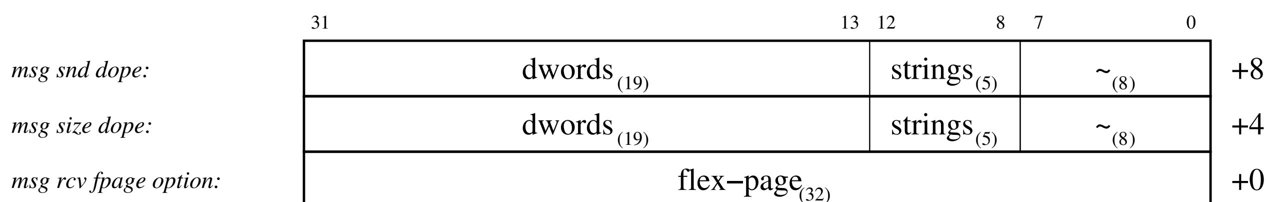

Figure 5.7: Format of the Message Header

Meaning of fields is following:

- **message size dope** (*msg size dope*) defines the size of the dword buffer, in dwords, (and hence the offset x of the string dopes from the end of the header), and the number of string dopes in the long IPC message;
- **message send dope** (*msg snd dope*) specifies how many dwords and strings are actually to be sent. Specifying message send dope values less than the message size dope values makes sense when the caller is willing to receive more data than it is sending;
- **message receive flex-page option** (*msg rcv fpage option*) describes the address range in which the caller is willing to accept flex-page mappings or grants in the receive part (if any) of the IPC. As described in section 3.4, a flex-page region is defined by providing its base address, b and size exponent, s. Note that the hot-spot, h, is provided by the sender and hence not required as part of the receive flex-page. It is common to provide flex-page that covers complete user address space as this option.

Using different message send/size dopes permits to specify not only pure send messages and pure receive message buffers. It is as well possible to send a message and receive the reply using the same data structure. It is also possible to receive a message in a message buffer and then forward this buffer as a message without changing the data structure.

x86 Implementation

Suggested definition located in file "`urriy.h`" (see Listing 5.4).

Listing 5.4: Message header declaration in "`urriy.h`"

```
typedef struct header
{
    l4_fpage_t rcv_fpage;
    l4_msgdope_t size_dope;
    l4_msgdope_t snd_dope;
} header_t;
```

5.5.5 Message `mwords`

Format

The (possibly zero) message `mwords` follow directly after the *message header* and contain the rest of the in-line data remaining after the short message. This in-line data is made up of a number (again, possibly zero) of flex-page descriptors followed by by-value data (see Figure 5.8).

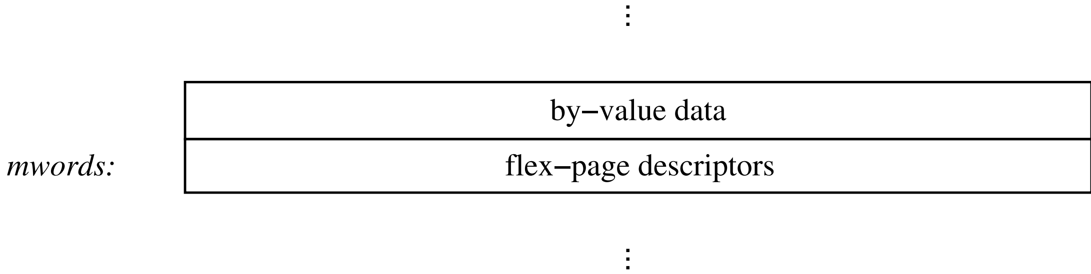

Figure 5.8: Format of the `mwords`

Note that on x86 architecture *the first three dwords of a message are always transferred via registers*. It is a simplification for user-level programming. This permits to handle long (memory) and short (registers) messages basically in the same way (the first three dwords are always in registers). Loading/storing those registers from/to the message/buffer data structure is *not* handled by the L4Ka kernel. It can be done at user level. Note also that

dwords field of the *msg size dope* of *message header* must be set correctly, even if it is a pure send message, because it defines the position x where the first string dope starts (see Figure 5.11).

Flex-page Descriptor

Flex-page descriptors are expected in memory (the long message) only if the m-bit is set in the message descriptor and all register data (the entire short message) consists of valid flex-pages. These flex-page descriptors (together with those in the short message) are provided by the sender for memory mapping purposes. The format of an flex-page descriptor is depicted in Figure 5.9.

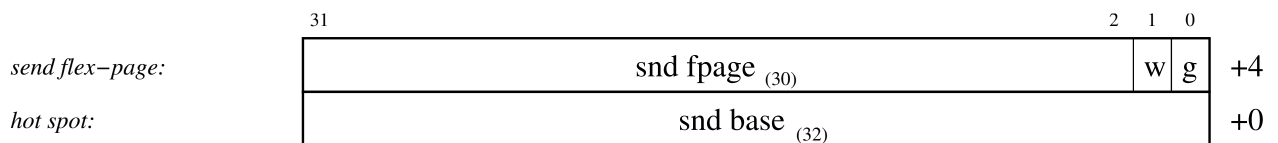

Figure 5.9: Format of the Flex-page Descriptor

Flex-page descriptor is often called “send flex-page” or “snd fpage”. It has the same format as the *message receive flex-page option* in the *message header* except the least significant two bits are no longer undefined. Setting the w bit will cause the flex-page to be mapped writable (rather than just read-only) and setting the g-bit causes the flex-page to be granted (rather than just mapped). The *snd base* is the mapping *hot-spot* as described in section 3.4.

The kernel will interpret each pair of dwords of the in-line part (starting from the short message part and continuing to the long message part, if present) as flex-page descriptors until an invalid descriptor is encountered. This and any feather dwords are then interpreted as by-value data. Note that all in-line data is copied to the receiver, including any initial parts, which are interpreted as flex-page descriptors.

Flex-page descriptor is defined as structure `l4_snd_fpage_t` in file “`types.h`” as presented in Listing 5.5.

Listing 5.5: Flex-page descriptor declaration in “`types.h`”

```
typedef struct {
  dword_t snd_base;
  l4_fpage_t fpage;
} l4_snd_fpage_t;
```

5.5.6 String Dopes

Definition

The last components making up a long message are string dopes. There can be zero or more string dopes, with the exact number specified in the *message size dope* of the *message header*. Each string dope describes a region in memory where out-of-line data can be copied from (on an IPC send) and copied to (on an IPC receive). The size and location of each string is specified in a string dope. The kernel copies data from sender memory, specified by the receiver's string dopes. Each string dope occupies four dwords and its format is presented in Figure 5.10.

The first part of the string dope specifies the size and location of the string the caller wants sent to the destination, while the second part specifies the size and location of a buffer where the caller is willing to receive a string. Note that strings do not have to be aligned, and that their size is specified in *bytes*.

Every string dope can specify a send string dope and a receive string dope. For a send message, the receive buffer part is ignored. For a receive buffer, the *rcv string* address specifies the buffer and *rcv string size* specifies maximum length; after message was received, the *snd string* address is set tot the beginning of the receive buffer and the *snd string size* specifies the current length. Thus, received dopes can be forwarded by a successive send operation without any change.

Format

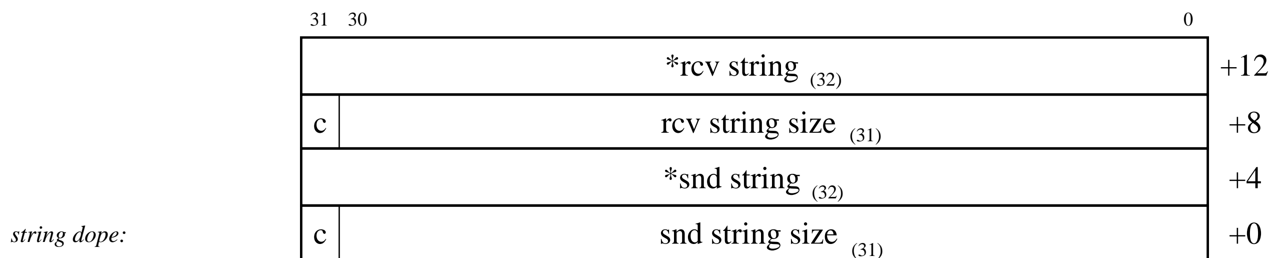

Figure 5.10: Format of the String Dope

Format of string dope is depicted in Figure 5.10. The c-bit enables *scatter/gather* functionality. $c = 0$ specifies the begin of a logical string; $c = 1$ specifies that this is a continuation of the last logical string. On the sender side, *logical string* means a sequence of one or more parts (dopes) that are transferred as if they were one contiguous string (*gather*). On the re-

ceiver side, *logical string* means a sequence of one ore more buffers that are treated as one logical buffer; the corresponding received string is *scattered* among them. Continuations can be arbitrarily combined on sender and receiver side. Note that length and size fields are always per part.

x86 Implementation

String dopes are defined as structure `l4_strdope_t` in file "`types.h`" and are presented in Listing 5.6.

Listing 5.6: String dopes declaration in "`types.h`"

```
typedef struct {
  dword_t snd_size;
  dword_t snd_str;
  dword_t rcv_size;
  dword_t rcv_str;
} l4_strdope_t;
```

5.5.7 Generic Long IPC Message

Generic long IPC message of arbitrary content can be constructed w.r.t. the Figure 5.6. It is shown in Figure 5.11. Note that the first three dwords are not taken from `mwords` buffer followed by the *message header*, but are transferred via registers (EDX, EBX, EDI in x86 architecture, see Figure 5.5). Note also, that if m-bit is set in the message descriptor, then content of `mwords` buffer must be treated according to Figure 5.8. Thus, it can contain arbitrary many flex-page descriptors (see Figure 5.9). And with the first incorrect descriptor by-value data begins.

5.6 L4Ka IPC Message Summary

In summary, an L4Ka IPC operation can have a send and a receive phase. The *snd msg* descriptor describes what is to be sent , and the *rcv msg* descriptor describes what can be received. For the IPC to be successful, the sender's send descriptor and the receiver's receive descriptor must be compatible.

Every successful IPC transfers some by-value data, the register (or short) part of the string (8 bytes on x86, 64 bytes on MIPS R4k). In addition, the following may be transferred, provided the *snd msg* descriptor says so, and the *rcv msg* descriptor allows it:

a) a bigger string, provided that:

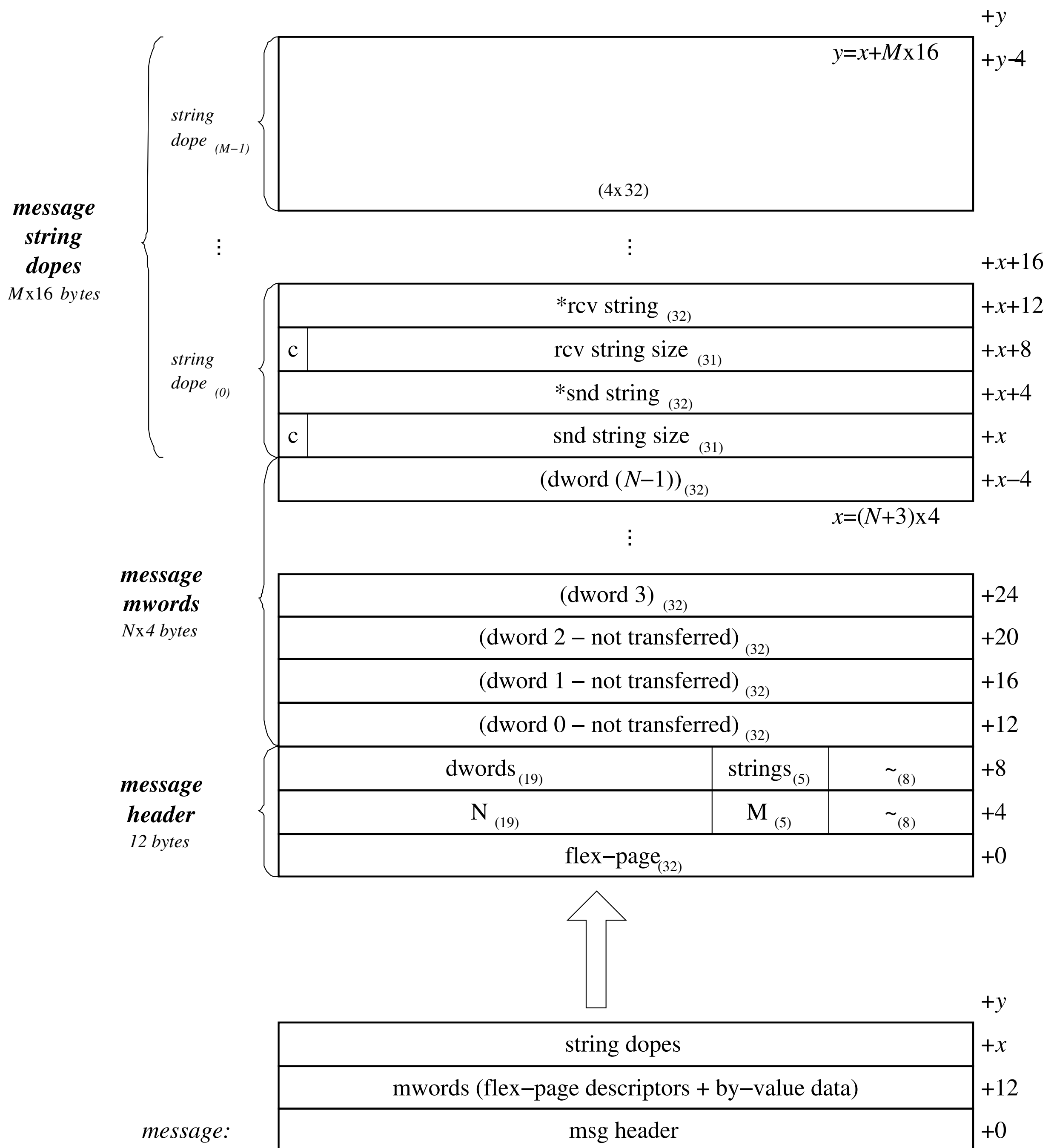

Figure 5.11: Generic Format of the Long IPC Message

1) the send descriptor points to a message descriptor specifying a non-zero number of dwords in the *message send dope*, and

2) the receive descriptor points to a message descriptor specifying a non-zero number of dwords in the *message size dope*, and

3) there is no error;

b) one or more indirect strings, provided that:

1) the send descriptor points to a message descriptor specifying a non-zero number of strings in the *send dope*, and
2) the receive descriptor points to a message descriptor specifying a non-zero number of strings in the *size dope*, and
3) there is no error;

c) one or more page mappings or grants, provided that:
1) the m-bit is set in the send descriptor, and
2) the beginning of the sender's direct string (starting with the register part) contains at least one valid flex-page descriptor, and
3) the receive descriptor either has the form of a valid *message receive flex-page* and has the m-bit set, or points to a message descriptor containing a valid *message receive flex-page*, and
4) there is no error;

Note that the structure of the message descriptor and the string dopes make it easy to use the same *message header* for sending and receiving (i.e., having the *send descriptor* and *receive descriptor* point to the same address). The *message send dope* specifies the size of the direct string and the number of indirect strings for the send part (if any) of the IPC, while the *message size dope* specifies the size of the buffer for the direct string and the number of buffers for indirect strings for the receive operation (if any).

Note that using the same message descriptor for sending and receiving implies using the same buffer for the direct string. For indirect strings, each string dope specifies separately the location and size of the buffers for the strings to be sent and received. If the IPC does not contain a send **and** a receive part, then some of the information in the *message header* is not used. Similarly, if the send and receive descriptors point to different message structures, some of the information in them is unused.

Obviously, if the same message descriptor is used for sending and receiving, receiving a direct string (longer than the register part) will overwrite the string sent. Similarly, if the *receive string* of some string dope points to the same address as the *send string* of the same or another string dope, then receiving may overwrite some of the data which has been sent. However, as the send part of the IPC is guaranteed to be concluded before any receive action takes place, this does not create any problems if the sender does not need the data any more. The data to be sent will have been safely copied to the receiver prior to the receive part of the IPC overwriting it ont the caller's end.

5.6.1 Send/Receive Protocol

In the overview of this chapter, it was mentioned that the sender and receiver of an IPC must make certain agreements. Sender and receiver must agree on the following points for the IPC:

- the size of data to be copied;
- the number and size of strings to be transferred;
- whether the IPC involves memory mappings (the presence of flex-pages).

The kernel does not provide the receiver with any information (e.g. size of data) concerning the incoming message. Thus, a user message protocol needs to be defined before hand to ensure agreement on the above points. In general, the receiver can expect more from the sender than is actually sent but not less.

Examples of such a protocols are:

- the sender arranges in-line data into a particular structure that the receiver is expecting. Depending on the structure and context, the receiver may not make use of the entire structure but must receive the entire structure regardless (e.g. one field of the structure may identify the context and consequently how the other fields are to be interpreted);
- the receiver always receives a maximum number of maximum size strings;
- use two IPCs. The first one allows the sender to establish an agreement with the receiver. The second IPC is the main message in the form established by the first IPC.

5.6.2 Sending Message (Generic Algorithm)

Following recommendations extends those, which are given in Section 5.4.

Sending an IPC message in general can be divided into following steps:

a) declare a result status variable (of type `l4_msgdope_t`);

b) decide the message format:

   1) in the case of the short IPC message only three dwords transferred to the receiver via registers (as by-value data, or mapping);

   2) in the case of the long IPC message number of `mwords` and strings must be defined in the *message header*;

   3) message descriptor must be defined according to the message format;

c) determine the thread id of the desired receiver thread;

d) determine the desired timeout period;

e) provide the parameters for `l4_ipc_call`.

5.6.3 Receiving Message (Generic Algorithm)

Following recommendations extends those, which are given in Section 5.4.

Receiving an IPC message in general can be divided into following steps:

– declare a result status variable (of type `l4_msgdope_t`);

a) decide the message format:

    1) in the case of the short IPC message only buffer for three dwords is needed;

    2) in the case of the long IPC message amount of buffers for `mwords` and strings must be defined in the *message header* (inside the *message size dope*);

    3) message descriptor must be defined according to the message format;

b) for a closed receive (*Receive from*), determine the thread id of the desired sender thread. For an *Open Receive*, declare a variable to store the thread id of the sender;

c) determine the desired timeout period;

d) provide the parameters for `l4_ipc_call`.

5.7 Examples

Suggested examples cover most frequent scenarios of use of L4Ka IPC.

5.7.1 Short IPC Messages between Threads

Listing 5.7: Short IPC messages between two threads

```
/************************************************************
    L4 IPC example
    Communication between threads using short messages.
    Protocol:
        1. foo2 sends data to foo1 in short message
        2. foo1 sends acknowledge if data is received ok
            or result status of IPC in case of fail
        3. foo2 outputs received acknowledge

************************************************************/
```

```c
#include "urriy.h"


void send (void); //function to send out data to another thread
void foo1 (void);
void foo2 (void);

#define FOOSTACKSIZE 1024

    l4_threadid_t foo1_tid, foo2_tid; //thread ids of communicating threads
    l4_threadid_t main_tid; //thread id of the root_task
    dword_t foo1stack[FOOSTACKSIZE]; //stacks
    dword_t foo2stack[FOOSTACKSIZE];
    dword_t dummy;

//________________________________FOO1
void foo1()
{

    outstring("\n foo1: started with thread id:"); outhex32(foo1_tid.raw);

    l4_msgdope_t dope; //msg_dope to store the result status
    dword_t dw0, dw1, dw2; //dwords to be received

receiving:
    outstring("\n foo1: blocked in receiving...");

//  ipc receive from prototype
// You can use l4_ipc_receive function from ipc.h for this type of IPC
l4_ipc_call2(
        foo2_tid,
        (void *) L4_IPC_NIL_DESCRIPTOR, //no send operation
        dummy,
        dummy,
        dummy,
        (void *) L4_IPC_SHORT_RECEIVE_FROM,//0 - receive from, 1 - open receive
        &dw0,
        &dw1,
        &dw2,
        L4_IPC_NEVER, //timeout never expires
        &dope); //result status


    if (L4_IPC_ERROR(dope))
    {
        outstring("\n foo1: error while receiving a message");
        outstring("\n foo1:  the result msgdope of IPC is ");
        outhex32(dope.raw);outchar('\n');
        dw0 = dope.raw;
```

```
    }
    else
    {
    outstring("\n foo1: has received IPC message successfully: ");
        outstring("\n foo1:  dw0= ");outhex32(dw0);
        outstring("\n foo1:  dw1= ");outhex32(dw1);
        outstring("\n foo1:  dw2= ");outhex32(dw2);
    outchar('\n');
        dw0 = 200;
    }

//now sending some kind of acknowlegde
    dw1=0;
    dw2=0;
//function from ipc.h
l4_ipc_send(
        foo2_tid, //source thread id
        (void *) L4_IPC_SHORT_MSG,
        dw0,
        dw1,
        dw2,
        L4_IPC_NEVER,
        &dope);

    if (L4_IPC_ERROR(dope))
        {
        outstring("\n foo1: acknowlegde send fails");
    outstring("\n foo1:  the result msgdope of IPC is ");
        outhex32(dope.raw);outchar('\n');
        }

    while(1){l4_sleep(5000000);outchar('+');}

}

//_________________________FOO2
void foo2()
{
   outstring("\n foo2: started with thread id: ");outhex32(foo2_tid.raw);

    dword_t dummy;
    dword_t dw0, dw1, dw2; //dwords to store received data
    l4_msgdope_t dope; //msg_dope to store result status

//  ipc send prototype
// You can use l4_ipc_send function from ipc.h for this type of IPC
l4_ipc_call2(
        foo1_tid, //destination thread id
        (void *) L4_IPC_SHORT_MSG, //message type: short
```

```
        1,
        2,
        3,
        (void *) L4_IPC_NIL_DESCRIPTOR, //no receive operation
        &dummy,
        &dummy,
        &dummy,
        L4_IPC_NEVER, //timeout never expires
        &dope); //result status

    if (L4_IPC_ERROR(dope))
    {
        outstring("\n foo2: error while sending a message");
    outstring("\n foo2:  the result msgdope of IPC is ");
        outhex32(dope.raw);outchar('\n');
    }


//now receiving acknowlegde

//function from ipc.h
l4_ipc_receive(
        foo1_tid, //source thread id
        (void *) L4_IPC_SHORT_MSG,
        &dw0,
        &dw1,
        &dw2,
        L4_IPC_NEVER,
        &dope);

    if (L4_IPC_ERROR(dope))
        {
        outstring("\n foo2: acknowlegde receive fails");
    outstring("\n foo1:  the result msgdope of IPC is ");
        outhex32(dope.raw);outchar('\n');
        }
    else
        {
        if (dw0==200) outstring("\n foo2: received acknowlegde is OK");
            else
                {
                outstring("\n foo2: received acknowlegde is an error:");
                outhex32(dw0);
                }
        }


    while(1){l4_sleep(5000000);outchar('-');};
}
```

					ДП.991137.ПЗ	Лист
						90
Изм.	Лист	№ докум.	Подпись	Дата		

```
//_______________________MAIN
int main(dword_t mb_magic, struct multiboot_info_t* mbi)
{
    //thread id of the root_task
    main_tid = l4_myself();
    l4_threadid_t my_pager = get_current_pager(l4_myself());

    foo1_tid=main_tid;
    foo1_tid.id.thread = 8;

// foo1 thread creation via system call l4_thread_ex_regs
l4_thread_ex_regs(
        foo1_tid,
        (dword_t) foo1,
        (dword_t) &foo1stack[FOOSTACKSIZE - 1],
        &my_pager,
        &my_pager,
        &dummy,
        &dummy,
        &dummy);

    outstring("\n root_task: first thread is created with thread id ");
    outhex32(foo1_tid.raw);

// foo2 thread creation using wrapper from helpers.h
//parameters differs, so be careful
foo2_tid=main_tid;
foo2_tid.id.thread=9; //this only needed by foo1, to know the dest thread id

create_thread(
        9, //needs only id.threadid field
        foo2,
        &foo2stack[FOOSTACKSIZE-1],
        my_pager); //thread id of pager, not a pointer to it

    outstring("\n root_task: second thread is created with thread id ");
    outhex32(foo2_tid.raw);

    enter_kdebug("\n root_task: two threads succ-ly created!");

    while(1){l4_sleep(5000000);};
}
```

5.7.2 Long IPC Messages between Threads

Listing 5.8: Long IPC message between two threads

```
/*****************************************************************
    L4 IPC example
    Communication between threads using long messages.
         Message consists of three strings and five mwords.
    Protocol:
        1. root_task creates foo thread to communicate with
        2. then sends long message with strings gathered into one
        3. then sends a message of same format,
                        but strings are not gathered.

*****************************************************************/

#include "urriy.h"

//define the maximum size of strings to receive
#define MAX_BUF 255



//  First, defining the structure of the message
// We assuming the simple protocol - messages have
// 5 mwords and 3 strings.
const int Mwords=5;
const int Strings=3;

typedef struct msg
{
        header_t        header;
        dword_t         buf[Mwords];
        l4_strdope_t    strdope[Strings];
}
 msg_t;

// Thread ids of communicating threads
l4_threadid_t main_tid; //root_task's main thread
l4_threadid_t foo_tid;  // another thread of root_task

dword_t dummy;

//_________________FOO function
void foo()
{

    foo_tid = l4_myself();
        outstring(" foo: foo thread is started with UID:"); outhex32(foo_tid.raw);

    l4_msgdope_t dope; //msg_dope to store result status
        dword_t dw0, dw1, dw2; //first three mwords will go via registers!
        char sbuf[Strings][MAX_BUF]; //buffers for strings to receive
```

```
        msg_t longmsg;          //long message of predefined type

receiving:
                //clear buffers
        for (int j=0; j<Strings; j++)
        for (int i=0; i<MAX_BUF; i++)
                        sbuf[j][i]=0;
                //fill in message header fields
        longmsg.header.size_dope.md.dwords = Mwords;
        longmsg.header.size_dope.md.strings = Strings;
        longmsg.header.snd_dope.raw = 0;
                //fill in string dopes
                // receiving without 'scatter' option enabled
        for (int i=0; i<Strings; i++)
        {
                longmsg.strdope[i].rcv_size = MAX_BUF; //maximum receive size
                longmsg.strdope[i].rcv_str = (dword_t) &sbuf[i]; //pointer to start of buffer
                longmsg.strdope[i].snd_size = 0;
                longmsg.strdope[i].snd_str = 0;
        }

        outstring(" foo is blocked in receiving...");

l4_ipc_call2(
        main_tid,
        (void *) L4_IPC_NIL_DESCRIPTOR,
        0,
        0,
        0,
        (msg_t *) (((dword_t) &longmsg) & ~ (3)),
        &dw0,
        &dw1,
        &dw2,
        L4_IPC_NEVER,
        &dope
                );

    if (L4_IPC_ERROR(dope))
    {
        outstring("\n foo: error while receiving a message");
    outstring("\n foo:  the result msgdope of IPC is ");
        outhex32(dope.raw);outchar('\n');
    }
        else
        {
    outstring("\n foo: has received IPC message successufully");
        outstring("\n foo:  dw0= ");outhex32(dw0);
        outstring("\n foo:  dw1= ");outhex32(dw1);
        outstring("\n foo:  dw2= ");outhex32(dw2);
```

```
        outchar('\n');

        for (int i=0; i<Strings; i++)
        {outstring("\n foo:  string=");outstring(sbuf[i]);}

        for (int i=0; i<Mwords; i++)
                {outstring("\n foo:  mword= ");outhex32(longmsg.buf[i]);}
        }

enter_kdebug("\n breakpoint before next receiving");

        goto receiving;
}



dword_t foostack[1024];

//_________________________SEND function
void send(char *send_str1, char *send_str2, char *send_str3, bool together)
{
        l4_msgdope_t dope;//result status of IPC
        char *sbuf;
        msg_t longmsg; //long message of predefined type

                //fill in message header fields
        longmsg.header.snd_dope.raw=0;
        longmsg.header.snd_dope.md.dwords=Mwords;
        longmsg.header.snd_dope.md.strings=Strings;
        longmsg.header.size_dope=longmsg.header.snd_dope;

        longmsg.header.rcv_fpage.fpage=0;

        longmsg.strdope[0].snd_size = strlen(send_str1);

        if (together)
                {//if strings must be sent together
                // use the 'gather' functionality of send string dopes
                // by turning on most significant bit ('c')
                longmsg.strdope[1].snd_size = 0x80000000 | strlen(send_str2);
                longmsg.strdope[2].snd_size = 0x80000000 | strlen(send_str3);
                }
        else
                {// or just send stings one by one
                longmsg.strdope[1].snd_size = strlen(send_str2);
                longmsg.strdope[2].snd_size = strlen(send_str3);
                }

        //addressed of strings to be sent
```

```
        longmsg.strdope[0].snd_str = (dword_t) send_str1;
        longmsg.strdope[1].snd_str = (dword_t) send_str2;
        longmsg.strdope[2].snd_str = (dword_t) send_str3;

        //no receive option
        for (int i=0; i<Strings; i++)
                {
                longmsg.strdope[i].rcv_size = 0;
                longmsg.strdope[i].rcv_str = 0;
                }

        //fill in mwords buffer, starting from mword 3,
        // because first three are transferred via registers
        for (int i=3; i<Mwords; i++)
                {
                longmsg.buf[i]=i+65;//some values just to see, that it works
                }

l4_ipc_call2(
            foo_tid,
            (msg_t *) (((dword_t)&longmsg) &~(3)), //send decriptor for long message without
                mappings
            1,
            2,
            3,
            (void *) L4_IPC_NIL_DESCRIPTOR,
            &dummy,
            &dummy,
            &dummy,
            L4_IPC_NEVER,
            &dope);

    if (L4_IPC_ERROR(dope))
    {
                outstring("\n main: error while sending a message");
        outstring("\n main:  the result msgdope of IPC is ");
                outhex32(dope.raw);outchar('\n');
    }


};

//______________________MAIN
int main(dword_t mb_magic, struct multiboot_info_t* mbi)
{
        main_tid = l4_myself();

                //foo thread creation
    create_thread(10, foo, &foostack[1024], get_current_pager(l4_myself()));
```

					ДП.991137.ПЗ	Лист
						95
Изм.	Лист	№ докум.	Подпись	Дата		

```
        l4_sleep(1000000); //little delay is necessary for thread to get started
                // beware! otherwise you will get an error code 0x0001, which says that
                // source/destination thread id not exists!

        kd_inchar(); //wait for user to press a key

        //sending strings together
        send("These", " three string", " comes together...\0", true);

        kd_inchar(); //before next sent operation
        send("And these", " are", " not", false);

while(1){l4_sleep(1000000); outchar('-');};

}
```

6 L4ka Additional Information

6.1 Bootstrap

At boot time, the μ-kernel image is loaded into resident memory (RAM). The first process started (in kernel mode) by the L4Ka bootstrap is σ_0 (see Section 6.3). σ_0 then in turn starts up all the initial servers in user mode. All initial servers have registered as their pager and exception handler. σ_0 does not allocate any stack space for the initial servers. The initial servers must allocate their own stack space and point the stack pointer variable to the beginning of that allocated space.

After booting, L4Ka enters protected mode if started in real mode, enables paging and initializes itself. It generates the basic address space-servers σ_0 and a root server task (also known as a *root task*) which is intended to boot the higher-level system. σ_0 and the root task are user-level tasks and are not parts of the μ-kernel. The predefined ones can be replaced by modifying the EIP (Entry Instruction Pointer) in the kernel configuration before starting L4Ka. The kernel debugger *kdebug* is also not part of the μ-kernel and can accordingly be replaced by modifying the table. For more detailed information about architecture independent kernel debugger *kdebug* for Hazelnut kernel, refer to [12].

6.2 Page Fault Handling

A page fault occurs when a task tries to access (read from or write to) memory that has not already been mapped into its virtual address space. A pager is a thread that handles page faults by determining the actions to be taken in the event of a page fault (usually give a mapping for the faulting address to the faulter).

When a new task/thread is created, a pager is registered with that task/thread (remember the task creation procedure in Section 4.2). The registered pager is then responsible for handling all the thread's page faults. When a client thread triggers a page fault, the L4Ka kernel intercepts the interrupt and sends an IPC to the pager on the client's behalf. That is,

the kernel sends the client's faulting address and instruction pointer in the first two words of a short IPC message to the registered pager pretending that the IPC actually comes from the faulting thread.

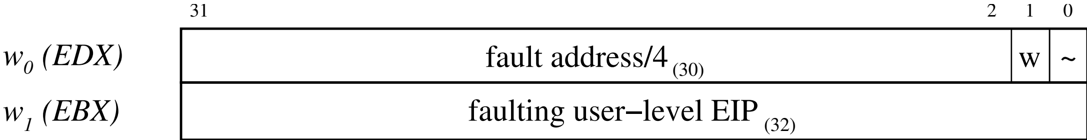

Figure 6.1: IPC from Kernel to the Pager of the Faulter

Faulting user-level Entry Instruction Pointer shows where faulter have caused page fault. *fault address* is truncated to thirty bits. Meaning of w-bit is following:

$$w = 0 \quad \text{Read page fault;}$$
$$w = 1 \quad \text{Write page fault.}$$

The pager will receive the page fault message as if it were directly from the faulter. It can then respond by sending a flex-page mapping for the faulting address back to the faulter or may implement another policy. The client does not actually receive the pager's mapping as the mapping is intercepted by the kernel on the client's behalf. The kernel restarts the client with the new mapping in place.

Interaction between the thread, its pager and the kernel is shown in Figure 6.2. Thread is trying to access memory for which no mapping exists. Hence, Memory Management Unit (MMU) raises a page fault exception. L4Ka looks up the pages of the running (and thus faulting) thread. The page fault is transformed into and IPC message describing the fault (in form described in Figure 6.1), and then is forwarded to the pager. The pager is also a thread, which may then decide how to react on the page fault. It encodes its answer as a page mapping IPC message, with which it replies to the faulter. L4Ka sees that this is a mapping message, intercepts it, and programs the MMU to provide the corresponding mapping in the address space of the faulter.

σ_0 (see Section 6.3) is an example of a pager service. It is in fact the main pager because it handles the initial address space and all mappings can be traced back to σ_0. The other type of pagers, which stem from σ_0 are the *external pagers*. External pagers are user-level threads that perform the task of page fault handling for other threads. External pagers themselves originally obtain their mappings from an intermediate pager (which is just another external pager).

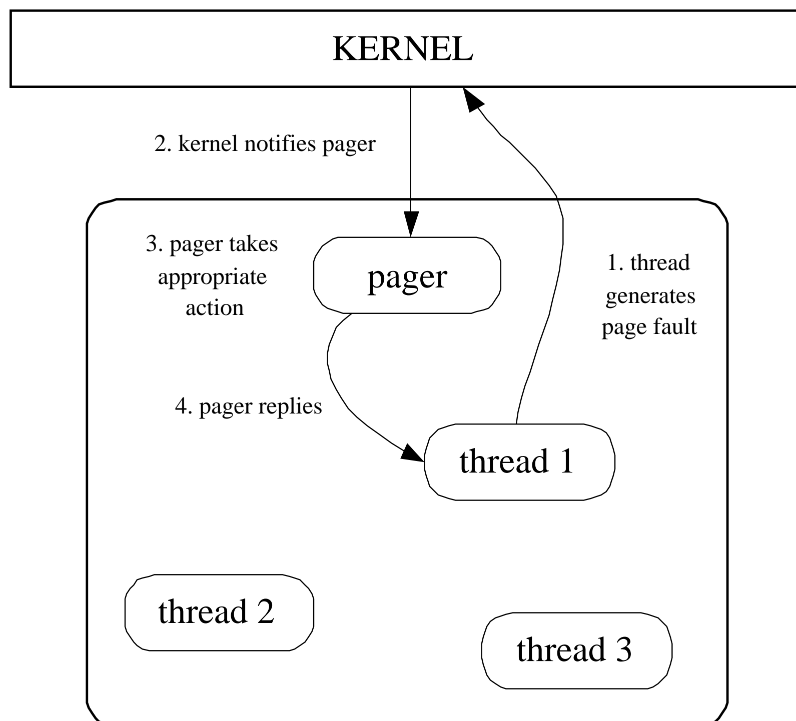

Figure 6.2: Interaction between Thread, its Pager and Kernel

Pager Thread

Basic tasks that a user-level pager needs to perform:

- wait for a page fault message from any client (to which it is assigned upon creation of task/thread);
- obtain a mapping of the faulting address for itself (if it doesn't already have it!) before it can pass the mapping on to the faulter;
- construct the send flex-page descriptor for the mapping to the faulting client. The single mapping to be sent is the same size as the single hardware page ($s = 12$) and has the fault address as the send flex-page base (b = fault address) and also the hot-spot. Function `l4_fpage` (defined in "`types.h`") can be useful to construct a flex-page;
- send the mapping to the client;
- return to the start to wait for the next client page fault.

Example of a simple pager thread was given in Section 4.2. As it was mentioned above, a valid page fault handler UID must be provided for `l4_task_new` system call to activate a task.

6.3 The Main Pager σ_0

An *initial address space*, called σ_0 automatically exists after the system is booted. Although it is *not* a part of the kernel, its basic protocol is defined with L4Ka::Hazelnut μ-kernel. Special σ_0 implementations may extend this protocol.

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

σ_0 gives pages to the kernel and to arbitrary tasks, but only once. The idea is that all pagers request the memory they need in the startup phase of the system so that afterwards σ_0 has spent all its memory. Further requests will then automatically denied (a null reply is sent). This enables the OS to have full control of memory (other than what is reserved by L4Ka). The OS can then provide its own pager, which maps memory to user tasks (with the appropriate checks) in response to user page faults.

Part of the kernel reserved space contains the *kernel information page* and other kernel information (their purpose is not covered in this document, refer to [11]). These special pages are mapped read-only upon request. Such mappings are part of the σ_0 protocol.

6.3.1 σ_0 Protocol

A task can request a mapping from σ_0 by sending a short message (see Section 5.5.2) to σ_0. The specific request is determined by up to the first two words in the register data of the request. If the request is valid, σ_0 sends a mapping to the requester. Note that σ_0 distinguish requests from kernel and non-kernel threads, and reacts accordingly.

User-level programmer need not to know σ_0 protocol, because in L4Ka::Hazelnut the pager of the root task is the intermediate pager, not σ_0 (see Figure 6.3). As it was mentioned above, σ_0 is a user-level application. The inclusion of σ_0 into the L4μ-kernel is specific only for L4/MIPS (see Figure 4.2 in [15]). Source code of σ_0 is located in applications directory (`apps/sigma0/main.c`). At the end of this chapter (in Examples section) presented extended σ_0 protocol (see Listing 6.2) and example of utilizing it in task creation issue (see Listing 6.1).

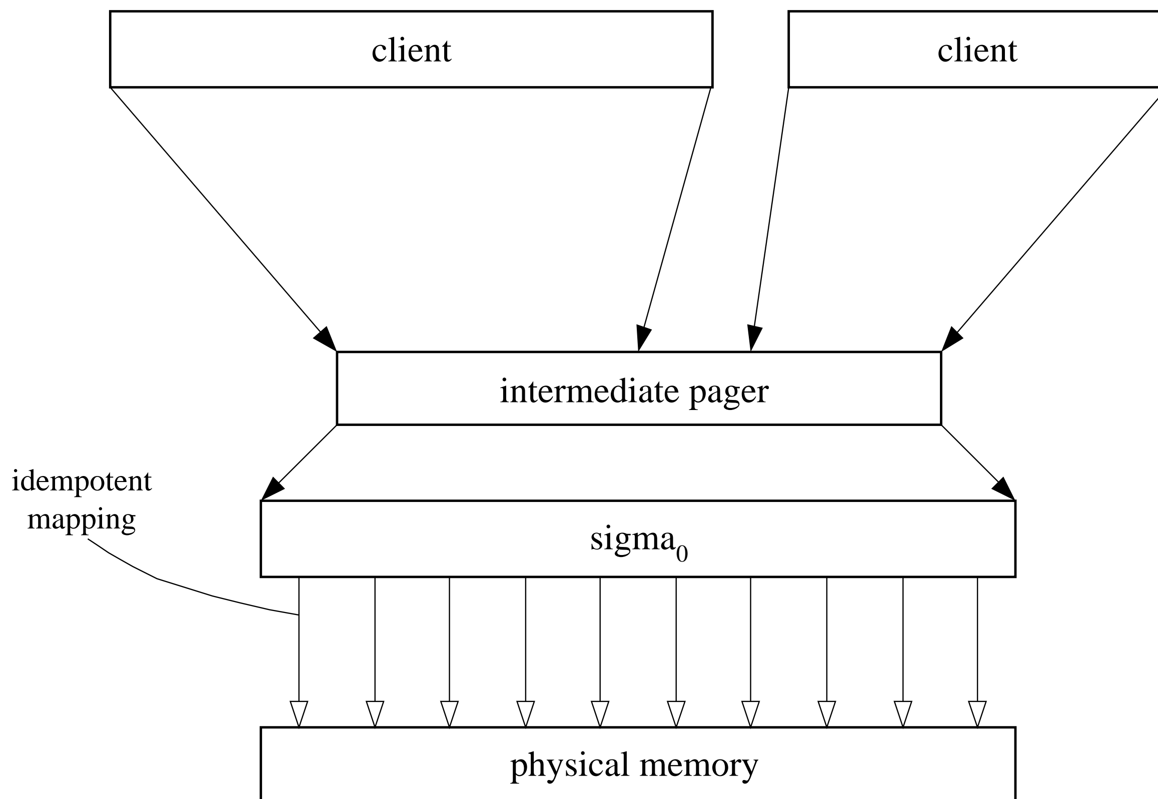

Figure 6.3: The Main Pager σ_0

6.4 Interrupt Handling

A thread can install itself as an interrupt handler for a particular interrupt by associating itself with the interrupt. A thread that wants to associate itself with an interrupt needs to invoke a receive IPC specifying the interrupt number (plus one) as the sender (*src* parameter) and a zero timeout.

Each interrupt is assigned to the first thread that attempts to associate with it. Any attempt to associate with an already associated interrupt will fail. A thread can dissociate itself as a handler for an interrupt by associating with a NULL interrupt.

Once associated with an interrupt, a handler waits for interrupts by invoking a receive IPC again but with non-zero timeout. Thus, in both actions needed to handle interrupt (interrupt association and interrupt receiving) `l4_ipc_call` is used. Example at the end of this chapter shows precise step-by-step procedure of interrupt handling (see Listing 6.3).

6.5 Preemption

There exists only one internal preempter for every thread, which is invoked after each timer interrupt if the time slice of running thread is expired or if it is called directly by kernel or user (`l4_thread_switch` system call). This internal preempter is essential for the system itself. The scheduling is done by the `l4_thread_schedule` function. External preempter: that is not a really preempter, but a user-level scheduler. The preemption is done by the kernel. The External preempter is scheduled by the kernel if no task/thread with higher priority is ready. The intended purpose of an external preempter is to refine the kernels scheduling method with any user-level method.

Preemption is not implemented in L4Ka::Hazelnut. But there are present related parameters in system calls: *external preempter*, *internal preempter*, *maximum controlled priority*.

6.6 Examples

6.6.1 σ_0 Extension

σ_0 protocol was extended. When sending long IPC message with `0xCCCCCCCC` in first dword and only one string attached σ_0 output this string on the screen. σ_0 writes data directly to text area of video memory (starting from address `0xb8000`).

Listing 6.1: Example of root task utilizing σ_0 extension

```
/**********************************************************************
 * File path:     root_task/main.c
 * Description:   utilizing sigma0 extension
 *                new task is created by the root task
 *                      and notifies its start by writing on
 *                      screen. Not using standard printf/outstring
 *                      stuff, but using sigma0 extension
 *                pager: on page fault asks pager of itself for
 *                      this page. then delivers it via short IPC message
 *                      with mapping to the faulter. works only with 4k pages.
 * 04.02.04 urriy@wjpserver.cs.uni-sb.de
 * Note that this example only works if file apps/sigma0/sigma0.c
 *   replaced by apps/sigma0/sigma0ext.c
 *********************************************************************/
#include "urriy.h"

l4_threadid_t main_tid;
l4_threadid_t subtask_tid;
```

```
#define PAGERSTACKSIZE 4096
dword_t pager_stack[PAGERSTACKSIZE];

#define SUBTASKSTACKSIZE        4096
dword_t subtask_stack[SUBTASKSTACKSIZE];

void pager(void);
void subtask(void);
l4_threadid_t pager_tid;
l4_threadid_t dummy_tid;

dword_t dummy;

        //structure of message to be sent to the sigma0
typedef struct msg
        {
        header_t header;
        l4_strdope_t strdope;
        } msg_t;

//_________________________SEND
void send(char *send_str)
{
        l4_threadid_t tid = l4_myself();
        l4_msgdope_t msgdope; //msgdope for result status
        char *sbuf; //string to be sent
        msg_t longmsg;

        longmsg.header.snd_dope.raw=0;
        longmsg.header.snd_dope.md.dwords=0;
        longmsg.header.snd_dope.md.strings=1; //sending one string
        longmsg.header.size_dope=longmsg.header.snd_dope;

        longmsg.header.rcv_fpage.fpage=0;

        sbuf=send_str;

        longmsg.strdope.snd_size = strlen(sbuf); //size of string
        longmsg.strdope.snd_str = (dword_t)sbuf; //starting address
        longmsg.strdope.rcv_size = 0;
        longmsg.strdope.rcv_str = 0;

                        //send string to output
l4_ipc_send(L4_SIGMA0_ID,
                        //long message without mappings
                (msg_t *) (((dword_t)&longmsg) & ~(3)),
                0xCCCCCCCC, //first dword according to extended protocol
                1,
                2,
```

```
                L4_IPC_NEVER,
                &msgdope);

                        //dummy reply from sigma0
l4_ipc_receive(L4_SIGMA0_ID,
                NULL,
                &dummy,
                &dummy,
                &dummy,
                L4_IPC_NEVER,
                &msgdope);
};

//________________________________SUBTASK
void subtask()
{
        send(" New task is started...\0");
        kd_inchar();
        send("  ... nothing to do, exiting.\0");

while(1) {l4_sleep(1000000);outchar('.');}

}

//________________________________PAGER
void pager()
{
        l4_threadid_t client;
        l4_msgdope_t dope;
        dword_t dw0, dw1, dw2;
        dword_t map=2;
        dword_t fault_addr;
        dword_t addr;

        l4_threadid_t s0 = L4_SIGMA0_ID;
        l4_threadid_t h_pager = get_current_pager(l4_myself());

while(4)
 {
        outstring("\n pager: waiting IPC message from kernel ");
        l4_ipc_wait(&client, 0, &dw0, &dw1, &dw2, L4_IPC_NEVER, &dope);

        while(5)
        {
                //fault address without mask
        fault_addr = (dw0 & (~(dword_t) 3));
                //let pager notify higher level pager that writing access is required
        dw0 = fault_addr | 2;
        outstring("\n pager: handling a page fault at address ");
```

```
outhex32(fault_addr);

        //asking page from higher level pager
l4_ipc_call(h_pager,
        L4_IPC_SHORT_MSG,
        dw0,
        dw1,
        dw2,
                //accepting page in whole address space
        (void *) l4_fpage(0, L4_WHOLE_ADDRESS_SPACE, 1, 0).raw,
        &addr, //address to be returned in the first dword
        &dummy,
        &dummy,
        L4_IPC_NEVER,
        &dope);

if (L4_IPC_ERROR(dope))
        {
        outstring("\n pager: error while asking for page ");
        break;
        }

        outstring("\n pager: memory page is given by higher level pager. address ");
        outhex32(addr);

        //first dword contains flex-page describing the memory region
dw0 &= L4_PAGEMASK;
        //second dword contains hot-spot which must be specified correct
dw1 = dw0 | (L4_LOG2_PAGESIZE << 2) | (L4_IPC_SHORT_MAPMSG);



        //send IPC message, which contains a mapping of desired page
        // and then wait for the next page fault
l4_ipc_reply_and_wait(client,
        (void *) L4_IPC_SHORT_MAPMSG,
        dw0,
        dw1,
        dw2,
        &client,
        L4_IPC_SHORT_MSG,
        &dw0,
        &dw1,
        &dw2,
        L4_IPC_NEVER,
        &dope);

if (L4_IPC_ERROR(dope))
        {
        outstring("\n pager: error reply and wait ");
```

```
                break;
                }
        }
 }
}

//_______________________MAIN
int main(dword_t mb_magic, struct multiboot_info_t* mbi)
{
        l4_threadid_t m_pager=get_current_pager(l4_myself());
        l4_threadid_t s0 = L4_SIGMA0_ID;

        main_tid = l4_myself();
        pager_tid = main_tid;
        pager_tid.id.thread = 1;

        //creating a pager thread
l4_thread_ex_regs
        (pager_tid,
        (dword_t) pager,
        (dword_t)&pager_stack[PAGERSTACKSIZE - 1],
        &m_pager,
        &m_pager,
        &dummy,
        &dummy,
        &dummy);

enter_kdebug("\n main: pager is created ");

        //creating a new task
        subtask_tid.raw = main_tid.raw;
        subtask_tid.id.task = 10;
        subtask_tid.id.thread = 0;

        //system call will return a thread id of new task
subtask_tid = l4_task_new
        (subtask_tid, //thread id of destination task
        255,    //MCP value to maximum possible priority
        (dword_t) &subtask_stack[SUBTASKSTACKSIZE - 1], //SP
        (dword_t) subtask,      //IP
        pager_tid       //thread id of pager
        );

        enter_kdebug("\n main: subtask is created ");

        while(1){l4_sleep(1000000);outchar('-');}
}
```

Listing 6.2: Extended σ_0 protocol

```
/**********************************************************************
 *
 * Copyright (C) 2001-2002,  Karlsruhe University
 *
 * File path:     sigma0/sigma0.c
 * Description:   old sigma0 implementation
 * 04.02.04 urriy@wjpserver.cs.uni-sb.de
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
 * 02111-1307, USA.
 *
 * sigma0.c,v 1.22.2.1 2001/12/13 07:46:23 uhlig Exp
 *
 *********************************************************************/
#include <config.h>

#if defined(CONFIG_L4_NEWSIGMA0)
#include "sigma0-new.c"
#else

#include <l4/l4.h>
#include <l4io.h>

#include "kip.h"

#if defined(CONFIG_ARCH_X86)
#define PAGE_BITS           12
#define SUPERPAGE_BITS      22
#endif
#if defined(CONFIG_ARCH_ARM)
#define PAGE_BITS           12
#define SUPERPAGE_BITS      20
#endif

#define PAGE_SIZE       (1 << PAGE_BITS)
```

```
#define PAGE_MASK       (~(PAGE_SIZE-1))
#define SUPERPAGE_SIZE  (1 << SUPERPAGE_BITS)
#define SUPERPAGE_MASK  (~(SUPERPAGE_SIZE-1))

//#define VERBOSE

#if 1
extern "C" void memset(char* p, char c, int size)
{
    for (;size--;)
        *(p++)=c;
}
#endif


#define iskernelthread(x)       (x.raw < myself.raw)
#define MB64    64L*1024L*1024L
#define MB128   128L*1024L*1024L
#define MB256   256L*1024L*1024L
#define MB512   512L*1024L*1024L

#define MAX_MEM MB256
static unsigned char page_array[MAX_MEM/PAGE_SIZE];



void dump_kip(kernel_info_page_t* kip)
{

#define kipel(x) printf(" kip: %s=\t%x\n", #x, kip->x);

    printf("%s: kernel_info_page magic is %c%c%c%c\n", __FUNCTION__,
           ((char*) kip)[0],
           ((char*) kip)[1],
           ((char*) kip)[2],
           ((char*) kip)[3]);

    kipel(main_mem_low); kipel(main_mem_high);

//enter_kdebug("foo");
    kipel(sigma0_low); kipel(sigma0_high);
    kipel(root_low); kipel(root_high);
//enter_kdebug("foo");
    kipel(reserved_mem0_low); kipel(reserved_mem0_high);
    kipel(reserved_mem1_low); kipel(reserved_mem1_high);
//enter_kdebug("foo");
    kipel(dedicated_mem0_low); kipel(dedicated_mem0_high);
    kipel(dedicated_mem1_low); kipel(dedicated_mem1_high);
//enter_kdebug("foo");
```

```
    printf("\nsigma0: Available main memory: %d MB (%d KB)\n",
           (kip->main_mem_high-kip->main_mem_low)/1024/1024,
           (kip->main_mem_high-kip->main_mem_low)/1024);
//enter_kdebug("foo");
}


extern "C" void sigma0_main(kernel_info_page_t* kip)
{

    l4_threadid_t client;
    int r;
    l4_msgdope_t result;
    dword_t dw0, dw1, dw2;
    dword_t msg;

    l4_threadid_t myself = l4_myself();

#ifdef VERBOSE
    printf("%s: kernel_info_page is at %p\n", __FUNCTION__, kip);
#endif
    if ((((char*) kip)[0] != 'L') |
        (((char*) kip)[1] != '4') |
        (((char*) kip)[2] != 0xE6) |
        (((char*) kip)[3] != 'K'))
        enter_kdebug("sigma0: invalid KIP!");

#ifdef VERBOSE
    dump_kip(kip);
#endif

    dword_t free_mem = kip->main_mem_low;
    dword_t kernel_mem = kip->main_mem_high-PAGE_SIZE;

#define IDX(x) ((x)-(kip->main_mem_low/PAGE_SIZE))

    if (sizeof(page_array) < (kip->main_mem_high-kip->main_mem_low)/PAGE_SIZE)
    {
        printf("sigma0: too much memory - %d < %d\n", sizeof(page_array), (kip->main_mem_high-kip->
            main_mem_low)/PAGE_SIZE);
        enter_kdebug("too much memory");
    };

#define PAGE_SHARED     0xFC
#define PAGE_GONE       0xFD
#define PAGE_FREE       0xFE
#define PAGE_RESERVED   0xFF
#define PAGE_ROOT       0x04
```

```
    /* initialize the page array */
    for (dword_t i = 0; i < sizeof(page_array); i++)
        page_array[i] = PAGE_RESERVED;
    for (dword_t i = kip->main_mem_low/PAGE_SIZE;
         i < kip->main_mem_high/PAGE_SIZE; i++)
        page_array[IDX(i)] = PAGE_FREE;

#define exclude(name,attrib)                                  \
    if ( kip->##name##_high ) {                               \
        if ((kip->##name##_low  >= kip->main_mem_low) &&      \
             (kip->##name##_high <= kip->main_mem_high)) {    \
            for ( dword_t i = kip->##name##_low / PAGE_SIZE;  \
                  i < (kip->##name##_high) / PAGE_SIZE;       \
                  i++ )                                       \
                page_array[IDX(i)] = PAGE_##attrib; }         \
        else                                                  \
            printf("kip->" #name " is wrong\n"); }

    exclude(dedicated_mem0, SHARED);
    exclude(dedicated_mem1, SHARED);
    exclude(dedicated_mem2, SHARED);
    exclude(dedicated_mem3, SHARED);
    exclude(dedicated_mem4, SHARED);

    exclude(reserved_mem0, RESERVED);
    exclude(reserved_mem1, RESERVED);
    exclude(root, ROOT);
    exclude(sigma0, RESERVED);
    exclude(sigma1, RESERVED);
    exclude(kdebug, RESERVED);

#if defined(CONFIG_ARCH_X86)
    page_array[IDX(0xb8000/PAGE_SIZE)] = PAGE_SHARED;
    page_array[IDX(0x01000/PAGE_SIZE)] = PAGE_RESERVED;
#endif

    dword_t one_shot_break = 0;

/*
  macros to figure out request type
  assumption: dw0:dw1:dw2 contains the received message
 */

#define GET_ANY_PAGE (dw0 == 0xFFFFFFFC)
#define GET_KMEM_PAGES ((dw0 == 1) && ((dw1 & 0xFF) == 0))
#define GET_INFO_PAGE ((dw0 == 1) && ((dw1 & 0xFF) == 1))
#define GET_THIS_PAGE_SUPER ((dw0 & 1) && (dw1 == (SUPERPAGE_BITS << 2)))

#if defined(CONFIG_ARCH_X86)
```

```
#define GET_THIS_PAGE (dw0 < 0x40000000)
#endif
#if defined(CONFIG_ARCH_ARM_EP7211)
#define GET_THIS_PAGE (1)
#endif

//urriy
//>--extending - adding some kind of print directly into video memory
//message format stuff
typedef struct header
    {
    l4_fpage_t rcv_fpage;
    l4_msgdope_t size_dope;
    l4_msgdope_t snd_dope;
    }
    header_t;
typedef struct prmsg
    {
    header_t header;
    l4_strdope_t strdope;
    }
    one_str_msg_t;

//message buffers stuff
#define MAX_BUF (2L*1024L*1024L)
char sigma0_buf[MAX_BUF];

//video memory stuff
#define VIDEO_MEMORY_EP (0xb8000)
#define totaloffset (((yoffset * 80) + xoffset)*2)
int xoffset=10, yoffset=0;

char *ptr; dword_t startaddr = VIDEO_MEMORY_EP;
int color=3;

//--<



//urriy
//>--
one_shot_break=1;//debug option
one_str_msg_t printmsg;
printmsg.header.size_dope.md.dwords = 0;
printmsg.header.size_dope.md.strings = 1;
printmsg.header.snd_dope.raw = 0;

printmsg.strdope.rcv_size = MAX_BUF;

printmsg.strdope.rcv_str = (dword_t) &sigma0_buf;
```

```c
printmsg.strdope.snd_size = 0;
printmsg.strdope.snd_str = 0;

void * rcv_dsc = (one_str_msg_t *) (((dword_t) &printmsg) & ~ (3));
//--<

    while(1)
    {
//printf("%s: do ipc_wait\n", __FUNCTION__);
//      r = l4_ipc_wait(&client, NULL, &dw0, &dw1, &dw2, L4_IPC_NEVER, &result);
        r = l4_ipc_wait(&client, rcv_dsc, &dw0, &dw1, &dw2, L4_IPC_NEVER, &result);

        while (1)
        {
            if (one_shot_break)
            {
                enter_kdebug("one_shot_break");
                one_shot_break = 0;
            }

            msg = 2; /* usually we map an fpage */

            if (iskernelthread(client))
            {
                switch (dw0)
                {
                case 0xFFFFFFFC:
                case 0xFFFFFFFE:        /* Jochen */
                    printf("sigma0: s0 receives %x from a kernel thread -> grant any page\n",
                           dw0);
                    /* grant any page */
                    while (kernel_mem > 0 && page_array[IDX(kernel_mem/PAGE_SIZE)] != PAGE_FREE)
                        kernel_mem -= PAGE_SIZE;

                    if (kernel_mem == 0)
                    {
                        //enter_kdebug("sigma0 out of memory");
                        msg = dw0 = dw1 = dw2 = 0;
                    }
                    else
                    {
                        page_array[IDX(kernel_mem/PAGE_SIZE)] = PAGE_GONE;
                        dw0 = kernel_mem;
                        dw1 = dw0 | (PAGE_BITS << 2) | 3;
                        dw2 = 0;

                        printf(" grant: %x:%x:%x-%x\n", dw0, dw1, dw2, msg);
                    }
```

```
                break;
            case 0x00000001:
                printf("sigma0: %x for number of recommended pages\n",
                       client.raw);
                if ((dw1 & 0xFF) == 0)
                {
                    /* reply number of pages recommended for kernel use */
                    msg = 0; dw0 = 0x100;
                }
                else
                    msg = dw0 = dw1 = dw2 = 0;
                break;
            default:
                printf("sigma0: unknown request %x:%x:%x from %x\n",
                       dw0, dw1, dw2, client.raw);
                msg = dw0 = dw1 = dw2 = 0;
                break;
            }
        }
        else
        {
            switch(dw0)
            {
            case 0xCCCCCCCC:
            //urriy
            //>-- from here our print routine goes
                ptr = (char *) startaddr + totaloffset;

                for (int i=0; (i < MAX_BUF) && (sigma0_buf[i/2]!=0);i++)
                    {
                    if ((i%2)==0) *ptr=sigma0_buf[i/2];
                       else *ptr = (char) 8 + color;
                    ptr++;
                    }
            //--<
            break;

            case 0xFFFFFFFC:
#ifdef VERBOSE
                printf("sigma0: s0 receives %x from %p -> map any page",
                       dw0, client);
#endif

                while (free_mem < MAX_MEM && page_array[IDX(free_mem/PAGE_SIZE)] != PAGE_FREE)
                    free_mem += PAGE_SIZE;

                if (free_mem >= MAX_MEM)
                {
                    //enter_kdebug("sigma0 out of memory");
```

```
                        msg = dw0 = dw1 = dw2 = 0;
                    }
                    else
                    {
#if defined(CONFIG_VERSION_X0)
                        page_array[IDX(free_mem/PAGE_SIZE)] = client.id.task;
#elif defined(CONFIG_VERSION_X1)
                        page_array[IDX(free_mem/PAGE_SIZE)] = PAGE_GONE;
#endif
                        dw0 = free_mem;
                        dw1 = dw0 | (PAGE_BITS << 2) | 2;
                        dw2 = 0;

//                      printf(" map: %x\n", dw0);
                    }
                    break;

                case 0x00000001:
                    if ((dw1 & 0xFF) == 1)
                    {
#ifdef VERBOSE
                        printf("sigma0: %x requests kernel-info page\n",
                               client.raw);
#endif
                        dw0 = 0;
                        dw1 = (((dword_t)kip) & PAGE_MASK) | (PAGE_BITS << 2);
                        dw2 = 0;
                    }
                    else
                        msg = dw0 = dw1 = dw2 = 0;
                    break;

#if defined(CONFIG_ARCH_X86)
                case 0x00000000:
                case 0x00000002 ... 0x40000000:
#elif defined(CONFIG_ARCH_ARM_EP7211) || defined(CONFIG_ARCH_ARM_BRUTUS)
                case 0xC0000000 ... 0xC0800000:
#endif
//                  printf("sigma0: s0 receives %x,%x from %x\n", dw0, dw1, client.raw);
                    if ( (dw0 & 1) && (dw1 & 0xFF) == (SUPERPAGE_BITS << 2) )
                    {
                        /* map superpage writeable
                           and uncacheable !!! */

                        dword_t adr = dw0 & SUPERPAGE_MASK;
                        dword_t i;
                        for (i = 0; i < SUPERPAGE_SIZE/PAGE_SIZE; i++)
                            if (page_array[IDX(adr/PAGE_SIZE + i)] != PAGE_FREE)
                            {
```

```
                        msg = 0;
                        break;
                    }

                if (msg)
                {
                    for (i = 0; i < 1024; i++)
#if defined(CONFIG_VERSION_X0)
                        page_array[IDX(adr/PAGE_SIZE + i)] = client.id.task;
#elif defined(CONFIG_VERSION_X1)
                    page_array[IDX(adr/PAGE_SIZE + i)] = PAGE_GONE;
#endif

                    dw0 = adr;
                    dw1 = dw0 | (SUPERPAGE_BITS << 2) | 2;
                    dw2 = 0;
#ifdef VERBOSE
                    printf("sigma0: map 4M page at %x to %x\n", adr, client.raw);
#endif
                }
            }
            else
            {
                if (page_array[IDX(dw0/PAGE_SIZE)] != PAGE_FREE &&
                    page_array[IDX(dw0/PAGE_SIZE)] != PAGE_SHARED
#if defined(CONFIG_VERSION_X0)
                    && page_array[IDX(dw0/PAGE_SIZE)] != client.id.task
#endif
                    )
                {
#if 1
                    printf("sigma0: page %x requested twice, old=%x, new=%x\n", dw0,
                        page_array[IDX(dw0/PAGE_SIZE)], client.id.task);
//                  enter_kdebug("page requested twice");
#endif
                    dw0 = dw1 = dw2 = msg = 0;
                }
                else
                {
                    /* mark only free pages - ignore the shared!!! */
                    if (page_array[IDX(dw0/PAGE_SIZE)] == PAGE_FREE)
#if defined(CONFIG_VERSION_X0)
                        page_array[IDX(dw0/PAGE_SIZE)] = client.id.task;
#elif defined(CONFIG_VERSION_X1)
                    page_array[IDX(dw0/PAGE_SIZE)] = PAGE_GONE;
#endif

                    /* map 4k page, writeable */
                    dw0 &= PAGE_MASK;
```

```
                              dw1 = dw0 | (PAGE_BITS << 2) | 2;
                              dw2 = 0;
                          }
                      }
                      break;

#if defined(CONFIG_ARCH_X86)
                case 0x40000004 ... 0xC0000000:
#elif defined(CONFIG_ARCH_ARM_EP7211)
                case 0xFFFFFFFF: /* dummy */
#endif
                {
#ifdef VERBOSE
                    printf("sigma0: s0 receives %x from %x\n",
                           dw0, client.raw);
#endif
                    /* map 4M page */
                    //enter_kdebug("s0: map superpage");

                    dw0 = (dw0 & SUPERPAGE_MASK) + 0x40000000;
                    dw1 = dw0 | (SUPERPAGE_BITS << 2) | 2;      /* map superpage writeable
                                                  and uncacheable !!! */
                    dw2 = 0;

                }
                break;

                }
            }
//urriy

//          r = l4_ipc_reply_and_wait(client, (void*) msg,
            r = l4_ipc_reply_and_wait(client, (void *) msg,
                                      dw0, dw1, dw2,
                                      &client, rcv_dsc,
                                      &dw0, &dw1, &dw2,
                                      L4_IPC_NEVER, &result);

            if (L4_IPC_ERROR(result))
            {
#ifdef VERBOSE
                printf("%s: error reply_and_wait (%x)\n",
                       __FUNCTION__, result.raw);
#endif
                break;
            }
        }
    }
}
```

```
#endif /* !CONFIG_L4_NEWSIGMA0 */
```

6.6.2 Interrupt Handling

Listing 6.3: Interrupt Handling

```
/**************************************************************
    L4 IPC example
    Interrupt handling.
    Root task creates a thread, which then becomes a handler for
         interrupt using ipc_call system call.
**************************************************************/
#include "urriy.h"

void foo (void);

#define FOOSTACKSIZE 1024

    l4_threadid_t foo_tid; //thread id of target thread
        l4_threadid_t main_tid; //thread id of the root_task
    dword_t foostack[FOOSTACKSIZE]; //stack
    dword_t dummy;

//________________________________FOO
void foo()
{

    outstring("\n foo: successfully started with thread id:");
        outhex32(l4_myself().raw);

        l4_msgdope_t dope;
        dword_t dw0, dw1, dw2;
        dword_t old0, old1, old2;
        int firsttime = 1;

        outstring("\n foo: handling iterrupt ");

        l4_threadid_t intr;

        intr.raw = 16 + 1;//remember -- interrupt number plus one

 while(1)
 {
        l4_ipc_receive(
                intr,
                L4_IPC_SHORT_MSG,
```

```
                &dw0,
                &dw1,
                &dw2,
                L4_IPC_TIMEOUT(0, 0, 0, 1, 0, 0),
                &dope);

                if (dope.raw == 0x20)
                        {//association succeeds, no interrupt is received
                        outstring(".");
                        if (firsttime)
                                {
                                enter_kdebug(" you can now check interrupt associations using key ""
                                    I""");
                                firsttime = 0;
                                }
                        }
                if (dope.raw == 0x10)
                        {//non existing
                        outstring("\n interrupt is already associated! ");
                        break;
                        }
                if (dope.raw == 0x00)
                        {//pending
                        outstring("———————————————————");
                        }

        outstring("\n  msgdope = ");outhex32(dope.raw);
        if ((dw0 != old0)||(dw1 != old1)||(dw2 != old2))
                {
                outstring("\n  dw0 = ");outhex32(dw0);
                outstring("\n  dw1 = ");outhex32(dw1);
                outstring("\n  dw2 = ");outhex32(dw2);
                old0 = dw0;
                old1 = dw1;
                old2 = dw2;
                }

 }

    while(1){l4_sleep(1000000);outchar('+');}
}

//_____________________MAIN -- root task
int main(dword_t mb_magic, struct multiboot_info_t* mbi)
{
    //thread id of the root_task
    main_tid = l4_myself();
        //thread id of the pager of the root task
    l4_threadid_t my_pager = get_current_pager(l4_myself());
```

```
    foo_tid=main_tid;
    foo_tid.id.thread = 8;

// foo thread creation using system call thread_ex_regs
l4_thread_ex_regs(
        foo_tid,        //target thread id
        (dword_t) foo,  //IP
        (dword_t) &foostack[FOOSTACKSIZE - 1], //SP
        &my_pager,      //preempter thread id of target thread
        &my_pager,      //pager thread id of target handler
        &dummy,         //old flags -- don't care
        &dummy,         //old IP -- don't care
        &dummy);        //old SP -- don't care

    outstring("\n root_task: new thread is created with thread id ");
    outhex32(foo_tid.raw);

    while(1){l4_sleep(1000000);outchar('-');}
}
```

7 Conclusions and Future Work

Every complete system would work reliable only in absence of errors in every distinguishable part and in protocols between these parts.

The results obtained in this thesis have showed that L4Ka::Hazelnut microkernel not only has negligible errors in implementation, but also vulnerable to malicious behavior of user-level tasks. Thus, even main principles of it fail. There are some Denial of Service attacks possible (see [16]) via page fault mechanism provided by L4Ka kernel. Scheduling principle also vulnerable to Denial of Service attacks.

Immediate cause of this error-prone behavior of operating system is that whole system built on top can not be surely reliable. This particular version can not be used in embedded systems, and correctness and fault-tolerance can not be proven because of problems even on abstraction level.

For all these reasons, our chair develops new microkernel operating system, which in the near future will be proven. Safety will be provided by correctness and liveness proof of processor VAMP (DLX processor with pipeline), programming language C0 (subset of C), compiler and operating system SOS (Simple Operating System). Last two to be written in C0, guarantee the reliability. All parts of a system will be proven in Isabelle (automated theorem proover).

Bibliography

[1] Jochen Liedtke. Towards real microkernels. Communications of the ACM, 39(9):70-77, September 1996.

[2] COMP9242 Teaching Team. Microkernels and client-server architectures. http://www.cse.unsw.edu.au/ cs9242/03/lectures/lect04.pdf, Aug 2003.

[3] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. Implementation and performance of the Mungi single-address-space operating system. Technical Report UNSW-CSE-TR-9704, University of NSW, University of NSW, Sydney 2052, Australia, Jun 1997.

[4] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The Sawmill multiserver approach. In *Proceedings of the 9th SIGOPS European Workshop*, pages 109114. ACM Press, 2000.

[5] L4Ka Team. L4Ka::Pistachio microkernel. `http://www.l4ka.org/projects/pistachio/`, 2004.

[6] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Marcus Haeberlen, and Marcus Völp. The L4Ka vision, Apr 2001.

[7] Jochen Liedtke. On μ-kernel construction. In Proceedings of the 15th ACM Symposium on OS Principles, pages 237-250, Copper Mountain, CO, USA, December 1995.

[8] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. To appear in The 3rd USENIX symposium on Virtual Machine Research and Technology, Oct 2003.

[9] Vasily Tsyba. Verification of the L4 Task Scheduler, Master Thesis, `http://www-wjp.cs.uni-sb.de/publikationen/VT03.pdf`, 2003.

[10] The L4Ka project. http://L4Ka.org/, 2004.

[11] Jochen Liedtke, L4 Nucleus Version X Reference Manual x86, Universität Karlsruhe, September 3 1999.

[12] Stephan Wagner. An Architecture Independent Kernel Debugger for Hazelnut, Studienarbeit. System Architecture Group, University of Karlsruhe, Sep 2001.

[13] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In Proceedings of the 16th ACM Symposium on OS Principles (SOSP), pages 6677, St. Malo, France, Oct 1997.

[14] DROPS - The Dresden Real-Time Operating System Project. http://os.inf.tu-dresden.de/drops/, 2004.

[15] Alan Au and Gernot Heiser. L4 user manual ver. 1.14, March 1999.

[16] Jonathan S. Shapiro. Vulnerabilities in Synchronous IPC Designs. Department of Computer Science, Johns Hopkins University. Oakland, CA 2003.

[17] Jochen Liedtke. Clans & chiefs. In 12. GI/ITG-Fachtagung Architektur von Rechensystemen, pages 294-305, Kiel, 1992. Springer Verlag.

[18] Jacob Gorm Hansen and Asger Kahl Henriksen. Nomadic operating systems, Master Thesis. Dept. of Computer Science, University of Copenhagen, December 2002.

[19] Home of the L4 community. http://l4hq.org/, 2004.

[20] L4Ka Team. L4Ka::Hazelnut microkernel.
http://L4Ka.org/projects/hazelnut/download.php, 2004.

[21] L4Ka Team. *L4 eXperimental Kernel Reference Manual*. Dept. of Computer Science, University of Karlsruhe, May 2003.

[22] Mauro Gargano. Documentation, Modeling and Verification of Parts of an OS core, Master Thesis, `http://www-wjp.cs.uni-sb.de/publikationen/Ga03.pdf`, 2003.

[23] Lok Sun Nelson Tam. A Comparison of L4 and K42 on PowerPC. School of Computer Science and Engineering, University of New South Wales, December 2003.