MASTER THESIS

# Documentation and Modelling of the IPC Mechanism in the L4 Kernel

Sergey Tverdyshev

Prof. Dr. W. J. Paul

Prof. Dr-Ing. H. Hermanns

Computer Science Department
University of Saarland

September 2003

*I would like to thank:*
*Prof. Wolfgang J. Paul for the invitation to study in Germany, encouragement and many suggestions,*
*Michael Klein for the help on the correction of this thesis,*
*and all research staff of Prof. Paul's chair for the good atmosphere.*

# Contents


**1 Introduction** **5**

**2 Classification of OS by Kernel Architecture** **7**

2.1 The OS without Kernel . . . . . 7

2.2 OS with a Kernel . . . . . 8

2.3 Monolithic Kernels . . . . . 8

2.4 Microkernel Architecture . . . . . 8

2.5 ExoKernel . . . . . 10

2.6 OS Summary . . . . . 10

**3 What is L4** **11**

3.1 L4 Design Philosophy . . . . . 11

3.2 Abstraction in the L4 . . . . . 12

3.2.1 Address Space . . . . . 12

3.2.2 Threads . . . . . 13

3.2.3 IPC . . . . . 13

3.2.4 Clans & Chiefs . . . . . 13

3.2.5 UIDs . . . . . 13

3.3 L4 Summary . . . . . 14

**4 Interprocess Communication** **15**

4.1 Processes . . . . . 15

4.2 Threads . . . . . 15

4.3 IPC . . . . . 16

4.4 Direct and Indirect Communication . . . . . 17

4.4.1 Direct Communication . . . . . 17

4.4.2 Indirect Communication . . . . . 18

4.5 Synchronous and Asynchronous Communication . . . . . 18

4.6 Buffering . . . . . 19

**5 Implementation of IPC in the L4** **21**

5.1 Message Data Types . . . . . 21

5.2 Messages . . . 21
5.2.1 Message Descriptor . . . 21
5.2.2 Short and Long Messages . . . 23
5.2.3 Long Message Header . . . 23
5.2.4 Message `mwords` . . . 24
5.2.5 String Dopes . . . 24
5.3 Timeouts . . . 25
5.4 UIDs . . . 26
5.5 Sending/Receiving IPC Message . . . 27

**6 Source Code Description** **29**
6.1 Data Structure . . . 29
6.2 Send Queue . . . 32
6.3 Thread States . . . 33
6.4 Function `ipc_sys` . . . 34
6.4.1 Input Arguments . . . 34
6.4.2 `ipc_sys` - Sending Part . . . 34
6.4.3 `ipc_sys` - Receiving Part . . . 37
6.5 Function "Transfer Message" . . . 41
6.5.1 Input Arguments . . . 41
6.5.2 Output Parameters . . . 41
6.5.3 `transfer_message` . . . 42
6.6 Function `ipc_copy` . . . 52

**7 Automated Theorem Provers and Proofs** **55**
7.1 What Automated Theorem Provers Are . . . 55
7.2 PVS . . . 55
7.3 Specification of the Message Passing Protocol in the Function `ipc_sys` 56
7.4 State Description . . . 57
7.5 Transition Relation . . . 58
7.6 Properties Specification and Proofs . . . 59

**8 Summary** **61**

**A List of all used L4-function** **63**

**B List of all used predicate over a message descriptor** **65**

# Chapter 1

# Introduction

Since many years software and hardware systems have become more and more complex. At the same time, those systems are used in more and more safety critical places. The problem of reliability must be solved. One way to solve this problem is a life-test of these products, but for example to measure a $10^{-9}$ probability of failure for an 1 hour NASA mission one must test for more than 114,000 years[1]. This is too exorbitant time. Another way is to verify these systems. In this case we make a formal specification of our system. After that we prove our formal specification either with pencil and paper or with some Computer Aided Verification tool. Formal specifications allow defects in requirements and designs to be detected earlier than they would be otherwise and greatly reduce the incidence of mistakes in interpreting and implementing correct requirements and designs.
My diploma thesis is part of the project "Verification of the L4 operating system kernel". The aim of this project is to create a reliable operating system for general and particular use. The term "reliable" implies both security and safety. This project is based on code of the L4 OS kernel[2] . Since the developers of L4 did not give any security or safety guarantees this OS can not be named reliable, therefore the main goal of this project is to prove that this OS has reliable properties. The project consists of several parts, one of these is IPC (interprocess process communication) mechanism. My part is to produce a documentation of the IPC mechanism, to specify and to verify a part of it. The IPC mechanism is very significant in the L4 operating system, since message-passing IPC is one of the core features of L4 and the key to the high performance of L4.

[1] taken from NASA Langley's Research and Technology Transfer Program in Formal Methods (http://shemesh.larc.nasa.gov)

[2] written by the L4-ka group (www.l4ka.de)

# Chapter 2

# Classification of OS by Kernel Architecture

[1] Whenever we talk about structure and efficiency of an **O**perating **S**ystems (OS) we will talk about its "kernel". OSs can be written so that most services are implemented as processes. The OS core then becomes a lot smaller, and we call it a kernel. When this kernel only provides the very basic services, such as basic memory and thread management, is called a microkernel or even nanokernel. We will here use the word kernel in the broad sense meaning the part of the OS supervising the hardware.
Most kernels offer 2 basic encapsulations of programs. The terms used to describe these differ between the OS's, but we will use the following:

- *processes/tasks*: a process has a certain (protected) memory area of its own. It has a status that can be: running, waiting for some event etc.

- *threads*: a thread is part of a process. It encapsulates an execution flow. Each thread has its own set of registers (thus, it's own virtual cpu) , but all threads in a process share the same memory space.

## 2.1 The OS without Kernel

Not all OS's have a "kernel" which is protected from user programs and which manages the hardware and the user programs. Some early operating systems 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. Thus, the user could do with the machine what he wanted to do by accessing the hardware directly. The strong side of this "architecture" is the speed of the system, but it can of course only be used as a personal system for a single user. It's not stable if running several programs at once, if this is, at all, possible.

[1] parts of this chapter are literally taken from [1], [2]

## 2.2 OS with a Kernel

This architecture evolved to an OS design with two process classes: one running in system mode, and another running in user mode. The kernel has full control of the hardware and provides abstractions for the processes running in user mode. A process running in user mode cannot access the hardware but 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.

## 2.3 Monolithic Kernels

The older monolithic kernels were written as a mixture of everything the OS needed without much of an organization. A monolithic kernel offers: processes, memory management, interprocess communication (IPC), device access, file systems, network protocols and whatever the OS should implement. Newer monolithic kernels have a modular design which offers run-time adding and removing of services. The whole kernel runs in "kernel mode". The processes running on top of the kernel run in "user mode". Although modern OSs have a big modularity, they are still monolithic. Examples for such OSs are Unix, Solaris.
The key features of monolithic kernels are:

- *Performance*: monolithic kernels have a relatively high performance since the kernel interacts directly with the hardware.
- *Configurability*: services can't be changed without need to restart the whole system, but the kernel can be optimized for a particular hardware architecture.
- *Robustness*: since many operating system components and system drivers run in kernel mode without memory protection, each component can destroy the system.

  The debugging and testing tools available for kernel-level development environments often lack the robustness of application-level development tools. Consequently, it is very difficult to eliminate software errors in drivers.
- *Portability*: the kernel is not very portable.

## 2.4 Microkernel Architecture

Microkernel designs run as many OS services as possible at user level. This makes the kernel a lot smaller and offers a far greater flexibility. File systems, device drivers, process management and even parts of the memory management can be put in processes running on top of the microkernel. This architecture is actually a client-server model: processes (clients) can call OS services by sending requests through IPC (will be discussed in chapter 4) to server processes e.g., a process that wants to read from a certain file sends a request to the file system process. The central processes that provide the process management, file system etc. are frequently called the *servers*. Microkernels are often also highly multithreaded, putting every different service in a different thread, offering greater speed and stability. The main difficulty of microkernels is to make IPC as fast as possible. This was a design problem in early microkernel design, because IPC, while being intended to

be the power of the architecture, often proved to be the bottleneck. Now, however, microkernels do offer fast IPC.
When talking about microkernels, one must really clearly make a difference between the first and the second generation. The first-generation microkernels, like Mach [3], are fat and provide lots of services, or multiple ways to do the same thing. The second-generation microkernels more follow the "pure" microkernel idea: very small kernels, only offering the abstractions really needed, with a clean and unambiguous Application Program Interface (API). Examples of the second generation are L4 [4] and QNX [5]. Microkernels move many of the OS services into "user space" that on other operating systems are kept in the kernel. This has significant effects in the following areas:

- *Robustness*: if there is a problem with a particular service, it normally can be reconfigured and restarted without having to restart the complete OS. This should be helpful for situations requiring high availability. Moreover, since services would now run in completely independent memory spaces (which is not the case for kernel-level services), bugs and misconfiguration will be less able to corrupt the kernel. The "true kernel" winds up being smaller in scope, and thus ought to be easier to understand and verify. The L4-microkernel occupies about 32K [6] of memory, which severely limits how complex it can realistically be.

- *Security*: services that run within the kernel effectively have kernel class privileges, which is to say that they can do anything, anywhere, at any time. Unix processes that run as root do not have that much control over the system. A problem at present with Linux is that any program that has graphics access must be set uid root because that is the only way of having permission to access the graphics hardware. This may have the unfortunate effect that the program gets "root" access to everything on the system and not just the screen. By running the services as lower level user processes, their access to system resources (e.g. the ability to mess things up) is far more restricted. Moreover, "security" becomes less monolithic, which allows even system services, and indeed security services for that matter, to themselves be forced to comply with security requirements.

- *Configurability*: services can be changed without need to restart the whole system. Work has been ongoing (for instance) to make Linux more dynamically reconfigurable, loadable kernel modules being a good example.

- *Makes coding easier*: kernel code usually requires the use of special memory allocation and output routines since the kernel cannot depend on a lower level to manage these things for it. User-mode code is thus simpler to write than "kernel" code, because it does not need to worry about hardware-specific restrictions.

- *Performance*: communication between components of the extended OS requires that formalized message-passing mechanisms are used. As a result, code must be written to use the formal mechanisms, rather than processes being able to informally use system memory. This may reduce performance. New kinds of deadlocks and other error conditions are possible between system components that would not be possible with a monolithic kernel.

## 2.5 ExoKernel

Exokernels [7] are a further extension of the microkernel approach where the "kernel" is almost devoid of functionality; it merely passes requests for resources to user "space" libraries. The exokernel's architecture implements nothing in kernel space. The exokernel's sole purpose is to securely multiplex hardware resources among user-space processes. Device drivers, virtual memory, even CPU multiplexing and process management are implemented in user space. Supervisor-mode hardware events, like timer ticks, page faults, etc., activate stub handlers in the kernel that simply pass the event to a user-level process that implements the relevant facility's policy. The same system can simultaneously implement forward and inverted page tables, compute-job-friendly or interactive-job-friendly process scheduling, and an application can pick and choose which ever policies will provide it with the best performance.

## 2.6 OS Summary

The different types of OS's allow to cover all needs in different areas. The popular OSs with monolithic kernel provide a high performance for specific hardware, but they have a relative low reliability. In the last time microkernel ideas have a growing tendency and good performance with high level safety and security. An example of a successful microkernel OS is Mach OS [3], but it has a rather big kernel. The L4 and QNX[5] are the last popular ideas in that direction. L4 has a small kernel (about 32k) and has only 7 system calls, that it is enough for building a complete OS with memory management, message passing, security polices etc. As perspective ideas to improve performances of OS's we observed Exo-kernel OS, as logical successors of the mikrokernel idea.

# Chapter 3

# What is L4

L4 by itself is not an OS, but rather constitutes a minimal base on which a variety of complete operating systems can be built. The basic idea of a $\mu$-kernel (microkernel) goes back to Brinch Hansen's Nucleus and Hydra and has been popularized by Mach [3]. The argument goes that by reducing the size of the kernel the part of the OS executing in privileged mode, it becomes possible to build a system which is more secure and reliable (because the trusted computing base is smaller) and easy to extend. A further advantage is that a $\mu$-kernel-based system can easily implement a number of different APIs (also called OS personalities) without having to emulate one within the other.
There was also hope of improved efficiency, as operating systems tend to grow as new features are added, resulting in an increase of the number of layers of software that need to be traversed when asking for service. (An example is the addition of the VFS layer in UNIX for supporting NFS.) A microkernel based system, in contrast, would grow horizontally rather than vertically: Adding new services means adding additional servers, without lengthening the critical path of the most frequently used operations. However, performance of these first-generation microkernels proved disappointing, with applications generally experiencing a significant slowdown compared to a traditional ("Monolithic") operating system. Liedtke [9,10], however, has shown that these performance problems are not inherent in the microkernel concept and can be overcome by good design and implementation. L4 is the constructive proof of this theorem.

## 3.1 L4 Design Philosophy

[1] The most fundamental task of an operating system is to provide secure sharing of resources, in essence this is the only reason why there needs to be an operating system. A $\mu$-kernel is to be as small as possible. Hence, the main design criterion of the $\mu$-kernel is minimality with respect to security: A service (feature) is to be included in the kernel if and only if it is impossible to provide that service outside the kernel without loss of security. The idea is that once we make things small, performance will look after itself. A strict application of this rule has some surprising consequence. For example device drivers: Some device drivers access physical memory and can therefore break security. They need to be trusted. This, however, does not mean that they need to be in the kernel. If they need not execute privileged instructions, and if the kernel can provide sufficient protection to run them at user

[1] parts of this chapter are literally taken from [9]

level, then this is what should be done, and, consequently, this is what L4 does. Another important principle is that it should be possible to implement arbitrary system on top of the microkernel. Together with the minimality principle this leads to a requirements for a small number of powerful and orthogonal abstractions, and for a strictly policy-free kernel.

## 3.2 Abstraction in the L4

Based on such philosophy Liedtke [11] concludes that the $\mu$-kernel needs to provide:

- address spaces - because they are the basis of protection.
- threads - because there needs to be an abstraction of program execution.
- inter-process communication (IPC) - as there needs to be a way to transfer data between address spaces.
- unique identifiers (UIDs) - for context-free addressing in IPC operations.

### 3.2.1 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 mappings[2] from virtual to physical memory (which is partial in the sense that many mappings are undefined, making the corresponding virtual memory inaccessible). Address spaces in L4 can be recursively constructed: A thread can map parts of its address space into another thread's address space (provided the receiver cooperates) and thereby share the data mapped by that 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 can not 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.) Note that while a grant is irreversible, the granter has, in general, received the address 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 is 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 L4, a task is the set of threads sharing an address space. Creating a new task creates 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. When we say that a task is created we mean that an inactive task is activated. 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. Inactive tasks can be donated to other tasks.
There is a hierarchy of tasks, with parents having some limited control over their

[2] For more explanations see [8],[20]

children. The main purpose of this is to be able to control (IPC-based) information flow between address spaces. It has nothing to do with process hierarchies a particular L4-based personality OS may implement. Such a hierarchy is under full control of the particular OS personality.

### 3.2.2 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 instruction pointer and a stack pointer), a page fault handler (pager), and an exception handler. 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 L4 and OS personalities built on top.

### 3.2.3 IPC

Message-passing IPC is the heart of L4. The $\mu$-kernel provides a total of seven system calls (IPC being one of them), which provide some very rudimentary OS functionality. 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 $\mu$-kernel copies the data between two address spaces) or by reference (using mapping or granting). Data by value can be transferred in two ways: in-line, when a limitted amount of such data is passed directly in registers (3 words) with any remainder in a message buffer. And out-of-line, an arbitrary out-of-line buffers are copied to the receiver.
IPC is also used for synchronisation (as it is blocking, so each successful IPC operation results in a rendez-vouz), wakeup-calls (as timeouts can be specified), pager invocation (the $\mu$-kernel converts a page fault into an IPC to a user-level pager), exception handling (the $\mu$-kernel converts an exception fault into an IPC to a user-level exception handler), and interrupt handling (the $\mu$-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).

### 3.2.4 Clans & Chiefs

Clans and chiefs are L4‘s basic mechanism enabling the implementation of arbitrary security policies. They allow controlling IPC and thus information flow. The basic idea is simple: A task‘s creator is that task‘s chief, all tasks (directly) created by a particular chief constitute that chief‘s clan. Threads can directly send IPC only to other threads in the same clan, or to their chief. If a message is sent to a thread outside the clan containing the sender, that message is instead delivered to the sender‘s chief (who may or may not forward the message). If a message is sent to a member of a subclan of the clan containing the sender, that message is delivered to the task in the clan whose clan (directly or indirectly) contains the addressee.

### 3.2.5 UIDs

A $\mu$ -kernel supplies unique identifiers (UIDs) for something, either for threads or tasks or communication cannels. UIDs are required for relible and efficient local

communication. If $S_1$ wants to send a message to $S_2$ it needs to specify the destonation $S_2$ (or some chanel leading to $S_2$). Therefore, the $\mu$-kernel must know which UID relates $S_2$. On the other hand, the receiver $S_2$ wants to be sure that the message comes from $S_1$. Therefore the identifier must be unique, both in space and time.
A UID of a thread is that of its task plus the number of the thread within the task. 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 (there are 256 tasks and within each task 64 threads). This means that tasks (or address spaces) and threads must be recycled. The $\mu$-kernel ensures uniqueness of task IDs by incrementing the version number whenever a task number is reused. As far as threads are concerned, the L4 view is that threads do not die as long as their task exists, they can only be blocked (waiting for IPC which will never arrive). This avoids the issue of lthread uniqueness. 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 L4's. How this is done is up to the OS personality, L4 only provides the tools.

## 3.3 L4 Summary

L4 is a successful implementation of mikrokernel ideas. It has a small reliable base and provide only seven system calls. The small reliable base provides ground to build a high safety system, if this base is well implemented. The seven system calls are enough to built a complete OS with memory management, security polices etc. The high abstract level is well suited to construct a highly portable kernel.

# Chapter 4

# Interprocess Communication

## 4.1 Processes

[1] Informally a processes is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity represented by the value of the program counter and the contents of the processor's register. A process generally also includes the process stack which contains temporary data (such as method parameters, return addresses, and local variables), and a data section which contains global variables. We emphasize that a program by itself is not a process; a program is a passive entity, such as the contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. Although two processes may be associated with the same program, they are nevertheless considered two separat execution sequences.
As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- NEW - the process is being created
- RUNNING - instructions are being executed
- WAITING - the process is waiting for some event to occur(such as an I/O completion or reception of a signal).
- READY - the process is waiting to be assigned to a processor.
- TERMINATED - the process has finished execution

## 4.2 Threads

The process model discussed so far has implied that a program can only perform a single thread of execution. For example, if a process is running a word processor program, there is a single thread of instructions being executed. This single thread of control only allows the process to perform one task at one time. The user could not simultaneously type in characters and run the spell checker within the same

[1] parts of this chapter are literally taken from [12]

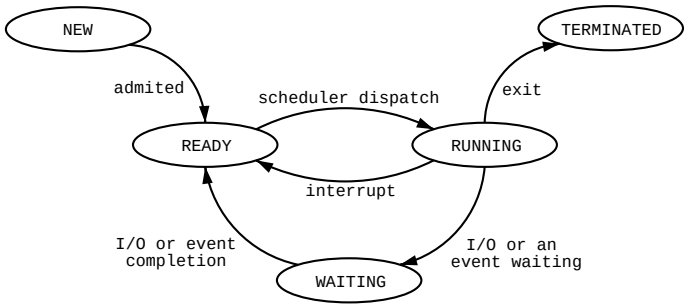


Figure 4.1: Final state machine of processes state

process. Many modern operating system have extended the process concept to allow a process to have multiple threads of execution. They allow the process to perform more than one task at a time.

### Cooperating Processes

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it can not affect or be affected by other processes executing in the system. Clearly, any process that does not share any data (temporary or persistent) with any other process is independent. On the other hand, a process is cooperating if it can affect or be affected by other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. There are several reasons for providing an environment that allows process cooperation:

- Information sharing - since several users may be interested in the same piece of information, we must provide an environment to allow concurrent access to these types of resources.
- Computation speedup - if we want a particular task to run faster, we can break it into subtask, each of which will be executing in parallel with the others. Notice that a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- Modularity - we may want to construct a system in a modular fashion, dividing the system functions into separate processes or threads, and we must provide a mechanism to communicate between them.
- Convenience - even an individual user may have many tasks on which to work at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution that requires cooperation among the processes requires a mechanism to allow processes (threads) to communicate with one another and synchronize their actions. This task is performed by the IPC mechanism.

## 4.3 IPC

In this section and further we will not distinguish between processes and threads, since IPC is used for communication between both processes and threads. IPC

provides a mechanism allowing processes to communicate and to synchronize their actions. Interprocess-communication can be implemented by a message system. Message systems can be defined in many different ways. Note that the shared-memory system and message-system communication schemes are not mutually exclusive and could be used simultaneously within a single operating system or even a single process. The function of a message system is to allow processes to communicate with each other without the need to resort to shared variables. An IPC facility provides at least the two operations: *send* and *receive* message. Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the physical implementations is straightforward. This restriction, however, makes the task of programming more difficult. On the other hand, variable-sized messages require a more complex physical implementation, but the programming task becomes simpler.
If process $P$ and $Q$ want to communicate, they must send messages to and receive from each other, so a communication link must exist between them. The link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus or network), but rather with the issue of its logical implementation, such as its logical properties. There are several methods for logically implementing a link and the send/receive operations:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages

## 4.4 Direct and Indirect Communication

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

### 4.4.1 Direct Communication

In the direct-communication principle, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send and receive primitives are defined as follows:

**send(P, message)** - send a message to process $P$

**receive(Q, message)** - receive a message from process $Q$.

A communication link in this scheme has the following properties:

- A link is establish automatically between every pair of processes that want to communicate.
- The processes need to know only each other's identity to communicate.

- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits a symmetry in addressing; that is, both the sender and the receiver processes have to determine the other to communicate. One calls this scheme *not open wait*. That means the receiver must know the sender.
A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to determine a sender. This scheme one calls *open wait*. In this scheme, the send and receive primitives are defined as follows:

**send(P, message)** send a message to process P.

**receive(id, message)** receive a message from any process (special id); the variable id is set to the name of the process with which communication has taken place.

The disadvantages in both of these schemes (symmetric and asymmetric) are the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found so that they can be modified to the new name.

### 4.4.2 Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes (WinNT), or ports (Mach), or pipe (Unix). A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which they can be removed. Each mailbox has a unique identification. In this case, a process can communicate with some other process via a number of different mailboxes. The send and receive primitives are defined as follows:

**send(A, message)** send a message to mailbox A.

**receive(A, message)** receive message from mailbox A.

Direct communications allows very fast communication, since we do not have unnecessary copy like the copies to and from a mailbox in indirect communication. L4 is based on direct communication (asymmetric and symmetric). The problem given above were successfully solved by creating a name-assignment mechanism.

## 4.5 Synchronous and Asynchronous Communication

Communication between processes takes place by system calls of the send and receive primitives. There are different design options for implementation each primitive. Message passing may be either blocking or non-blocking - also known as synchronous and asynchronous.

- Synchronous send - the sending process is blocked until the message is received by the receiving process or by the mailbox.

| OS | Key features |
|---|---|
| UNIX | Source and receiver have a socket as "door". All messages are buffered. Data passing is asynchronous, buffered, indirect communication. |
| WinNT | Two methods 1) sharing kernel objects (without IPC) 2) passing message. There are two functions for synchronous and asynchronous interaction. |
| L4 | Unbuffered, synchronous. |

Table 4.1: IPC in different OS's

- Asynchronous send - the sending process sends the message and resumes operation.
- Synchronous receive - the receiver blocks until a message is available.
- Asynchronous receive - the receiver retrieves either a valid message or a null.

Different combinations of send and receive are possible. When both the send and receive are blocking, we have a "rendez vous" between the sender and receiver.

## 4.6 Buffering

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, there are three ways in which such a queue can be implemented:

**Zero capacity** - the queue has a maximum length 0; the link can not have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

**Bounded capacity** - the queue has finite length $n$; thus, at most $n$ messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.

**Unbounded capacity** - the queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as automatic buffering, since we have to copy to a buffer at first. The different implementations of the IPC mechanism in popular OS's are depicted in Table 4.1

# Chapter 5

# Implementation of IPC in the L4

[1] Message passing is the basic IPC mechanism in L4. It allows L4 threads to communicate via messages. All L4 IPC is synchronous and unbuffered. Synchronous IPC requires an agreement between both the sender and the receiver. The main implications of this agreement is that the receiver is expecting an IPC and provides the necessary buffers. If either the sender or the receiver is not ready, then the other party must wait. Unbuffered IPC reduces the amount of copying involved and thus is the key to high performance IPC.

## 5.1 Message Data Types

Data can be transferred in three ways using L4 IPC:

1. In-line by-value data (data aligned by words): a limited amount of data (the first 3 words) is passed directly via registers.
2. Strings: Arbitrary large out-of-line buffers (buffers not necessary aligned by words) which are copied to the receiver.
3. Virtual memory mappings (by-reference data): Data transfer via mappings is described by flex-pages (fpages)[21]. Alternatively, virtual memory can be granted: mapped to the receiver and unmapped from the sender simultaneously.

Figure 5.1 illustrates the difference between in-line by-value data and strings.

## 5.2 Messages

### 5.2.1 Message Descriptor

A message descriptor is either a pointer to the start of a message buffer or an indicator that the IPC is purely register based. There are two types of message

[1] parts of this chapter are taken from [9],[13] and are optimized to the given implementation of L4.

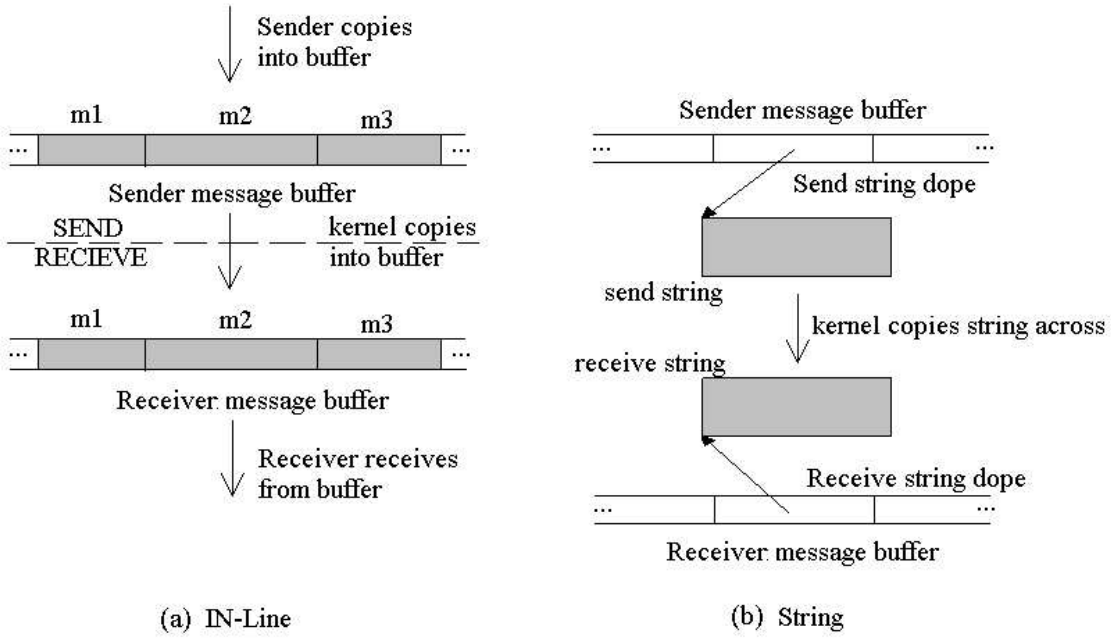


Figure 5.1: Data passing in L4

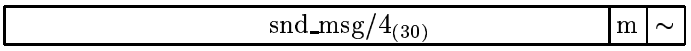


Figure 5.2: The sender message descriptor

descriptors: one for sending and one for receiving IPC. The format of the send message descriptor is depicted in the figure 5.2.

A zero value indicates a purely register based IPC. Such a message we will call *short message*. A non-zero message descriptor address (snd_msg) is interpreted as the start address of the sender's message buffer. Such a message we will call *long message*

Setting the m-bit indicates that the IPC includes mappings, i.e. we can extract the needed fpage descriptor from sender. A zero value for the m-bit indicates that the message contains only by-value data and no fpages.

The format of the receive descriptor is similar and is shown in figure 5.3:

If the receiver message descriptor is not equal to 0 and the m-bit is not set, then the message descriptor address (rcv_msg) is interpreted as the start address of the receiver's message buffer, which may contain a receive fpage, indicating the caller's willingness to accept mappings . Setting m-bit inicates that the receiver is willing to accept an fpage but no long message. In this case the receive fpage is supplied 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 wait) while a zero value for the o-bit allows receiving only from the specified sender.

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.

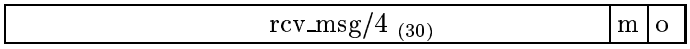


Figure 5.3: The receiver message descriptor

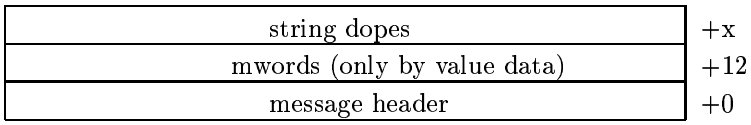


Figure 5.4: Format of the long message

### 5.2.2 Short and Long Messages

Every successful IPC operation will always copy at least three dwords[2] to the receiver. These three dwords contain the first 12 bytes of a message's in-line data and are referred to as the short message. The long message (see section 5.2.1) is optional and its presence is indicated by the m-bit in the message descriptor (snd_msg/recv_msg). If the long message is present, then it is a dword-aligned memory buffer pointed to by the 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 dwords, excluding those copied in registers) and string dopes is specified in the message header (see section 5.2.3). Figure 5.4 depicts the format of the long message. This structure is used to describe both the sending and the recieving messages.
Note that to simplify user-level programming the first three dwords of a message are always transfered via registers. This permits to handle long and short messages basically in the same way. Also loading/storing those registers from/to the message/buffer data structure is not handled by the kernel, i.e. can be done at user-level.

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

### 5.2.3 Long Message Header

The message header describes the format of the long message (figure 5.5). The size dope specifies the message which the recever is willing to receive, i.e. on the sending side size dope is ignored. It defines the number of the mword (field *#mword*) in dwords, and hence the offset of the string dopes from the end of the header (the $x$ in figure 5.4). The size dope also defines the number of string dopes (field *#strings*) in the long message.

The send dope specifies how many dwords and strings are actually to be sent, i.e. this field is ignored in the reciever's side and is filled in by the sender. (Specifying send dope values less than the size dope values makes sense when the caller is willing to receive more data than it is sending.) The receiver's fpage describes the address range in which the caller is willing to accept fpage mappings or grantings in the

[2] dwords in the current implementation are bit strings of length 32

| | | | | |
|---|---|---|---|---|
| *size dope* | #mwords (19) | #strings (5) | ~ (8) | +8 |
| *send dope* | #mwords (19) | #strings (5) | ~ (8) | +4 |
| | receiver's fpage (32) | | | +0 |

Figure 5.5: Message header

| | |
|---|---|
| *receive string (32) | |
| c | receive string size(31) |
| *sender string (32) | |
| c | sender string size(31) |

Figure 5.6: String dope

receive part (if present) of the IPC. Notice that sender's fpage and hot spot [8] are stored in processor registers, and the user should store these data there. It is also important to note, that field *#mwords* of send and size dope is at least 3, because the first 3 mwords are always transfered via short message (see section 5.2.2).
This structure allows us to perform combined *send-recieve* operation without unnecessary redefining of the buffer's fileds.

### 5.2.4 Message `mwords`

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. Notice that a `mword` is just a bit string of length 32.

### 5.2.5 String Dopes

The last component making up a long message are string dopes. There can be zero or more string dopes, with the exact number specified in the 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 kernel copies data from sender memory, specified by the sender string dopes, to the receiver's memory, specified by the receiver's string dopes. Each string dope occupies four dwords and its format is depicted in figure 5.6.

The first part of the string dope specifies the size and location of the string the caller wants to send to the destination, while the second part specifies the size and location of a buffer where the caller is willing to receive the string. It is important to note that strings do not have to be aligned, and that their size is specified in bytes. By setting bit $c$ we can make logical string (we will also call it compound string). $c$=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 that the sequence of strings that are transfered are considered as one contiuous string. This process is called gathering. On the receiver side, logical string means that a sequence of strings is to be considered as one logical buffer. The corresponding received string is scattered among them. Continuations can be arbitrary combined on sender and receiver side.
Figure 5.7 shows a complete long message. This buffer contains all message data of a long message. The first three words are message header (section 5.5). After the

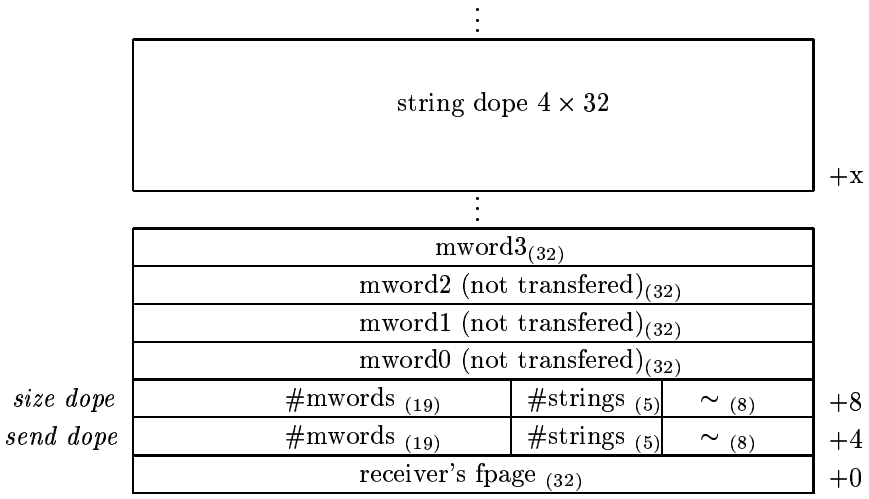


Figure 5.7: A complete long message

message header follows mwords, which forms in-line data of a message. Note that the first three mwords are not transfered and we do not use them. These dwords are reserved for further functional extension. After in-line data follows string dopes, which forms out-of-line data. We can calculate the offset $x$ (see figure 5.7) and it is equal 12+(amount of in-line data)*4, where the first number is the size of the message header. Note that all computations are in bytes. It is also important to note that we use either *send dope* or *size dope* at the time. Thus whenever we send we use only *send dope*, similary whenever we receive we use only *size dope*. In case we want to perform a send-receive operation, we have to construct two buffers one for send and another for receive operation. For more details see section 6.4.

In the next section we cover two important mechanisms: timeouts and UIDs. We will need them to understand the IPC mechanism.

## 5.3 Timeouts

Timeouts are used to control IPC operations. Each IPC operation specifies four timeout values [8]. The first two timeouts are with respect to the time before the message transfer starts. These timeouts are no longer relevant once the message transfer was started.

**receive timeout** - Specifies how long to wait for an incoming message. The receive operation fails if the timeout period is exceeded before the message transfer starts.

**send timeout** - Specifies how long the IPC should try to send a message. The send operation fails if the timeout period is exceeded before the message transfer starts.

*Send* and *receive pagefault timeouts* are used if a page fault occurs during an IPC operation. A page fault is converted to an IPC by the kernel.

**send pagefault timeout** - In the case of a pagefault in the receivers's address space, the corresponding IPC to the pager uses this timeout for both send and receive timeout. The timeout is specified by the sender.

| $m_r(8)$ | $m_s(8)$ | $p_s(4)$ | $p_r(4)$ | $e_s(4)$ | $e_r(4)$ |
|---|---|---|---|---|---|

Figure 5.8: Format of the timeouts word

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

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

$$\text{sender pagefault timeout} = \begin{cases} \infty, & \text{if } p_s = 0 \\ 4^{15-p_s} * m_s \ \mu s, & \text{if } 0 < p_s < 15 \\ 0, & \text{if } p_s = 15 \end{cases}$$

$$\text{receive pagefault timeout} = \begin{cases} \infty, & \text{if } p_r = 0 \\ 4^{15-p_r} * m_s \ \mu s, & \text{if } 0 < p_r < 15 \\ 0, & \text{if } p_r = 15 \end{cases}$$

Figure 5.9: Timeouts calculations

**receive pagefault timeout** - In the case of a pagefault in the sender's address space, the corresponding IPC to the pager uses this timeout for both send and receive timeout. The timeout is specified by the receiver.

Besides the special timeouts: 0 - do not wait at all and $\infty$ - wait for ever, periods from $1\mu s$ up to approximately 19 hours can be specified. The complete quadruple is packed into a 32-bit word (figure 5.8). Figure 5.9 depicts the timeouts calculations.

## 5.4 UIDs

Unique IDs (UIDs) identify tasks, threads and hardware interrupts. Each unique id is a 32-bits value which is unique in time. Figure 5.10 shows the format of a thread id. The UID of a thread is that of its task plus the number of the thread within the task (local thread number - lthread). It also contains the id of the chief of its clan [14]. The last component making up the UID is the version number. The $\mu$-kernel ensures uniquess of task IDs by incrementing the version number whenever a task number is reused.

The format of the task id is similar and is depicted in figure 5.11. The one distiction is that we do not use the lthread field.

There are two special IDs: NIL id is $0^{32}$ and invalid id is $1^{32}$. We can see that available task ids are in the range from 1 till 254, so we have got 254 tasks. The number of available threads per task is 64 (from 0 till 63). Likewise we have got 1024 version numbers (0-1023). Altogether we have $1.67117 \times 10^7 + 2$ UIDs.

| chief(8) | task(8) | lthread(6) | ver(10) |
|---|---|---|---|

Figure 5.10: Format of the thread id

| chief(8) | task(8) | 0(6) | ver(10) |
|---|---|---|---|

Figure 5.11: Format of the task id

## 5.5 Sending/Receiving IPC Message

Before considering how to send or receive it is necessary to decide what form of data is to be sent. In particular, a decision must be made on whether to send data in-line 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 needs to be copied to a buffer first and must also be aligned. Thus the in-line option is best for small amounts of data. Sending short strings as in-line data is more efficient as it avoids setting up string dopes and may also be done in registers. 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) same number and size of strings that the sender sends. The C interface to L4's system calls provides a number of IPC operations. They are differentiated in the following ways:

- Single send or receive versus a combined send and receive.
- Receive from specific sender (not open wait) or any sender (open wait).

To send or receive a message, certain parameters must be provided:

- dest/src - Identifier of message destination/source thread respectively.
- snd_msg/rcv_msg - Descriptor for long part of message, for send/receive part of IPC respectively.
- Timeout - Timeout specification. Used to ensure that a thread need not be blocked indefinitely.

In section 6.4.2 we describe the possible IPC operations and input/output arguments in more details.

# Chapter 6

# Source Code Description

The IPC mechanism code consists of:

- the interaction protocol between sending and receiving sides;
- transfer of a message;
- processing of special cases.

The interaction protocol (message passing protocol) is represented as the function `sys_ipc`. This function describes three possible cases of an interaction: *send, receive* and *combined send receive*. The transfer of a message is described in the function `transfer_message`. This function performs the transfer of a message from the sender to the receiver with the proper copying of in-line and out-of-line data (see chapter 5). If a thread receives a messsage from the kernel, this is a special case and we perform a special receiving. Both functions are in file *ipc.c* of L4 implementation.

## 6.1 Data Structure

The IPC mechanism works over the following C data types:

**dword_t** (file types.h)
: This type is the same as `unsigned int`. In our case it is a double word of length 32 bits.

**ptr_t** (file types.h)
: This type is the pointer to `dword_t`.

**fpage_t** (file ipc.h)
: This type is an implementation of a flexpage and describes the format of the flexpage.

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

We make a union with `dword_t` to make it possible to perform mask operations with the descriptor, i.e., we can work on the descriptor as a whole.

```
typedef union{
        dword_t raw;
        fpage_struct_t fpage;
} fpage_t;
```

**msg_dope_t** (file ipc.h)
This type is an implementation of a message dope. The type is used to represent *size dope* and *send dope* in the message header (see section 5.2). The fields `dwords` and `strings` are used to describe *size dope* and *send dope*. This type also describes the received message.

```
typedef struct{
        unsigned msg_deceited:1;
        unsigned fpage_received:1;
        unsigned msg_redirected:1;
        unsigned src_inside:1;
        unsigned snd_error:1;
        unsigned error_code:3;
        unsigned strings:5;
        unsigned dwords:19;
}msg_dope_struct_t;
```

The fields `dwords` and `strings` after an IPC operation contain the number of dwords/strings actually copied. The fields `error_code, snd_error, snd_inside, msg_redirected, fpage_received, msg_decited` describe the status of the IPC operation. Note that after the IPC operation this dope word is available in the register (for x86 architecture it is register EAX). The kernel does not store it in the receiver's message buffer. The user program may store it or use directly from the register.

```
typedef union{
        dword_t raw;
        msg_dope_struct_t msgdope;
} msgdope_t;
```

**memmsg_t** (file ipc.h)
This type descibes the message header that was explained in section 5.2

```
typedef struct{
        fpage_t rcv_fpage;
        msgdope_t size_dope;
        msg_dope send_dope;
        dword_t dwords[0];
}memmsg_t;
```

The first three fields are the message header. The last element is the pointer to the first in-line word to send. It is declarated in that way to simplify pointer arithmetic, i.e. dwords[3] will point to the third word.

**stringdope_t** (file ipc.h)
This type is an implementation of the string dopes discussed in section 5.2.5.

```
typedef struct{
        dword_t send_size:31;
        dword_t send_continue:1;
        ptr_t send_address;
        dword_t rcv_size:31;
        dword_t rcv_continue:1;
        ptr_t rcv_address;
}stringdope_t;
```

The fields `send_continue` and `rcv_continue` allow us to make composed string dopes. If the field `send_continue` (`rcv_continue`) is true, then the next string dope is the continuation of the current string dope. If that field is false, then the string dope is not composed or that was the first dope in the composed string dope.

**timeout_t** (file thread.h)
This type is an implementation of the timeout data structure described in section 5.3.

```
typedef struct{
        unsigned rcv_exp:4;
        unsigned snd_exp:4;
        unsigned rcv_pfault:4;
        unsigned snd_pfault:4;
        unsigned rcv_man:8;
        unsigned snd_man:8;
}timeout_struct_t;
typedef union{
        dword_t raw;
        timeout_struct_t timeout;
} timeout_t;
```

In our notation from section 5.3 `rcv_exp` is $e_r$, `snd_exp` is $e_s$, `rcv_pfault` is $p_r$, `snd_pfault` is $p_s$, `rcv_man` is $m_r$ and `snd_man` is $m_s$.

**l4_threadid_t** (file thread.h)
This type is an implementation of the UIDs (see section 5.4). Note that task id and thread id are composed in one union structure. So, this type describes both id's.

```
typedef union{
        struct{
                unsigned version:  L4_X0_VERSION_BITS;// 10 bits
                unsigned thread:  L4_X0_THREADID_BITS+L4_X0_TASKID_BITS; //
6+8 bits
                unsigned chief:  L4_TASKID_BITS;// 8 bits
        }id;
        struct{
                unsigned version:  L4_X0_VERSION_BITS;// 10 bits
                unsigned thread:  L4_X0_THREADID_BITS;// 6 bits
                unsigned task:  L4_X0_TASKID_BITS; // 8 bits
                unsigned chief:  L4_TASKID_BITS;// 8 bits
        }x0id;
        dword_t raw;
}l4_threadid_t
```

The names of some well known id's are:
L4_KERNEL_ID : 0,1,0,0 i.e. version filed is equal to 0, thread field to 1, task

filed to 0, chief field to 0.
L4_SIGMA0_ID : 1,0,2,4.

**tcb_t** (file thread.h)
This type describes a thread control block (tcb) which implements a thread (see section 4). It contains the current thread activities: UID, queues, thread states etc. In the next structure we present only the fields which we use.

```
typedef struct{
        dword_t ipc_buffer[3];
```
*This buffer allows us to access the first three dwords (or short message) of in-line data. If the sender has an fpage to send, the first two words are used to describe this fpage. ipc_buffer[0] is used to describe* ***send base*** *while the ipc_buffer[1] is used to describe* ***send fpage****.*

```
        timeout_t ipc_timeout;
        dword_t msg_desc;
```
*Message descriptor see section 5.2.*

```
        l4_threadid_t myself;
        dword_t thread_state;
```
*The description of the thread states is presented in chapter 6.3.*

```
        tcb_t* send_queue;
        tcb_t* send_prev;
        tcb_t* send_next;
```
*The description of the send queue is presented in chapter 6.2.*

```
        l4_threadid_t partner;
                    ⋮
}tcb_t;
```

## 6.2 Send Queue

The *send queue* is a list with threads which want to send to the thared. It is implemented as a cyclic double linked list. It works on the three fields from the thread structure:

**tcb_t *send_prev** describes the previous thread to send to

**tcb_t *send_queue** describes the current thread to send to

**tcb_t *send_next** describes the next thread to send to

Note that the first part in a bold text means the data type while the second means the name of a variable e.g. **tcb_t *send_prev** means that we have the pointer `*send_prev` of type `tcb_t`.

To explain how this queue works we break down the work of the queue into two cases:

1. The *send queue* is empty. Then, after adding a thread into the queue, it will look like figure 6.1: the pointers `send_prev` and `send_next` point to the same added thread, i.e. `send_queue`

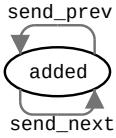


Figure 6.1: Addition of a thread into an empty queue

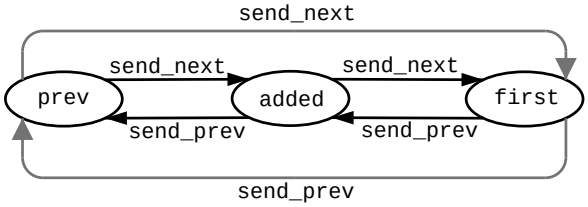


Figure 6.2: Addition of a thread into a not empty queue

2. There are threads in the *send queue*. Then, after adding a thread, it will look like figure 6.2. We added the thread in the queue as follows: let *first* be the thread in the *send queue*, i.e., `send_queue`. Then the pointer `send_prev` of the added thread will point to the previous thread of the *first*, `send_next` of the added thread will point to the *first*. The pointer `send_prev` of the *first* will point to the added thread, and `send_next` of the `send_prev` of the added thread will point to added thread. So we have an ordinary insert operation into a cyclic double-linked list.
   Note that since IPC in L4 is synchronous a thread can be only in one send queue at the time.

## 6.3 Thread States

As we said in section 4.2, a thread changes its state during execution. In this section we list and describe all possible states. The numbers in the parentheses means the value of the corresponding first bits in the thread state variable.
Note that the developers use here the inverse logic, because it is easer to test. For example `110` describes `TS_RUNNING`, in the ordinary logic it will be `001`.

**TS_RUNNING (110)** The thread is ready to be scheduled, thus the thread is ready to be executed at the user-level.

**TS_POLLING (101)** This state is possible only in the sending part of the IPC, i.e, the sender is waiting for the receiver.

**TS_WAITING ($1^{32}$)** The thread is waiting for an event e.g. for a message. If the receiver is in this state, it means that the sender was not ready to send.

**TS_LOCKED_WAITING (011)** This state is possible only in the receiving part of IPC. It means that the receiver detected that the sender is ready to send a message and the receiver is ready to receive it.

**TS_LOCKED_RUNNING (010)** In this case both the sender and the receiver are ready to interact. When the scheduler selects the sender thread, it immediately sends message to the receiver and ends its IPC part.

**TS_ABORTED (111)** A thread is unable to be executed or does not exist.

## 6.4 Function ipc_sys

As mentioned in chapter 5, all L4 IPC are synchronous and unbuffered. Synchronous IPC requires an agreement between both the sender and the receiver. The main implication of this agreement is that the receiver is expecting an IPC and provides the necessary buffers. If either the sender or the receiver is not ready, then the other part must wait. That function (protocol) also guarantees that after a successful (or unsuccessful) message transfer both threads are ready to be executed again, i.e., there are no dead-locks.

### 6.4.1 Input Arguments

The function `sys_ipc` has the following input arguments:

**l4_threadid_t dest** the id of the destination thread for sender or the id of the source thread for receiver.

**dword_t snd_desc** the sender message descriptor.

**dword_t rcv_desc** the receiver message descriptor.

### 6.4.2 ipc_sys - Sending Part

We describe the function as a flow-chart (figure 6.3). The current implementation of the function supports three modes: *send, receive* and *send-receive*. We can distinguesh modes with the help of predicates `IS_SEND?` and `IS_RECEIVE?`. They are defined as follows:

```
 IS_SEND? = snd_desc != 1^32
IS_RECEIVE? = rcv_desc != 1^32
```

In case both predicates are true we have *send-receive* mode. This mode allows to speed up IPC since there are cases when we need to receive an information as reply to the previous sending.

Step by step we describe all blocks from the figure 6.3. The function starts from the declaration of auxiliary arguments.
At first we get the pointer to the current thread structure with the help of L4-function `get_current_tcb()`. This function returns the pointer to the executing thread. We also declare the pointer to the partner structure and assign it the value NULL:

```
tcb_t* current = get_current_tcb();
tcb_t* to_tcb = NULL;
```

Then we check whether IPC has the sending part of the operation. If IPC has it, then we get the pointer to the receiver thread from its message descriptor. If it does not, then we go to process the receiving part of the IPC (section 6.4.3).

```
if (IS_SEND?)
```

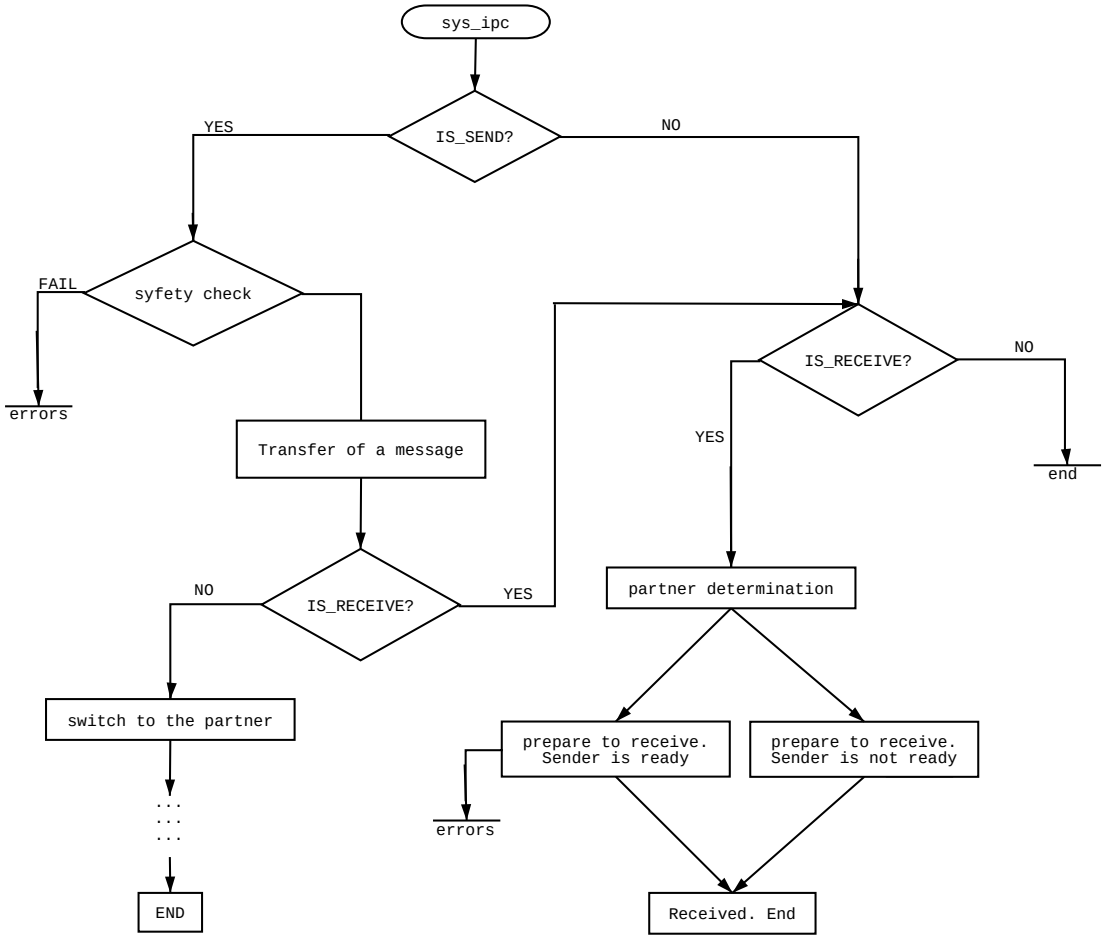


Figure 6.3: Function ipc_sys

## "Safety check"

At first we get pointer to the partner structure. We do it with the help of L4-function `tid_to_tcb()`. This function takes as argument a thread id and returns pointer to the corresponding thread structure.

```
to_tcb = tid_to_tcb(dest)
```

Safety check means that if the id of the thread destination is valid. So that function `tid_to_tcb(dest)` returns an existing and valid thread:

```
if ( !l4_is_invalid_id(dest) && tcb_exist(to_tcb) && (to_tcb → myself ==
dest) )
```

If the process did not pass this test, i.e. the destination is invalid, we cancel the function. To cancel the function we use l4-function `return_ipc`. This function ends an IPC and saves the status of the operation in the processor's register.

```
return_ipc(IPC_ERR_EXIST);
```

If the safety check was successful then we set up the field partner in the current

thread to dest:

```
current → partner = dest;
```

## "Transfer message"

At first we check whether the receiver is ready to receive:

```
if( !IS_WAITING(to_tcb) || ((to_tcb → partner != current → myself) &&
(!l4_is_nil_id(to_tcb → partner))) )
```

If the receiver is not ready, or ready to receive but not from this sender, the sender needs to wait for the receiver. At first we check whether we have a non infinty send timeout. If we do, then we calculate the absolute timeout. Otherwise we have infinite timeout and we will wait for the receiver anyway. Predicate IS_INFINITE_SEND_TIMEOUT and function TIMEOUT which are used below work as described in section **??**

```
if ( !IS_INFINITE_SEND_TIMEOUT(current→ipc_timeout))

compute absolute timeout
qword_t absolute_timeout = TIMEOUT(current→ipc_timeout.timeout.snd_exp,
current→ipc_timeout.timeout.snd_man);

if absolute timeout is equal to 0, we have to leave the function. Since
if we are here the receiver is not ready and the sender does not have time to
wait for it. This allows to speed up the quitting of "bad" IPC with the error
message.
if ( !absolute_timeout ) return_ipc(IPC_ERR_SENDTIMEOUT);

compute the time for a wakeup event
current→absolute_timeout = absolute_timeout + get_current_time();

we add the sender thread to the wakeup queue with the help of L4-function
thread_enqueue_wakeup:
thread_enqueue_wakeup(current);
```

Now the sender can tell the receiver that it is ready to send. So, we add the sender thread to the send queue of the receiver and set the sender's state to the TS_POLLING:

```
thread_enqueue_send(to_tcb, current);
current→thread_state = TS_POLLING;
```

After the preparations we switch to the idle thread, i.e. let the scheduler execute the idle theread or the next ready thread if any exists. This is done by L4-function ipc_switch_to_idle.

```
ipc_switch_to_idle(current);
```

After this command the sender thread is not active; we will describe the behaviour of the sender after its reactivation. At first we check the state of the sender. If it equals to TS_RUNNING, then the sender thread was reactivated because of a timeout and we must end the function with an error message:

```
if ( current→thread_state == TS_RUNNING )
return_ipc(IPC_ERR_TIMEOUT);
```

If the state does not equal to `TS_RUNNING`, then the receiver is ready to interact and we dequeue the sender thread from the *wakeup queue*:

```
thread_dequeue_wakeup(current);
```

Notice that *wakeup queue* is the same data structure as *send queue*. It is used to control IPC operation.

Now the receiver is ready to receive the message, so we call the function `transfer_message`. If the partner does open wait, it needs to know who we are. If the test at the start of "Transfer message" block is true, then we execute the following code without performing the actions described above.

```
transfer_message(current, to_tcb, snd_desc);
to_tcb→partner = current→myself;
```

### "Switch to the partner" and "End"

After the successful or unsuccessful transfer we need to end the interaction or perform preparations for the receiving part (if there is one). To end the operation we set the sender thread state to `TS_RUNNING`. We also switch to the receiver since it needs to also end its part of the interaction.

```
if(!IS_RECEIVE){
current→thread_state = TS_RUNNING;
ipc_switch_to_thread(current, to_tcb);
return_ipc_args(dest, 0, current);}
```

Note that after an execution of the function `ipc_switch_to_thread` the current thread is not active. When the scheduler activates that thread next time, we will perform the function `return_ipc_args`, i.e. after a call of the function `ipc_switch_to_thread` that thread quasi ends the IPC.
The function `return_ipc_args` just copies necessary arguments to the processor registers, e.g. after the interaction the thread field `msg_desc` contains the error status. Thus the status will be copied to the register *EAX*.

We can desribe the chnage of the sender's and the receiver's states in the sending part of function `ipc_sys` as a transition system. Figure 6.4 depicts such a system. Each state of the system is a pair: sender state and receiver state. Star in a system state means the undefined state and *MPP* is abbreviation for *Message Passing Protocol*.

### 6.4.3 `ipc_sys` - Receiving Part

This section describes the behaviour of the receiving side of the IPC. At first we check whether the current thread has the receiving part:

```
if(IS_RECEIVE?)
```

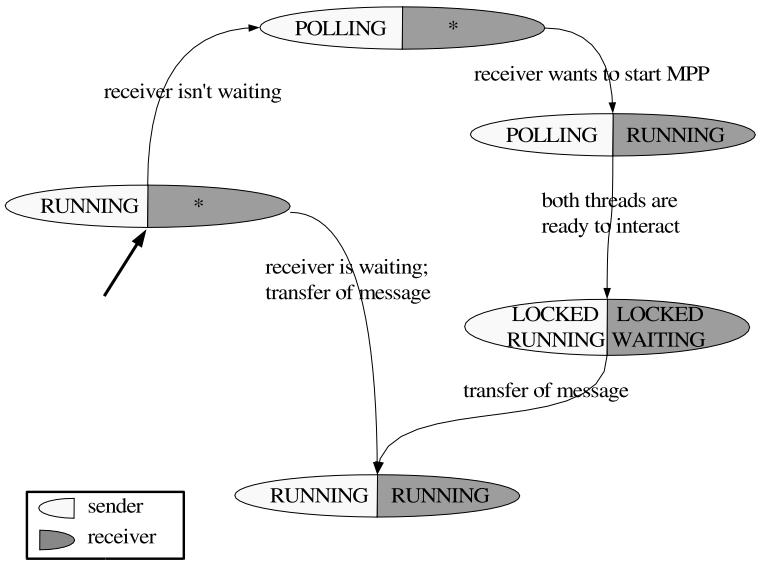


Figure 6.4: The change of the sender's and receiver's states in the sending part of function `ipc_sys`

If we have receiving part, then we start from declaration of the pointer to the partner structure:

```
tcb_t* from_tcb;
```

## "Determination of the partner"

As we considered in the chapter 4.4.1 the receiving side can work in two modes:

**open wait** - receiving of a message from any thread

**not open wait** - receiving a message from the strictly defined thread.

Now we check whether the receiver does *not open wait*, with the help predicate `IS_OPEN_WAIT`. This predicate tests the first bit of receiver message descriptor as described in section 5.2.1:

```
IF(!IS_OPEN_WAIT)
```

if the receiving thread does, then we have to get the partner's structure. We do it by function `tid_to_tcb`:

```
from_tcb = tid_to_tcb(dest);
```

Now we have to check the result of the L4-function `tid_to_tcb`. It can be wrong if we only have a receiving part, otherwise it was tested in the sending part ("Safety check"). So we test if the returned thread structure `from_tcb` does not exist or in the returned thread structure the field `myself` does not equal to the given thread id `dest`, then we have an error. We end the IPC and save status of the operation in processor's register by L4-function `return_ipc`.

```
if( !tcb_exist(from_tcb) || (from_tcb→myself.raw != dest.raw) ){
return_ipc(IPC_ERR_EXIST);}
```

If the receiving thread does **not** *not open wait*, i.e., it does *open wait*, then the receiver takes id of the partner from its *send_queue*.

```
from_tcb = current→send_queue;
```

The result can be either the valid id of the partner or an invalid id, namely 0. So, we have to test the result.
As input argument we have the descriptor of the message which has to be received. We assign that argument to the receiver's descriptor:

```
current→msg_desc = rcv_desc;
```

Now we have to test whether the sender is ready to send or not. The sender is ready if the receiver has got a valid id:

```
if ( (!from_tcb) || !IS_POLLING(from_tcb) || (from_tcb→partner != current→myself))
```

## Preparation to receive. Sender is not ready

At first we perform a timeout calculation procedure. This is the same as for the sending part. So, if we have a receiving timeout, then we calculate it and enqueue the thread into wakeup queue:

```
if ( !IS_INFINITE_RECV_TIMEOUT(current→ipc_timeout) ){
       qword_t absolute_timeout =
          TIMEOUT(current→ipc_timeout.timeout.rcv_exp,
          current→ipc_timeout.timeout.rcv_man);
       if ( !absolute_timeout ) return_ipc(IPC_ERR_RECVTIMEOUT);
        current→absolute_timeout = absolute_timeout + get_current_time();
        thread_enqueue_wakeup(current);}
```

After the execution of the timeout calculation we need to make a trick:
`from_tcb` can be only zero in the case if the receiver has an *open wait*. Otherwise the thread does not exist (and we do not reach this path) or we have a valid thread id. Thus, by checking for an open wait we are safe:

```
current→partner = IS_OPEN_WAIT ? L4_NIL_ID :from_tcb→myself;
```

Now we are ready to wait for a message:

```
current→thread_state = TS_WAITING;
```

If we also sent to the partner, then we switch to him:

```
if ( to_tcb )
       ipc_switch_to_thread(current, to_tcb);
else
       ipc_switch_to_idle(current);
```

So, assume that the receiving thread has been reactivated. If it was because of timeout, then an error occurs:

```
if ( current→thread_state == TS_RUNNING )
return_ipc(IPC_ERR_TIMEOUT);
```

otherwise we have got the message and dequeue the receiver's thread from the wakeup queue:

```
thread_dequeue_wakeup(current);
```

and end the operation. The next working block is *Received. End* (see figure 6.3)

## Preparation to receive. Sender is ready

The partner is ready to send a message. We change its state to `TS_LOCKED_RUNNING`, that tells the sender to send the message. We enqueue the sender into the ready queue, i.e., it can be selected by the scheduler:

```
from_tcb→thread_state = TS_LOCKED_RUNNING;
thread_enqueue_ready(from_tcb);
```

The sender is waiting for us. As it is in our *send_queue*, we dequeue it:

```
thread_dequeue_send(current, from_tcb);
```

Then we switch the receiver's state to "receiving of a message" and switch to its partner:

```
current→thread_state = TS_LOCKED_WAITING;
ipc_switch_to_thread(current, from_tcb);
```

## Received. End

We have got the message. We switch the receiver's state to `TS_RUNNING` and add the receiver thread to the ready queue and end the IPC:

```
current→thread_state = TS_RUNNING;
thread_enqueue_ready(current);
return_ipc_args(current→partner, 0, current);
```

Similar to the sending part of function `ipc_sys` we can describe the change of the sender's and the receiver's state in the receiving part as a transition system (figure 6.5).

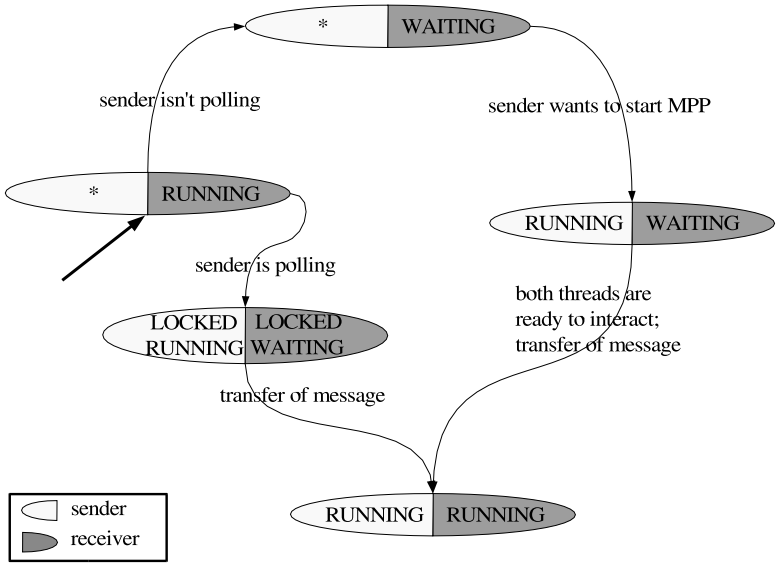


Figure 6.5: The change of the sender's and receiver's states in the receiving part of function `ipc_sys`

## 6.5 Function "Transfer Message"

The function described in this chapter is based on the data structure described in the section 6.1. The aim of the function is a correct transfer of a message. As we said in the section 5.2, a message may consist of three data types: in-line, out-of-line and fpage. This function transfers the rest of in-line data to the receiver. Remember that the first three words are not transfered because they are in processor registers. This function should also provide correct scattering/gathering and prepare the data to map an fpage[1].

### 6.5.1 Input Arguments

The function `transfer_message` takes three arguments as inputs. For the description of the data structure `tcb_t` and message descriptor see section 6.1.

**tcb_t* from** pointer to the sender's structure

**tcb_t* to** pointer to the receiver's structure

**dword_t snd_desc** sender's message descriptor

### 6.5.2 Output Parameters

Although this function has type `void` it returns a status of transfer procedure. This status is represented as a message dope and is in the processor's register.

[1] description of an fpage see in [8,20]

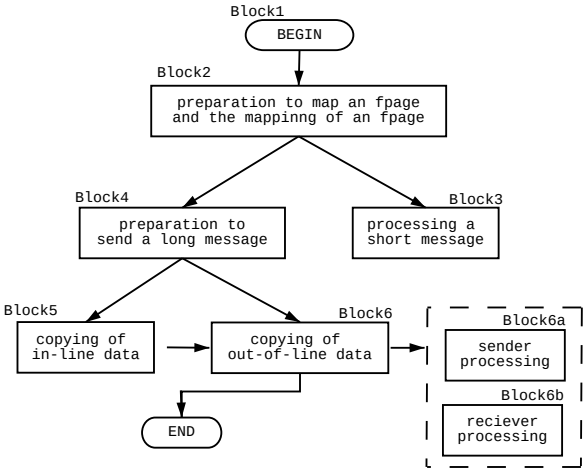


Figure 6.6: Transfer of a message

### 6.5.3 `transfer_message`

We will consider this function in form of a flow-chart. Figure 6.6 shows the overview of the function. To simplify the description task we have broken the function into several blocks, and we will describe each of them.

## BEGIN (Block1)

At first we perform an IPC optimization, i.e. the transfer of a short message. We copy the pointers to the processor's registers to the receiver.

```
to→ipc_buffer[0] = from→ipc_buffer[0];
to→ipc_buffer[1] = from→ipc_buffer[1];
to→ipc_buffer[2] = from→ipc_buffer[2];
```

Since the current version does not supply autopropogation[2], we assume that the control bit for autopropogation is zero. We test `snd_desc` to zero. Remember that the zero value of `snd_desc` means pure registered message. So, if `snd_desc` is equal to zero then the function is over. We set the message dope of the receiver to zero to say that a short message was successfully transfered. Otherwise we call the sub-function `extended_transfer`, that performs a message transfer. The input arguments are still the same.

```
if( snd_desc &  0x1 )
        extended_transfer(from, to, snd_desc);
else
        to→msg_desc = 0;
```

We declare and initialize the key parameters. The parameters of type `memmsg*` point to dword aligned memory message buffers. `rcv_fpage` describes the fpage to receive.

```
memmsg_t* snd_msg;
memmsg_t* rcv_msg = NULL;
fpage_t rcv_fpage;
```

Since the receiver thread can wait for the sender in the wakeup queue, we should

[2] Autopropogation is the part of the Clan&Chief mechanism. For more comments see [14]

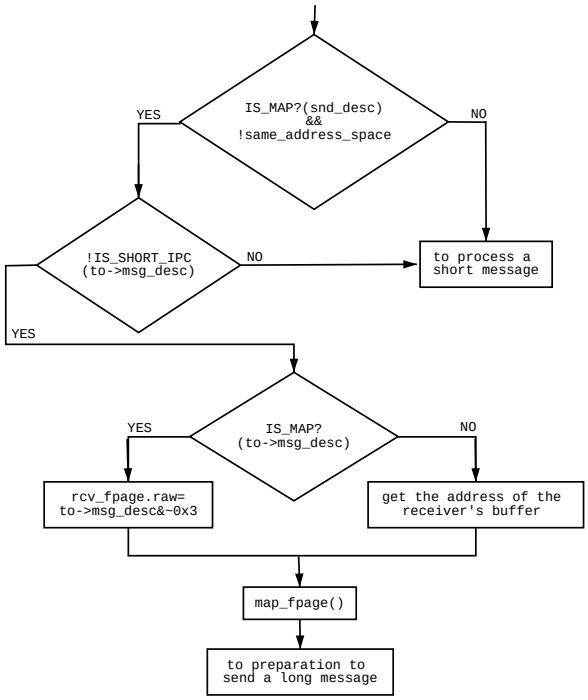


Figure 6.7: Transfer of a message (Block 2)

dequeue it. This is done by L4-function `thread_dequeue_wakeup`:

```
thread_dequeue_wakeup(to);
```

## Block2

Figure 6.7 shows the internal structure of the Block2. That part of the code makes preparations to call the function `map_fpage` and then executes it. These preparations consist of:

- determination whether the sender wants to send and the receiver is willing to receive an fpage

- correct determination of the receiver's fpage.

Note that if we have an fpage to send, then *snd_base* and *send_fpage* must be in the registers ,i.e. in ipc_buffer[0] and ipc_buffer[1] respectively. At first we check whether the sender has an fpage to send and whether the sender and the receiver are in the same address space. If the sender and the receiver are in the same address space, then the mapping is useless, because the receiver can access this page directly.

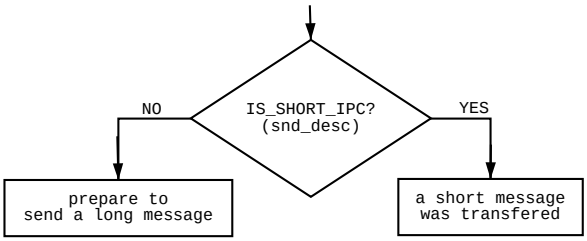


Figure 6.8: Transfer of a message (Block 3)

```
if( IS_MAP(snd_desc) && !same_address_space(from,to) )
```

*The receiver has to accept a long message (i.e. no zero message descriptor) for mapping, else we set up the flag error ,i.e., the sender sends an fpage, but the receiver does not accept it*

```
        if ( !IS_SHORT_IPC(to→msg_desc) ) msgdope.raw |= IPC_ERR_CUTMSG;
```

*The receiver is willing to receive an fpage, but not a long message, so receive fpage is contained in the receiver's message descriptor (we mask the last two bits, since this is an additional information about fpage*

```
        if ( IS_MAP(to→msg_desc) ) rcv_fpage.raw = to→msg_desc&  0x3;
```

*If the receiver is ready to accept a long message then we take receiver's fpage from the memory message buffer. At first we get the pointer to receiver's buffer and then rcv_fpage.*

```
        rcv_msg = (memmsg_t *) get_copy_area(from, to, (ptr_t) (to→msg_desc
&  0x3));
         rcv_fpage = rcv_msg→rcv_fpage;
```

Now we have got all arguments to call the function `map_fpage`:

```
map_fpage(from, to,
         from→ipc_buffer[0], /* snd base */
         (fpage_t) raw:from→ipc_buffer[1],/* snd fpage */
         rcv_fpage);/* receiver's fpage */
```

## Block 3

This block is just a simple check whether the sender is sending a short message (Figure 6.8). If it is so, we are done, otherwise we start to process a long message.

```
if ( IS_SHORT_IPC(snd_desc) ){
         to→msg_desc = msgdope.raw;
         return;}
```

## Block 4

In this block we start to process a long message. Figure 6.9 describes this block. At first we get the pointer to the sender message buffer. Then we get the number of dwords and the number of strings to copy (Figure 5.5).

```
snd_msg = (memmsg_t *) (snd_desc &  0x3);
dword_t num_dwords_src = snd_msg→send_dope.msgdope.dwords;
dword_t num_strings_src = snd_msg→send_dope.msgdope.strings;
```

If the number of dwords is less than IPC_GAP_DWORDS (in current version 3, see

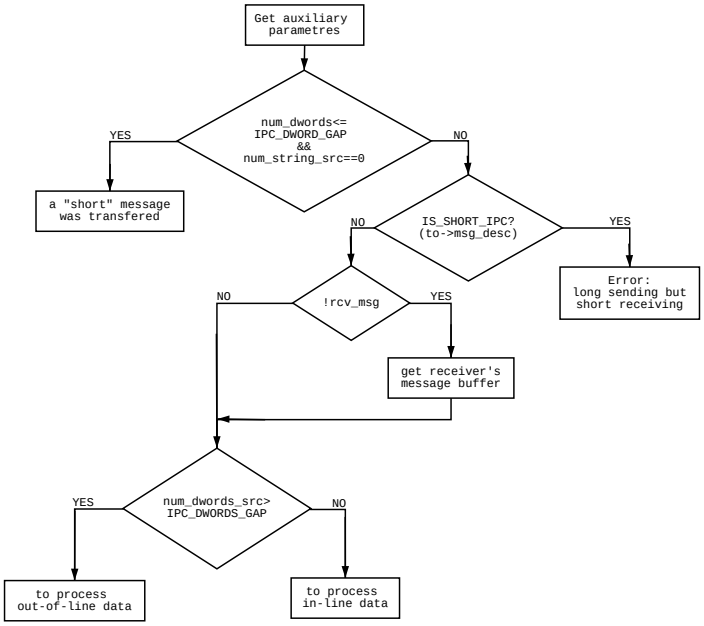


Figure 6.9: Transfer of a message (Block 4)

figure 5.5 for comments) and we do not have strings to copy, then such long message is leading to short. So we are done.

```
if( (num_dwords_src <= IPC_DWORD_GAP) && (num_strings_src == 0) ){
        to→msg_desc = msgdope.raw;
        return;}
```

If the sender sends a long message, but the receiver receives a short message, then we set the error code, which means the message was cut.

```
 if (IS_SHORT_IPC(to→msg_desc) ){
        to→msg_desc = msgdope.raw | IPC_ERR_CUTMSG;
        return;}
```

If we have not yet obtained the pointer to the receiver's message buffer, i.e. we did not have mapping, then we take the pointer:

```
if( !rcv_msg )
rcv_msg = (memmsg_t *) get_copy_area(from, to, (ptr_t) (to→msg_desc &
0x3));
```

Now we are ready to process the data to send. If we have the rest of the in-line data, then we go to the block 5. If we do not have then we go to the sending of a long message (block 6):

```
if (num_dwords_src > IPC_DWORD_GAP)
```

## Block 5

This block is depicted in figure 6.10. The aim of this block is to check whether the receiver wants to receive at least the number of dwords that the sender sends or

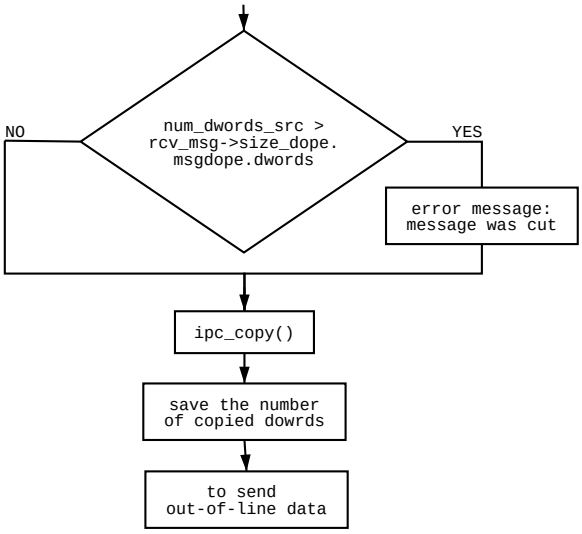


Figure 6.10: Transfer of a message (Block 5)

not. If it does not want, then we add the code "cut message error code" to the status.

```
if (num_dwords_src > rcv_msg→size_dope.msgdope.dwords){
      num_dwords_src = rcv_msg→size_dope.msgdope.dwords;
      msgdope.raw |= IPC_ERR_CUTMSG;}
```

Although the message was probably cut, we copy the maximal possible number of dwords to the receiver. Maximal possible number means the smallest dwords number between the sender and the receiver. We call the function for coping the necessary number of dwords. Description of the function `ipc_copy` see in section 6.6. Note that this function takes as arguments pointers to the start of sender's and receiver's data and the number of bytes to be copied, i.e. this function copies the given number of bytes from/to the given address.

```
ipc_copy(&rcv_msg→dwords[IPC_DWORD_GAP],
      &snd_msg→dwords[IPC_DWORD_GAP],
       (num_dwords_src - IPC_DWORD_GAP) * 4);
```

After copying we set the amount of copied dwords to the received dope.

```
msgdope.msgdope.dwords = num_dwords_src;
```

## Block 6

In this block we copy out-of-line data. Since this block is complex we break down the description task into three cases: We start with the declaration of key variables. We also describe the process of copying the necessary number of bytes[3] from the current sender's string dope to the receiver's one. Then we observe two cases:

- all bytes from the sender's current string dope were copied
- all bytes from the receivers current string dope were copied

[3]Remember that the size of a string is defined in bytes. We call the function `ipc_copy` to copy the given number of bytes from the sender's dope to the receiver's dope.

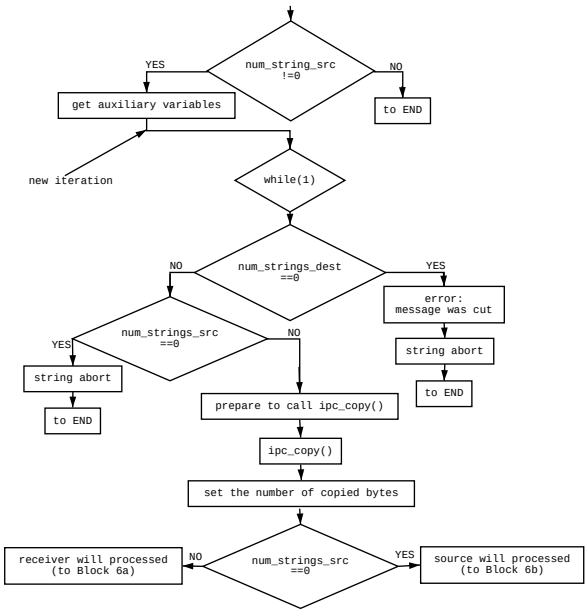


Figure 6.11: Transfer of a message (Block 6)

Figure 6.11 shows the first case. At first we check whether we have any strings to copy. If it is not the case then we are done, otherwise copying is started:

```
if(num_strings_src)
```

Then we get the number of strings which the receiver is willing to accept. We also need the address (for the sender and the receiver) of the first string dope, which is shifted on amount in-line data.

```
dword_t num_strings_dest = rcv_msg→size_dope.msgdope.strings;
stringdope_t* dope_src = (stringdope_t *)
        &snd_msg→dwords[snd_msg→ size_dope.msgdope.dwords + 1];
stringdope_t* dope_dest = (stringdope_t *)
* &rcv_msg→dwords[rcv_msg→ size_dope.msgdope.dwords + 1];
```

Now we need to get address of the first byte to send. We take it from the first sender's string dope in the field `send_address`. Similarly we need to get the address of the first destination byte. We take it from the first receiver's string dope in the field `rcv_address`. We also need the number of bytes to send/receive for the first string dope.

```
char* p_src = (char* ) dope_src→send_address;
char* p_dest = (char *) dope_dest→rcv_address;
dword_t size_src = dope_src→send_size;
dword_t size_dest = dope_dest→rcv_size;
```

We define two auxiliary variable's to keep trace of the operation: the number of all copied bytes and the number of bytes for one copy operation.

```
dword_t size_total = 0;
dword_t copy_size;
```

Now we are ready to start copying. This procedure is implemented as an infinite loop. We will break this loop when we have all data processed or when an error occurs.

```
for(;;){
We do not have any place to accept more strings
        if ( num_strings_dest == 0 ){
            msgdope.raw |= IPC_ERR_CUTMSG;
            break;}
We have already copied all strings (previous iteration)
        if( num_strings_src == 0 ) break;
```

We determinate the smaller size between the number of bytes to send and the number of bytes to receive in order to perform a correct copy operation. Then we copy the determined number of bytes. We also update the "size" variables: we decrease the variable `snd_size` and the variable `rcv_size` by the number of copied bytes. We also increase the variable `total_size` by the number of copied bytes.

```
copy_size = size_src < size_dest ?  size_src :  size_dest;
ipc_copy(get_copy_area(from, to, (ptr_t) p_dest),
          (dword_t *) p_src, copy_size);
size_src -= copy_size;
size_dest -= copy_size;
size_total += copy_size;
```

## Block 6a

In this part we consider the case where the sender has copied all bytes from the current sting dope. Figure 6.12 shows that case. So, we have copied all bytes from the source:

```
 if( size_src == 0 )
Decrease numbers of string dopes to send by 1.
num_strings_src--;
```

If we have already copied all string dopes, then we end the procedure. The end means that we assign to the field size in the last receiver's dope the size of the copied bytes i.e. prgtxtsize_total. In the IPC status variable we increase the number of successfully copied strings by one. Then we go to the block "END".

```
if( num_strings_src == 0 ){
       dope_dest→send_size = size_total;
       dope_dest→send_address = dope_dest→ rcv_address;
       msgdope.msgdope.strings++;
       goto string_abort;}
```

Note that for the sending, the receiving part of a string dope is ignored. On the receiving side, after the message was received, the field address in the sending part of the string dope is assigned to the value of the receiving part. Now the field size in the sending part of the receiver's string dope is equal to the length of the received message. Thus, received parts can be forwarded after a successful send operation without any change.
If we have more string dopes to process we set the send pointer to the next string

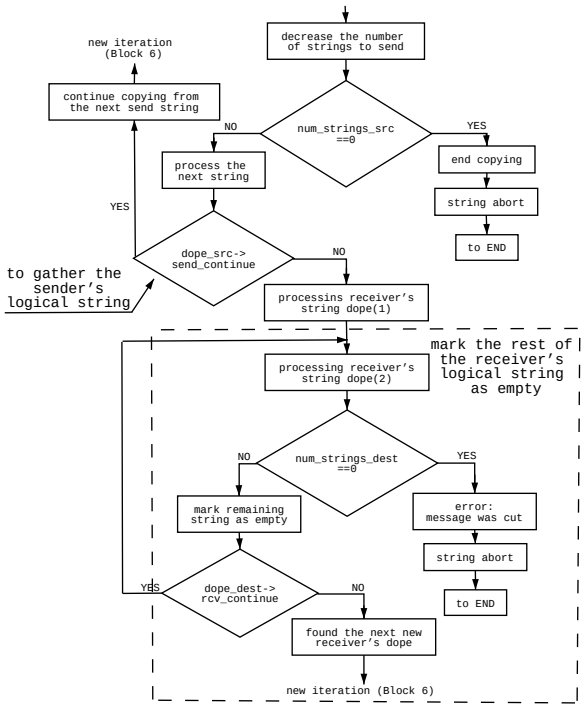


Figure 6.12: Transfer of a message (Block 6a)

dope:

```
dope_src++;
```

As we discussed in the section 5.2.5 each string dope has the bit *continues* to compose logical strings. So, we check this bit and if it is set up we continue copying from next string to send.

```
if( dope_src→send_continue ){
p_src = (char*) dope_src→send_address;
size_src = dope_src→send_size;
p_dest += copy_size;
continue;}
```

Note that we ignore the case when `dest_size` is equal to 0, since we can determinate this in the next iteration.
If the current string dope is not composed, then it means that the sender has more string dopes to send. At first in order to "close" the copying of the current sender's string, we set the field sender string in the receiver's string dope to the receiver string (to enable sending without any changes).

```
dope_dest→send_address = dope_dest→ rcv_address;
```

The rest of this block describes a process of scattering/gathering which we discussed in the section 5.2.5. At this point we know that the source is not a compound string, but may be the destination string. In the next loop we skip to the next receive dope which is not a part of the current scatter/gather string. Figure 6.13 shows scatter and gather processes.

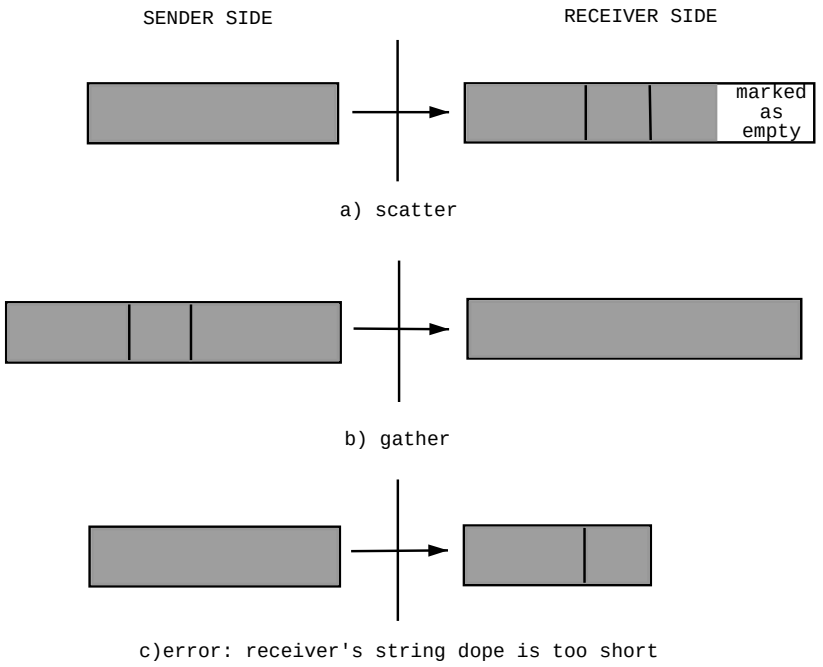


Figure 6.13: Scatter and Gather processes

```
do{
We set in the receiving dope the number of successfully copied bytes.
      dope_dest→send_size = size_total;
We set the receiving dope pointer to the next string dope.
      dope_dest++;
W decrease the number of string dopes to receive by 1.
      num_strings_dest--;
We increase (status of the operation) the number of received string dopes by 1.
      msgdope.msgdope.strings++;
We have more data to send, but the receiver does not have any more place.
      if ( num_strings_dest == 0 ){
         msgdope.raw |= IPC_ERR_CUTMSG;
         goto string_abort;}
This will make the remaining scattering/gathering receive strings marked as
empty, since the source is not a compound string.
       size_total = 0;
This will work until the end of compound string, i.e. the next string is found.
}while( dope_dest→rcv_continue );
```

After termination of the last loop we found the next new receive dope. We repeat the same procedure as in the case when the sender's string is continuous (logical):

```
if( dope_src→send_continue ){
p_src = (char*) dope_src→send_address;
size_src = dope_src→send_size;
p_dest += copy_size;
```

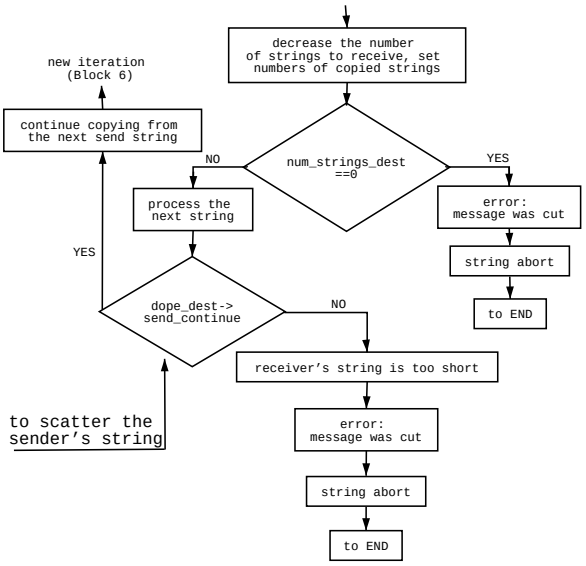


Figure 6.14: Transfer of a message (Block 6b)

## Block 6b

This block describes a case when during copying the receiver does not have any space left in the current dope. Figure 6.14 shows this case. At first we close the session for the current string dope, i.e.: we set up in the receiver dope the number of successfully copied bytes, decrease the number of string dopes to receive by 1 etc.

```
dope_dest→send_size = size_total;
dope_dest→send_address = dope_dest→rcv_address;
num_strings_dest--;
msgdope.msgdope.strings++;
```

At this point we know that the sender has more data to send, so we must check if the receiver has more string dopes to receive. If it has not, we set an error message:

```
if (num_strings_dest == 0 ){
msgdope.raw |= IPC_ERR_CUTMSG;
goto string_abort;}
```

The next check is the same as in the sender part i.e. if the receiver dope is a compound. We continue copying into the next receiver's string (namely substring):

```
if( dope_dest→rcv_continue ){
p_dest = (char*) dope_dest→rcv_address;
size_dest = dope_dest→rcv_size;
p_src += copy_size;
size_total = 0;
continue;}
```

If the receiver current dope is not *continues*, then the receiver does not have place in the current logical string, so the message will be cut. We set an error message and abort the operation:

```
else{ msgdope.raw |= IPC_ERR_CUTMSG;
goto string_abort;}
```

Note that from the last two blocks we can conclude that continuations can be arbitrary combined on sending and receiving sides. The size fields in string dope are always per string. Figure 6.13(a) shows a process of scattering a string in receiver's address space. Figure 6.13(b) depicts gathering process and Figure 6.13(c) shows the case when the receiver's string dope is too small to receive all data from the sender.

### END

This block describes forced exit from the infinitely loop and the end of the operation. The next command clears the copy area of the current IPC to avoid sending wrong data in case of an immediate start of the next IPC (see [15]).

```
string_abort:  free_copy_area(from);
```

We save the status of the transfer procedure in the receiver's dope (in the message descriptor).

```
to→msg_desc = msgdope.raw;
```

## 6.6 Function `ipc_copy`

This function copies the given number of bytes from the given source address to the given destination address. This function takes the following input arguments:

**dword_t * to** : address of the source

**dword_t * from** : address of the destination

**dword_t len** : the number of bytes to copy

If the number of bytes is word aligned, then this function just copies these words to the destination. If the number of bytes is not aligned, then at first it copies all aligned words, i.e., the number of bytes divided by four. In principle we can control the rest of the division by mask operations. If the second bit in the `len` is equal to one, then we have at least a rest of two bytes. The first bit shows that we have as rest at least one byte. So, we need to check these cases.
In case the second bit is equal to one then we convert input double word pointers to word pointers; therefore we can access two bytes by these word pointers. Afterwards we copy these two bytes. In case the first bit is equal to one, then we convert double word pointers to byte pointers. Then we copy the byte. Listing of the function is depicted below:

```
static void ipc_copy(dword_t * to, dword_t * from, dword_t len){
      int i = len / 4;
      while ( i-- )
         *(to++) = *(from++);
      if ( len & 2 ){
         *((word_t *) to) = *((word_t *) from);
          to = (dword_t *) ((word_t *) to + 1);
          from = (dword_t *) ((word_t *) from + 1);}
      if ( len & 1 ) *(byte_t *) to = *(byte_t *) from;}
```

# Chapter 7

# Automated Theorem Provers and Proofs

## 7.1 What Automated Theorem Provers Are

[1] Automated theorem prover (Prover): A mechanized reasoning tool (automated theorem prover) often supplements a formal language. Theorem provers (with formal languages) are typically university development programs that are continually evolving to include additions to the knowledge base. A prover's input language, a formal notation, can be used for formal specification alone or as a tool to break the model down into abstract objects of reasonable size for submission to the prover.
Automated theorem proving is an iterative process. The user must develop and submit a theorem, and any additional information about the theorem, to the prover. The prover applies rules of deduction and specific knowledge (if available) to the theorem to attempt a proof. It is up to the user to determine if the results are satisfactory. If not, information from the prover is used as additional knowledge and resubmitted until the proof is satisfied. A completed proof can be added to the knowledge base of the prover giving it additional knowledge to apply to future proofs.

## 7.2 PVS

PVS is the most recent in a line of specification languages, theorem provers, and verification systems developed at SRI, dating back over 20 years. That line includes the Jovial Verification System, the Hierarchical Development Methodology (HDM was one of the first tools for specification algorithms and them verification), and Ehdm (Enhanced Hierarchical Development Methodology). PVS stands for "Prototype Verification System", because it was built partly as a lightweight prototype to explore "next generation" technology for Ehdm, though it has now outgrown that role.
PVS consists of a specification language, a number of predefined theories, a theorem prover, various utilities, documentation, and several examples that illustrate different methods of using the system in several application areas. PVS exploits the synergy between a highly expressive specification language and powerful automated deduction; for example, some elements of the specification language are made pos-

[1] this section is literally taken from [16]

sible because the typechecker can use theorem proving. This distinguishing feature of PVS has allowed perspicuous and efficient treatment of many examples that are considered difficult for other verification systems. The specification language of PVS is based on classical, typed higher-order logic. The base types include uninterpreted types that may be introduced by the user, and built-in types such as the booleans, integers, reals, and the ordinals; the type- constructors include functions, sets, tuples, records, enumerations, and recursively-defined abstract data types such as lists and trees. Predicate subtypes and dependent types can be used to introduce constraints, such as the type of prime numbers. These constrained types may incur proof obligations during typechecking, but greatly increase the expressiveness and naturalness of specifications. In practice, most of the obligations are discharged automatically by the theorem prover. PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. Definitions are conservative (i.e., cannot introduce inconsistencies); to ensure this, recursive function definitions generate proof obligations to guarantee termination. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers, with a traditional syntax. Names may be freely overloaded, including those of the built-in operators such as AND and +. A case expression provides pattern-matching over the constructors of abstract data types, and tables allow piecewise-continuous functions to be specified in a visually appealing manner.
The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic over both integers and reals. The implementations of these primitive inferences are optimized for large proofs: for example, propositional simplifications are cached for efficiency. User-defined procedures can combine these primitive inferences to yield higher-level proof strategies, such as those for induction and CTL model checking. Proofs yield scripts that can be edited, attached to additional formulas, and rerun. This allows many similar theorems to be proved efficiently, permits proofs to be adjusted economically to follow changes in requirements or design, and encourages the development of readable proofs.

## 7.3 Specification of the Message Passing Protocol in the Function `ipc_sys`

As far as we considered the function **ipc_sys** we can conclude that besides the C-code, another important part is the message exchange protocol. As we said in the chapter 5, the main aspect of that protocol is that the receiver is expecting an IPC and provides the necessary buffer. In this chapter we specify this protocol.
A common way to model protocols, concurrent or reactive systems is by means of a transition relation. The instantaneous state of the system (protocol) is represented by an assignment of values to its variables. As it executes, the system progresses from one state to another, and the transition relation specifies the possible successors to each state. The sequence of states visited in one run of the system is called a trace. The set of all traces is called the behavior of the system. Verification questions one might ask of transition relations include whether the behavior induced by one (regarded as an implementation) implies that of other (regarded as a specification), whether a certain property is true of all reachable states, i.e., an invariant, and whether a state having a certain property is reachable on some or all traces starting from some given state (liveness properties). Many such properties of sets of traces can be specified compactly by means of temporal logic. To ask whether the behavior

specified by a certain transition relation satisfies a property specified by a certain formula of temporal logic can be viewed as asking whether the relation is a Kripke model of the formula. For some temporal logics and for transition relation over a finite state space, this model checking question can be decided very efficiently i.e. in linear time. The invention and popularization of this approach is due to Edmund Clarke and his students [16,17].
Using an efficient decision procedure[2] based on BDDs for a logic known as the $\mu$-Calculus[3], PVS provides model checking for a temporal logic known as Computation Tree Logic (CTL)[17] and transition relation defined on finite types.
In this chapter we specify our protocol in the form which is suitable to perform model checking. To do this we specify states space and transition relation of our protocol. We also check important invariant and liveness properties in our specification.

## 7.4 State Description

The state of a system is a collection of attributes that represents the system's operation. It is important to find a set of attributes that enables a full description of the function behavior and efficient method of representing these attributes. We have chosen several parameters (presented below) from the function data structure. These parameters allow us to describe the interaction protocol without loss of generality. To model a system state we introduce some PVS data types:

**tcb_state_t** This type describes the current thread state. It is implemented as enumeration of thread states (section 6.1):

```
tcb_states_t:  TYPE+ = {POLLING, WAITING, LOCKED_WAITING, LOCKED_RUNNING,
RUNNING, ABORTED}
```

**partner** In the source code this field is represented as a 32-bit vector. In our protocol we are only interested whether our partner is really our partner or not. We model this type as enumeration with two items:

```
tcb_id_t:  TYPE+ = {PARTNER, NON_PARTNER},
```

where PARTNER - partner is correct and NON_PARTNER - wrong id.

**msg_t** A message is a complex structure and consits of many parts. For us only the call of the function `transfer_message` is important. In assumption of the correctness of that function, we model a message as enumeration:

```
msg_t :  TYPE+ = {MSG_CORRECT, MSG_INCORRECT},
```

where MSG_CORRECT - a correct message was transfered. That is equal to a call of the function `transfer_message` during an execution of the protocol. MSG_INCORRECT - a wrong message, i.e., there was not call of the function `transfer_message`.

**msg_desc** In this protocol we just check whether IPC has sender/receiver phase, i.e., whether the message descriptor is not equal to 0xFFFFFFFF. So, with-

[2]This procedure, and also the BDD based propositional simplifier invoked by PVS's `(bddsimp)`,`(model-check)` commands, were provided by Geet Janssen, Eidhoven University of Technology

[3]$\mu$-Calculus is basically quantified Boolean logic with lest and greatest fixed point operators[18]

out loss of generality we model it as a bit vector of length 1.

```
m_desc_t:  TYPE+ = bvec[1]
```

We model a thread state as a record:

```
tcb_st :TYPE+ = [#
 state:  tcb_states_t,          %thread state
 partner:  tcb_id_t,            %id of partner
 msg:  msg_t,                   %message, abstract value
 snd_desc, rcv_desc:  m_desc,   %message descriptors.
 active:  bool                  %active variable, 1-thread is active, 0-no
#]
```

Note that we added the fields `snd_desc` and `rcv_desc` to the thread structure since we need to check whether IPC has the sending/receiving part. So, for the sender we will ignore `rcv_desc` and for the receiver `snd_desc`. This does not affect the protocol, since these two descriptors are input arguments for the function `ipc_sys` and we can access them at any time.
To simplify the specification of the protocol, we model a system state as tuple of sending thread and receiving thread:

```
statet:  TYPE = [#snd:  tcb_st,rcv:  tcb_st#]
```

## 7.5 Transition Relation

Once the state descriptor is defined, the next step is to define a function to describe the protocol in terms of state transitions. To simplify the description of the protocol we introduce some useful predicates over the data structure defined above:

`is_send?(th :tcb_st):  bool = snd_desc /= fill[1](TRUE) ;`
This predicate is true whenever an IPC has a sending part.

`is_rcv?(th:  tcb_st):  bool = rcv_desc /= fill[1](TRUE);`
This predicate is true whenever an IPC has a receiving part.

`init_snd?(s):  bool = RUNNING?(state(snd(s))) AND active(snd(s)) AND is_send?(snd(s));`
This predicate defines the initial state of the sender, i.e., the sender can begin an interaction if its thread state is RUNNING, it has a sending part of the operation, and the scheduler has selected this thread.

`init_rcv?(s):  bool = RUNNING?(state(rcv(s))) AND active(rcv(s)) AND is_rcv?(rcv(s));`
This predicate defines the initial state of the receiver, similarly as for the sender.

`is_rcv_waiting?(s):  bool = WAITING?(state(rcv(s))) AND (PARTNER?(partner(rcv(s))));`
This predicate is true whenever the receiver is waiting for a/the sender. In other words, the receiver thread is WAITING and its partner is either the sender or L4_NIL_ID, that corresponding to *open wait*.

`is_snd_polling?(s):  bool = (POLLING?(state(snd(s)))) AND (snd(s) /= NONEXIST_TH) AND (PARTNER?(partner(snd(s))));`
This predicate is true whenever the sender is waiting for the receiver. So, the

```
next(s1,s2):  bool =
((init_snd?(s1) AND is_rcv_waiting?(s1)) AND
       (RUN_snd?(s2) AND RUN_rcv?(s2) AND transf_cmpl?(s2))) OR
((init_snd?(s1) AND (NOT is_rcv_waiting?(s1))) AND
       (is_snd_polling?(s2) AND active(rcv(s2)) AND (NOT
transf_cmpl?(s2)))) OR
((init_rcv?(s1) AND is_snd_polling?(s1)) AND
       (is_rcv_ready_receive?(s2) AND active(snd(s2)) AND (NOT
transf_cmpl?(s2)))) OR
((init_rcv?(s1) AND (NOT is_snd_polling?(s1))) AND
       (is_rcv_waiting?(s2) AND active(snd(s2)) AND (NOT
transf_cmpl?(s2)))) OR
((is_rcv_ready_receive?(s1) AND active(snd(s1))) AND
       (RUN_rcv?(s2) AND RUN_snd?(s2) AND transf_cmpl?(s2)))
```

Figure 7.1: Specification of `next` function

sender thread exists and its state is POLLING. The sender's partner is the receiver.

`RUN_snd?(s):  bool = RUNNING?(state(snd(s))) AND NOT is_send?(snd(s));`
`RUN_rcv?(s):  bool = RUNNING?(state(rcv(s))) AND NOT is_rcv?(rcv(s));`
These predicates say whether sender/receiver thread is ready to be executed after the IPC operation. We reset sender and reciever messsage descriptors since they are no longer relevant after the IPC operation.

`transf_cmpl?(s):  bool = MSG_CORRECT?(msg(rcv(s)))`
This predicate is true whenever a message was successfully transfered.

To describe the transition relation we label a state with a set of predicates. In that set we include only the predicates which hold in that state. The predicates which are not included in the set will be treated as arbitrary values. We call a set of predicates true if and only if all predicates in that state are true.
We refer to a transition as a conjunction of two states i.e. the start state and the descendant state. Note that we can evaluate conjunction of two states as boolean value, since it can be viewed as conjunction of two sets of predicates. For example:

```
(init_snd?(s1) AND is_rcv_waiting?(s1)) AND (RUN_snd?(s2) AND RUN_rcv?(s2)
AND transf_cmpl?(s2))
```

This transition describes the case where the sender starts an interaction and the receiver is ready. So, this is the case where the transfer is completed at once and after IPC operation both threads are ready to be executed. Disjunction of all transitions gives us complete transition relation of our protocol. Figure 7.1 shows this transition relation.

## 7.6 Properties Specification and Proofs

One of the main requirements is the termination of the protocol. In other words, whenever the protocol is started, both sides will be able to be executed again. We specify this requirement as a PVS theorem with the help of the CTL:

```
no_dead_lock:  THEOREM
AG( next, LAMBDA(st:statet):
    (init_rcv?(st) OR init_snd?(st) IMPLIES
       AF(next, LAMBDA s1:  RUN_snd?(s1))(st) AND
       AF(next, LAMBDA s1:  RUN_rcv?(s1))(st)))(s)
```

This means that whenever the sender or the receiver starts an interaction, then in all possible computation paths from the current position exists a state where the sender is ready to be executed again. Similarly exists a state where the receiver thread is ready to be executed again. This guarantees that there are not any dead locks in the protocol. A call of the PVS function `model-check` is enough to check this statement and say that our system has this invariant.
Another main aspect is that a message should be transfered. We assume that if the sender has some data to send then the receiver is willing to receive this data. We also assume that the function `transfer_message` is correct. The appropriate theorem is presented below.

```
transf_cmpl:  THEOREM
AG( next, LAMBDA(st:statet):
    init_snd?(st) OR init_rcv?(st) IMPLIES
       AF(next, LAMBDA (t:statet):  transf_cmpl?(t))(st))(s)
```

This statement also was successfully checked via PVS command `model-check`.

# Chapter 8

# Summary

My diploma thesis is a part of the project "Verification of the L4 operating system kernel". The aim of this project is to formally verify the L4 kernel in order to guarantee its correctness. In my thesis I have documented the implementation of the IPC mechanism in the L4 kernel. I have also proved a part of IPC mechanism of the L4 kernel. I proved the correctness of the message passing protocol of the IPC mechanism. This was done in three steps: At first I have created the formal specification of that protocol. After that I built a model of the IPC mechanism in PVS and at last I proved that the created model fulfils the specification. The model which is built in this work is not meant to be a precise representation of the original protocol. The proof of such a model is not the same as proofs for the original C code, but they may allow to write a correct source code. Therefore for software verification we need some tool that can work with C-code or any programming language in order to verify its correctness.

# Appendix A

# List of all used L4-function

get_current_tcb
tid_to_tcb
l4_is_invalid_id
l4_is_nil_id
tcb_exist
return_ipc
return_args_ipc
IS_WAITING
IS_POLLING
IS_INFINITE_SEND_TIMEOUT
TIMEOUT
thread_enqueue_wakeup
thread_dequeue_wakeup
thread_enqueue_send
thread_enqueue_ready
ipc_switch_to_idle
ipc_switch_to_thread
IS_MAP
IS_SHORT_IPC
same_address_space
get_copy_area
free_copy_area
ipc_copy
ipc_sys
transfer_message
extended_transfer

# Appendix B

# List of all used predicate over a message descriptor

IS_SEND?
IS_RECEIVE?
IS_OPEN_WAIT

# Bibliography


[1] *Christopher B. Browne's Home Page*, http://www.cbbrowne.com/info/microkernel.html

[2] *Operating System Technical Comparison*, http://www.osdata.com/

[3] *The Mach Project* http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html

[4] *The L4ka Project* www.l4ka.de

[5] *QNX* http://www.qnx.com/

[6] Jochen Liedtke. *$\mu$-Kernels Must And Can Be Small*

[7] *MIT Exokernel Operating System* http://www.pdos.lcs.mit.edu/exo.html

[8] Alan Au, Gernot Heiser. *L4 User Manual*, School of computer since and engineering the university of New South Wales Sydney 2052, Australia, 1999

[9] Jochen Liedtke. *On-$\mu$-kernelconstruction*, In Proceeding of the 15th ACM Symposium on OS Principles, December 1995

[10] Jochen Liedtke. *Towards real microkernels*, Communications of the ACM, September 1996

[11] Abraham Silzberschatz, Peter Baer Galvin, Greg Gange. *Applied Operating System Concepts, First Edition*, 2000 John Wiley & Sons, Inc. ISBN 0-471-36508-4

[12] *L4 eXperemental Kernel Reference manual*, L4ka Team, University Karlsrue, Germany, 2002.

[13] Jochen Liedtke. *Clan&Chiefs*, In 12. GI/ITG-Fachtagung Architektur von Rechensystemen, Kiel, 1992. Springer Verlag

[14] Jochen Liedtke. *Improving IPC by Kernel Design* 14th ACM Symposium on OS Principles, December 1993

[15] J.Crow, S. Owre, N. Shankar, J.M. Rushby, M. Srivas. *A Tutorial Introduction to PVS*, SRI-International 1995. http://www.csl.sri.com/sri-cls-fm.html

[16] J.R. Burch, E.M. Clarke, K.L. McMillian. *Symbolic model checking: $10^{20}$ states and beyond*, Information and computation, 98(2):142-70, June 1992

[17] E.M. Clarke, Orna Grumberg. *Model Checking*, MIT Press, Cambridge, Massachusetts, 1999

[18] David Park. *Finiteness in mu-ineffable*, Theoretical Computer Science, 3:173-181, 19

[19] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert. *PVS Language Reference*, SRI-International 1999. http://pvs.csl.sri.com/

[20] E.Petrova *Documentation of memory managment functions in the L4 microkernel*, University of Saarland, 2003

This Master Thesis has been written on my own without any unpermitted help and using mentioned materials only.
**Sergey Tverdyshev, September 2003**