Embedded, Real-Time, and
Operating Systems Program
http://ertos.nicta.com.au/

L4 User Manual
NICTA L4-embedded API

Ihor Kuz

Document Version 1.11 of October 5, 2005

ihor.kuz@nicta.com.au
http://www.cse.unsw.edu.au/~disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia

This document is a user manual for the L4 μ-kernel, specifically the NICTA L4-embedded N1 API. It gives an introduction to the main concepts and features of L4, and explains their use by a number of examples.

The manual is platform independent, however, examples showing the compiling and linking of programs are based on the L4 implementation for the UNSW u4600 platform (which uses a MIPS R4x00 processor). Compiling and linking may differ on other platforms.

This document supplements, rather than replaces, the L4 Reference Manual, and anyone intending to write applications on top of L4 should obtain the L4 Reference Manual.

Preface

This manual is a complete re-write of the original *L4 User Manual* [AH98] written by Alan Au and Gernot Heiser for the MIPS implementation of the Version 2 API of L4. It has been adapted to the V4 API and extended by a more general introduction in order to make it more approachable to the uninitiated. There are presently two versions of this document: for the Karlsruhe X.2 API and the NICTA L4-embedded N1 API. This version is for the latter, adapted from the former by Carl van Schaik.

The original user manual grew out of lecture notes for a course on Advanced Operating Systems given by Gernot Heiser at UNSW in July–November 1997. In that course students had to build a small operating system on top of L4/MIPS which had previously been developed at UNSW by Kevin Elphinstone with the help of Jochen Liedtke. The 44 students of that course (one of whom was Alan Au) were guinea pigs who need to be thanked for their interest and patience, as well as for their questions which prompted the lecturer to write and provide further documentation. Thanks are also due to the late Jochen Liedtke, who provided the L4 specification and the original reference manual, Kevin Elphinstone for the MIPS implementation as well as for explaining L4's idiosyncrasies, the OS group at the Technical University of Dresden under Herrman Härtig for example code, C bindings and manual pages, and Jerry Vochteloo and many others for their contributions.

Thanks for this manual are due also to Gernot Heiser.

This manual is still as an evolving document — a number of chapters are still empty waiting for a volunteer to fill them in. It is hoped that it will continue to grow as more people learn to use L4, and consequently discover shortcomings of the documentation. We welcome comments, corrections and enhancements and will try to incorporate them quickly into the manual. The latest version of this manual (as well as of the L4 Reference Manual [NIC05] and other L4-related documentation) are available from `http://l4hq.org` and `http://ertos.nicta.com.au` .

Contents

Chapter 1

Introduction

1.1 About This Guide

L4 is an operating system microkernel (μ-kernel). That is, L4 by itself is not an operating system (OS), but rather a minimal base on which a variety of complete operating systems can be built. This guide will provide programmers[1] new to L4 with an easy way to familiarise themselves with the concepts behind L4, its programming model, and its API.

Because of L4's minimality, the API provides quite a low-level interface. As such it does not include services available in traditional OS environments such as page swapping, device drivers, and file systems. If such services are required it is up to the L4 programmer to provide them. This guide does not cover the building of such services, which is what OS courses and textbooks are for. It does, however, attempt to provide enough of an understanding of L4 and its programming model that such services can be built to run on L4.

1.2 Quick Start

For those impatient readers who do not want to read about L4's histroy, design philosophy, core concepts, or the underlying architectural model, Chapter 4 provides a guide to the key issues involved in programming for L4. Readers looking for a quick start are advised to skip straight to this chapter and refer back to Chapter 2 for clarification of terms and concepts when necessary. Appendix A provides a complete example application, and can be used as a guide to structuring a simple L4 program.

1.3 L4 History

The basic idea of a μ-kernel goes back to Brinch Hansen's Nucleus [BH70] and Hydra [WCC+74] and has been popularised by Mach [RTY+88]. 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 with a smaller *trusted computing base*, which results in a more secure and reliable system. Likewise, by moving most OS services outside the kernel, it becomes easier to extend and customise and operating system. A further advantage is that μ-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. Operating systems tend to grow when as 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 being the addition of the VFS layer in UNIX for supporting

[1]Note that this is a guide for programmers writing software using the L4 API. It is not an L4 kernel hacker's guide.

NFS). Unlike traditional (“monolithic”) operating systems, a microkernel-based system grows horizontally rather than vertically: Adding new services means adding additional servers, without lengthening the critical path of the most frequently-used operations.

Unfortunately, however, performance of first-generation microkernels proved disappointing, with applications generally experiencing a significant slowdown compared to traditional monolithic operating systems [CB93]. Liedtke, however, has shown [Lie93, Lie95, Lie96] 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, as has been clearly demonstrated by Härtig *et al.* [HHL$^+$97].

1.4 L4 Design Philosophy

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 μ-kernel should to be as small as possible. Hence, the main design criterion of the μ-kernel is *minimality* with respect to security: *A service (feature) is to be included in the μ-kernel if and only if it impossible to provide that service outside the kernel without loss of security.* The idea is that once we make things small (and do it well), performance will look after itself.

A strict application of this rule has some surprising consequences. Take, for example, device drivers. Some device drivers access physical memory (e.g. DMA) and can therefore break security. They need to be trusted. This does not, however, mean that they need to be in the kernel. If they do not need to execute privileged instructions and if the kernel can provide sufficient protection to run them at user level, then this is what should be done. Consequently, this is what L4 does.

According to Liedtke, and based on such reasoning, a μ-kernel must provide:

address spaces: the basis of protection,

threads: an abstraction of program execution,

interprocess communication (IPC): a mechanism to transfer data between address spaces,

unique identifiers (UIDs): providing context-free addressing in IPC operations.

We return to these and other fundamental concepts in Chapter 2.

1.5 L4 Resources

The main NICTA L4-embedded resource and official API document is the L4 Reference Manual [NIC05] , which contains a complete specification of the API. Besides the L4 source code itself, this is currently also the only complete documentation. While documents describing older versions of L4, as well as user guides describing these older versions, are also available, the L4 API and programming model have changed significantly. This makes the older documents interesting only from a historical perspective.

1.6 Overview

The rest of this guide is structured as follows. Chapter 2 introduces the core concepts of L4. It also describes the fundamental L4 architectural model. Chapter 3 explains how to run an L4-based system. Chapter 4 provides practical information about programming L4. It describes how standard L4 API procedures are best used and provides code to illustrate this. Chapter 5 continues where Chapter 4 leaves off and provides examples of typical L4 programming practice or idioms. Chapter 6 introduces Kenge and the L4 Environment, both of which provide an extended programming environment for L4. Chapter 8 provides information about the L4 kernel debugger and explains how to use it to debug L4 programs. Chapter 9 provides a list of things that are not, or not satisfactorially implemented in L4, while Chapter 10 provides some informaion about using L4 on specific platforms.

Appendix A provides an example of the implementation of a simple L4 program. Appendix B discusses how to compile a program, link and combine it into a single L4 loadable image, and boot this image on a machine.

Note that this guide purposefully does not provide instructions on how to download and compile L4 itself. This is due to the fact that there are various versions of L4 (and also various ports of L4) around and they are in constant flux. As such, the instructions would become obsolete quickly leaving a large part of this guide obsolete as well. This kind of information is best presented on a website where it can be updated as necessary. The best bet for finding information on downloading and installing L4 is, therefore, to look on appropriate L4 websites (e.g., `http://www.l4hq.org`), or to ask someone already familiar with L4.

Chapter 2

Basic L4

2.1 Core Concepts

This section introduces key concepts that underly L4 and the L4 architecture. Most are general concepts and common to most (μ-kernel) operating systems. The application of the concepts and the relationships between them are, however, in some ways unique to L4. It is important that a programmer understands how these concepts are used in L4 and how they fit into the general L4 architecture.

2.1.1 Memory and Address Spaces

Memory

In operating systems we distinguish between two types of memory: physical and virtual. Physical memory refers the real memory available on the computer and is referenced using physical addresses. A physical address can be interpreted directly by a computer as a reference to a location in RAM or ROM. On most architectures, however, programs running on the CPU generate virtual addresses. A memory-management unit (MMU) has to translate virtual addresses into physical addresses before they can be used to address real memory. We will look at the relationship between virtual and physical addresses when we discuss address spaces below.

Registers

Besides regular (virtual) memory, L4 also provides programs with access to virtual registers. Virtual registers offer a fast interface to exchange data between the microkernel and user threads. They are registers in the sense that they are static per-thread objects. Depending on the specific processor type, they can be mapped to hardware registers or to memory locations. Mixed mappings, where some virtual registers map to hardware registers while others map to memory, are also possible.

There are two classes of virtual registers: Thread Control Registers (TCRs) and Message Registers (MRs) Each will be discussed in more detail later.

In general, virtual registers can only be addressed directly, not indirectly through pointers. The L4 API provides specific functions for accessing the three different classes of registers. Loading illegal values into virtual registers, overwriting read-only virtual registers, or accessing virtual registers of other threads in the same address space (which may be physically possible if some are mapped to memory locations) is illegal and can have undefined effects on all threads of the current address space.

Address Space

An address space contains all the data that is directly accessible by a thread. It consists of a set of mappings from virtual to physical memory. This set of mappings is *partial* in the sense that many mappings may be undefined, making the corresponding virtual memory inaccessible. Figure 2.1 shows an example of how the virtual memory in an address space may map onto physical memory. The regions of virtual memory that are not mapped are inaccessible to threads running in that address space.

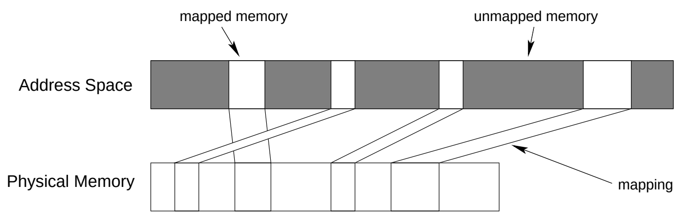


Figure 2.1: Mapping of virtual memory to physical memory

Typically an address space will contain a number of standard regions. These include the text, data, heap, and stack regions. The text region contains program code, the data region contains pre-initialised data used by the program, the heap region is used for dynamically allocated data, and the stack region is used for storing temporary data during execution. The text and data regions usually have a fixed size and do not change during execution of a program. The heap and stack regions, on the other hand, can grow or shrink while the program is executing. Note that by convention on most architectures the heap grows up toward high memory, while the stack grows down toward lower memory.

Figure 2.2 shows a typical arrangement of these regions in an address space. Note that L4 does not enforce any specific layout of address spaces. The layout is generally determined by a combination of the compiler and higher-level operating system services built on top of L4.

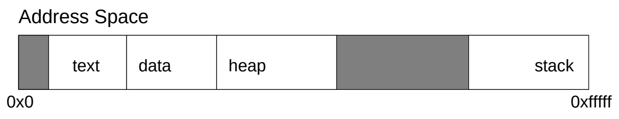


Figure 2.2: A typical address space layout

2.1.2 Threads

A *thread* is the basic execution abstraction in L4. L4 threads are light-weight and cheap to manage. The light-weight thread concept together with fast IPC are the keys to the efficiency of L4 and OS personalities running on top of L4.

A thread is always associated with a specific address space. The relationship between threads and address spaces is one to many, that is, a thread is associated with exactly one address space, while any address space can have many threads associated with it.

Each thread has its own set of virtual registers called the *thread control registers* (TCRs). These virtual registers are static (they keep their values until explicitly modified) and store the thread's private state (e.g., parameters for IPCs, scheduling information, the thread's identifiers, etc.). They are stored in an area of the address space called the *thread control block* (TCB). The TCB is actually split into two parts, there is the *user TCB* (UTCB), which is accessible by the thread, and the *kernel TCB* (KTCB) which is accessible only by the kernel. Programmers are generally only interested in the UTCB, therefore, when we refer to the TCB we will generally mean the UTCB. Details about the TCB and the information stored in TCRs can be found in the L4 Reference Manual.

Each thread is further associated with a *page-fault handler* and an *exception handler*. These are separate threads that are set up to handle page faults and other exceptions caused by the thread. Page faults, pagers, exceptions and exception handlers will be discussed in more detail later.

Each thread in an address space has its own stack. A thread's stack address is explicitly specified during thread creation. It is up to the thread to determine where to place its heap — this is a design issue, but typically several threads in the same address space will share a heap.

L4 also distinguishes between *privileged* and *non-privileged* threads. Any thread belonging to the same address space as one of the initial threads created by the kernel upon boot-time (see Section 3.1) is treated as privileged. Some system calls can only be executed by privileged threads.

Threads can be created as *active* or *inactive* threads. Inactive threads do not execute but can be activated by active threads that execute in the same address space. As discussed later, inactive threads are typically used during the creation of address spaces. A thread created *active* starts executing immediately after it is created. The first thing it does is execute a short receive operation waiting for a message from its pager. This message will provide the thread with an instruction and stack pointer and allow it to start executing its code.

Tasks

A *task* is the set of of threads sharing an address space. The terms task and address space are often used interchangeably, although strictly speaking they are not the same. We will try to avoid using the term task where possible, prefering to specifically refer to an address space or thread.

Identifying Threads and Address Spaces

A thread is identified by its unique identifier (UID). A thread can actually have two identifiers, a *local* identifier and a *global* identifier. While a global identifer is valid in any address space, a local identifier is only valid within the thread's address space. That is, a global identifier can be used by any thread, while a local identifier can only be used by threads that are part of the same task. In different address spaces, the same local thread ID may identify different and unrelated threads.

Unlike threads, address spaces do not have identifiers. Instead, an address space is identified by the UID of any thread associated with that address space. This means that an address space must always have at least one thread associated with it. That thread does not, however, have to be active.

2.1.3 Communication

One of the main activities that threads engage in is to communicate with other threads (for example, in order to request services from each other, in order to share results of computations, etc.). There are two ways that threads communicate: using shared memory (see Section 2.1.4), or using L4's Interprocess Communication (IPC) facilities (see Section 2.1.5).

Communication Within an Address Space.

When communicating with threads in the same address space, it is easiest (and most efficient) to use shared memory. Threads in the same address space automatically share memory, so they do not have to make use of L4's memory mapping facilities. As long as both threads agree on which shared memory region (or variables) to use, they are free to communicate in this way.

When threads communicate using shared memory it is necessary to avoid race conditions. This is best done by enforcing mutual exclusive access to shared memory. Note that L4 does not provide any mutual exclusion primitives (such as semaphores) to do this, it is expected that these are provided at user level (possibly using mechanisms provided by the underlying hardware). Various implementations of user level mutual exclusion primitives are available.

It is possible for threads in the same address space to communicate using IPC. The main use for this is thread synchronisation. Addressing a thread in the same address space can be done using local or global thread Ids.

Communication Between Address Spaces.

When communicating between address spaces (i.e., when threads in different address spaces communicate with each other) both the use of shared memory and IPC are valid approaches. IPC is generally used for smaller messages and synchronisation, while shared memory is used to exchange larger amounts of data.

Communicating between address spaces using shared memory requires that all (communicating) threads have access to the same memory region. This is achieved by having one thread map a region of its address space into the address spaces of the other threads. The concept of mapping memory is explained in Section 2.1.4. Once a shared region of memory has been established, the threads communicate by simply reading from or writing to the particular memory region. Note that, as mentioned previously, when communicating using shared memory it is necessary to avoid race conditions.

Communication using IPC requires that the threads send messages to one another. A message is sent from a sender thread and addressed to a particular receiver thread. Messages can be used to directly share data (by sending it back and forth), to indirectly share data (by sending memory mappings), or as a control mechanism (e.g., to synchronise).

2.1.4 Memory Mapping

Address spaces can be recursively constructed. A thread can *map* parts of its address space into another thread's address space and thereby share data. Figure 2.3 shows an example of two address spaces with a region of address space A mapped into address space B. Both the thread running in address space A and the thread running in address space B can access the shared region of memory. Note, however, that the memory may have different virtual addresses in the different address spaces. Thus, in the example, a thread running in address space A accesses the shared memory region using virtual address `0x1000000` while a thread in address space B uses virtual address `0x2001000`.

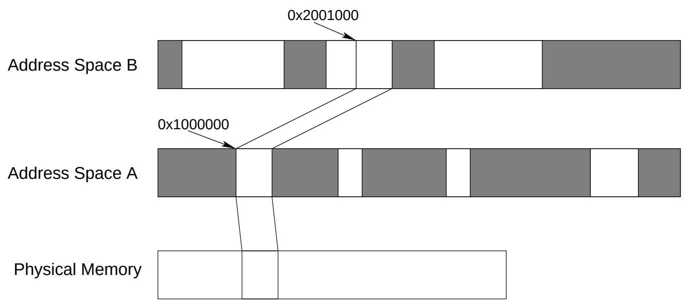


Figure 2.3: Two address spaces sharing a region of memory

A *mapper* (that is, the thread making the memory available) retains full control of the mapped region of memory. In particular, the mapper is free to revoke a mapping at any time. Revoking a mapping is called *unmapping*. After a mapping has been revoked, the receiver of that mapping (the *mappee*) can no longer access the mapped memory. This is shown in Figure 2.4. Here a thread in address space B can no longer access the region of virtual memory that used to contain the mapped memory.

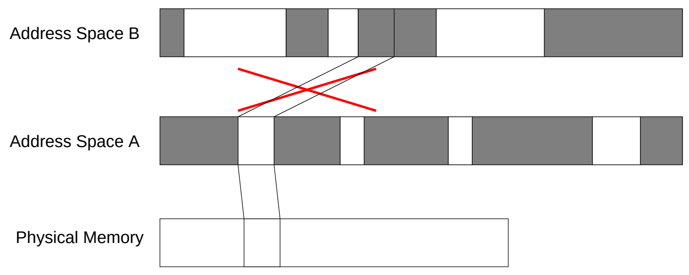


Figure 2.4: A shared region of memory is unmapped

Access to mapped memory is limited by the access permissions set by the mapper. These access permissions specify read permission, write permission, and execute permission and determine how the mappee can access the mapped memory. Note that a mapper cannot grant access rights that it does not itself have. Thus, if a thread does not have write access to a particular region of memory, it cannot map that memory with write permission into another address space.

It is also possible for a region of one address space to be *granted* to another address space. Granting differs from mapping in that after the grant has succeeded, the granter loses access to that region of its address space (i.e., it no longer has a valid mapping for that region). Figure 2.5 shows an example of the situation before and after a region of address space A has been granted to address space B. Unlike revocation of mappings, a granter cannot revoke a grant, as it no longer has access to the page.

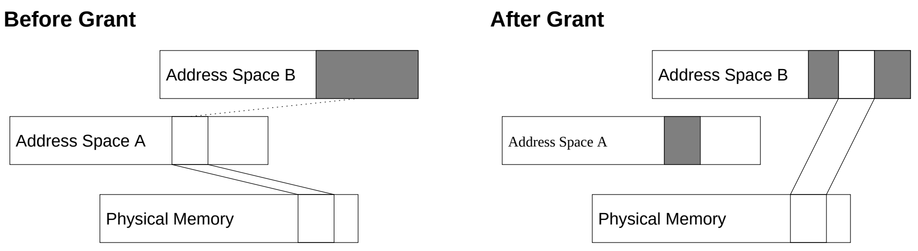


Figure 2.5: A region of one address space is granted to another

Note that, for security reasons, when mapping or granting memory the receiver must always explicitly agree to accept maps or grants.

Mapping and granting of memory are implemented using L4's IPC mechanism (IPC is discussed in Chapter 4). In order to map memory, a mapper sends the intended mappee a message containing a *map item* specifying the region of memory to be mapped. The mappee must explicitly specify that it is willing to receive mappings. It also specifies where, in its own address space, the memory should be mapped. The receiver does not actually have to do anything with the received map item (or grant item). The mapping is performed as a side effect of the IPC.

Note that although mappings apply to an address space, map messages are sent via IPC to a thread in that address space.

2.1.5 Interprocess Communication (IPC)

Message passing is the basic *interprocess communication* (IPC) mechanism in L4. It allows L4 threads in separate address spaces to communicate by sending messages to each other. This message-passing IPC is the heart of L4. It is used to pass data between threads (either by value, with the μ-kernel copying the data between two address spaces, or by reference, through mapping or granting). L4 IPC is normally synchronous and is used for synchronisation (each successful IPC operation results in a *rendez-vouz*). It is even used for memory management (the μ-kernel converts a page fault into an IPC to a user-level pager), exception handling (the μ-kernel converts an exception fault into an IPC to a user-level exception handler), and interrupt handling (the μ-kernel converts an interrupt into an IPC from a pseudo-thread to a user-level interrupt-handler).

Messages

A message consists of one mandatory and two optional sections. The mandatory *message tag* is followed by the optional *untyped-words* section which is followed by the optional *typed-items* section. The message tag contains message control information and a message label. The message control information specifies the size of the message and the kind of data contained in it. The μ-kernel associates no semantics with the message label; it allows threads to identify a message and is often used to encode a request key or to identify the function to be invoked upon reception of the message.

The untyped-words section holds arbitrary data that is untyped from the μ-kernel's point of view. The data is simply copied to the receiver. The μ-kernel associates no semantics with it. The typed-items section contains typed data such as map items and grant items. Map items and grant items were introduced earlier and are used to map and grant memory.

Message Registers

IPC messages are transferred using *message registers* (MRs). A sender writes a message into the message registers associated with its own thread and a receiver reads the message out of the message registers associated with its thread. Each thread has 32/64 MRs (architecture defined)(n), numbered MR_0 to MR_{n-1} (inclusive). A message can use some or all MRs to transfer untyped words and typed items. The message tag is always transfered in MR_0.

MRs are transient read-once virtual registers. Once an MR has been read, its value is undefined until the MR is written again. The send phase of an IPC implicitly reads all MRs; the receive phase writes the received message into MRs.

MRs can be implemented as either special purpose (hardware) registers, general memory locations, or general purpose (hardware) registers. It is generally up to the particular L4 implementation (and the hardware that it is implemented on) whether any MRs are implemented as hardware registers and if so which ones. For example, in the MIPS implementation MR_0 to MR_9 are implemented as hardware registers.

Acceptor TCR

In order to be able to handle a received message, the receiver must explicitly agree to accept messages of that type. The *acceptor* is used to specify which typed items will be accepted when a message is received. If an acceptor specifies that map or grant items are accepted, then it also specifies where the associated memory will be mapped in the receiver's address space.

Send and Receive

Messages are sent and received through the *IPC* system call. IPC is the fundamental operation for inter-process communication and synchronization. It can be used for intra- and inter-address-space communication. Normally, communication is synchronous and unbuffered: a message is transferred from the sender to the recipient if and only if the recipient has invoked a corresponding IPC operation. The sender blocks until this happens or until a period specified by the sender has elapsed without the destination becoming ready to receive. Similarly, the reciever blocks until a message has been recieved. This unbuffered operation reduces the amount of copying involved and is the key to high-performance IPC.

A single IPC call combines an optional send phase and an optional receive phase. Which phases are included is determined by specific parameters to the IPC call. It is also possible to specify blocking/non-blocking IPCs. Different combinations of blocking and send and receive phases lead to fundamentally different kinds of IPCs. For example, including both a send and a receive phase with noblocking implements a synchronous IPC that blocks until a reply is received, while an IPC that includes only a receive phase and no blocking, implements a blocking call that waits for a message to arrive. The L4 API provides convenience interfaces to these different kinds of IPC calls.

IPC messages are always addressed to specific threads using their unique (local or global) identifiers.

2.2 Kernel Data Structures

The *kernel-interface page* (KIP) contains API and kernel version data, system descriptors including memory descriptors, and system-call links. The page is a μ-kernel object and is directly mapped through the μ-kernel into each address space upon address space creation. It is not mapped by a pager, cannot be mapped or granted to another address space and can not be unmapped. The creator of a new address space can specify the address where the kernel interface page will be mapped. This address will remain constant through the lifetime of that address space.

2.3 L4 Architecture

2.3.1 Kernel

2.3.2 Privileged Tasks (or Threads??)

2.3.3 Servers

2.3.4 Programs

2.3.5 Memory Management

Sigma0

Page Faults

Pagers

2.3.6 Interrupt Handling

2.3.7 Exception Handling

2.3.8 Scheduling

Chapter 3

Running L4

Chapter 4

Programming L4

4.1 NICTA L4-embedded API

The NICTA L4-embedded API is described in the L4 Reference Manual [NIC05]. This manual contains specifications of all the data structures and functions provided by the API. This includes kernel data structures, message formats, system calls, and convenience functions. The API is presented in a generic language-neutral way (based on a pseudo C++ syntax) and is valid for all 32-bit and 64-bit architectures. Language bindings provide language-specific versions of the API. The generic API specification makes use of function overloading (that is numerous functions have the same name and are distinguished only by their formal parameters). Because this causes problems for bindings to languages that do not support overloading (e.g., C), the API also specifies alternative, non-ambiguous, names. These alternative names are provided (enclosed in square brackets []) alongside the official names.

The code examples in this document use the C language binding of the API. This binding prepends an `L4_` prefix to all type and function names, and appends a `_t` suffix to all type names. Thus, `Send` becomes `L4_Send`, and `MsgTag` becomes `L4_MsgTag_t` in the C binding.

4.2 Threads and Address Spaces

4.2.1 Thread Lifecycle

A new thread is created using the `L4_ThreadControl()` function. A call to `L4_ThreadControl()` must specify a thread id, an address space identifier, a scheduler, a pager , a send redirector, a recieve redirector and a pointer to the UTCB area (i.e., where the thread's UTCB will be placed). A thread can be created active or inactive. An inactive thread (i.e., one whose `pager` parameter is `NIL`) is simply brought into being but does not run. A thread is activated by assigning it a pager. When a thread is activated the first thing it does is wait for an IPC (from its pager) containing the IP and SP (instruction pointer and stack pointer) of the code to execute.

The following function shows an example of the code required to create and activate a thread.

```
L4_ThreadId_t create_thread (L4_ThreadId_t tid) {
  L4_Word_t utcb_location;
  L4_ThreadId_t space_specifier = L4_Myself();
  L4_ThreadId_t scheduler = L4_Myself();
  L4_ThreadId_t pager = L4_Myself();
  int res;
  /* Create active thread */
  res = L4_ThreadControl(tid, space_specifier, scheduler, pager,
                         L4_anythread, L4_anythread,
                         (void *) utcb_location);
  if (res != 1) {
    /* error */
  }

  return tid;
}
```

In this example the thread is created in the same address space as the creator (space_specifier = L4_Myself() — recall that an address space is identified by the UID of a thread associated with it), and it is assumed that the creating thread will take on the role of both the scheduler and pager. This is not always the case, and will depend on the context that the thread creation takes place in.

Note that L4_ThreadControl() can only be invoked by a privileged thread[1].

The following function shows how a pager might start a thread by sending it an instruction pointer and a stack pointer.

```
void start_thread(L4_ThreadId_t tid, void *ip, void *sp,
                  L4_ThreadId_t pager) {
  L4_Msg_t msg;
  L4_MsgTag_t tag;

  L4_MsgClear(&msg);
  L4_MsgAppendWord(&msg, (L4_Word_t)ip);
  L4_MsgAppendWord(&msg, (L4_Word_t)sp);

  /*
    The start thread message must come from the thread's pager.
  */
  L4_Set_Propagation(&msg.tag);
  L4_Set_VirtualSender(pager);

  L4_MsgLoad(&msg);
  tag = L4_Send(tid);

  if (L4_IpcFailed(tag)) {
    /* error */
  }
}
```

We will discuss the code used to create and send a message (i.e., L4_MsgClear(), L4_MsgAppendWord(), etc.) later. Of importance here is the setting of the propagation bit[2] with

[1]Recall (from Section 2.1.2) that a thread is privileged if it belongs to the same address space as one of the initial threads created by the kernel at boot-time.

[2]A discussion of propagation is beyond the scope of this document. Please see the L4 Reference Manual for more details

L4_Set_Propagation() and the virtual sender TCR with L4_Set_VirtualSender(). This ensures that the message is seen to come from the new thread's pager.

L4_ThreadControl() is also used to destroy threads. This is done by calling L4_ThreadControl() with a space_specifier parameter of NIL (L4_nilthread). For example:

```
  L4_ThreadControl(tid, L4_nilthread, scheduler, pager, (void *)
                   L4_anythread, L4_anythread, utcb_base);
```

4.2.2 Address Space Lifecycle

To create a new address space it is sufficient to create a thread in that address space. Since an address space does not have an identifier of its own (but is simply identified by the threads that it is associated with) this new thread must be created with the same UID for both its thread Id and its address space identifier. After being created in this way, the address space must be initialised before it can actually be used. Initialisation of an address space involves setting up appropriate memory areas, and is done using the L4_SpaceControl() function as shown in the following code.

```
int new_task(L4_ThreadId_t task, L4_ThreadId_t pager, void *ip, void *sp) {
  L4_Word_t control;
  int res;

  /* Create an inactive thread */
  res = L4_ThreadControl (task, task, L4_Myself(), L4_nilthread,
                          L4_anythread, L4_anythread, (void *) -1);
  if (res != 1) {
    /* error */
  }

  /* Initialise address space */
  res = L4_SpaceControl (task, 0, kip_area, utcb_area, &control);
  if (res != 1) {
    /* error */
  }

  /* Activate thread */
  res = L4_ThreadControl (task, task, scheduler, pager,
                          L4_anythread, L4_anythread,
                          (void *) utcb_base);
  if (res != 1) {
    /* error */
  }

  start_thread(task, ip, sp, pager);

  return 0;
}
```

In this example the address space is created by creating a thread using the same UID (`task`) for both its thread Id and its address space identifier. The thread is originally inactive (because `L4_nilthread` is given as its pager thread Id). After creating the thread `L4_SpaceControl()` is called to initialise the address space. In particular this sets up the address space's KIP and UTCB areas. The original thread is subsequently activated by assigning it a pager process. Finally the thread is started by sending it the given instruction pointer (`ip`) and stack pointer (`sp`).

Note that, like `L4_ThreadControl()`, `L4_SpaceControl()` can only be invoked by a privileged thread.

Destroying the last thread in an address space implicitly also destroys that address space.

4.3 Interprocess Communication (IPC)

4.3.1 Finding Threads

In order to directly communicate with another thread it is necessary to know its thread Id. In particular, in order to communicate with a thread in another address space it is necessary to know thread's global thread Id. Unfortunately, when a thread is started it does not know of any other thread (or thread Id), and therefore cannot initiate any communication.

The only function that L4 provides for finding another thread is `L4_Pager()`, which returns the thread ID of the calling thread's pager. Other ways of finding threads include: to receive a thread Id from another thread (e.g., a thread's parent, a naming service, etc.), to read it in from persistent storage (e.g., a file stored on disk), to remember it after creating a thread (e.g., save a child's thread Id in a local variable or table), or to read it from a known location in memory (for example, a parent thread may store ThreadIds on a child's stack before starting the child).

Note that σ_0 ("sigma-zero") is the root thread's pager. Therefore, calling `L4_Pager()` in the root thread will return σ_0's thread ID. L4 also provides the `L4_Myself()` function which returns the calling thread's thread Id.

4.3.2 Sending Messages

Before being sent, a message must first be constructed (i.e., the headers and body must be filled in) and placed in the message registers. A message consists of a header and a body. The following subsection shows how a message may be constructed manually. Normally this is not necessary, as the L4 interface generator, IDL^4 (discussed in Chapter 5) can do this work.

Message Header

The message header (also called a message tag or `L4_MsgTag`) consists of a label, message flags, and two fields specifying the size of data in the body (see Figure 4.1). The label identifies the message type. Neither labels nor label formats are specified by L4, so it is up to the programmer to choose a labelling scheme for messages. The semantics of the message flags are specified in the L4 Reference Manual, however, they are not often used and are usually set to 0. The size field *t* specifies the number of typed items in the body, and *u* specifies the number of untyped words.

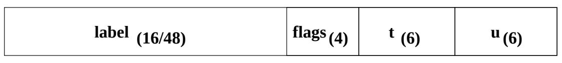


Figure 4.1: The fields of a message header. Size in bits is included in parentheses. Where two values are shown the first is the size on 32-bit architectures and the second is the size on 64-bit architectures.

The following code can be used to set the values of a message header.

```
#define LABEL 0x1

L4_Msg_t msg;

L4_MsgClear(&msg);
L4_Set_MsgLabel(&msg, LABEL);
```

Note that before any values are entered into the header, the whole message is first cleared using `L4_MsgClear()`. After this the label is set using `L4_Set_MsgLabel()`. The message flags do not have to be set unless they differ from 0. It is also not necessary to set *u* and *t* explicitly, as they will be set appropriately when data is added to the message.

Message Body

The message body contains of between zero and 63/31 data words. The data can be in the form of typed items and untyped words. A typed item can be a `L4_MapItem` or `L4_GrantItem`. These are used to map memory and grant memory respectively. Untyped words are placed in the body first followed by the typed items.

The following code shows how untyped data is added to a message.

```
L4_Word_t data1;
L4_Word_t data2;
...
L4_MsgAppendWord(&msg, data1);
L4_MsgAppendWord(&msg, data2);
```

The following code shows how a typed item (an `L4_MapItem`) is added to a message.

```
L4_MapItem_t map;
...
L4_MsgAppendMapItem(&msg, map);
```

`L4_GrantItems` are added in a similar way using `L4_MsgAppendGrantItem()` .

Note that the above code does not directly write into the message registers, it simply stores all the header and content data in an `L4_Msg_t` data structure. Thus, before a message can be sent, the data must first be loaded into the registers. This is done as follows.

```
L4_MsgLoad(&msg);
```

Send

Once the message registers have been loaded the message can be sent. This is done as follows.

```
L4_ThreadId_t dest_tid;
L4_MsgTag_t tag;
        ...
tag = L4_Send(dest_tid);
```

The `L4_Send()` function takes the destination thread Id as a parameter and returns an `L4_MsgTag_t`, which is used to signal an error. Note that the `L4_Send()` function does not take an `L4_Msg_t` as a parameter. It sends the message previously loaded into the *MR* register set. Note that `L4_Send()` is blocking. This means that an `L4_Send()` call will block until the message has been successfully received by the intended recipient.

Error Handling

If an error occurred, causing the send to fail, the fourth bit of the returned message tag's message flag field will be set to 1. Error details can be retrieved from the appropriate *ErrorCode* TCR. To make error checking easier, the L4 API provides convenience functions to check the error bit and retrieve the error code. These functions can be used as follows.

```
L4_Word_t error;
        ...
if (L4_IpcFailed(tag)) {
  error = L4_ErrorCode();
}
```

The L4 API also provides a similar `L4_IpcSucceeded()` function. Please see the L4 Reference Manual for details about possible error values.

Call, Reply, ReplyWait

Besides `L4_Send()`, there are three other ways to send a message. The first is `L4_Call()`, which sends a message and waits for a reply from the receiver. The second is `L4_Reply()` which is used to send a reply message (i.e., when the sender sent a message using `L4_Call()`). The third is to use `L4_ReplyWait()`, which is similar to `L4_Call()` in that it sends a message and waits for a reply, except that in this case, it sends a reply and waits for a new incoming message from any thread (see the discussion of `L4_Wait()` below).

Examples of these three approaches follow.

In the first example, showing `L4_Call()`, a message is sent to a given destination thread (specified by `dest_id`), after which the sender waits for a reply from that same thread. The return value is used to indicate an error on failure as above, however, on success it will contain the header (i.e., `L4_MsgTag`) of the received message

```
L4_ThreadId_t dest_tid;
L4_MsgTag_t tag;
        ...
tag = L4_Call(dest_tid);
```

In the second example, showing `L4_Reply()`, the return value and parameters are the same as for `L4_Send()`.

```
L4_ThreadId_t dest_tid;
L4_MsgTag_t tag;
        ...
tag = L4_Reply(dest_tid);
```

Finally, in the third example, showing `L4_ReplyWait()`, the second parameter (`src_tid`) is used to return the thread Id of the sender of the new message. Unlike `L4_Call()` this might not be the same as the thread that the message was sent to. As with `L4_Call()`, however, the return value is used to indicate an error on failure and will contain the received message's header on success.

```
L4_ThreadId_t dest_tid;
L4_ThreadId_t src_tid;
L4_MsgTag_t tag;
        ...
tag = L4_ReplyWait(dest_tid, &src_tid);
```

There are variations of these functions that take extra timeout parameters. Please see the L4 Reference Manual for more details.

4.3.3 Receiving Messages

Before receiving any messages, a thread must specify the kinds of messages that it is willing to accept, as well as any receive windows and string buffers, (both of which will be discussed below). This is done by setting values in the *acceptor*, which is stored in the BR_0 buffer register. Note that since the buffer registers are static (i.e., they don't change unless explicitly changed), it is sufficient for a thread to set BR_0 (the acceptor) only once. For example:

```
L4_Accept(L4_UntypedWordsAcceptor);
```

This specifies that the thread is willing to accept messages containing only untyped words. No typed items will be accepted.

Other possible parameters for `L4_Accept()` include `L4_AsynchItemsAcceptor()` and `L4_MapGrantItems()`. The use of these will be discussed later in Section 4.4. The L4 API defines several other convenience functions to modify the values in the acceptor. Please see the L4 Reference Manual for more details.

Wait

Wait is used to receive a message from any sender, either in the same address space or a remote one. The following code shows how a message is received using `L4_Wait()`.

```
L4_MsgTag_t tag;
L4_ThreadId_t tid;
      ...
tag = L4_Wait(&tid);
```

`L4_Wait()` blocks until a message is received. The thread Id of the sender is returned through the first parameter, while the header of the received message is passed back in the return value. When `L4_Wait()` fails, the return value is used to indicate an error and the error details can be extracted as discussed earlier in Section 4.3.2. `L4_Wait()` can fail only if it timed out, or was cancelled by another thread.

Call, ReplyWait, Receive

Similar to the sending of messages there are several alternative ways to receive a message. As mentioned above, `L4_Call()` and `L4_ReplyWait()` involve both a sending and a receiving phase. Furthermore, the `L4_Receive()` function allows a receiver to wait for a message from a specific thread. This function takes a thread Id as a parameter and blocks until a message from that thread is received. Its use is shown in the following example.

```
L4_MsgTag_t tag;
L4_ThreadId_t src_tid;
      ...
tag = L4_Receive(src_tid);
```

When L4_Receive() fails, the return value is used to indicate an error and the error details can be extracted as discussed earlier in Section 4.3.2. L4_Receive() can fail if it timed out, was cancelled by another thread, or the given thread does not exist.

As with the send functions, there are also variations of these functions that take extra timeout parameters. Please see the L4 Reference Manual for more details.

Message Header

When a thread receives a message it should first check the label to see what kind of message it is. Based on its knowledge of what the body of that type of message should contain, it can then unpack the data from the message.

Here is example code showing a thread receiving a message and extracting the label:

```
L4_Msg_t msg;
L4_MsgTag_t tag;
L4_ThreadId_t tid;
L4Word_t label;
      ...
tag = L4_Wait(&tid);

/* Load the message from the MRs */
L4_MsgStore(tag, &msg);

label = L4_Label(tag);
```

In order to access the message data it is necessary to copy the data from the message registers into an L4_Msg_t data structure using L4_Msg_Store(). Once the message has been copied out of the registers the label is extracted from the header using L4_Label().

Message Body

Once a message has been received and the receiving thread knows what kind of message it is (based on the value of the label), it can begin to extract data from the message body. We first look at extracting untyped words. One way to do this is to explicitly extract each individual untyped word. For example:

```
L4_Word_t u;
L4_Word_t data1;
L4_Word_t data2;
      ...
u = L4_UntypedWords(tag);
assert(u >= 2);

/* Extract word 0 then word 1 */
data1 = L4_MsgWord(&msg, 0);
data2 = L4_MsgWord(&msg, 1);
```

The second parameter of L4_MsgWord() specifies the word to extract, 0 being the first data word, 1 being the second, and so on. Note that if the second parameter specifies an invalid word (i.e., >= u) then invalid data is returned or a pagefault may be generated. For this reason it is advisable to ensure that only correct parameters are passed to the function. In the example this is acheived using an assert statement (`assert(u >= 2);`).

Another approach is to extract the untyped words all at once into an array. For example:

```
L4_Word_t data[2];
      ...
u = L4_UntypedWords(tag);
assert(u == 2);

/* Extract both words at once */
L4_MsgGet(msg, data, 0);
```

The last parameter of `L4_MsgGet()` can be a pointer to an array of typed items, in which case the typed items will also be extracted from the message.

Typed items can also be extracted individually as follows.

```
L4_Word_t t;
L4_MapItem_t map;
      ...
t = L4_TypedWords(tag);

/* Extract first typed item */
L4_MsgGetMapItem(msg, 0, &map);
```

Extracting an `L4_GrantItem` proceeds in a similar fashion. Note that it is usually not necessary to extract typed items. As we will discuss later, simply successfully sending[3] `L4_MapItem` or `L4_GrantItem` is enough to map or grant a page.

4.3.4 Reusing Messages

Message headers and bodies can be reused. This is often useful when forwarding a message on to another receiver, or when replying to a message. In the first case it is sufficient to simply reload the message and send it off again. In the second case the receiver might replace the data, keeping the header (i.e., the label) intact. Alternatively the receiver may keep the data intact and simply replace the label. Note that in all cases the message contents should be reloaded into the message registers before invoking a send function. This is to allow for compiler optimisations (see below).

4.3.5 Optimisation

The process of loading and accessing message registers shown in the preceding sections is somewhat non-optimal as it first stores the message data in memory, then copies it into the registers, and at the receiving end copies them from registers back into memory. This somewhat defeats the purpose of having a highly-optimised IPC mechanism that attempts to transfer as many messages in registers as possible.

A highly optimising compiler should be able to optimise this overhead away, and transfer the message directly to and from registers. However, many widely-used compilers (in particular, gcc on RISC or EPIC architectures) are presently not up to this task, and therefore will produce rather

[3]Note that strictly speaking, it is the act of receiving a `L4_MapItem` or `L4_GrantItem` that causes the memory to be mapped. However, successfully sending a message implies that the message was also successfully received

non-optimal code. It is not unheard off that the cost of moving data around in user space exceeds the cost of the actual L4 IPC system call.

An alternative approach would be to use hand-optimised assembly stubs around the system call, which are adapted to the parameters of the specific IPC operation. This may be reasonable approach for a system that uses raw L4 IPC in only a few places, and otherwise uses higher-level abstractions that ultimately map to L4 IPC. However, this is not a very likely scenario.

... Insert info on Magpie ...

4.4 Sharing Memory — Memory Mapping

In order for threads in separate address spaces to access the same memory, that memory must be mapped into all of those address spaces. Mapping is achieved by sending a message containing an `L4_MapItem` or `L4_GrantItem`. Upon receipt of such a message the receiver will have a region of the sender's address space mapped into its own address space.

4.4.1 Flexpages

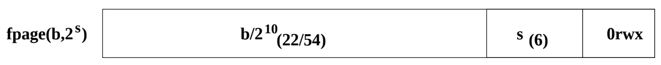


Figure 4.2: The flex page layout

In order to create an `L4_MapItem` it is necessary to first define a *flex page* (*fpage*). An fpage is a generalisation of a hardware page. It represents a region of memory, specifying a start address, a size, and access bits (see Figure 4.2). Similar to hardware pages, there are restrictions on the size of fpages. Specifically, the size of an fpage must be:

- a power of 2
- at least 1024 bytes
- greater than or equal to the smallest hardware page size.

Note also, that, like a hardware page, the fpage must be aligned to its own size. For fpages that are equal to a hardware page size this is not a problem. For larger fpages it is up to the programmer to make sure the fpage is aligned properly. Most useful are fpages that correspond to actual hardware pages, in which case they allow the user to specify to the kernel which super-page size to use. However, the implementation may not support all hardware page sizes. Fpages which do not correspond to supported hardware page sizes are treated as a range of individual pages.

In order to share the region of memory specified by an fpage with other threads, the fpage must be put into an `L4_MapItem` and that `L4_MapItem` must be sent to those threads. Upon receiving the message, the region specified by the fpage is mapped into the receiver's address space.

4.4.2 Sharing Memory the Right Way

The simplest way to communicate using shared memory is for the communicating threads to agree beforehand on a shared memory region that will be reserved specifically for communication. Agreeing on the location and size of this shared memory area makes setting up the shared memory easier. Also, storing only shared data in this region protects the threads' internal state from unauthorised access or unintentional modification. An example of this approach is shown in Figure 4.3.

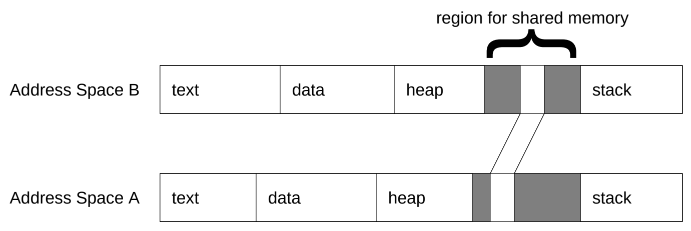


Figure 4.3: An example of sharing a predetermined region of memory

4.4.3 Sharing Memory the Wrong Way

A different approach is to directly share the region of memory that contains the data to be shared (e.g., share the page(s) containing a particular variable, array, etc.). This can be done in two ways. The first is to define an fpage that contains the shared data region. The problem with this approach is that if the data crosses page boundaries the final (size aligned) fpage may be many times larger than the actual shared data. An example of this problem is shown in Figure 4.4; here the smallest fpage fully containing a sub-page-sized object is four times the size of the base hardware page size.

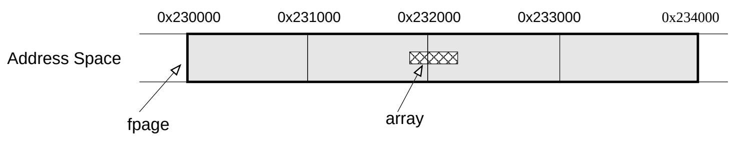


Figure 4.4: An example of making a shared array available using a single fpage. Note that the resulting fpage is many times larger than the actual shared data

The second approach is to define several contiguous fpages that together contain the desired shared data. These are then mapped using separate map items (which may be sent in the same message). An example of this approach is shown in Figure 4.5.

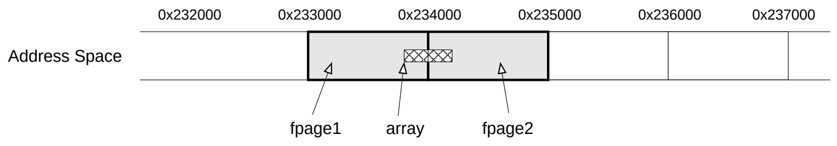


Figure 4.5: An example of making a shared array available using multiple fpages

As mentioned, a major problem with the first approach is the possible ballooning of fpage size. While the second approach does not suffer from this problem, both approaches might unintentionally share more than just the desired data region. For example, any data on the sharer's side that falls on the same page as the shared data will also be made available to the sharee. The sharee can then (purposefully or accidentally) read and overwrite data that was not meant to be shared. An example of this is shown in Figure 4.6.

4.4.4 Mapping Details

Sharer

The following example code shows how an fpage is defined.

```
L4_Fpage_t fpage;
```

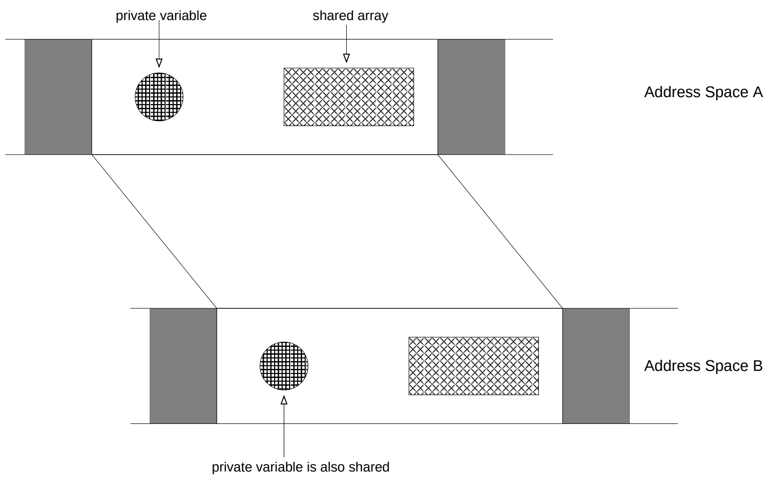


Figure 4.6: An example of unintentional sharing of data

```
void *addr = SHARED_REGION;
int size = PAGE_SIZE;

fpage = L4_Fpage(addr, size);
L4_Set_Rights(&fpage, L4_FullyAccessible);
```

In this code the fpage is created using `L4_Fpage()` and both the fpage's starting address and size have been predefined by the programmer. Note that besides creating the fpage the code also sets the fpage's access permissions. The L4 API provides several convenience constants for various combinations of access permissions.

Next, a map item is created as follows:

```
L4_MapItem_t map;
L4_Word_t base;

base = (L4_Word_t)addr;
map = L4_MapItem(fpage, base);
```

The `base` variable determines how the fpage will be mapped onto the receive window, its use is explained in Section 4.4.5 below. Once an `L4_MapItem` has been created it is added to a message and sent to the sharee.

```
L4_Msg_t msg;
L4_MsgTag_t tag;

L4_MsgClear(&msg);
L4_Set_MsgLabel(&msg, LABEL);
L4_MsgAppendMapItem(&msg, map);
L4_MsgLoad(&msg);

/* send message */
tag = L4_Send(tid);
```

Sharee

In order to receive a mapping the sharee must specify a receive window, i.e. an area of the sharee's local memory that the shared memory will be mapped into. A receive window is specified by copying an fpage structure into the acceptor (BR_0). The L4 API provides convenience functions for this: `L4_Accept()`, and `L4_MapGrantItems()`. They are used as follows:

```
#define RCVWIND 0x035000
#define RCVWIND_SIZE PAGE_SIZE

L4_Acceptor_t acceptor;
L4_Fpage_t rcv_fpage;

rcv_fpage = L4_Fpage(RCVWIND, RCVWIND_SIZE);
acceptor = L4_MapGrantItems(rcv_fpage);
L4_Accept(acceptor);
```

The address and size of the receive window (`RCVWIND` and `RCVWIND_SIZE`) can be predetermined by the programmer or generated at run time by the program.

It is also possible for a thread to open up its whole address space to be a receive window. The following code shows how to do this.

```
  L4_Accept(L4_MapGrantItems(L4_CompleteAddressSpace));
```

Note that this should only be done in the rare occasion that the receiver fully trusts the mapper; a typical instance of this is where the sender is the receiver's pager, in which case the kernel sets up full-size receive window on the faulting thread's behalf.

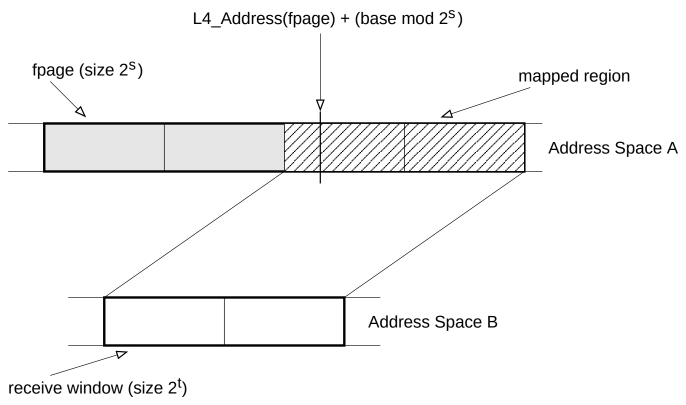
4.4.5 Resolving Mapping Ambiguities: The `base` Parameter

If the fpage is larger than the receive window, `base` indicates which part of the fpage will be mapped to the mappee. If the receive window is larger than the fpage, the base indicates where in the receive window the fpage is mapped. A precise description of how the value of `base` is used to determine this is given in the L4 Reference Manual. Here we provide a more informal explanation.

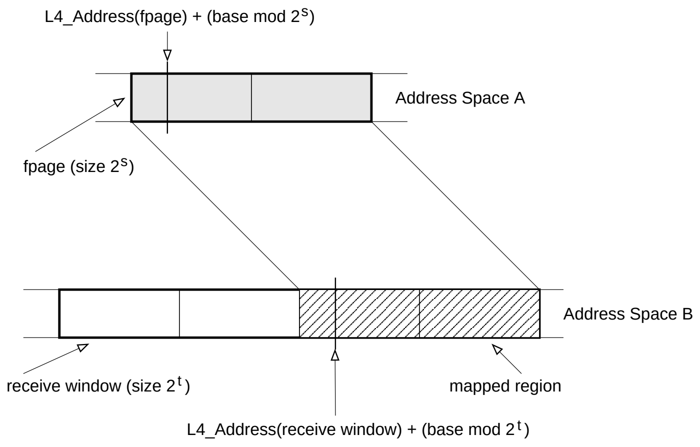
Figure 4.7 shows an example of the two situations (mapping an fpage to a smaller receive window and mapping an fpage to a larger receive window). In order to understand how the `base` parameter is used, it is best to imagine the larger of the two fpages (the receive window is also specified by an fpage) as being *tiled* with instances of the smaller fpage. The `base` parameter (modulo the larger fpage size) identifies one of these tiles, which is the part of the larger fpage that corresponds to the smaller fpage in each specific case. The actual mapping occurs between the tile identified by `base` and the full smaller fpage.

From the above explanation it can be seen that the `base` parameter is completely irrelevant if the send and receive fpages are of the same size, as there is no ambiguity to resolve. Assume now that the two fpages differ in size, say the larger one is of size 2^s and the smaller one of size 2^t, with $s > t$. Then the bits of `base` which determine the offset within the smaller fpage, i.e. the least significant t bits of `base`, are also irrelevant, as they are not needed to identify the specific tile. Furthermore, as `base` is only used modulo the larger fpage size, the most significant $N - s$ bits of `base`, N being the word size of the machine, are also irrelevant. The only bits of `base` which are of any relevance are bits $s - 1 \cdots t$ (i.e., none if $s = t$).

All this can be ignored (and an arbitrary value supplied for `base`) if the two fpages are of the same size. However, because the sender may not know the size of the receive window provided



(a) An example of an fpage being mapped onto a smaller window


L4_Address(fpage) + (base mod 2^s)
Address Space A
fpage (size 2^s)
Address Space B
receive window (size 2^t)
mapped region
L4_Address(receive window) + (base mod 2^t)

(b) An example of an fpage being mapped into a larger window

Figure 4.7: An example of an fpage being mapped onto a smaller window, and an fpage being mapped into a larger window

by the receiver, ignoring `base` is not always possible. In fact, a common case is that the receiver specifies a large area, or even the full address space, as its receive window (the latter is what happens on a page fault). The page fault scenario actually provides the simplest way of attaching a meaning to `base`: it is the address that triggered the page fault. In this case the sender is the receiver's pager, and is expected to provide a mapping that resolves the page fault. Rather than sending a single page that would suffice to restart the faulting thread, the pager may choose to supply a larger amount of virtual memory, in order to reduce the number of future page faults. In this case the pager uses the fault address as `base`, to identify which part of the fpage should cover the address where the fault happened.

4.4.6 Granting Details

Granting of pages is similar to mapping (the main differences being the use of `L4_GrantItem` and `L4_MsgAppendGrantItem()`. The following code shows how to create and send an `L4_GrantItem`.

```
L4_Msg_t msg;
L4_MsgTag_t tag;
L4_GrantItem_t grant;
L4_Word_t base;
L4_Fpage_t fpage;
void *addr = SHARED_REGION;
int size = PAGE_SIZE;

/* prepare grant item */
fpage = L4_Fpage(addr, size);
L4_Set_Rights(&fpage, L4_FullyAccessible);
base = (L4_Word_t)addr - L4_Address(fpage);
grant = L4_GrantItem (fpage, base);

L4_MsgClear(&msg);
L4_Set_MsgLabel(&msg, LABEL);
L4_MsgAppendGrantItem(&msg, grant);
L4_MsgLoad(&msg);

/* send message */
tag = L4_Send(tid);
```

The code for receiving a grant item is the same as that for receiving a map item.

4.4.7 Unmapping

In order to revoke access to memory that has previously been mapped, that memory must be unmapped by the mapper. This is done using `L4_UnmapFpage()` as follows:

```
L4_UnmapFpage(fpage);
```

A similar function, `L4_UnmapFpages()`, supports unmapping of several fpages in a single system call.

Note that, because the granter loses ownership of the page as part of the grant process, granted pages cannot be unmapped by the granter.

4.5 Interrupts

4.5.1 Interrupt Handler

4.5.2 Registering for Interrupts

4.5.3 Handling Interrupts

4.5.4 Turning Interrupts On and Off

4.6 Exceptions

4.6.1 Exception Handler

4.6.2 Registering for Exceptions

4.6.3 Handling Exceptions

4.6.4 Turning Exceptions On and Off

Chapter 5

Recipes and Best Practice

Chapter 6

User Environments

6.1 L4e and Kenge

6.2 Iguana

Chapter 7

L4 Internals

7.1 Thread states

When debugging an L4 based systems it is useful to know about the various states a thread may be in, and the state transitions. Figure Figure 7.1 shows this information.

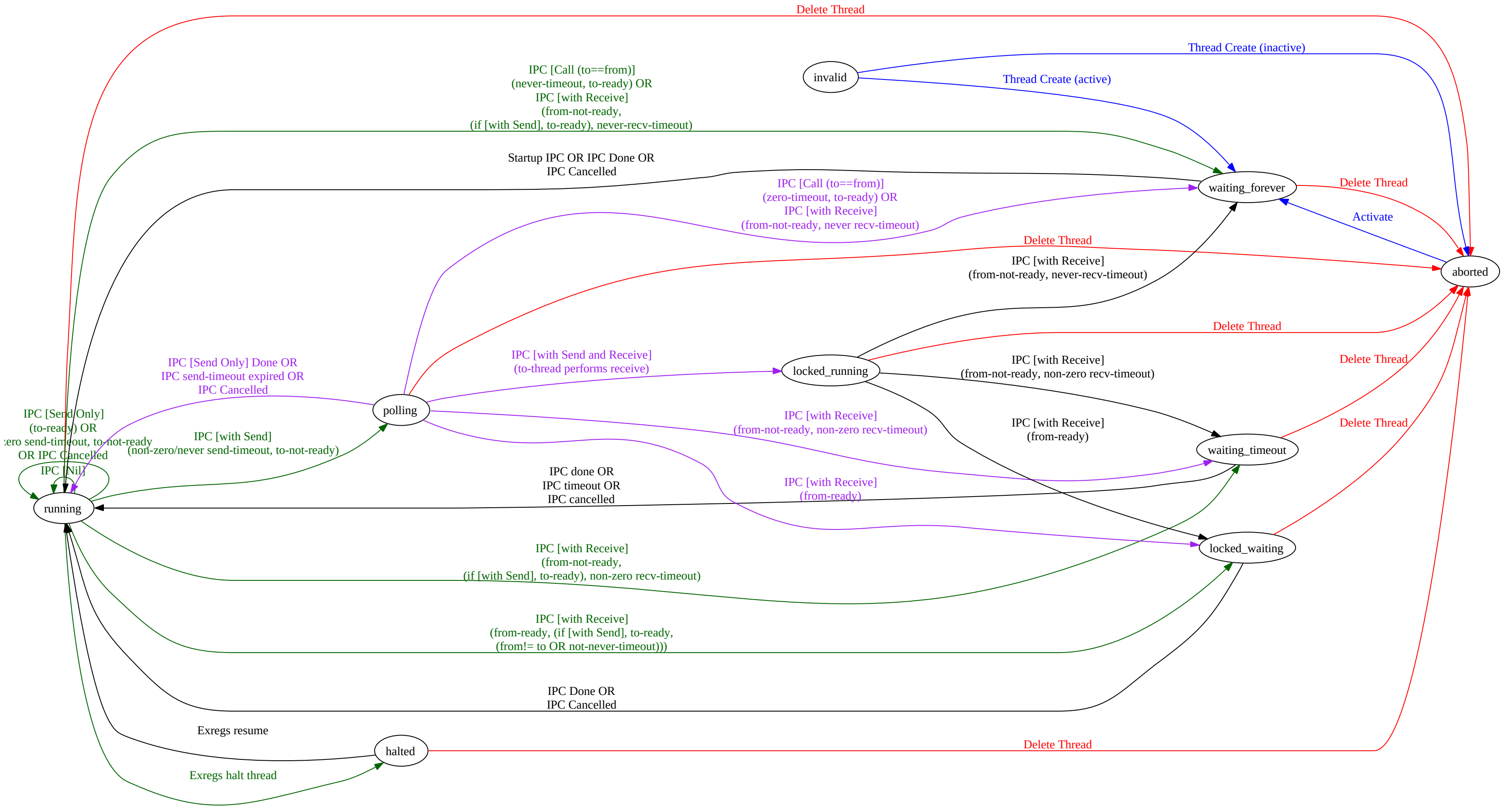


Figure 7.1: Thread states and transitions

7.2 Mapping database

L4 supports recursive mappings of flex-pages between address spaces. The L4 mapping database (MDB) is the kernel data structure that represents and tracks these recursive mappings.

Conceptually each mapping consists of a tree of address spaces, however this tree is implemented as a linked list.

Chapter 8

Debugging

Chapter 9

Not Implemented

At the time of writing, the kernel does maintain page access bits in software. This means that the output parameters of the `L4_UnmapFpage()` system call are only delivered on architectures where those bits are hardware-maintained (IA32). This is not likely to change in the near future.

Chapter 10

Ports of L4

Appendix A

An Example

Appendix B

Compiling and Linking

Appendix C

Glossary

Bibliography

[AH98] Alan Au and Gernot Heiser. L4 User Manual — version 1.0. Technical Report UNSW-CSE-TR-9801, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, April 1998. Latest version available from `http://www.disy.cse.unsw.edu.au/Software/L4`. iii

[BH70] Per Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13:238–250, 1970. 1

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 120–133, Asheville, NC, USA, December 1993. 2

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 66–77, St. Malo, France, October 1997. 2

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–88, Asheville, NC, USA, December 1993. 2

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995. 2

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996. 2

[NIC05] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N1*, October 2005. `http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf`. iii, 2, 15

[RTY+88] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, C-37:896–908, 1988. 1

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17:337–345, 1974. 1