Inside L4/MIPS

Anatomy of a High-Performance Microkernel

Gernot Heiser

Version 2.19 (5 syscalls+scheduling) of January 30, 2001

disy@cse.unsw.edu.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

Abstract

This document is an attempt to document the internal structure of L4 and its operations. It is based on the L4 implementation for the MIPS R4x00 (L4/MIPS), kernel version 79 (February 1999). The document is meant as an aid in teaching operating systems internals, and as a guide for kernel implementors. While the actual code discussed is very specific to the MIPS processor, much of the overall structure and logic of L4 is quite uniform across platforms.

The present version of this report documents L4/MIPS data structures, exception handling and the IPC system call. Documentation of the implementation of the other system calls, and issues such as scheduling, will be added in the near future.

Acknowledgments

This report would not have been possible without Jochen Liedtke, who created L4, and Kevin Elphinstone, who did an outstanding job in designing and implementing the MIPS version. Special thanks are due to both.

Thanks are also due to a number of other people who helped to get L4 into what is described here. Alan Au added a number of missing components to the kernel, and fixed a few bugs. UNSW undergraduate thesis students Brett Nash, Victoria Ong, Cristan Szmajda and Adam Wiggins studied the code, suggested a number of improvements, and helped with the documentation. Students taking Advanced Operating Systems at UNSW in 1997–1999 exercised the kernel and contributed to its stability.

Contents

List of Code Listings

List of Figures

List of Tables

List of Bugs and Restrictions

Chapter 1

Introduction

L4/MIPS is the implementation of the L4 microkernel [Lie93, Lie95] on R4000 series CPUs [R4695]. The R4600/R4700 is an implementations of the MIPS R4x00 architecture [KH92] aimed at embedded applications. The L4/MIPS API is based on the ix86 version of L4 [Lie96a] as of mid-1996 (“Version 2”) with a few minor extensions (compatible to “Version 4”).

L4/MIPS was designed by Kevin Elphinstone, then a PhD student at UNSW, and Jochen Liedtke, then a researcher at GMD, Germany, in 1995–6. It was mostly implemented by Kevin Elphinstone between mid 1995 and mid 1997. Improvements were since made by Kevin Elphinstone and Alan Au, the latter also a PhD student at UNSW. This document is based on L4/MIPS kernel version 79 [L4M99], which was released in February 1999.

L4/MIPS is now in regular use at UNSW for research and teaching. The Mungi single-address-space operating system [HEV+98] has been implemented on top if it, Linux has been ported to it [HHL+97], and the kernel is used in teaching *Advanced Operating Systems*.

This report is an attempt to document the internal operations of the kernel, and to shed some light on what “makes it tick” and where it gets its performance from.

1.1 Intended Audience

This document is written to serve two purposes:

- to be used in operating systems teaching.

  L4/MIPS has been used as a platform on which students built operating systems in UNSW’s COMP9242 *Advanced Operating Systems* course since August 1997. Since 1999 the course took a closer look at microkernel implementation issues, and the first version of this document grew out of that. It is planned to expand the coverage of microkernel internals in the future, based on a more complete version of this document. Arizona State University is also planning to use this report as an aid in teaching an operating systems internals course.

- as an aid to future kernel implementors.

  L4 implementations are highly optimised, contain lots of assembler code, and tend to employ plenty of architecture-specific tricks. However, the general structure, and the “course-grain logic” is remarkably similar across platforms. An in-depth study of L4 source code for one particular architecture will therefore provide a good guide on how to approach a re-implementation on a different platform.

```
        mtc0    t0, C0_STATUS
        dli     a6, 0
        j       k_ipc
        move    a5, a4
3:      lui     a0, KERNEL_BASE
        ld      a0, K_GPT_POINTER(a0)
        move    a1, s3
        and     a2, a0, a4
```

Listing 1.1: Sample code listing.

In both of these aspects, this report aspires to follow the example of the famous *Lions Book* [Lio77, Lio96], also from UNSW, without pretending to rival the Commentary's impact and significance.

This report, as well as the source code it describes, is available from the UNSW L4 web site.[1] That site also contains the latest releases of L4 for various architectures, associated documentations and forms to register for mailing lists with L4-related announcements. Feedback on this report is highly welcomed and should be directed to `l4.inside@cse.unsw.edu.au`.

1.2 Why MIPS?

Why did we go through the trouble of writing this report on L4 for the MIPS architecture, when it is more than doubtful whether this architecture has any future?

The answer comes in two parts:

1. For the purposes stated in the previous section, it doesn't matter much. For teaching microkernel internals the specific platform it is of little relevance. Teaching use is also supported by a R4x00 simulator which can run L4, and thus provides independence from actually available hardware. ***Real Soon Now...*** Similarly, as an aid for future kernel builders the MIPS source as relevant as any other.

2. The MIPS implementation is very clean, certainly the most readable one available at this time. It is about half assembler, half C, but even the assembler code is mostly very readable and not too hard to understand. The same cannot be said about other implementations.

And, of course, it helped that the code was developed locally, and as a result a number of people were familiar with at least parts of it, and could therefore answer questions.

1.3 Conventions

Most of this report consists of verbatim source code listings and associated annotations. Source code is shown in displays called "Listings", such as the example Listing 1.1. Wherever feasible, listings and annotations appear on the same or on opposite pages. To make this possible the listings have been condensed as much as possible, by eliminating empty lines, instrumentation code, compiler directives, and even the (rare) comments.

The listing shows the format of MIPS assembler instructions. In general an instruction consists of an opcode and up to three operands. The first operand is **always** the destination register. The second operand is normally the source register. The third operand is a second input to the instruction, and can be a register or an immediate value. If a three operand instruction is written with two operands only, then the first operand represents both source and destination registers. In the case of a store instruction, the first ("destination") operand is the register to be saved.

[1]The URL of the UNSW L4 web site is `http://www.cse.unsw.edu.au/~disy/L4/`.

Assemblers on RISC architectures such as the MIPS generally do a good job of scheduling instructions into load and branch delay slots. For some of the tightest system code, however, the “DIY” approach is still better. Consequently, much of the critical kernel code is covered by `.set noreorder` directives, which prevent the assembler from changing the order of instructions. To avoid cluttering up the listings with these directives, any code for which a `noreorder` directives is in force, is shown in the listings with **bold** line numbers. In Listing 1.1 this is the case for Lines 0–3. Here, Line 3 is in Line 2’s branch delay slot; logically Line 3 precedes Line 2.

In the reminder of this document we will make an attempt to point out (often non-obvious) implementation decisions/simplifications as follows:

Implementation choice: This highlights an implementation choice.

Any known bugs in the exhibited code will be indicated like this:

Bug/Restriction 0: Sample bug.
This is how bugs are shown.

A bug is code which causes the kernel to operate in a way that is inconsistent with the specification. This includes unimplemented features.

There are other shortcomings of the implementation, for example affecting efficiency, which are not bugs as they do not affect correctness. These are highlighted like this:

Implementation criticism: This criticises the implementation.

Note that these shortcomings are often “quick-and-dirty hacks” which were put in place to get things going, and were never removed because there was not enough pressure for having them removed. Much of this is actually nitpicking. The frequent use of *implementation criticisms* should not create an impression that the L4/MIPS implementation is highly deficient. In fact, it’s great code!

Text in ***bold italics*** indicates something that needs fixing up in this document.

1.4 Structure

Chapter 2 provides background information required for the understanding of the remainder of the report. There are two parts to this: L4 in general, and the MIPS R4x00 architecture. The description of L4 is kept extremely brief, for the simple reason that we consider it essential to have the L4 Reference Manual [EHL99] handy when reading this document. Hence it makes little sense to repeat much of what is said in the Reference. The R4x00 architecture description is much more detailed, and, hopefully, sufficient for the understanding of the code.

Chapter 3 provides the introduction to the L4 code. It contains a short overview of the structure of the source code, and then presents and explains the main kernel data structures and their use.

The remaining chapters consist mostly of annotated source code listings. Chapter 4 presents the exception handling code, Chapter 5 the IPC code, and Chapter 6 the code implementing the other system calls.

Chapter 2

Background

This chapter provides some background information on L4 and the MIPS R4x00 architecture.

2.1 L4 Overview

Here we give a quick summary of the main features of L4. However, the reminder of this document assumes a fair amount of familiarity with L4 and its API, well beyond what is presented in this section. The reader is advised to have the L4/MIPS reference manual [EHL99] handy when reading on. A practical guide to using L4 and its various features is provided by the User Manual [AH98].

2.1.1 L4 philosophy

The design of the L4 microkernel is based on the principle of minimality: *A feature should be in the microkernel if and only if security requires that the feature be implemented in privileged mode*. The term "kernel" refers to code which executes in the hardware's privileged mode. This implies, for example, that device drivers should not be part of the kernel. While drivers have access to physical memory, and are thus part of the system's *trusted computing base*, security of the system does not depend on running device drivers in privileged mode, only on protecting the drivers from interference by untrusted code [Lie96b].

Another important principle is that *it should be possible to implement arbitrary systems on top of the microkernel*. Together with the minimality principle this leads to a requirement for a small number of powerful and orthogonal abstractions, and for a strictly *policy-free* kernel.

It is important to note that Liedtke does not claim that the present versions of L4 fully satisfies these requirements. In fact, the L4 API is still developing in an attempt to better meet its design goals.

Interestingly, efficiency is not a primary design principle. However, all features have been carefully analysed and were only included into the interface if it was clear that they could be implemented efficiently. Smallness itself is a very good basis for efficiency. The performance of modern computer systems is critically dependent on maintaining high hit rates in the CPU caches, and the kernel's cache footprint has a dramatic impact on performance [Lie95]. Small is certainly beautiful in the world of microkernels.

But that is not the full story. A careful implementation which makes the best possible use of hardware features is a major factor for kernel efficiency. A careful design of kernel data structures and algorithms is, maybe, the most important factor in making a kernel fast.

2.1.2 L4 abstractions

L4's main abstractions are:

Threads represent program execution. The CPU is multiplexed between threads, each of which has some context, including its view of hardware registers. Each thread is uniquely identified via a *thread ID* (TID).

Address spaces form the basis of protection. A thread can access data which are mapped into its address space. Address spaces are constructed by **mapping** sections of other address spaces.

Inter-process communication, based on synchronous message-passing, provides for a controlled way to communicate between address spaces. L4 IPC also serves as a **synchronisation** primitive between threads.

Address spaces and tasks

Each thread belongs to a *task*. There is a one-to-one correspondence between tasks and address spaces.[1] Creating and deleting tasks implies creating and deleting address spaces. This is done via the `task_new` system call.

Logically, the number of tasks is fixed (2048 in L4/MIPS Version 79). Hence tasks are strictly speaking not created or destroyed, but their state is changed from *inactive* to *active* or vice versa. Inactive tasks consume (almost) no resources.

Creation (activation) of a task implies creation of a fixed-size set of threads (128 in L4/MIPS Version 79). These are numbered consecutively, starting at zero. All of a task's threads, except local thread zero ("lthread zero", or l_0) are logically created executing an idle loop (but without consuming any resources). L_0 is immediately active and starts execution at a start address passed to the `task_create` syscall.

A new task's address space is initially empty, and can only be populated by mapping-in parts of other address spaces. Consequently, l_0, when it begins execution, will immediately trigger a page fault when attempting to fetch its first instruction. It is the responsibility of its *page fault handler* (pager) to provide a mapping for that page.

A task has a number of attributes:

chief or owner. For an active task, this is the task to which the thread belongs which has activated the task. Only a task's chief can activate or deactivate a task. As a side effect of deactivation, a task can be *donated* to a new chief. Task IDs are effectively capabilities conferring rights over tasks.

maximum controlled priority (MCP). This is the upper limit to which a thread of the task can influence the priority of other threads (including its own). The MCP is *not* a priority. A task's MCP is specified at the time the task is activated (and cannot be higher that the MCP of its chief).

Further attributes are (explicitly or implicitly) specified when a task is activated, but they are attributes of l_0 rather than the task's.

Threads

Threads are the active entities in L4, they are the source and destination of IPC messages. Threads have the following attributes:

task: the task (and thus address space) to which the thread belongs;

pager: the thread responsible for handling this thread's page faults. On a page fault the kernel will send a page fault IPC message to the pager, and the pager is expected to reply with a message containing a mapping for the faulting page;

[1]For that reason, tasks are not a primitive concept, and it can be argued that the task concept is redundant. Not surprisingly, it is due for removal in a future version of the L4 specification. But for the time being tasks are an essential component of L4.

excepter (exception handler)[2]: the thread responsible for handling this thread's exceptions. On an exception, the kernel will send an exception IPC message to the excepter. It is then up to the excepter to implement whatever policy it chooses for handling exceptions;

internal preempter: not implemented on L4/MIPS Version 79.

external preempter: not implemented on L4/MIPS Version 79.

scheduling priority: determines when a thread will be scheduled by the kernel. L4 implements hard priorities (values 0–255): The scheduler will select the highest-priority runnable thread, using round-robing within priority levels;

time-slice length: the value a thread's *current time-slice length* will be set to by the scheduler to the value of the thread's time-slice length when selecting the thread to run;

current priority determines preemptability of a thread. A thread may be preempted by another thread whose priority is higher than the presently running thread's current priority. The current priority of a thread may be set at scheduling time, or by time-slice donation;

current time-slice length determines the time a thread is allowed to run until preempted (unless a higher-priority thread becomes runnable). The current time slice is obtained either by scheduling or by time-slice donation.

A thread's stack pointer, program counter, excepter, internal preempter and pager can be obtained and manipulated via the `lthread_ex_regs` syscall. This call can be performed by any thread on another thread belonging to the same task. It is the means by which new threads can, for example, be activated (taken away from their logical idle loops to more productive endeavours), or moved out of harm's way (by forcing them to block on an IPC which will never succeed). An excepter can use this system call to redirect a thread to some code which saves its user state, and later restores it.

A thread's scheduling parameters (time-slice length, scheduling priority and external preempter) can be obtained and manipulated via the `thread_schedule` system call. A thread may only perform such an operation on another thread whose priority does not exceed the caller's MCP.

Time-slice donation can happen in one of two ways:

- explicitly via the `thread_switch` system call. This call donates the remainder of the caller's current time slice to a specific thread. If that thread does not exist, or is blocked, the system call becomes a "yield" operation, i.e., the caller forfeits the remainder of its present time slice, and the scheduler is invoked to select another thread to run with a new time slice;
- implicitly via IPC. IPC operations are often accompanied by a context switch from the sender to the receiver, in which case the sender's current time slice is implicitly donated to the receiver.

IPC and mapping

IPC is performed via the `ipc` system call. All L4 IPC is blocking — a message transfer only takes place when both he sender and receiver are ready for it. This implies a synchronisation between the communication partners. It also means that messages only have to copied once, no buffering of messages in the kernel is required.

An IPC system call specifies an optional send operation and an optional receive operation. This reduces the number of system calls required in many frequent situations, such as RCP (or "call": send message and wait for answer) or server-style reply-and-wait-for-next-request.

To prevent indefinite blocking, timeouts are specified for each IPC. An IPC system call has four timeout arguments:

[2]Excepters are a MIPS-specific feature. Other L4 implementations handle exceptions differently, e.g., by mirroring hardware exceptions [Lie96a].

send timeout: the maximum amount of time the caller is willing to block on the send operation (if any), measured from the time of trapping into the kernel until the partner becomes ready to receive;

receive timeout: the maximum amount of time the caller is willing to block on the receive operation (if any), measured from the time of concluding the send operation (if any, time of trapping into the kernel otherwise) until the partner becomes ready to send (if any);

send page-fault timeout: the timeout value to use for the send and receive parts of any page-fault IPC that may be required for the partner's address space during delivery of the send message (if any);

receive page-fault timeout: the timeout value to use for the send and receive parts of any page-fault IPC that may be required for the partner's address space during delivery of the receive message.

Timeouts may be specified as zero (poll partner) or infinity (block indefinitely).

IPC messages may transfer data in two ways:

By value: data is copied from sender to receiver. By-value data is supported in three forms:

- registers — up to 64 bytes (on MIPS R4x00) may be transferred in registers. This is in many cases a zero-copy operation, and is therefore highly efficient;
- direct strings — a message buffer contains data which is copied by the kernel into the receiver's message buffer. Direct strings must be word aligned;
- indirect strings — the message buffer contains an arbitrary number of pointers to data buffers, which are copied by the kernel into the corresponding buffers pointed to by the receiver's message buffer.

Direct and indirect strings are copied only once, from the sender's address space directly into the receiver's address space.

By reference: the sender can designate a range of pages in its address space, which get mapped into the receiver's address space during the IPC. The receiver must specify a window where the pages may be mapped. By-reference data transfer can happen in two ways:

- mapping — sender and receiver share the mappings after a successful IPC map operations. The sender may revoke ("flush") the mappings at any time via a `fpage_unmap` system call;
- granting — the pages are implicitly unmapped from the sender's address space, who loses any access or control over them. This operation is not reversible by the sender.

Bug/Restriction 1: No granting.
Granting is not implemented in L4/MIPS Version 79.

Address-space regions are specified via an abstraction of variable size pages, called a *flexpage* or *fpage*. An fpage is a section of virtual address space whose size is a power-of-two multiple of the smallest page size supported by the hardware, and must be aligned to its size. A single IPC operation may specify several fpages for mapping. The receiver specifies its window via a single *receive fpage*.

Clans and chiefs

Clans and chiefs are a mechanism allowing control over information flow. Tasks, via the relation to their chiefs, form a hierarchy. All tasks which are owned by the same chief form that chief's *clan*. A thread can send an IPC message only to

- a thread belonging to its chief task,
- a thread belonging to a task which is part of the same clan as the sender's task, or
- a thread belonging to a task which is (directly) part of the sender's task's clan.

All other IPC is *redirected* to the *nearest* chief. The nearest chief is

- the sender's chief, in the case of a message directed to a task not directly or indirectly inside the caller's clan,
- otherwise the task inside the caller's clan whose clan directly or indirectly contains the intended receiver.

The `id_nearest` syscall returns the ID of the "nearest" chief which would be the actual receiver of such an IPC message.

For the purpose of IPC redirection, the chief is thread l_0 of the chief task. The receiving chief is informed, via a bit in the IPC syscall's result word, that the message was redirected, and is also informed of the intended recipient. The chief has then the option of forwarding the message to the intended receiver (or the next nearest chief along the way). In order to maintain RPC semantics, the chief can send on the message using *deceiving* IPC. A deceiving IPC specifies a *virtual sender*, different from the actual sender, which will be returned to the receiver as the originator of the message. The receiver is alerted to the deceit via a bit in the IPC result code. The kernel allows a deceit only if it is *direction preserving*. Loosely speaking, a deceit is direction preserving if the actual sender is within the sequence of chiefs a message from the virtual sender to the receiver had to take.

System calls

The above mentioned seven syscalls, `ipc`, `fpage_unmap`, `task_new`, `id_nearest`, `lthread_ex_regs`, `thread_switch`, and `thread_schedule` comprise the full set of L4 system calls.

Interrupts

Interrupts are modelled as threads sending empty messages spontaneously. Each interrupt has a TID, and a single user thread can be *associated* with each interrupt, thus becoming a handler for that interrupt. If an interrupt has an associated hander, the kernel will convert an occurance of that interrupt into a message to the handler thread.

Initial address space

Mapping IPC (particularly in combination with pagers) can be used to build up an address space from others, but somewhere the recursion must bottom out. For that reason, L4 provides a (somewhat magical) *initial address space* called σ_0. It is created at system initialisation time with an address space containing a one-to-one image of physical memory (other than the parts reserved for kernel use).

L4's boot protocol also contains a notion of *initial servers*, which are started up by σ_0 once the kernel has booted. σ_0 is the pager of all initial servers. σ_0 will satisfy any faults within the range of its own initialised address space by mapping the corresponding page, and ignores faults outside its valid address space range (leading to the server becoming blocked). Pages are only mapped to one initial server, thus ensuring protection. Special protocols exist to request mappings for device regions.

Initial tasks

Tasks other than σ_0 and initial servers remain inactive and owner-less, until someone claims them. A task is claimed by performing a deactivating form of the `task_new` syscall, specifying oneself as the new chief.

2.2 Relevant Features of the MIPS R4x00 Processor

2.2.1 Target systems

The kernel code described in this document is for a uniprocessor R4600/R4700 system. There are a number of minor differences between various processors of the R4x00 family. For the purpose of kernel code, no significant differences exist between the R4600 and the R4700, and we will use the term “R4600” to represent both processor models. Similarly, the differences between the R4000 [Hei93] and the R4400 are very minimal, and we will use the term “R4000” to refer to both. For most of our purposes there is no need to distinguish between processors of the family, and we will use the generic term “R4x00” to refer to any of them.

Other related processors, such as the R5000 and the R10000 will probably run L4/MIPS without major changes. Particularly the R5000’s MMU seems to be similar enough to the R4x00 to allow the code to run virtually unchanged. However, the R5000 and R10000 are multi-issue CPUs, and no attempt has been made in the kernel to schedule instructions for multiple issue.

2.2.2 R4x00 general features

The R4x00 processor family is a 64-bit architecture which supports full compatibility with the 32-bit MIPS CPUs R2000 and R3000. This is achieved by supporting a 32-bit execution mode.

Implementation choice: 32-bit execution is not supported by L4/MIPS and is therefore not covered here.

The processor is a RISC design which issues one instruction per clock cycle. The only addressing mode is base-register plus 16-bit, signed immediate offset. Most instructions execute in a single cycle.

On the R4600, which has a 5 stage pipeline, jump and branch instructions have an additional one cycle delay, and load instructions also have a one cycle delay. On the R4000, which has a 8 stage pipeline, the branch delay is 3 cycles and the load delay 2 cycles. On all R4x00 processors, the instruction immediately following a jump or branch (the *load/branch delay slot*) is always executed while the target instruction is being fetched.

The pipeline will stall in the case of an attempted access to the result of a load before it is available. Hence scheduling instructions in a load delay slot will hide the delay but is not necessary for correct execution.

Multiplication and division instructions require between 10 and 133 cycles to complete. They leave their results in two special registers, `HI` and `LO`, and the pipeline stalls until the result is available.

The processor can be configured to operate little-endian or big-endian, and can also switch endianess between user and kernel mode.

Implementation choice: L4/MIPS uses big-endian only.

MIPS instructions support four data types: *byte* (`b`, 8 bits), *half word* (`h`, 16 bits), *word* (`w`, 32 bits), and *double word*, or *dword* (`d`, 64 bits). Load and store instructions support all four sizes, but data must be aligned to size.[3]

The processor features 32 general-purpose registers, r0–r31, all 64 bits wide. Assembler programs use symbolic names based on compilers’ usage conventions. These are summarised in Table 2.1. Register `r0` reads as zero and ignores writes. Register `r31` is implicitly used by the *jump-and-link* (`jal`) instruction.

The following register conventions are important to observe when writing kernel code:

- The `AT` register is used by the assembler to store intermediate results of pseudo-instruction macros. If used for any other purpose the appropriate instructions must be surrounded by `.set at` and `.set noat` directives to prevent interference from assembler macros.

[3]A special instruction is `lui`, which loads the specified immediate value into the second-least significant byte, zeroing the least significant byte, and sign extending.

- The `k0` and `k1` registers are used as temporary registers by the kernel's exception handlers. As various exceptions can occur in kernel mode, these must not be used except in situations where it is certain that no exceptions can occur. This means that interrupts must be disabled. It also means that no mapped addresses may be used, or it must be ensured that any virtual pages used are already mapped.

register	menmonic	convention
`r0`	`zero`	always zero
`r1`	`AT`	assembler temporary
`r2–r3`	`v0–v1`	integer function results
`r4–r11`	`a0–a7`	first eight integer function arguments
`r12–r15`	`t0–t3`	temporary (callee saved)
`r16–r23`	`s0–s7`	caller-saved
`r24–r25`	`t8–t9`	temporary (callee saved)
`r26–r27`	`k0–k1`	kernel reserved
`r28`	`gp`	global (data segment) pointer
`r29`	`sp`	stack pointer
`r30`	`s8/fp`	frame pointer (caller saved)
`r31`	`ra`	return address

Table 2.1: R4x00 general purpose register set, mnemonic names and usage conventions (for 64-bit API).

In addition to the general-purpose registers, the processor has three special purpose registers:

- the program counter register, `PC`;
- the multiplication and division result registers, `HI` and `LO`. These registers can only be accessed via special instructions, `mfhi`, `mflo`, `mthi`, and `mtlo`. The first two read the and the last two write the respective register.

MIPS assembly code general uses the format

op *dst*, *src*, *trgt*

where *dst* is the register receiving the result of the operation, and *src*, *trgt* are the operands. If only two registers are specified, the *src* is taken to be the same as *dst*. One exception to the general scheme is the *store* instructions, where *dst* designates the register whose contents are to be stored to memory.

The processor has a number of *co-processors*, which have their own register sets. Co-processor zero (CP0) is the *system coprocessor*, which contains the the memory-management unit (MMU) as well as the status register and a number of other registers relevant to exception handling. Co-processor 1 (CP1) is the optional floating-point unit (FPU). Further co-processors are optional.

The processor has separate primary instruction and data caches on chip and accessible in parallel. For the R4600 these are both 16kB big, are two-way associative and feature a 32B line size (8 instructions in the case of the I-cache). Best case cache miss penalty on a system with 80ns DRAM and no secondary cache is 13–14 cycles for the I-cache and 15–16 cycles for the D-cache.

2.2.3 Memory management unit

The MMU is part of the CP0. Virtual addresses are translated in one of two ways:

- *mapped* VM addresses are translated by the *translation lookaside buffer* (TLB),
- *unmapped* VM addresses are translated by masking out the most significant bits of the address.

Which mechanism is used depends on the address, as explained in Section 2.2.4 below. Here we summarise the translation of mapped addresses.

The R4x00 TLB is fully associative and holds 48 entries, each mapping a pair of 4kB pages. TLB entries are *tagged* with an *address-space identifier* (ASID), an 8-bit number. The TLB is *software loaded*, a translation miss raises a TLB miss exception. The miss handler uses the `tlbwi` or `tlbwr` instruction to load a TLB entry. The former instruction loads the entry indicated by the `C0_INDEX` register, while for the latter instruction the target entry is indicated by the `C0_RANDOM` register. The `C0_RANDOM` register is decremented at each instruction execution. The `C0_WIRED` register can be used to define a lower limit of `C0_RANDOM` register values, and is used to protect some TLB entries from “random” replacement.

Implementation choice: L4/MIPS only uses random replacement (except to update an existing entry) and does not wire down any entries.

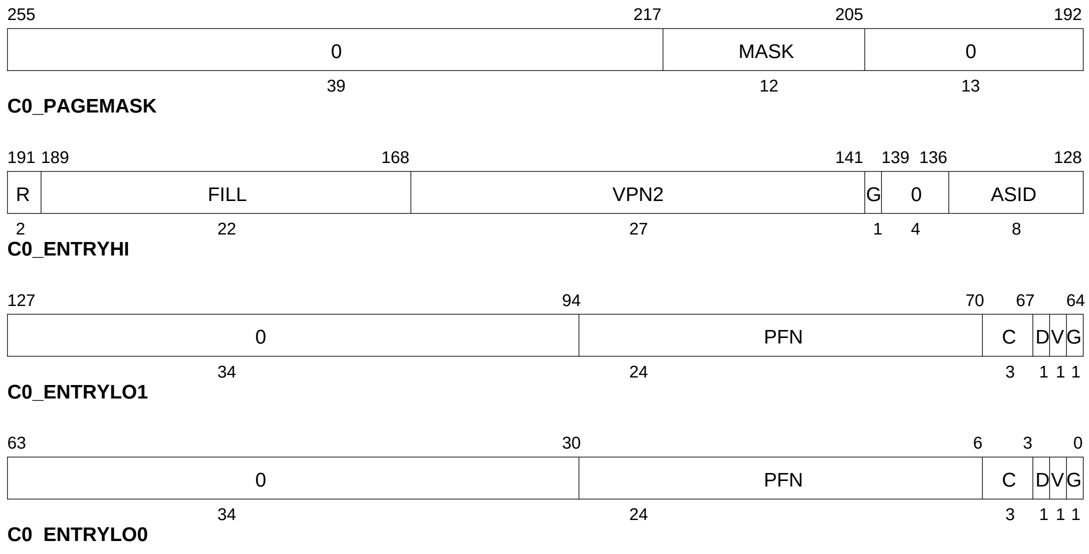


Figure 2.1: The format of R4x00 TLB entries and of the corresponding coprocessor registers.

Figure 2.1 shows the format of a TLB entry and the corresponding coprocessor registers from which the `tlbwi` or `tlbwr` load the entry. There is one minor difference between the format of a TLB entry and the coprocessor registers: The `G` (global) bit shown in the EntryHi word exists in the TLB entry but must be zero in the `C0_ENTRYHI` register. Conversely, the `G` bit in the EntryLo words is settable in the `C0_ENTRYLO0`, `C0_ENTRYLO1` registers but is zero in the TLB entry. When a TLB entry is loaded its `G` bit is set to the logical AND of the `G` bits in `C0_ENTRYLO0` and `C0_ENTRYLO1`. The ASID field of `C0_ENTRYHI` is also used during address translation to match the ASID value of a TLB entry.

The meaning of the fields are:

`MASK`: Defines the page size. Valid page sizes vary from 4kB (`MASK`=0) to 16MB (`MASK`=0xfff) in powers of 4.

Implementation criticism: L4/MIPS presently only uses the smallest page size, 4kB.

`R`: Indicates that the mapping is valid in kernel mode only (`R`=11), in kernel or supervisor mode (`R`=01) or always (`R`=00). During address translation the `R` field is matched against bits 63:62 of the virtual address (except for CKSEG addresses) which ensures that the processor is in the reight mode.

`FILL`: In `C0_ENTRYHI` must have all bits equal to the MSB of `R`. Zero in the TLB entry.

`VPN2`: Virtual page number (in units of the page size defined by the `MASK` field) divided by two.

`G`: If set the ASID is ignored when the TLB is looked up.

`ASID`: Used to distinguish mapping for the same page belonging to different processes. If the `G` is unset on a

TLB entry, it will only match if the `ASID` field matches the ASID value presently set in the `C0_ENTRYHI` register. If the TLB entry's `G` bit is set the ASID is ignored.

`PFN`: Physical frame number (in units of the page size defined by the `MASK` field).

`C`: Cacheability and cache coherency setting (see Table 2.2).

`D`: Dirty bit — iff set (and the `V` bit is set also) the page is writable.

`V`: Valid bit — iff unset a TLBL or TLBS exception occurs when accessing the page mappede by this entry.

C	Cacheability and coherency	XKPHYS starting address
0†	Cacheable, non-coherent, write-through, no write allocate	`8000 0000 0000 0000`
1†	Cacheable, non-coherent, write-through, write allocate	`8800 0000 0000 0000`
2	Uncached	`9000 0000 0000 0000`
3	Cacheable, noncoherent, write back	`9800 0000 0000 0000`
4‡	Cacheable, coherent exclusive	`A000 0000 0000 0000`
5‡	Cacheable, coherent exclusive on write	`A800 0000 0000 0000`
6‡	Cacheable, coherent update on write	`B000 0000 0000 0000`
7	Reserved	`B800 0000 0000 0000`

Table 2.2: Memory cacheability and coherency attributes on the R4x00. "C" represents the cacheability field of a TLB entry (bits 5:3 of EntryLo) for mapped accesses, or the `K0` field (bits 2:0) of the `C0_CONFIG` register, or bits 61:59 of a `XKPHYS` address for unmapped accesses. Attributes marked † are R4600 only and are unavailable ("reserved") on the R4000. Attributes marked ‡ are R4000 only and are unavailable ("reserved") on the R4600/R4700.

2.2.4 Address space layout

The MIPS R4x00 features a 64-bit address space. The processor can run in three different modes: *user*, *supervisor* and *kernel*. In each of these modes some regions of the address space are accessible while others lead to addressing exceptions. The valid address range in supervisor mode is a subset of the user mode address range, and the kernel address range is a superset of the supervisor address range. Figure 2.2 shows the address map, indicating for each address-space region the minimum privilege (kernel, supervisor or user) required.

As the figure shows, the MIPS' virtual address range is 1TB (40 bits). Address-space regions denoted as *mapped* are translated by the TLB, whereas regions denoted as *unmapped* are translated by masking out the most significant bits of the address.

The processor supports up to 64GB of physical memory, all of which can be addressed unmapped via the various `XKPHYS` segments. When using `XKPHYS` addresses, address bits 61:59 are interpreted as the *cacheability* bits. Their interpretation is the same as the corresponding bits in a TLB entry (see Figure 2.2.3). Table 2.2 summarises cacheability and coherency attributes selected by bits 61:59 of a `XKPHYS` address.

The `CKSEG` regions (top 2GB of the address space) are called *compatibility spaces*, as they correspond, via sign extension of the address, to kernel and supervisor regions in 32-bit mode.

Implementation choice: L4/MIPS does not make use of supervisor mode. The supervisor address space `XKSSEG` is used, but only as another kernel mapped area (for temporary mappings). Supervisor mode does not provide sufficient privilege to be useful inside the kernel. The design of L4 does not seem to make it necessary to use supervisor privilege outside the kernel.

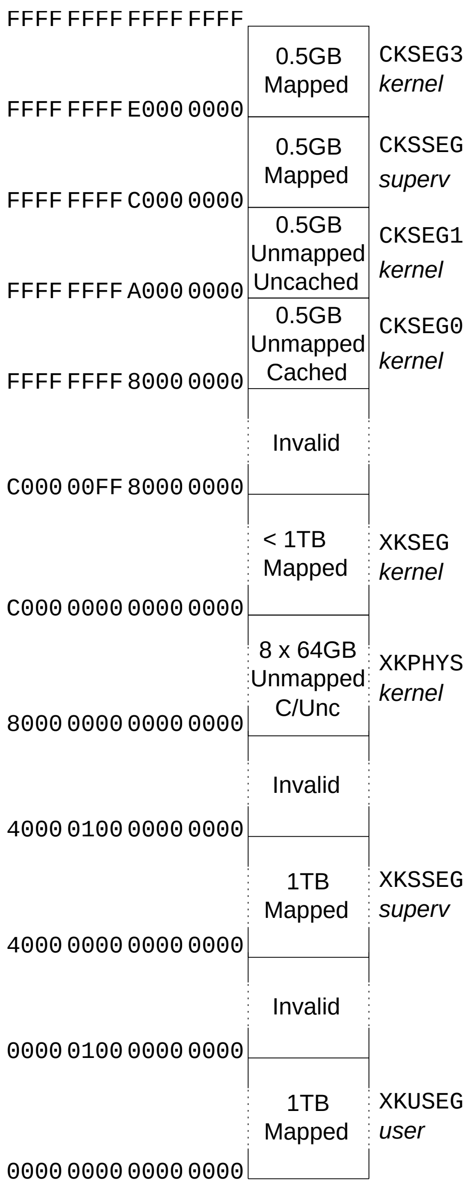


Figure 2.2: MIPS R4x00 address space.

Implementation choice: L4/MIPS only uses the compatibility spaces `CKSEG0`, `CKSEG1` in kernel mode. The reason is that these addresses allow the kernel to use 32-bit address constants, which are faster to load than 64-bit constants. In some cases, in particular, page tables, pointers are also stored in 32-bit form, resulting in more compact data structures. Furthermore, 0.5GB is more than enough memory for kernel use.

Note, however, that the kernel presently only supports 0.5GB as the maximum RAM size. Addressing based on `CKSEG0`, `CKSEG1` means that on a machine with more than 0.5GB of RAM the kernel's dynamic memory pool could no longer reside at the high end of physical memory. The necessary changes to the kernel's startup code would be straightforward.

One drawback of the use of `CKSEG0` for unmapped cached memory accesses is that there is no control over the cache coherency protocol via the address. This is not an issue on the R4600/R4700, which does not offer a choice of coherency protocols. On other processors the cacheability of `CKSEG0` accesses is defined via the K0 field (bits

2:0) of the *config register* `C0_CONFIG`.

2.2.5 Exception processing

Exceptions are events which disrupt the normal flow of instruction execution. On the R4x00 this includes the synchronous events of TLB miss, arithmetic overflow and system calls as well as asynchronous interrupts.

When an exception occurs, the CPU enters kernel mode, saves some context in CP0 registers and jumps to the appropriate exception handler. The architecture defines four different exception handlers:

TLB at PA 0x000: Used during 32-bit execution for primary TLB refill exceptions.

> **Implementation choice:** The 32-bit TLB miss handler is not used by L4.

XTLB at PA 0x080: Used during 64-bit execution for primary TLB refill exceptions.

CACHE at PA 0x100: Used for cache errors.

> **Implementation choice:** L4/MIPS Version 79 handles cache errors by panicking.

GENERAL PA 0x180: Used for all other exceptions (overflow, system call, interrupt, or nested TLB miss).

Exception handlers are addressed via `CKSEG0` during normal operation, except cache errors which use `CKSEG1` (for uncached access). During boot time the exception vector base address is shifted up to 0xfff ffff bfc0 0200 (a ROM address), but this is only of relevance when writing ROM monitors.

The relevant CP0 registers are:

`C0_BADVADDR`: Contains the virtual address causing a TLB or cache exception.

`C0_STATUS`: Contains the processor status, see below.

`C0_CAUSE`: Contains, among others, a field indicating the cause of the exception and a mask indicating pending interrupts.

`C0_EPC`: Contains the saved PC value during exception processing. The `eret` instruction will reload the PC from this register, if it finds the `C_STATUS.EXL` bit set and the `C_STATUS.ERL` bit unset.

`C0_ErrPC`: Contains the saved PC value during error processing. The `eret` instruction will reload the PC from this register, if it finds the `C_STATUS.ERL` bit set.

In addition there is a `C0_COUNT` register which is decremented on every second clock cycle, and the `C0_COMPARE` register, which triggers a timer interrupt when it matches the contents of the `C0_COUNT` register.

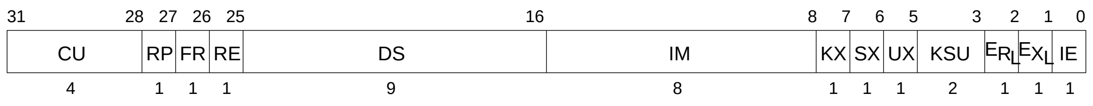


Figure 2.3: R4x00 status register format.

Figure 2.3 shows the contents of the status register, `C0_STATUS`. The relevant fields are:

IM: Interrupt mask; a bit, if set, enables the corresponding interrupt, providing that the `IE` bit is set.

KX, SX, UX: If set turns on 64-bit addressing in kernel, supervisor and user modes, respectively.

KSU: Processor mode — values of 00, 01, 10 for kernel, supervisor and user mode, respectively.

ERL: Error level — if set, the CPU is executing in *error mode*, which implies kernel mode irrespective of the setting of the `KSU` field.

EXL: Exception level — if set, the CPU is executing in *exception mode*, which implies kernel mode irrespective of the setting of the `KSU` field.

IE: Interrupt enable — if set, interrupts are enabled or disabled according to the setting of `IM`; if unset, interrupts are disabled irrespective of the setting of `IM`.

Chapter 3

L4/MIPS Organisation and Data Structures

3.1 L4/MIPS Source Structure

The directory structure of the L4/MIPS distribution is relatively simple:

`example/`	a simple example program
`doc/`	Ref man, User man, C inferface man pages
`src/`	source tree
`Makefile.conf`	master configuration file
`include/`	header files
`*.h`	rudimentary C library
`kernel/`	kernel-internal header files
`l4/`	C bindings for L4 syscalls
`lib/`	rudimentary C library
`tools/dit`	Boot file builder
`kernel/`	kernel source
`vm/*/`	various versions of page table code
`drivers/`	user-level device drivers (shouldn't be in kernel directory)
`indy/`	Indy-specific kernel code
`p4000i/`	code for Algorithmics board
`u4600/`	code for UNSW R4700 board
`libkern/`	minimal libraries for kernel code
`test/`	a number of test programs

Most directories contain a `Makefile` which contains generic rules and include `src/Makefile.conf` for parameterisation.

In the reminder of this document, unless there is an explicit statement to the contrary, we will denote file path names relative to the `src/` directory.

Two header files should be mentioned at this stage: `include/regdef.h` defines the mnemonic register names. `include/r4kc0.h` defines architecture-dependent constants defining the address-space layout of Figure 3.1, the exception vectors, CP0 register names and status, cause and TLB register fields. It also defines constants for setting the caching `C` field and cache instructions.

3.2 Kernel Data Structures

3.2.1 Kernel memory allocation

The kernel is loaded across an ethernet interface by the resident boot monitor, PMON. The boot image is built from the kernel executable (in ELF format) via a tool called DIT. The kernel must be linked to start at virtual memory address 0xffff ffff 8005 0000, the `CKSEG0` address corresponding to physical address 0x50000. (This is for the P4000i and U4600 systems, on the Indy the addresses are shifted but the layout is the same.) This address is defined as `Makefile.conf:LINKADDR`.

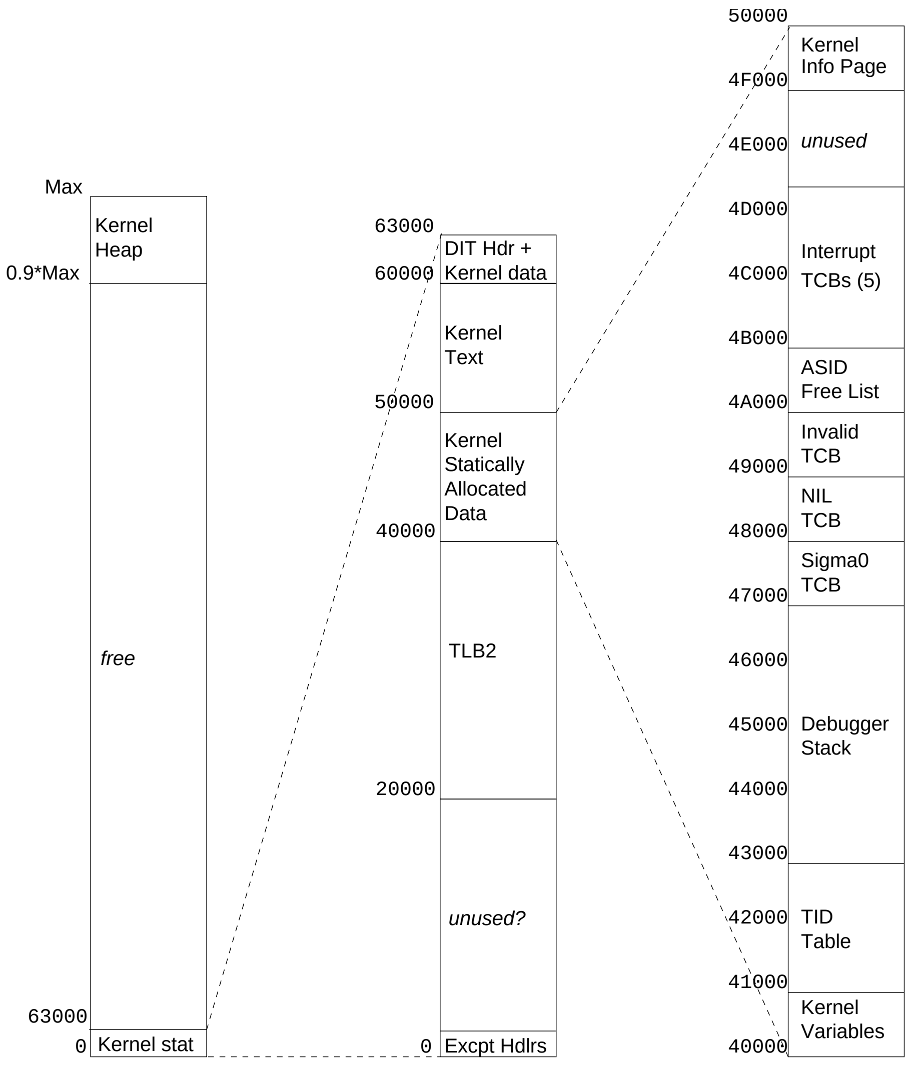


Figure 3.1: Kernel physical memory map.

At boot time the kernel sets up physical memory as shown in Figure 3.1. The region from 0x1000–0x20000 (124kB) seems presently unused. Possibly this is used by the resident boot monitor on either the U4600 or the Indy.

Implementation criticism: There is to be no reason why this memory is not used once L4 has booted up. It should be added to the kernel's memory pool.

The region marked as “TLB2” is used for the software TLB, it supports an STLB of up to 128kB.

Implementation criticism: Ideally, the STLB size should be determined as a function of physical memory size. This would imply allocating it at a dynamically determined location, but that could be done without slowing down the TLB miss handler.

The memory layout is defined by constants in `include/kernel/kernel.h`.

3.2.2 Miscellaneous kernel data: `kernel_vars`

Page 0x40 contains a structure, called `kernel_vars`, defined in `include/kernel/kernel.h`. It contains global kernel data, mostly simple types. It also contains the jump tables for the system call despatcher and general exception handler, and the array of pointers to the ready queues for each of the 256 scheduling priorities. Its fields are (somewhat out of order for the purpose of grouping related ones):

`stack_bottom`: the kernel stack for the current thread (top of the current TCB);

`s0_save`–`s4_save`: place for exception handlers to save `s0`–`s4` as long as no kernel stack is set up;

`soon_wakeup_list`, `wakeup_list`, `late_wakeup_list`: pointer to the heads of the three wakeup lists of threads blocked with a timeout. The thread is inserted into the “soon” wakeup list if its timeout is at most 16ms, into the medium-term wakeup list if its timeout is not more than 1024ms, and the “late” wakeup list if it has a finite timeout of more than 1024ms. The three lists are used to reduce the amount of list processing required at each 1ms clock tick;

`present_list`: list of all valid TCBs, grouped by task;

`int_list`: list of threads preempted by interrupts;

`clock`: system clock, mirrored in KIP;

`timeslice`: remaining current time slice (i.e., time until next preemption);

`priority`: current priority (as assigned by last scheduler invocation);

`memory_size`: total RAM size;

`trace_reg_save`: place where instrumentation code may save a register;

`tlb_t0_save`–`tlb_t8_save`: place for TLB miss handlers to save `t0`–`t8`;

`gpt_pointer`: root of running thread’s page table;

`profile_addr`: when kernel profiling is enabled, this is used to store the address where execution was interrupted by the timer;

`tcb_gpt_pointer`, `tcb_gpt_guard`: root and root guard of kernel page tables (mapping TCBs) — ***apparently unused***;

`frame_list`: pointer to head of list of free kernel heap frames (i.e., high physical memory);

`free_asid_list`, `gpt_free_list`, `gpt_leaf_free_list`, `mt_free_list`: pointer to head of free lists for: ASIDs, GPT internal nodes, GPT leaf nodes, mapping tree nodes, respectively. A separate free list is maintained for each data structure of (potentially) different size. The ASID free list is in static kernel memory, all others are on the kernel heap;

`break_on`: used for setting kernel break points for debugging;

`sigz_tcb`: TCB of σ_0

`asid_fifo_count`: pointer to next ASID due for FIFO preemption;

`int0_thread`–`int4_thread`: TID of handler thread of user-visible interrupts 0–4;

`fp_thread`: TID of thread “owning” the FPU;

`tlb_miss`, `tlb_miss_time`, `tlb2_miss`: used in instrumented TLB miss handler to store miss counts and handling times;

`syscall_jmp_table[8]`: jump table used by syscall dispatcher;

`exc_jmp_table[16]`: jump table used by exception handler dispatcher;

`prio_busy_list[MAX_PRIORITY+1]`: pointers to each priority’s circular list of runnable threads, null if a particular list is empty. The array entries actually point to the *tail* of the list, which is the TCB of the thread that was last scheduled (and on whose time slice the present thread, whoever it might be, is executing). If `prio_busy_list[p]` is non-null, then `prio_busy_list[p]->busy_link` is the TCB at the head of the list;

`frame_table_base`, `frame_table_size`, `frame_table_pointer`: unused.

3.2.3 TCBs

TCBs are allocated as a large array in virtual memory (`CKSEG`). The structure of a TCB is defined in `include/kernel/kernel.`
Each TCB is half a page (2kB) in size. Only about 0.5kB of this is needed for various data structures describing a thread, the reminder is used as the stack during execution in kernel mode.

TCBs are allocated on demand: When a thread with no previously allocated TCB is created, a frame on the kernel heap is allocated and mapped to the appropriate entry in the TCB array. If an unmapped TCB of a non-existing thread is touched (e.g., when someone attempts to IPC to that thread), the Invalid TCB is mapped (by the TLB miss handler).

The individual fields of the TCB are listed here (again somewhat reordered). Entries marked “†” are task attributes which are only defined for l_0’s TCB; all others are defined for any active thread (even though some are logically task attributes):

`sndq_start`, `sndq_end`: head and tail pointers for the doubly linked ***FIFO???*** queue of send operations pending for this thread;

`sndq_next`, `sndq_prev`: link fields for send queue, only relevant if thread’s state is PENDING;

`soon_wakeup_link`, `wakeup_link`, `late_wakeup_link`: link field for short term, medium term, long term *wakeup* lists, respectively;

`wakeup`: time at which the thread’s timeout expires (defined for threads presently linked in a wakeup list);

`busy_link`: link field for busy list (prioritised ready queue), zero if thread is not in the busy list;

`int_link`: link field for list of interrupted threads;

`present_next`: link field for TCB *present* list. This links all allocated TCBs of a task to allow efficient cleanup on task destruction;

`wfor`: TID of partner this thread is waiting to receive from;

`stack_pointer`: top of thread’s kernel stack;

`asid`: ASID value of task, -1 if task has no ASID allocated at present;

`gpt_pointer`: root of task’s page table;

`myself`: this thread’s ID (TID);

`coarse_state`: state of this TCB: *unused, used, invalid*. The latter is used to identify TCBs which are presently mapped to the shared *Invalid TCB*;

`fine_state`: thread state. Set of:

- `BUSY`**:** ready to run or running,
- `WAITING`**:** blocked on a receive,
- `POLLING`**:** blocked on a send; also called *pending*,
- `WAKEUP`**:** waiting or polling thread with a finite timeout; is in a wakeup queue,
- `LOCKS`**:** in long IPC, sending,
- `LOCKR`**:** in long IPC, receiving,
- `DYING`**:** task being killed (during `task_new`),
- `INACTIVE`**:** not activated but has a TLB allocated anyway (because it's the buddy of an active thread).

Only `WAKEUP` can occur in combination with other state values; ***CHECK!***

`timeout`: timeout value in L4 timeout format (only some fields still relevant by the time it gets stored here...);

`recv_desc`: receive descriptor for receive phase of IPC, stored here during send and while blocked on receive;

`child_task`†: pointer to TCB of first child task (or NULL);

`sister_task`†: pointer to TCB of next task in same clan (or NULL);

`rem_timeslice`: remaining time slice of preempted thread;

`timeslice`: thread's time slice length;

`mcp`†: maximum controlled priority (MCP) as defined by the `task_new` system call;

`bpad1`: byte padding (should never be accessed);

`tsp`: scheduling priority of current thread;

`ctsp`: scheduling priority under which the current thread is presently executing. This can be different from `tsp` due to *time-slice donation* and *priority inheritance*;

`pager_tid`: TID of thread's pager;

`excpt_tid`: TID of this thread's exception handler;

`comm_partner`: pointer to TCP of thread we are waiting (or polling) for or are presently communicating with;

`wdw_map_addr`: the base address, in the receiver's address space, of the temporary mapping area. This is used during long IPC to allow mapping a page fault address inside the temporary mapping area back to an address in the receiver's address space;

`interrupt_mask`: if thread is associated with an interrupt, this holds the corresponding interrupt mask. It has a single bit set which corresponds to the interrupt the thread is associated with;

`stacked_fine_state`, `stacked_comm_prtnr`: values of `fine_state` and `comm_partner` saved while in recursive (page fault) IPC;

`fp_regs[32]`, `fp_control`: thread's FPU context;

`pt_size`, `pt_number`: page table space usage and number of mappings, used by instrumentation code;

`cpu_time`: thread's accumulated CPU time.

3.2.4 Other kernel data structures

Other statically allocated data are:

TID Table: Array of task version numbers indexed by task ID. The format of a (32-bit wide) entry is:

i	0 $_{(3)}$	chief $_{(11)}$	0 $_{(2)}$	o	vers $_{(14)}$

Here *chief* is the present owner (chief) of the task, *vers* is the present task version number, *i*, if set, indicates that the task is inactive, and *o* indicates version overflow (task activated too many times). Note that the *chief* field is at the same position as in the upper word of a thread ID, and can therefore be matched without additional shift operations.

Debugger stack: Used as kernel stack during boot time, later as the stack for the kernel debugger.

Sigma0 TCB: Thread control block for σ_0. Only first half of page is used.

NIL TCB: TCB of NIL thread ***what is it used for???***.

Invalid TCB: Page of two TCBs whose `course_state` is marked invalid. Mapped on demand to TCB of non-existing thread (when faulting on an unmapped TCB).

ASID Table: Array, indexed by ASID. For each ASID presently in use the entry contains the task ID it is associated with. For unused ASIDs it contains a pointer to the next free one, zero indicates the end of the list. The list head (pointer to first free entry) is in `kernel_vars.free_asid_list`.

Interrupt TCBs: TCBs of virtual threads representing interrupts. Inherit scheduling parameters from their respective interrupt handler threads (if associated).

Page tables are dynamically allocated on the kernel heap. Their structure is discussed with the miss handling code in Section 4.1.2 below. ***Still TO DO: mapping database.***

Chapter 4

Exception Processing

4.1 TLB Miss Handling

TLB miss handling is, together with “short” IPC, the kernel operation most critical to performance of systems built on top of L4.

The L4/MIPS distribution contains a number of page table implementations, each in a separate subdirectory of `kernel/vm/`: multi-level page table (`vm-mpt/`), inverted page table (`vm-ipt/`), clustered page table (`vm-cpt/`), and guarded page table (`vm-gpt/`). The guarded page table (GPT) structure is also implemented in combination with two versions of a software TLB (STLB, `vm-tlbcache-gpt/` and `vm-tlbcache-gpt-pair/`). The version to be built is specified by `Makefile.conf:VM_CODE`. The kernel-internal API for the page tables is defined in `include/kernel/vm.h`.

Implementation choice: Elphinstone [Elp99b] showed that `vm-tlbcache-gpt-pair/` is, of the ones implemented, the most appropriate version for L4. We will therefore not consider any of the other implementations.

The page table implementation in `vm-tlbcache-gpt-pair/` supports four different implementation of the STLB: `tlb2-1way-8.S`, `tlb2-1way-128.S`, `tlb2-2way-128.S`, and `tlb2-2way-8.S`. The one used is selected by `Makefile.conf/TLB2_OBJ`.

Implementation choice: Again, the study has shown that the most appropriate version is `tlb2-1way-128.S` and only it will considered in this report. See Elphinstone’s PhD thesis [Elp99b] for details on the various page table implementation and their analysis.

In summary, the kernel’s page tables consist of a global, direct-mapped STLB of 128kB size, tagged with VPN and ASID. It contains 8k entries, each mapping two pages. Hence the total coverage of the STLB is 16k pages, or 64MB best case. Note that equals times the physical memory size for which the configuration was optimised. This results in reasonably high STLB hit rates.

On a miss the STLB is reloaded from a per-address-space GPT. Given that the STLB is configured for high hit rates, the GPT lookup costs are not very critical. However, GPTs can potentially grow rather deep (7–10 levels are not extraordinary), the GPT lookup code is therefore highly optimised.

4.1.1 Fast miss handler

Owing to the architecture-determined allocation of exception vectors, the TLB miss handler (like other exception vectors) can be up to 128 bytes (32 instructions) long, any extra code must be allocated elsewhere and accessed via

```
0         lui     k0, KERNEL_BASE          # kernel vars ptr
1         sd      t0, K_TLB_T0_SAVE(k0)    # save t0
2         lui     k1, TLB2_BASE            # TLB cache base adr
3         dmfc0   k0, C0_ENTRYHI           # has VPM/2
4         dsll    t0, k0, 38
5         dsrl    t0, t0, 47               # STLB index
6         daddu   k1, t0, k1
7         ld      t0, (k1)                 # EntryHi
8         bne     t0, k0, 2f               # check tag
9         lwu     t0, 8(k1)                # EntryLo1
10        lwu     k0, 12(k1)               # EntryLo0
11        dmtc0   t0, C0_ENTRYLO0
12        dmtc0   k0, C0_ENTRYLO1
13        lui     k0, KERNEL_BASE
14        tlbwr                            # load TLB
15        ld      t0, K_TLB_T0_SAVE(k0)    # restore t0
16        eret
17  2:    j       tlb2_miss                # TLB cache miss
18        lui     t0, KERNEL_BASE
```

Listing 4.1: TLB refill handler `xtlb_refill`.

a jump instruction. The handler, `xtlb_refill` in `kernel/vm/vm-tlbcache-gpt-pair/tlb2-1way-128.S`, reloads the TLB from the STLB. It is shown in Listing 4.1.

Line 0 loads the address of `kernel_vars` into the kernel-reserved register k0. Note that the `lui` instruction does not introduce a load delay, so `k0` can be used in the next cycle.

Line 1 saves register `t0` in `kernel_vars`, as the two reserved registers `k0` and `k1` are not sufficient for this TLB miss handler.

Line 2 loads the address of the STLB into `k1`. The result is not used until Line 6. This means that, provided that the STBL base was allocated to the same cache line as `K_TLB_T0_SAVE`, *there would be no extra cost for loading the STLB base from* `kernel_vars`, as would be required for a dynamically sized STLB.

Line 3 loads the CP0 EntryHi register into `k0`. On a TLB exception this coprocessor register is loaded by the hardware with the VPN2 value corresponding to the faulting virtual address, and the present ASID value.

Lines 4–5 load `t0` with the value of `(VPN2*16)%TLB2_SIZE`, ready to use as an index into the STLB. Lines 6–7 load `t0` with the first half of the 16-byte STLB entry, which contains the EntryHi value.

Line 8 compares the EntryHi value of the STLB entry with the value obtained from CP0 (note the pipeline stall due to the immediate use of `t0`). A mismatch indicates a cache miss and the code branches to Line 17 from where it jumps to the STLB miss handler (outside frame zero). The address of `kernel_vars` is reloaded into `t0` in the delay slot of the jump.

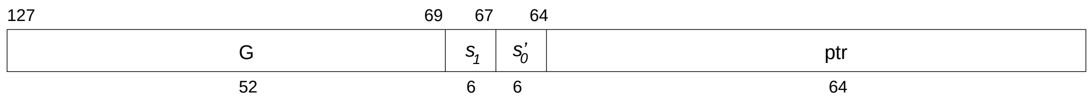
Line 9 (executed in Line 8's delay slot, although not used if the branch is taken) loads the third quarter of the STLB entry into `t0`, and the following line loads the final part of the entry into `k0`. These two values are the compressed EntryLo values for the two pages represented by VPN2. Lines 11–12 load them into the corresponding CP0 registers. Note that the top 34 bits of the EntryLo registers are always zero on the R4x00, so both are effectively 32-bit quantities. Line 14 loads the TLB with the new entry, randomly replacing an existing one.

Lines 13 and 15 restore the original value of `t0` and Line 16 performs the return from exception handling.

The fast TLB miss handler code is 16 instructions long. It contains four memory loads, one of which has an unused delay slot. There is one branch instruction which is usually not taken. It may take a maximum of two data cache misses.

4.1.2 STLB miss handler

The STLB miss handler, `tlb2_miss` in `kernel/vm/vm-tlbcache-gpt-pair/tlb2-1way-128.S`, reloads the cache from the page table proper, i.e., the GPT belonging to the faulting task. Each GPT node is an array of GPT entries of the form



where:

s_1: # untranslated address bits (guard & node index stripped)

s_0: # address bits to be translated at node (s'_0 for child node)

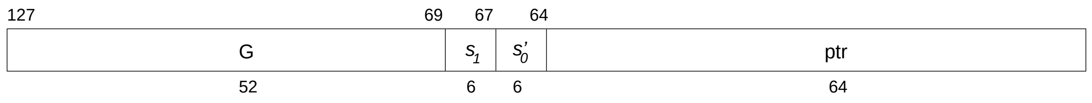
$s_1 - s'_0$: $\log$ of child node size (# child index bits)

$s_0 - s_1$: # guard bits

G: extended guard (index bits + guard)

u: node index

For a given partially translated virtual address these fields look as follows:



Implementation choice: The pointer could be stored in 32-bit, relying on sign extension during the load. However, this would not by itself make more compact page tables feasible due to the resulting unaligned data. However, since virtual addresses on the R4x00 are only 40 bits long, 28 bits would suffice for the extended guard, and the pointer could be packed into the remaining 24 bits. This would half the page table size, but the requirement for unpacking the data would result in a significant slowdown of page table lookup and manipulation code.

The STLB miss handler is entered with EntryHi in `k0`, the address of the cache bucket to load in `k1`, and the address of `kernel_vars` in `t0`. The code is shown in Listing 4.2.

Lines 0–2 free up additional work registers `t1`, `t2`, and load `t2` with the address of the root of the current page table from `kernel_vars`. Line 3 is redundant (but see comments to Line 30).

Line 4 loads the shift count used in Line 5 to extract the top bits of the VPN as an index into the root node of the GPT. `GPTROOTSIZE` is the number of index bits required (the root node has in the present implementation a constant size of $2^{\texttt{GPTROOTSIZE}}$ entries), and 2^4 is the size of each GPT node entry. Note that bits 3:0 of the value resulting from Line 5 are not masked out and are therefore undefined. To save the masking operation Line 6 constructs a GPT entry pointer not by adding but by or-ing the base pointer with the offset. This leads to a defined result if it is ensured that the "node pointer" in `t2` has bits 3:0 all set. It is in this form that all GPT pointers are stored, including the root pointer in `kernel_vars`.

This becomes evident in Lines 7–8, where the 128-bit GPT entry is loaded into `t0` and `t2`. The specified offsets of -15 and -7 result in the correct address when added to the result of Line 6.

```
 0          sd      t1, K_TLB_T1_SAVE(t0)
 1          sd      t2, K_TLB_T2_SAVE(t0)
 2          ld      t2, K_GPT_POINTER(t0)
 3          dmfc0   k0, C0_ENTRYHI
 4          dli     t1, WORDLEN - 4 - GPTROOTSIZE

 5  1:      dsrlv   t1, k0, t1
 6          or      t2, t1
 7          ld      t0, -15(t2)
 8          ld      t2, -7(t2)
 9          xor     k0, t0
10          dsrl    t1, k0, t0
11          beql    t1, zero, 1b
12          dsrl    t1, t0, 6

13          dsrl    k0, t0, 6
14          dsllv   t1, t1, k0
15          bne     t1, zero, xtlb_refill_fail
16          nop
17          lw      t0, (t2)
18          lw      t1, 4(t2)
19          dmfc0   t2, C0_ENTRYHI
20          sw      t0, 8(k1)
21          sw      t1, 12(k1)
22          sd      t2, (k1)
23          dmtc0   t0, C0_ENTRYLO0
24          dmtc0   t1, C0_ENTRYLO1
25          lui     k0, KERNEL_BASE
26          tlbwr
27          ld      t0, K_TLB_T0_SAVE(k0)
28          ld      t1, K_TLB_T1_SAVE(k0)
29          ld      t2, K_TLB_T2_SAVE(k0)
30          nop     /* avoid a potential 48 instruction routine */
31          eret
```

Listing 4.2: STLB miss handler `tlb2_miss`.

Lines 9–11 implement the matching of the guard. The extended guard from the first word of the GPT entry is xor-ed with the as yet untranslated part of the address (the full EntryHi value on the first iteration of the loop). This destroys bits 11:0 of the value, as the GPT entry stores s_1, s'_0 in those bits, but since these bits are not part of the page number this is irrelevant. (In fact, bits 7:0 of EntryHi contain the ASID.) Line 10 shifts the xor result to the right by s'_0, which leaves the guard bits and the bits already used for indexing the GPT node. Both fields are stored in the GPT node as part of the *extended guard*. The index bits are guaranteed to match (and are therefore zero in the result left in `t1` after Line 10), so `t1` contains zero iff the guard matched. This is tested in Line 11.

Note that Line 10 relies on the shift instruction only using bits 5:0 (s'_0) of `t0` for determining the shift count. This field is shifted out by Line 12 which is in Line 11's branch delay slot and is executed prior to executing Line 5 the next time, which then uses s_1 as the shift count. Line 11 uses the `beql` instruction which nullifies the branch delay slot if the branch is **not** taken, so Line 12 is not executed if the loop is exited due to a guard mismatch.

Lines 13–15 test the reason the loop terminated: guard mismatch or a leaf has been reached. Loop exit at a leaf entry is forced by storing "incorrect" index bits in the extended guards of leaf nodes. The xor result from Line 9 is shifted right by s_1, which only leaves the xor of the guard proper, and is zero iff the guard matched (indicating a leaf entry). Otherwise a page fault has occurred, which is handled in `xtlb_refill_fail`. The latter restores registers and jumps to `fail_tlb_rfl_ent`, which is part of the general exception handling code in `kernel/exc.S`, to invoke the faulting thread's pager (see Listing 4.5).

Lines 17–19 load the EntryLo values from the leaf node and the EntryHi value from CP0 and Lines 20–24 and 26 load the complete entry into the STLB and the TLB. Lines 25 and 27–31 restore registers and return.

The `nop` instruction in Line 30 is to avoid the possibility that the whole miss will be handled in exactly 48 cycles. This would mean that the `C0_RANDOM` register would point to the same TLB entry (remember, the TLB has 48 entries) and would thus run the risk of the entry just loaded being replaced right away. This would bear the risk of getting into an infinite loop if a single instruction faults on two pages (as is possible for a load or store instruction).

More details on this very tight code can be found in [LE95].

4.2 General Exception Handling

All exceptions, other than cache errors and TLB misses occurring outside exception mode, are vectored to the *general exception handler*. This includes all system calls.

4.2.1 General exception handler

The general exception handler is the last exception vector, so it is not restricted to a length of 128B (and is, in fact, 180B long). The code is in `kernel/exc.S:gen_exc` and is shown in Listing 4.3.

Lines 0–7 load the `ExcCode` from CP0 and invoke `other_excpt` to handle exceptions other than system calls. As well, the Status register value is loaded into `k1` and the pointer to `kernel_vars` into `k0`.

Lines 8–11 reset the error level and exception level flags, disable interrupts and turn on kernel mode. Kernel mode was already active while in exception mode, but turning off exception mode (by resetting the `EXL` flag) would

```
 0          mfc0    k0, C0_CAUSE
 1          mfc0    k1, C0_STATUS
 2          andi    k0, CA_EXC_CODE
 3          subu    k0, CA_Sys
 4          beq     k0, zero, 1f
 5          lui     k0, KERNEL_BASE
 6          j       other_excpt
 7          nop
 8  1:      move    t0, k1
 9          srl     k1, 5
10          sll     k1, 5
11          mtc0    k1, C0_STATUS
12          andi    k1, t0, ST_KSU
13          beq     k1, zero, 1f
14          move    t2, sp
15          ld      sp, K_STACK_BOTTOM(k0)
16  1:      dmfc0   t1, C0_EPC
17          sd      t2, -8(sp)
18          daddiu  t1, t1, 4
19          sd      t1, -16(sp)
20          sb      t0, -24(sp)
21          bne     AT, zero, 1f
22          dsubu   sp, 24
23          j       k_ipc
24          tcbtop(t8)
25  1:      slti    t1, AT, MAX_SYSCALL_NUMBER + 1
26          beq     t1, zero, 2f
27          dsll    AT, 3
28          daddu   t0, k0, AT
29          ld      t0, K_SYSCALL_JMP_TABLE(t0)
30          jr      t0
31          ori     t8, sp, TCBO
32  2:      syscall_ret
```

Listing 4.3: General exception handler `gen_exc`.

otherwise return the CPU to user mode. Similarly for interrupts; these are implicitly disabled when in exception mode and need to be disabled explicitly when resetting exception mode.

Note that during a system call the kernel is allowed to trash certain registers, particularly the "callee-saved" `t` registers.

Lines 12–15 check whether the CPU was in kernel mode (`ST_KSU`) prior to the exception. If not, the stack pointer is set to the kernel stack area in the executing thread's TCB, after temporarily saving the old `sp` value in `t2` (in Line 13's branch delay slot). Note that this allows kernel threads, such as the idle thread and σ_0, to perform L4 system calls without changing the kernel stack.

Lines 16–20, 22 set up an exception frame, as shown in Figure 4.1, on the kernel stack. The old stack pointer is stacked as well as the exception PC (adjusted to restart execution *after* the syscall instruction) and the least significant byte of the old status register value (which will still differ from the pre-exception value by the `ST_EXL` bit).

Lines 21, 23 invoke the IPC handler `k_ipc` if the syscall number in `AT` is zero. Line 24, executed in Line 23's branch delay slot, loads `t8` with the address of the top of the running thread's TCB, by masking in the least significant bits of the stack pointer.

Lines 25–31[1] check whether the syscall number is within range and, if yes, invoke the appropriate system call via the jump table stored in `kernel_vars`.

Implementation criticism: Line 31 is identical to Line 24 and *should really use the same macro*.

Implementation choice: L4/MIPS uses the `AT` register to hold the system call number, while Linux on the MIPS uses the `v0` register. This prevents binary compatibility of statically linked binaries between native Linux and L4-based systems. However, statically linked binaries are rarely used in the Linux world, mostly for some maintenance tool, whose source code is readily available. We therefore do not consider this choice problematic.

An out-of-range syscall is silently ignored, as Line 32 simply continues execution of the caller.

Implementation criticism: Ignoring invalid system calls is a bad idea, an exception should be raised instead. This would then allow the use of a *trampoline* mechanism to emulate other system calls.

The code uses all load and branch delay slots. However, the memory access at Line 15 is likely to cause a cache miss. The total execution time for an IPC system call is 25 cycles plus one possible D-cache miss.

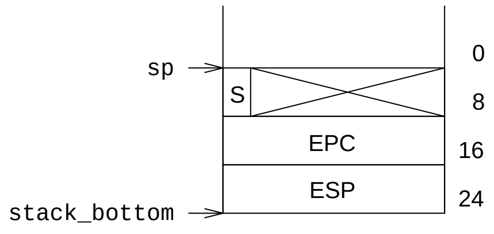


Figure 4.1: Exception stack frame set up on the kernel stack by the general exception handler. The field denoted by `S` is the least significant byte of the Status word.

[1]The `slti` instruction set the destination register to one if the source register (here same as destination) is less than the signed immediate operand; otherwise the destination is zeroed.

4.2.2 Return from exception

The `syscall_ret` macro, defined in `include/kernel/macros.h` is worth looking at. It generates the code shown in Listing 4.4.

Lines 0–2 turn the exception mode bit back on in the Status register. This is required for correct operation of the `eret` instruction, which checks the exception and error mode bits to determine whether to reload the PC from the EPC or ErrorPC CP0 registers. ("Errors" include cache errors and reset, which can happen during exception handling.) The interrupt-enable bit `IE` is also turned off as a side effect. Remember that interrupts are disabled while in exception mode irrespective of the setting of `IE`.

Lines 3–6 combine into `k0` bits 31:8 of the status register with the old value of bits 7:0 which was stacked in Line 20 of the general exception handler. This leaves, among others, the interrupt mask unchanged, and thus allows the interrupt mask to change during the execution of a system call. This is necessary to allow correct interrupt handling, see Listing 5.11. The pre-exception value of `IE` (normally on) is restored. Interrupts remain disabled as the exception flag is on.

Note, however, that the mask in Line 3 does not have the top four bits set, which mask the `ST_CU` (coprocessor usable) bits in the status register. As a result, all coprocessor use is unconditionally disabled upon return to the user. This includes the FPU, and any subsequent FPU use by the user will result in a *coprocessor unusable* exception, which will be handled by `exc_cpu` (Listing 4.18).

Lines 7–8 load the exception PC, stacked in Line 19 of the general exception handler, back into the appropriate coprocessor register, from where the `eret` instruction will reload the PC to return to the caller. Line 9 resets the status register so that its least significant byte now contains the value from immediately after the exception was taken (which will be the pre-exception setting with the exception of the `ST_EXL` bit). This changes the processor status from normal kernel-mode back to exception mode.

Line 10 restores the stack pointer and Line 11 leaves exception mode and returns to the caller.

The load delay slots of the two load instructions (Lines 5 and 7) are unused, leading to a pipeline stall. The total execution time is therefore 14 cycles, plus a possible D-cache miss on Line 5.

Note that the syscall macro uses the `k0`, `k1` registers. These registers are trashed during exception handling, including TLB reloads. Hence **this code must not cause a page fault, or the `k` register contents might get lost, with unpredictable results**. The (only) point where a page fault can happen is Line 5, where the stack pointer is first dereferenced. Remember, the stack pointer points to the kernel stack, which is part of the TCB, which is in kernel mapped memory.

The way to avoid a page fault at this point is to touch the TCB before invoking the `syscall_ret` macro, **after**

```
 0          mfc0    k0, C0_STATUS
 1          ori     k0, k0, ST_EXL
 2          mtc0    k0, C0_STATUS
 3          li      k1, 0x0fffff00
 4          and     k0, k0, k1
 5          lbu     k1, (sp)
 6          or      k0, k0, k1
 7          ld      k1, 8(sp)
 8          dmtc0   k1, C0_EPC
 9          mtc0    k0, C0_STATUS
10          ld      sp, 16(sp)
11          eret
```

Listing 4.4: The `syscall_ret` macro.

any other references to mapped memory.

4.2.3 Exception dispatcher `other_excpt`

The code for handling general exceptions other than system calls is in `kernel/exc.S:other_excpt`. This entrypoint is aliased to `fail_tlb_rfl_ent`, which is invoked when a TLB refill fails due to a page not being mapped in the page table (a proper page fault). The handler is shown in Listings 4.5, 4.6.

Lines 0–5 load the processor status and save a number of temporary registers in `kernel_vars`.

Lines 6–13 reset exception mode and set kernel mode, and load the kernel stack pointer if coming from user mode. This is identical to the code of Lines 8–15 of the general exception vector. Note that this needs to be done here as the corresponding code in the general exception vector was not executed prior to jumping here, nor was it executed in the TLB miss handler.

Lines 14, 17–19 set up the exception stack frame consisting of the excepting thread's PC, stack pointer and processor status, as in Figure 4.1. This is like in the general exception vector, except that the PC is not adjusted, so the excepting instruction will be restarted upon return from exception.

Lines 20–54 stack the whole register set, other than the kernel-reserved `k0` and `k1`, and the stack pointer, which is already stacked. This includes the registers already saved in `kernel_vars` which are now also pushed onto the stack (Lines 36–45; note the order of loads and stores which makes use of all load delay slots).

Having saved all registers the exception handler is now able to call C functions. Some registers which are not used by kernel assembly code would not need saving for the sake of invoking C functions. However, a context switch may occur as a side effect of an exception, and it is therefore important that the full processor context is saved.

Bug/Restriction 2: HI/LO not saved.
The code as shown fails to save the multiplication and division result registers, `HI` and `LO`.

Lines 55–61 dispatch the appropriate exception handler function via the jump table in `kernel_vars`, using the Cause register field `CA_EXC_CODE` as the index. The handlers are invoked with the address causing the exception (`C0_BADVADDR`) in s3.

General exception return: `other_excpt_ret`

The return code, `other_excpt_ret`, is a straightforward extension of the `syscall_ret` macro. It restores all stacked registers prior to performing the same operations as `syscall_ret` (mindful of the different stack state, Listing 4.7).

4.2.4 TLB exceptions

The MIPS R4x00 knows four types of TLB exceptions:

TLB Refill: No TLB entry matches the virtual address. This exception vectors to the fast TLB miss handler discussed in Section 4.1.1, unless the CPU is in exception mode (as for a secondary TLB miss). However, remember that the fast TLB miss handler reverts back to the general exception handler code in the case of a page fault (when no mapping is found in the page table, see Line 15 of Listing 4.2).

In the latter case the exception is vectored to the general exception handler like a TLB Invalid exception. Note that L4 does not access mapped memory in exception mode (except in the `syscall_ret` macro, which must be used with care), so this type of exception indicates a kernel bug.

TBL Invalid — Load or Store: A matching entry for a virtual address was found in the TLB but it is marked *invalid* (i.e., the `V` bit is off). The `CA_EXC_CODE` is set to `CA_TLBL` or `CA_TLBS` depending on whether the exception occurred on a memory read or write operation.

TLB Modified: A store is attempted to a page which is mapped read-only in the TLB (`V` bit is on but `D` bit is off). The `CA_EXC_CODE` is set to `CA_Mod`.

These exceptions (except the Refill exception vectored to the fast TLB exception handler) can have three causes: a kernel bug (as indicated above), a page fault in kernel mapped memory (i.e., the TCB array), or a user page fault. In the latter case the user thread's pager must be invoked by sending an IPC message to it.

The TLB Modified and TLB Store exceptions use the same handler (`exc_tlbs` being an entrypoint in `exc_mod`), except that `exc_mod` first checks for a documented processor bug (and panics it that bug is tripped). The handler for TLB Load exceptions (`exc_tlbl`) is almost identical, except it omits one instruction.

Implementation criticism: These functions should share some code.

```
        mfc0    k1, C0_STATUS
        sd      s0, K_S0_SAVE(k0)
        sd      s1, K_S1_SAVE(k0)
        sd      s2, K_S2_SAVE(k0)
        sd      s3, K_S3_SAVE(k0)
        sd      s4, K_S4_SAVE(k0)
        move    s0, k1
        srl     k1, 5
        sll     k1, 5
        mtc0    k1, C0_STATUS
        andi    k1, s0, ST_KSU
        beq     k1, zero, 1f
        move    s1, sp
        ld      sp, K_STACK_BOTTOM(k0)
1:      dmfc0   s2, C0_EPC
        dmfc0   s3, C0_BADVADDR
        mfc0    s4, C0_CAUSE
        sd      s1, -8(sp)
        sd      s2, -16(sp)
        sb      s0, -24(sp)
        sd      AT, -32(sp)
        sd      v0, -40(sp)
        sd      v1, -48(sp)
        sd      a0, -56(sp)
        sd      a1, -64(sp)
        sd      a2, -72(sp)
        sd      a3, -80(sp)
        sd      a4, -88(sp)
        sd      a5, -96(sp)
        sd      a6, -104(sp)
        sd      a7, -112(sp)
        sd      t0, -120(sp)
        sd      t1, -128(sp)
```

Listing 4.5: Exception dispatcher `other_excpt`, first part.

```
        sd      t2, -136(sp)
        sd      t3, -144(sp)
        lui     k0, KERNEL_BASE
        ld      t0, K_S0_SAVE(k0)
        ld      t1, K_S1_SAVE(k0)
        sd      t0, -152(sp) /* s0 */
        ld      t0, K_S2_SAVE(k0)
        sd      t1, -160(sp) /* s1 */
        ld      t1, K_S3_SAVE(k0)
        sd      t0, -168(sp) /* s2 */
        ld      t0, K_S4_SAVE(k0)
        sd      t1, -176(sp) /* s3 */
        sd      t0, -184(sp) /* s4 */
        sd      s5, -192(sp)
        sd      s6, -200(sp)
        sd      s7, -208(sp)
        sd      t8, -216(sp)
        sd      t9, -224(sp)
        sd      gp, -232(sp)
        sd      s8, -240(sp)
        sd      ra, -248(sp)
        daddiu  sp, sp, -(ST_EX_SIZE)
        andi    t0, s4, CA_EXC_CODE
        dsll    t0, 1
        daddu   t0, t0, k0
        ld      k1, K_EXC_JMP_TABLE(t0)
        nop
        jr      k1
        nop
```

Listing 4.6: Exception dispatcher `other_excpt`, second part.

```
        ld      ra, (sp)
        ld      s8, 8(sp)
        ...
        ld      v0, 208(sp)
        ld      AT, 216(sp)
        mfc0    k0, C0_STATUS
        ori     k0, k0, ST_EXL
        mtc0    k0, C0_STATUS
        li      k1, 0x0fffff00
        and     k0, k0, k1
        lbu     k1, 224(sp)
        or      k0, k0, k1
        ld      k1, 232(sp)
        dmtc0   k1, C0_EPC
        mtc0    k0, C0_STATUS
        ld      sp, 240(sp)
        eret
```

Listing 4.7: General exception return code `other_excpt_ret`.

Listing 4.8 shows the first part of the TLB Modify/TLB Store handler. It is invoked by `other_excpt` with the exception PC in `s2`, the exception virtual address in `s3` and the value of the Cause register in `s4`.

Line 0 checks whether the fault address is negative, which indicates a `CKSEG` address and thus a fault on a non-existent TCB. If so, it jumps to the TCB handling code at the end, which is discussed below.

Lines 2–27 of the code deal with page faults during long IPC. Understanding of this code requires some understanding of the operation of long IPC, in particular the implementation of the memory copy operation between address spaces. We defer its description to Section 4.2.6 below.

```
0         bltz     s3, 3f
1         move     t2, sp
```

Listing 4.8: TLB fault handler `exc_tlbs`, prologue.

The reminder of the TLB fault handler is shown in Listing 4.9. The instruction (Line 40) missing in `exc_tlbl` is flagged by a comment.

Line 28 is the continuation point for faults other than long-IPC page faults. The address of `other_excpt_ret` is loaded as the restart address into `t3` and an infinite timeout is loaded into `a2`. Long IPC page fault code will have loaded the appropriate values into these registers when it joins this code at line 30.

Lines 30–33 enable interrupts. The reason is that the user's pager may fail to send a valid mapping. TLB exceptions take priority over interrupt exceptions on the MIPS, so this would result in the fault being re-triggered immediately on return from the exception. Interrupts would never become enabled again and the system would be livelocked. Hence we enable interrupts for a short while to introduce a *preemption point*. Interrupts are disabled again (Line 47) prior to invoking the pager IPC (Line 49).

Implementation criticism: The kernel should prevent unlimited repetition of page faults.

Line 34–50 set up the IPC message to the pager. A pseudo-exception stack is set up (Lines 34–37) containing a saved PC value pointing to the `other_excpt_ret` code, so that the IPC code's `syscall_ret` sequence in fact "returns" to the exception return code, which does the proper return to user code. (In the case of a page fault during long IPC the return address is `ipc_fault_ret`, see Line 9). The register message containing the fault address, writable bit (except for `exc_tlbl`) and exception PC, is set up (Lines 38–41). A send descriptor (Line 43), receive descriptor (Line 44), timeouts (Line 29), destination (= pager) TID (Lines 42 and 45) and wait-for TID (Line 50) are set up and the IPC code is called as if the call came directly from `gen_exc`. Note that long IPC page faults skip lines 28–29, as they have already had their restart address and timeout values set up.

Implementation criticism: Line 48 (setting the virtual sender to zero) is redundant (deceive bit is off in send descriptor), as is Line 46 (redone in the branch delay slot, Line 50).

Implementation criticism: This code triggers Bug 15, which results from the way an error status is returned if an IPC operation is aborted. The page fault IPC may be aborted or cancelled (e.g., by a handler using the `lthread_ex_regs` system call to save the faulter's state). The cleanup code will then overwrite the saved value of `v0` with an error value. If the faulter is then restarted, its `v0` register is restored from the modified value in the TCB, resulting in a trashing of the thread's register. It is necessary to save `v0` separately, like the thread state and communication partner are stacked in Lines 10 and 12.

The end of the routine (Lines 51–56) is concerned with handling page faults in the TCB array.

The code calls the C function `vm_tcb_insert` (in `kernel/vm/*/vm.c`), passing as parameters the pointer (from `kernel_vars`) to the current page table, the fault address, and the (physical) address of the *invalid TCB* statically allocated in low memory (see Figure 3.1).

This function allocates a GPT leaf node (by calling `gpt.c:gpt_insert`) and inserting the physical address of the frame containing the invalid TCB. This is ok, as an unmapped TCB can only belong to an invalid (not yet

```
4:      dla     t3, other_excpt_ret
        dli     a2, L4_IPC_NEVER
5:      mfc0    t0, C0_STATUS
        li      t1, ST_IE
        or      t1, t0, t1
        mtc0    t1, C0_STATUS
        daddiu  sp, sp, -24
        sb      t0, (sp)
        sd      t3, 8(sp)
        sd      t2, 16(sp)
        dli     s0, ~(L4_FPAGE_RW_MASK | L4_FPAGE_GRANT_MASK)
        and     s0, s0, s3
        ori     s0, s0, L4_FPAGE_RW_MASK /* TLB Mod/Store ONLY! */
        move    s1, s2
        tcbtop(t8)
        dli     a0, L4_IPC_SHORT_MSG
        dli     a1, L4_IPC_SHORT_FPAGE|(L4_WHOLE_ADDRESS_SPACE<<2)
        ld      a4, T_PAGER_TID-TCBO(t8)
        move    a5, a4
        mtc0    t0, C0_STATUS
        dli     a6, 0
        j       k_ipc
        move    a5, a4
3:      lui     a0, KERNEL_BASE
        ld      a0, K_GPT_POINTER(a0)
        move    a1, s3
        dli     a2, INVALID_TCB_BASE
        jal     vm_tcb_insert
        j       other_excpt_ret
```

Listing 4.9: TLB fault handler `exc_tlbs`, main part.

activated) thread. The kernel must obviously avoid modifying the invalid TCB, so when a thread is created it must check whether its TCB is the invalid one, and, if yes, allocate, initialise and map a new one. The code of `gpt_insert` is inherently ugly and we leave it to greater masochists to delve in it. We note, however, that the function can be called with either a GPT or a TCB pointer as its first argument. In the latter case (obviously not usable if the fault is on a TCB) the GPT pointer is taken from the TCB. The two types of pointers can be distinguished by the fact that TCB pointers are 8-byte aligned while GPT pointers have bits 3:0 set (see Listing 4.1.2).

`vm_tcb_insert` turns on the *global bit* in the page table entry, to ensure that the mapping is valid no matter which user thread executes. The function also calls `tlb2-1way-128.S:tlb2_sync_shared` which removes any matching TLB and STLB entries. Normal TLB refill handling is relied on to activate the mapping.

4.2.5 Exceptions passed to the user

A number of exceptions (address error on load, address error on store, bus error on fetch, bus error on load or store, breakpoint, reserved instruction, arithmetic overflow, and floating-point exceptions) are handled by invoking the excepting thread's user-level exception handler. They all use the same kernel exception handler, with the aliases `exc_adel`, `exc_ades`, `exc_ibe`, `exc_dbe`, `exc_bp`, `exc_ri`, `exc_ov`, and `exc_fpe`. A coprocessor-unusable exception, handled by `exc_cpu`, is also diverted here if it originated in user mode for a coprocessor other than the FPU. The code is almost identical to the TLB exception code in Section 4.2.4 (ignoring the part dealing with long IPC page faults. The differences are the that the IPC goes to the exception handler rather than the pager and is “short” (no mapping). Kernel exceptions are not meant to occur and result in a kernel panic.

Implementation criticism: This code triggers Bug 15 as does the TLB miss handlers. In the case of user exceptions this is actually more serious than for page faults, as it makes it impossible for a user-level exception handler to save an excepting thread’s complete user state.

4.2.6 TLB misses during long IPC

As mentioned above, the middle part of the TBL fault handlers `exc_mod`, `exc_tlbs` and `exc_tlbl` deal with TLB misses occurring during cross-address-space memory copies while processing long IPC. We will examine this code here.

As stated earlier, this code is unlikely to make much sense to someone who does not understand the implementation of long IPC. The reader is therefore encouraged to skip this section for now and return to it after reading Section 5.5.

Listing 4.10 shows the relevant part of the TLB miss handlers. Here we are dealing with a TLB miss which occurred in kernel mode but outside the TCB array. Faults of this kind should only result from long IPC processing. They can be either in user space (resulting from “normal” page faults while the kernel is trying to access the sender’s message buffers) or in `XKSSEG`, which is used as the `temporary mapping area` allowing the kernel to access the receiver’s data from within the sender’s context. See Listing 5.16, page 67 for details.

```
        tcbtop(a0)
        lw      t0, T_FINE_STATE-TCBO(a0)
        andi    a1, t0, FS_LOCKS
        beq     a1, zero, 4f
        dsrl    a1, s3, 62
        bne     a1, zero, window_fault
        nop
        dla     t3, ipc_fault_ret
        sw      t0, T_STACKED_FINE_STATE-TCBO(a0)
        ld      t0, T_COMM_PARTNER-TCBO(a0)
        sd      t0, T_STACKED_COMM_PRTNR-TCBO(a0)
        li      a2, 0x01010000
        lw      t1, T_TIMEOUT(t0)
        lw      a3, T_TIMEOUT-TCBO(a0)
        andi    a3, a3, 0xff00
        andi    t1, t1, 0x0f00
        srl     t1, t1, 4
        or      a2, a2, t1
        srl     t1, t1, 4
        or      a2, a2, t1
        or      a2, a2, a3
        daddiu  sp, sp, -8
        move    t2, sp
        sd      s3, (sp)
        b       5f
        nop
```

Listing 4.10: TLB fault handler `exc_tlbs`, long IPC faults.

Line 2 loads a0 with the address of the faulting thread's TCB. Note that a0 points to the top, not the bottom of the TCB (compare the k_ipc code, Listing 5.1).

Lines 3–5 test whether the faulting thread is in the LOCKS state, indicating that it is in the middle of performing long IPC. If so then we test whether the fault address is outside the user-mode address range (Lines 6–7). A TLB miss outside the user address range and outside the TCB array can only result from faulting on the temporary mapping area (or from a kernel bug). This case is handled by jumping to window_fault, see page 38.

If we reach Line 9 we must be dealing with a fault in user space resulting from the kernel accessing the sender's message buffers during a long IPC operation. Handling this requires invoking the faulting thread's pager. That means that a (nested) IPC must be performed by the sender while it is in the middle of a (long) IPC. In order to allow clean unwinding of the stack if the page fault IPC is aborted, Lines 10–12 save the thread's state and its communication partner in special locations (stacked_fine_state, stacked_comm_prtnr) of the TCB.

The send and receive timeout for the page-fault IPC (in a2) are constructed from the receive page-fault timeout value of the faulting thread's communication partner (Lines 13–22); the faulter's page-fault timeouts are preserved (so they would be used if the pager itself faults).

The real fault address is stacked (Lines 23, 25) for later use by ipc_fault_ret. The address of the restart code ipc_fault_ret is loaded into t3 as the "exception PC" (Line 9), and the "exception SP" into register t2 (Line 24); these will further down be used to set up an "exception stack" (Lines 34–37). We are now ready to handle this like any user page fault (but we use a different restart routine from other faults). The code jumps to Label 5, which is at Line 30 of Listing 4.9.

Restart after long IPC page fault: `ipc_fault_ret`

The TLB exception handlers, prior to invoking the sender's pager for handling a page fault during long IPC, push the address of this function as the post-pager-IPC resumption code (Line 9 of Listing 4.10 and Line 36 of Listing 4.9).

```
 0          tcbtop(s2)
 1          lw      t0, T_STACKED_FINE_STATE-TCBO(s2)
 2          sw      t0, T_FINE_STATE-TCBO(s2)
 3          sd      zero, T_STACKED_FINE_STATE-TCBO(s2)
 4          ld      s3, T_STACKED_COMM_PRTNR-TCBO(s2)
 5          sd      s3, T_COMM_PARTNER-TCBO(s2)
 6          ld      s0, (sp)
 7          daddiu  sp, sp, 8
 8          andi    t0, v0, L4_IPC_ERROR_MASK
 9          bne     t0, zero, 1f
10          ld      a0,  T_GPT_POINTER-TCBO(s2)
11          move    a1, s0
12          jal     vm_lookup_pte
13          beq     v0, zero, 1f
14          lw      a0, (v0)
15          and     t1, a0, EL_Valid
16          beq     t1, zero, 1f
17          j       other_excpt_ret
18  1:      daddiu  sp, s2, 1 - 24
19          dli     v0, L4_IPC_RESNDPFTO
20          move    t8, s2
21          move    t9, s3
22          ld      v1, T_MYSELF-TCBO(t8)
23          b       send_only_short
```

Listing 4.11: Resumption code after pager IPC: `ipc_fault_ret`

Lines 0–5 restore the sender's state which was saved in the TCB prior to the pager IPC. Lines 6–7 restore the original fault address, which was stacked by the TLB exception handler (Lines 23, 25 above). The return value of the IPC is checked, diverting to the error code at the end if the IPC failed (Lines 8–9).

Lines 10–12 call the page table lookup function to check wether a mapping now exists for the page (as expected from a successful pager invocation), and, if found, the validity of the page table entry is verified (Lines 13–16). If there is a valid entry, `other_excpt_ret` is invoked to perform a return-from-exception and continue long IPC processing (Line 17).

Lines 18–23 constitute the cleanup code for the case of an unsuccessful pager invocation. The stack is unwound back to the original frame set up by the system call handler (Line 18). The return value is set to indicate a page-fault timeout on the sender's side (Line 19), and the register conventions expected by the short IPC code (see Table 5.1) are re-established (Lines 20–22). The code then branches to `send_only_short` to perform delivery of the register message.

TLB misses in temporary mapping area: `window_fault`

The `window_fault` code shown in Listings 4.12– 4.15 is invoked from the TLB exception handlers `exc_tlbl`, `exc_tlbs`, `exc_mod` if the fault address is in kernel space but outside the TCB array (Line 7 of Listing 4.10). Such a miss address indicates a page fault in the temporary mapping area during long IPC (see Listing 5.17).

Line 0 loads the faulting thread's ID into `t0` (`a0` points to the top of the faulter's TCB). Lines 1–6 compute from the local thread number the base of the thread's slot in the temporary mapping area (compare this with the C code in `do_long_ipc`, Listing 5.17, which sets up the temporary mapping area).

Line 7 subtracts this from the fault address (left in register `s3` by `other_excpt`). This yields the offset of the fault address from the base of the mapping window. Lines 8–9 add the window's base address in the receiver's address space (left in the TCB field `wdw_map_addr` by `do_long_ipc`). The result (in `s5`) is the fault address in the receiver's address space, which is what is needed to look up the mapping in the receiver's page table.

Lines 10–13 call `vm_lookup_pte` to find the mapping for the fault address. The function's parameters are the base address of the receiver's page table (`a0`) and the fault address (`a1`). The function's implementation is very similar to the STLB miss handler `tlb2_miss` discussed in Section 4.1.2. The difference is in register usage (`k0`, `k1` cannot be used here), and the calling convention (`tlb2_miss` obtains the fault address from CP0 and reloads the TLB and the STLB, which `vm_lookup_pte` does not do). The return value of `vm_lookup_pte` is the address of the `ENTRYLO` word containing the mapping. Note that this is a 32-bit entity.

Line 14 tests the return value for zero, indicating that no mapping exists. A non-existing mapping requires invocation of the receiver's page fault handler, which is done below starting at Line 56.

Lines 15–18 check whether the page table entry is *valid* and *writable* (i.e., has the "dirty" bit set). If not, the receiver's pager must be called after all. Line 23 constructs the correct `ENTRYHI` value containing the sender's ASID (Lines 19–20) and the VPN2 (half of the faulting page number, Lines 21–22).

Lines 24–30 check whether the TLB already contains a mapping for `ENTRYHI`. If not, indicated by a negative value left in `C0_INDEX` after the TLB probe instruction, execution diverts to Line 42.

A matching entry can exist either because the same entry was loaded earlier and then invalidated, or, more likely,

```
0           ld       t0, T_MYSELF-TCBO(a0)
1           dli      t1, TID_THREAD_MASK
2           and      t0, t0, t1
3           dsrl     t0, t0, TID_THREAD_SHIFT
4           dsll     t0, t0, RECV_WINDOW_SHIFT
5           dli      t1, RECV_WINDOW_BASE
6           daddu    t0, t1, t0
7           dsubu    a1, s3, t0
8           ld       a2, T_WDW_MAP_ADDR-TCBO(a0)
9           daddu    s5, a1, a2
10          ld       a2, T_COMM_PARTNER-TCBO(a0)
11          ld       a0, T_GPT_POINTER(a2)
12          move     a1, s5
13          jal      vm_lookup_pte
14          beq      v0, zero, 1f
15          lw       a0, (v0)
16          li       t0, EL_Valid|EL_Dirty
17          and      t1, a0, t0
18          bne      t1, t0, 1f
19          tcbtop(t8)
20          ld       a1, T_ASID-TCBO(t8)
21          dli      t1, ~(8192-1)
22          and      a2, s3, t1
23          or       a3, a2, a1
```

Listing 4.12: Mapping window TLB misses: `window_fault`, first part.

because a mapping exists for the faulting page's buddy. Remember that on the R4x00 a TLB entry always maps a pair of pages.

Lines 27 and 32 test whether the faulting page number is even or odd, corresponding to the TLB's `ENTRYLO0`, `ENTRYLO1` words respectively. The matched TLB entry is read (Line 31), the `ENTRYLO` word constructed in Line 23 is written to the appropriate coprocessor register (Lines 34, 37) and the new entry is loaded into the TLB (Line 39). We can now return from the exception (via `other_excpt_ret`).

Lines 42–55 deal with the case that the TLB does not yet contain a matching entry. It is a straigthforward variation of Lines 32–41, except that now both `ENTRYLO` words are set, one as above, the other to zero (for an invalid mapping).

```
24          dmtc0   a3, C0_ENTRYHI
25          nop
26          tlbp
27          andi    t0, s3, 4096
28          mfc0    t2, C0_INDEX
29          bltz    t2, 2f
30          nop
31          tlbr
32          beq     t0, zero, 3f
33          nop
34          dmtc0   a0, C0_ENTRYLO1
35          b       4f
36          nop
37  3:      dmtc0   a0,  C0_ENTRYLO0
38          nop
39  4:      tlbwi
40          j       other_excpt_ret
41          nop
42  2:      beq     t0, zero, 3f
43          nop
44          dmtc0   zero,  C0_ENTRYLO0
45          dmtc0   a0, C0_ENTRYLO1
46          nop
47          tlbwr
48          j       other_excpt_ret
49          nop
50  3:      dmtc0   a0,  C0_ENTRYLO0
51          dmtc0   zero,  C0_ENTRYLO1
52          nop
53          tlbwr
54          j       other_excpt_ret
55          nop
```

Listing 4.13: Mapping window TLB misses: `window_fault`, second part.

Note: The window mappings are truly "temporary", they are entered into the TLB but not into any page tables, not even the TLB cache. This is appropriate, as they are, by definition, very short lived and unlikely to be replaced from the TLB while still active. The cost of adding the entries to, and removing them from, the page table would not be justified.

Line 56 is reached if the destination buffer is not presently mapped and the copy operation therefore triggers a user-visible page fault in the receiver's address space. This fault must be handled by sending a message to the

```
56 1:      daddiu  sp, sp, -8
57         dla     a0, window_fault_ret
58         sd      a0, (sp)
59         tcbtop(t8)
60         li      t0, FS_LOCKS
61         sw      t0, T_FINE_STATE-TCBO(t8)
62         sd      sp, T_STACK_POINTER-TCBO(t8)
63         ld      t8, T_COMM_PARTNER-TCBO(t8)
64         ld      sp, T_STACK_POINTER(t8)
65         ld      t0, T_ASID(t8)
66         bgez    t0, 9f
67         dmtc0   t0, C0_ENTRYHI
68         jal     asid_get
69         nop
70 9:      ld      t0, T_GPT_POINTER(t8)
71         lui     t9, KERNEL_BASE
72         sd      t0, K_GPT_POINTER(t9)
73         li      t0, FS_LOCKR
74         sw      t0, T_STACKED_FINE_STATE(t8)
75         ld      t0, T_COMM_PARTNER(t8)
76         sd      t0, T_STACKED_COMM_PRTNR(t8)
77         li      a2, 0x01010000
78         lw      t1, T_TIMEOUT(t0)
79         lw      a3, T_TIMEOUT(t8)
80         andi    a3, a3, 0xff00
81         andi    t1, t1, 0xf000
82         srl     t1, t1, 8
83         or      a2, a2, t1
84         srl     t1, t1, 4
85         or      a2, a2, t1
86         or      a2, a2, a3
```

Listing 4.14: Mapping window TLB misses: `window_fault`, third part.

receiver's pager from the receiver's context.

The address of the continuation code for the sender, `window_fault_ret` is pushed onto the stack (Lines 56–58). The sender's state is set to `LOCKS`, indicating it is blocked during a send operation. (The previous state would have been `LOCKS|BUSY`.)

Lines 62–72 perform a context switch to the receiver. This code is analogous to `thread_switch_fast` (Listing 5.3). The difference is that the `k` registers cannot be used here as this code is pre-emptible. Instead general-purpose registers can be used, as user state is already saved.

Lines 73–86 are completely analogous to Lines 10–22 of Listing 4.10 (the TLB exception handler), with two exceptions: in Line 73 the (receiver) thread's state is explicitly set to `LOCKR` (blocked during receive operation), and in Line 82 the sender's send-pagefault timeout value is used to construct the timeout value for the pagefault IPC.

```
87          daddiu  t2, sp, -16
88          daddiu  sp, sp, -40
89          dla     t3, 1f
90          mfc0    t0, C0_STATUS
91          sb      t0, (sp)
92          sd      t3, 8(sp)
93          sd      t2, 16(sp)
94          dli     s0, ~(L4_FPAGE_RW_MASK | L4_FPAGE_GRANT_MASK)
95          and     s0, s0, s5
96          ori     s0, s0, L4_FPAGE_RW_MASK
97          move    s1, zero
98          sd      s5, 24(sp)
99          sd      s3, 32(sp)
100         tcbtop(t8)
101         dli     a0, L4_IPC_SHORT_MSG
102         dli     a1, L4_IPC_SHORT_FPAGE|(L4_WHOLE_ADDRESS_SPACE<<2)
103         ld      a4, T_PAGER_TID-TCBO(t8)
104         j       k_ipc
105         move    a5, a4
106 1:      ld      s0, (sp)
107         ld      s1, 8(sp)
108         daddiu  sp, sp, 16
109         tcbtop(t8)
110         lw      t0, T_STACKED_FINE_STATE-TCBO(t8)
111         sw      t0, T_FINE_STATE-TCBO(t8)
112         sd      zero, T_STACKED_FINE_STATE-TCBO(t8)
113         ld      t1, T_STACKED_COMM_PRTNR-TCBO(t8)
114         sd      t1, T_COMM_PARTNER-TCBO(t8)
115         li      t0, FS_LOCKS | FS_BUSY
116         sw      t0, T_FINE_STATE(t1)
117         lui     t2, KERNEL_BASE
118         thread_switch_fast(t8, t1, t2)
```

Listing 4.15: Mapping window TLB misses: `window_fault`, final part.

Lines 87–105 are similar to Lines 34–50 of Listing 4.9 (TLB exception handler). The main difference is that here the fault address and its equivalent in the receiver's address space is saved (Lines 87, 98–99) above the "exception stack frame", for later continuation. The "exception PC" stacked (Line 92) is actually the address of Line 106 below. The `syscall_ret` executed by the IPC code will therefore return to that line. The `k_ipc` entry point is invoked to deliver the page-fault IPC (Line 104).

Line 106 is invoked by the IPC's `syscall_ret` as discussed above. The fault address in the receiver's address space (Line 106) and the original fault address in the temporary mapping window (Line 107) are restored from the stack. The receiver's state is restored (Lines 109–114) in analogy to Lines 0–5 of `ipc_fault_ret` (Listing 4.11). The sender's state is set back to `LOCKS|BUSY`, indicating that it is in the process of delivering long IPC (Lines 115–116) and a context switch is performed back to the sender. This will leave the receiver blocked on the IPC, until resumed by `send_only_short`. The sender will resume execution at the restart address, `window_fault_ret`, stacked in Line 57.

Note that a fast thread switch is possible here as the register message has not yet been delivered to the receiver. It is therefore ok to trash the receiver's registers. The sender's registers were stacked by the general exception handler, and will be restored upon return from exception.

`window_fault_ret`

The `window_fault_ret` code shown in Listings 4.16–4.17 is invoked by the context switch at the end of the `window_fault` routine. The invocation is a result of a receiver-side page fault, which was handled by invoking the receiver's pager. This code then restarts the long IPC (remember, all IPC is performed in the sender's context).

The routine is called with the return status of the page fault IPC in `v0`, and the fault address, and the receiver's equivalent of the fault address, in registers `s1`, `s0`, respectively.

Line 0 pops the restart address (its own address) off the stack. Lines 1–4 check for the success of the IPC, and divert to Line 49 for error handling if it failed. Lines 5–16 are essentially identical to Lines 11–23 of `window_fault` (Listing 4.12), except for the different fault handling code (Label 1, Line 49 above or Line 56 of `window_fault`).

Implementation criticism: Line 12 aborts the IPC if a page fault is handled by an invalid or read-only mapping, instead of taking a repeated fault. This behaviour is not strictly correct, the IPC should only be aborted when the page fault timeout is reached. However, the definition of page fault timeouts implies that the timeout is restarted with each repeated fault, so that timeouts do not help in this case. Hence the present implementation is probably the best that can be done to stop denial-of-service attacks. Note that this problem does not exist in the `window_fault` code.

Lines 17–48 are identical to Lines 24–55 of `window_fault` (Listing 4.13), except for a different register assignment affecting Line 20.

Implementation criticism: Lines 17–48 should be replaced by a branch to the appropriate `window_fault` code, the different register assignment could easily be fixed. The run-time overhead introduced by the backwards branch would most likely be offset by a reduction of the number of cache missess resulting from denser code.

```
 0        daddiu  sp, sp, 8
 1        tcbtop(s2)
 2        ld      s3, T_COMM_PARTNER-TCBO(s2)
 3        andi    t0, v0, L4_IPC_ERROR_MASK
 4        bne     t0, zero, 1f
 5        ld      a0,  T_GPT_POINTER(s3)
 6        move    a1, s0
 7        jal     vm_lookup_pte
 8        beq     v0, zero, 1f
 9        lw      a0, (v0)
10        li      t0, EL_Valid|EL_Dirty
11        and     t1, a0, t0
12        bne     t1, t0, 1f
13        ld      a1, T_ASID-TCBO(s2)
14        dli     t1, ~(8192-1)
15        and     a2, s1, t1
16        or      a3, a2, a1
17        dmtc0   a3, C0_ENTRYHI
          ...
47        j       other_excpt_ret
48        nop
```

Listing 4.16: Sender-side continuation after receiver-side page fault: `window_fault_ret`.

```
1:      daddiu  sp, s2, 1 - 24
        dli     v0, L4_IPC_RERCVPFTO
        move    t8, s2
        move    t9, s3
        ld      v1, T_MYSELF-TCBO(t8)
        b       send_only_short
```

Listing 4.17: Error handling code of `window_fault_ret`.

Lines 49–54 handle unsuccessful page-fault IPC exactly as Lines 18–23 of `ipc_fault_ret` (Listing 4.11) except that the error code is set to indicate a page-fault timeout on the receiver's side.

4.2.7 Coprocessor unusable exception

The *coprocessor unusable exception* is handled by the `exc_cpu` routine. This exception occurs as a result of a user thread attempting to access a coprocessor. L4/MIPS presently only supports two of the possible four coprocessors:

- CP0, the system coprocessor (incorporating the MMU) is never enabled for user mode. Any attempted access is treated as a "normal" user exception and handled by invoking the user's excepter.
- CP1, the FPU, is enabled on demand. The first user-mode access will trigger the `exc_cpu` exception, which L4 handles by enabling the FPU and returning to the user.

Saving and restoring FPU state (a total of 33 registers) is an expense which the kernel tries to avoid as much as possible, by keeping the FPU disabled by default (compare `syscall_ret`, Listing 4.4). It keeps track of which thread has last used ("owns") the FPU. If an exception happens as a result of a user trying to use the FPU, the kernel checks whether that thread owns the FPU already. If not it saves and restores FPU state prior to enabling access.

```
        lbu     t0, 224(sp)
        andi    t0, t0, ST_KSU
        beq     t0, zero, 3f
        move    t2, sp
        li      t0, CA_CE_MASK
        and     t1, s4, t0
        li      t0, CA_CE_FP
        bne     t1, t0, exc_user
        nop
        mfc0    t0, C0_STATUS
        li      t1, ST_CU1
        or      t0, t0, t1
        mtc0    t0, C0_STATUS
        lui     a0, KERNEL_BASE
        tcbtop(t8)
        ld      a1, K_FP_THREAD(a0)
        ld      a2, T_MYSELF-TCBO(t8)
        beq     a2, a1, 2f
        nop
```

Listing 4.18: Coprocessor-unusuable exception handler: `exc_cpu`, first part.

```
19          tid2tcb(a1,a3)
20          ld      a4, T_MYSELF(a3)
21          bne     a4, a1, 1f
22          nop
23          cfc1    t2, $31
24          nop
25          sd      t2, T_FP_CONTROL(a3)
26          sdc1    $f0,T_FP_REGS+0(a3)
            ...
57          sdc1    $f31,T_FP_REGS+248(a3)
58  1:      ldc1    $f0,T_FP_REGS+0(a3)
            ...
89          ldc1    $f31,T_FP_REGS+248(a3)
90          ld      t2, T_FP_CONTROL(a3)
91          ctc1    t2, $31
92          sd      a2, K_FP_THREAD(a0)
```

Listing 4.19: Coprocessor-unusuable exception handler: `exc_cpu`, second part.

This lazy saving of FPU state works well unless there is a thread which uses the FPU heavily and also performs very frequent system calls. Such a thread would trigger a coprocessor-unusuable exception after every system call, resulting in significant overhead. However, such a behaviour is unusual for heavy users of floating point operations, and the approach chosen works well in practice (as indicated by the SPEC-FP benchmarks presented in [Elp99b]).

Listings 4.18–4.20 show the `exc_cpu` routine. Remember, this is invoked by `other_excpt` with the exception PC in `s2`, the exception virtual address in `s3` and the value of the Cause register in `s4`.

Lines 0–3 check whether the exception occurred in kernel mode, if yes, it jumps to Label 3 (Line 133) to cause a kernel panic, as the kernel does not use the FPU, and cannot trigger this exception for the system coprocessor.

Lines 4–8 examine CE (coprocessor number) field in the Cause register. Any value other than one (indicating CP1, i.e, the FPU) leads to diverting to `exc_user`, which invokes the user's excepter. Lines 9–12 enable the FPU, so the kernel can access its registers.

Lines 13–18 check whether the excepting thread is already the "owner" of the FPU. If so, no further action is required other than returning to the user without disabling the FPU. This is done from Line 93 on.

Line 19 locates the TCB of the present owner of the FPU. The TID recorded in that TCB is then compared to the TID of the owner (Lines 20–24). If there is no match, the original owner no longer exists (its task was killed in the meantime) and saving of FPU state can be skipped.

Lines 25–57 save the FPU state in the previous owner's TCB. FPU state consists of the *FPU Control Register* and 32 general floating point registers. The FPU Control Register is copied to a general purpose register by the `cfc1` (*move control word from CP1*, Line 23) instruction and subsequently stored in the TCB. The general floating point registers can be stored directly to memory by the `sdc1` (*store double word from CP1*) instruction (Lines 26–57).

Lines 58–92 perform the corresponding restore of FPU state from the excepting thread's TCB. That thread is then recorded as the new owner of the FPU (Line 92).

Bug/Restriction 3: FPU state incorrectly restored.
The code in Lines 58–91 restores the FPU state from the previous owner's context, rather than the excepting thread's. The effective address `T_FP_REGS+0(a3)` should read `T_FP_REGS-TCBO+0(t8)` etc.

```
 93 2:      ld      ra, (sp)
 94         ld      s8, 8(sp)
            ...
120         ld      AT, 216(sp)
121         mfc0    k0, C0_STATUS
122         ori     k0, k0, ST_EXL
123         mtc0    k0, C0_STATUS
124         li      k1, 0xffffff00
125         and     k0, k0, k1
126         lbu     k1, 224(sp)
127         or      k0, k0, k1
128         ld      k1, 232(sp)
129         dmtc0   k1, C0_EPC
130         mtc0    k0, C0_STATUS
131         ld      sp, 240(sp)
132         eret
133 3:      dla     a0, kern_exc_msg
134         j       panic
```

Listing 4.20: Coprocessor-unusuable exception handler: `exc_cpu`, final part.

The remainder of the code is straightforward. It is almost identical to `other_excpt_ret`, the general exception return code: restore registers (Lines 93–120), followed by the equivalent of `syscall_ret`. The only difference is in Line 124: Contrary to Line 3 of `syscall_ret` (Listing 4.4) this line does *not* mask out the coprocessor-enable bits, and thus leaves the FPU enabled for user code.

Lines 133–134 performs the kernel panic resulting from a coprocessor exception in kernel mode.

Chapter 5

IPC Path

5.1 Introduction

IPC is the “heart” of L4, and its efficiency of paramount importance to any L4-based system. This applies in particular to the “short” IPC path, which, as a consequence, is highly optimised.

IPC is considered “short” if it only passes a “short” message, i.e., only uses registers to transmit data. “Short” IPC does not involve fpage mappings or messages in memory buffers (direct or indirect strings). Note that the distinction between “short” and “long” IPC has nothing to do with whether the operation is between local partners (threads of the same task) or not, but has to do with the time it takes to perform.

L4 IPC is blocking, hence an IPC can only be started when one of the partners is currently blocked (waiting for the IPC to happen) and the other is running (trying to perform the IPC). Most IPC processing is done in the sender’s context. In the case of a sender being originally blocked, waiting for the receiver to be available, and the IPC is consequently initiated from the receiver’s context, this implies a context switch to the sender before any other processing. At the end a context switch is performed from the sender to the receiver, which delivers the register part of the message to the receiver. The present implementation always continues the receiver first after a successful IPC.

Implementation criticism: This is not always the correct behaviour, see comments in Section 5.3.1.

Remember, L4 terminology calls the state of a thread blocked on a send “polling” or “pending”, while a thread blocked on a receive is considered “waiting”. The latter can be an “open” wait, if the thread is willing to receive from any thread, or a “wait-for” if it is trying to receive from a specific partner.

The thread state is recorded in `tcb.fine_state`. The distinction between open wait and wait-for is by the `tcb.wfor` field, which is zero for an open receive.

5.2 Short IPC

The short IPC code is contained in `kernel/exc.S:k_ipc`, which is called by the general exception handler `gen_exc`. The code uses a register convention and introduces mnemonic aliases for a number of them, as shown in Table 5.1. The *type* indicates whether a register is an input (I), output (O), input and output (I/O) to the system call or a temporary (T). I/O registers are delivered to the receiver unchanged. The `stcb` register is set up by the general exception handler prior to invoking `k_ipc`, the other temporaries and outputs are set up by the `k_ipc` code prior to the `deliver` label.

alias	*standard*	*type*	*usage*
`sdesc`	`a0`	I	send descriptor
`rdesc`	`a1`	I	receive descriptor
`timeout`	`a2`	I	timeout struct for IPC
`dthrd`	`a4`	I/O	(intended) destination TID
	`a3`	T	various
`wfor`	`a5`	I	wait-for TID
`vsend`	`a6`	I	virtual sender TID
	`s0–s7`	I/O	register message
	`v0`	O	result word
	`v1`	O	sender TID (may be deceived)
`stcb`	`t8`	T	source TCB pointer (+TCBO)
`dtcb`	`t9`	T	(actual) destination TCB pointer

Table 5.1: Register usage and naming convention in IPC code.

5.2.1 Send & receive: `k_ipc`

The `k_ipc` code is shown in Listings 5.1 and 5.2. The function contains the code for a complete short IPC message delivery in the sender's context. Send-only and receive-only IPC operations use separately optimised code (`send_only_short`, `receive_only`, respectively), which get invoked from `k_ipc` if appropriate.

```
        bltz    sdesc, receive_only
        tid2tcb(dthrd, dtcb)
        ld      t0, T_MYSELF(dtcb)
        ld      v1, T_MYSELF-TCBO(stcb)
        lw      t3, T_FINE_STATE(dtcb)
        xor     t1, t0, v1
        dsll    t1, 4
        dsrl    t1, 53
        bne     t1, zero, to_chief
        move    v0, zero
return_to_chief1:
        bne     t0, dthrd, invalid_dest
return_to_chief2:
        andi    t3, t3, FS_WAIT
        bne     sdesc, zero, ipc_long
        nop
        beq     t3, zero, pending
        ld      t2, T_WFOR(dtcb)
        beq     t2, zero, deliver
        nop
        bne     v1, t2, pending
        nop
deliver:
```

Listing 5.1: Prologue of `k_ipc`.

Prologue

Listing 5.1 shows the prologue of the IPC code. It performs various validity checks: is the destination TID valid, is redirection required, is an attempted deceit legal. None of these are relevant for receive-only IPC: redirection and deception do not apply to receive operations, and receiving from an invalid thread is legal, and is in fact the way sleeps are implemented in L4. A receive from a non-existent thread is guaranteed to block the caller until the specified timeout is exhausted.

Line 0 consequently diverts to `receive_only` if the system call does not request a receive operation, as indicated by a nil (-1) send descriptor.

Line 1 uses the `tid2tcb` macro to convert the destination TID into a TCB pointer. That macro uses the concatenated task and thread numbers from the TID as an index into the TCB array. Note that the `tid2tcb` macro expands into five instructions (none of which access memory). The first of these falls into the branch delay slot of Line 0. Its result is ignored if the branch is taken.

Lines 2–3 load the TIDs of source and destination. The source (caller's) TID was not known before, and the destination TID, although supplied by the caller, cannot be trusted. The caller could be using an incorrect destination TID (a thread which has not yet been created, or a task with an incorrect version number). Also the code requires the `chief` field in the TID, which it cannot trust in a user-supplied TID.

Here it becomes obvious that the source (`stcb`) and destination (`dtcb`) TCB pointers are aligned differently: The former has been obtained by masking in the least significant bits of the (kernel) stack pointer and points to the end of the thread's TCB, while the destination TCB pointer has been obtained from the TID and points to the beginning of the respective TCB. Consequently, `TCBO` must be subtracted from all `stcb` offsets. This approach saves one cycle in the shortest IPC path.

Note that Line 2 may result in a page fault, if the caller has supplied an invalid TID which was not referenced before. The kernel page fault handler (in `exc_tlbl`, see page 34) will handle this by establishing a mapping to the Invalid TCB, which a TID of zero, inconsistent with whatever entry in the TCB array it is mapped to. The only thread with a task number of zero and a thread number of zero is the idle thread. It is intentionally given an "inconsistent" TID to make sure it can never be the destination of an IPC operation.

Lines 5–8 check whether source and destination are part of the same clan (by checking whether the chiefs are the same). If not, the IPC may need to be redirected, and `to_chief` is called to determine the real destination. That function returns to Line 10 or Line 12.

Line 9 sets up the return value (optimistically) as "successful, undeceived, not redirected, no mappings". Note that this line is in the branch delay slot of Line 8. The `to_chief` function modifies `v0` as appropriate if it finds that the IPC is to be redirected.

Line 11 tests whether the caller-supplied destination TID agrees with the one recorded in the TCB corresponding to the task and thread number in the user-supplied TID. If not, `invalid_dest` is invoked, which sets an appropriate error code in the return value and returns via `syscall_ret`. As a consequence of what was said for Line 2, the branch will be taken if the user supplied the TID of a non-existing thread. Line 13 is in the branch delay slot, but its execution is irrelevant if the branch is taken.

The test in Line 11 is skipped by the `to_chief` function if redirection is required, as `to_chief` has already verified a valid receiver.

Line 14 diverts to `ipc_long` if the send operation includes memory messages, mappings or deceiving.

Lines 4, 13, 16–20 check whether the message can be delivered without blocking. This requires that the receiver is in the WAIT state (Line 16) and the wait-for partner is zero (indicating an open receive, Lines 17–18) or equal to the sender (Line 20). Otherwise the caller must enter the PENDING state, which is done in the `pending` routine.

The `nop`s in Lines 15, 19, 21 are there to make unfilled branch delay slots obvious, as an aid in cycle-counting. All other delay slots are utilised. If there are no cache misses, this code executes in 20 cycles (open receive) or 22 cycles (closed receive). Lines 2 and 17 may miss on different cache lines in the destination TCB, and Line 3 may cause a cache miss in the source TCB.

Delivery

Having made it to Line 22 we now know that we are ready to deliver the message. All registers listed in Table 5.1 are set up (although the value of `v0` might still change if something goes wrong). This label is jumped to by some other parts of the IPC code, once they are ready to do the delivery: `ipc_long` to do the "fast" bit after processing mappings and memory operations, and `pending_restart` when a sender becomes unblocked.

Line 23 diverts to `send_only_short` if there is no receive part in the IPC, so the *send&receive* and pure send can be optimised independently. The instruction in the branch delay slot is irrelevant if the branch is taken.

The reminder of the function contains the "magic" of L4 IPC. At the very end (line 52) a limited context switch is performed to the receiver, leaving most of the sender's general purpose registers unchanged. This is partially where L4 IPC gets its high performance from: The limited context switch reduces the amount of context that needs to be saved and restored, and at the same time transfers part of the message (the register message in `s0`–`s7`).

```
22 deliver:
23          bltz    rdesc, send_only_short
24          ori     t0, zero, FS_BUSY
25          sw      t0, T_FINE_STATE(dtcb)
26          bne     wfor, zero, 1f
27          lui     t1, KERNEL_BASE
28          ld      t0, T_SNDQ_START-TCBO(stcb)
29          beq     t0, zero, 1f
30          daddiu  sp, sp, -32
31          sd      rdesc,  24(sp)
32          sd      wfor,   16(sp)
33          sd      timeout, 8(sp)
34          dla     t0, sender_restart_receiving
35          sd      t0, (sp)
36          daddiu  t3, stcb, -TCBO
37          ins_busy_list(t3, t1, t0)
38          dli     t0, FS_BUSY
39          b       3f
40 1:       andi    s8, timeout, L4_RCV_EXP_MASK
41          beq     s8, zero, 2f
42          dli     t0, FS_WAIT
43          li      t0, FS_WAIT+FS_WAKEUP
44          daddiu  t3, stcb, -TCBO
45          receive_timeout(timeout, t2, t2)
46          ins_wakeup(t2, t3, t1)
47 2:       sd      rdesc,  T_RECV_DESC-TCBO(stcb)
48          sw      timeout, T_TIMEOUT-TCBO(stcb)
49          sd      wfor, T_WFOR-TCBO(stcb)
50 3:       sw      t0, T_FINE_STATE-TCBO(stcb)
51          thread_switch_fast(stcb, dtcb, t1)
52          syscall_ret
```

Listing 5.2: Delivery part of `k_ipc`.

The code in Lines 24–51 serves to set up that context switch. The caller/sender must be set up so it will complete its IPC (by performing the receive part) once it gets to execute again. Part of L4's context switch protocol (see `to_next_thread`, Listing 5.8) is that the top of the kernel stack of a runnable thread which is not currently executing points to the instruction where execution is to continue.

Lines 24–25 mark the receiver as BUSY, as the IPC will unblock it.

Lines 28–38 are executed only if the receive part of the caller's IPC operation (remember, we are doing a send and a receive operation with the same system call) is an open wait (i.e., we're really in a *reply&wait* call, but see further comments below).

The send-queue of the caller's TCB is checked for any pending sends to this thread (Lines 28–29). If there are any, the receive will be processed the next time the caller is dispatched. This is achieved by pushing the address of `sender_restart_receiving` as the restart address on the stack (Lines 30, 34–35). This function will perform the receive part of the IPC. The parameters for the restart function, the receive descriptor, wait-for TID (zero in this case) and timeout are pushed on the stack as well (Lines 31–33). When called `sender_restart_receiving` will simply pop these parameters off the stack and call `receive_only`.

Note that the timeout is irrelevant in this case: the source thread of the receive part of the IPC was already pending, so no receive timeout can occur. As well, we are in short-only IPC, so no page faults (with associated timeouts) can occur. However, the restart function is common to all cases and therefore takes a timeout parameter.

In Line 37 the caller is inserted into the busy list by the `ins_busy_list` macro (see Listing 7.3). Even though it is currently executing, it may not have been entered into the busy list, as it may be executing on time slice donated by the receiver (lazy scheduling). Note that the list insertion macros expect a properly aligned TCB pointer, so it is adjusted in Line 36 (similarly in Line 44).

The caller is then marked BUSY (Lines 38, 50) and the context switch is performed (Line 51). **Consequently, the `syscall_ret` macro at Line 52 returns not to the caller, but to the thread which received the caller's message.**

If the receive is not an open one we are assumed to be in a *call* IPC, send and receive from the same partner. The receive part of the IPC will necessarily block, until the partner gets to do its send operation to the caller thread (or a timeout occurs). Note that it is not necessary to stack a continuation address in this case: Once the receive part of the IPC is performed (as a send operation in the partner's context) it will, after message delivery, involve a (fast) context switch back to the caller, who has then nothing more to do than to return from the system call. This is done by the `syscall_ret` macro, which uses the exception stack frame, which is already on the top of the stack.

Since the partner is (by definition) not yet ready to send, the receive part of the IPC can time out. If a finite receive timeout is specified, the TCB is added to the wakeup queue (Lines 45–46) so it will be unblocked when the receive times out. The thread state is set to WAIT+WAKEUP to indicate that it is blocked but also in the wakeup queue. The zero timeout case is not handled separately: the thread cannot be dispatched anyway (as we are switching to the receiver) and processing can be left to the next time the wakeup queue is processed.

Implementation criticism: The code shows that a "zero" timeout really means a timeout of less than the timeout resolution (1ms).

Implementation criticism: If the receive is not an open one, the code assumes that it is to receive from the same partner (consistent with L4/ix86 Version 2 specification). However, the L4/MIPS IPC interface allows the specification of different TIDs for destination and wait-for. The above code should therefore check whether the wait-for thread's send is already pending, and threat this like the open receive. The present implementation will probably block until timeout and then return a failure status if the send is pending.

Lines 47–49 save the receive descriptor, timeout value and wait-for TID in the caller's TCB prior to switching context. Note that since the send phase is completed and the receive timeout already taken care of via the wakeup queue, the only remaining timeout value of interest at this stage would be the *receive page-fault timeout*. As we are in short IPC, that one is not ever used either.

The fastest path through this code, with an open receive (*reply&wait* semantics) and an infinite timeout, requires 12 cycles, plus the time taken by the `thread_switch_fast` and the `syscall_ret` macros. The fast path does not cause D-cache misses or pipeline stalls.

5.2.2 Fast context switch: `thread_switch_fast`

The last bit of IPC magic is hidden in the `thread_switch_fast` macro, defined in `kernel/macros.h`. It implements the limited context switch which is at the heart of the fast IPC. It expands into the code shown in Listing 5.3. (`t1` still contains the address of `kernel_vars`.)

Line 0 saves the caller's (= sender's) stack pointer in its TCB. That frees up `sp` to use as a temporary register, `AT` is also used as a temporary.

Line 1 loads the receiver's ASID value from its TCB. If this is non-negative (Line 3) it is defined (Line 4) as the new ASID for the MMU to use for matching TLB tags (see page 12).

The current page table is set to the one of the receiver thread (Lines 2, 12) and the stack pointer is set to the one saved in the receiver's TCB (Line 13) — this is the point where the context switch "really" happens. `kernel_vars.stack_bottom` is set appropriately (TCB address plus TCB size, Lines 13–14). This is to allow unwinding the stack later on.

```
0           sd      sp, T_STACK_POINTER-TCBO(stcb)
1           ld      sp, T_ASID(dtcb)
2           ld      AT, T_GPT_POINTER(dtcb)
3           bgez    sp, 255f
4           dmtc0   sp, C0_ENTRYHI
5           sd      AT, K_GPT_POINTER(t1)
6           ld      sp, T_STACK_POINTER(dtcb)
7           daddiu  AT, dtcb, TCB_SIZE
8           jal     asid_get
9           sd      AT, K_STACK_BOTTOM(t1)
10          b       254f
11          nop
12  255:    sd      AT, K_GPT_POINTER(t1)
13          ld      sp, T_STACK_POINTER(dtcb)
14          daddiu  AT, dtcb, TCB_SIZE
15          sd      AT, K_STACK_BOTTOM(t1)
16  254:
```

Listing 5.3: The `thread_switch_fast` macro.

A negative ASID value implies that the destination thread does not currently have an ASID allocated, and `asid_get` is called (Line 8) to obtain one, possibly preempting another task's ASID. Otherwise the code is the same as in the case of a valid ASID.

As the stack pointer is now switched to the destination's kernel stack, the `syscall_ret` macro (see page 30) will use the *receiver's* exception stack frame to restore processor status, including its PC. The final `eret` instruction will thus "return" to the receiver, completing the IPC operation for the receiver, and leaving the sender blocked until it can perform its own receive operation.

If the receiver thread has an ASID already allocated, the macro executes in 9 cycles, assuming no cache misses. All delay slots are utilised. Lines 1 and 2 may cause D-cache misses in different lines of the destination TCB.

5.2.3 Discussion

As can be seen from the above, only very minimal context is saved and restored. The sender thread gets restarted with very few of its registers in tact. Only the receive descriptor, the timeouts and the wait-for id will be restored, or just enough to perform the receive part of the IPC. The `s` registers will be overwritten by the receive anyway, and `v0` is implicitly known to be zero, as otherwise no receive would be attempted.

The other point to note is that, while the sender thread is put into the busy list to allow it to be scheduled again, the context switch to the receiver is actually performed without any scheduling (*lazy scheduling* [Lie93]). The receiver simply continues in the remainder of the sender's time slice. This is an instance of *time-slice donation* in L4.

The best-case execution time of the short IPC path (*reply&wait* semantics with infinite timeouts) is $25 + 20 + 12 + 9 + 14 = 80$ cycles. It was benchmarked at 99 cycles. The difference is most likely due to cache conflict misses. It should be possible to construe an example which avoids any cache misses (but what would it prove?)

Note that interrupts remain disabled throughout the whole short IPC path.

5.3 Other Short IPC Send Code

5.3.1 Non-blocking send: `send_only_short`

This code (Listing 5.4) is a simplified version of the delivery part of `k_ipc`. The main difference is that it does not need to check for a sender to match the receive phase of the IPC and that a different restart code, `send_only_short_restart`, is used.

Lines 8–14 mark sender and receiver both as BUSY. (This code is also invoked to finish up long IPC, during which the sender may have become blocked.)

Line 15 performs the context switch to the receiver and Line 16 returns (to the receiver).

Bug/Restriction 4: Priority inversion in `send_only_short`.
This unconditional context switch is incorrect. A send-only operation with waiting receiver should, after message delivery, continue executing the sender rather than donating the sender's time slice to the receiver [Elp99a].

```
0           daddiu  sp, sp , -16
1           dla     t0, send_only_short_restart
2           sd      t0, (sp)
3           andi    t1, v0, L4_IPC_ERROR_MASK
4           beq     t1, zero, 1f
5           sd      zero, 8(sp)
6           ori     t1, v0, L4_IPC_SND_ERR_MASK
7           sd      t1, 8(sp)
8   1:      dli     t0, FS_BUSY
9           sw      t0,  T_FINE_STATE-TCBO(stcb)
10          ori     t0, zero, FS_BUSY
11          sw      t0, T_FINE_STATE(dtcb)
12          daddiu  t3, stcb, -TCBO
13          lui     t1, KERNEL_BASE
14          ins_busy_list(t3, t1, t0)
15          thread_switch_fast(stcb, dtcb, t1)
16          syscall_ret
```

Listing 5.4: Send-only delivery: `send_only_short`.

5.3.2 Blocking send: `pending`

The `pending` routine (Listing 5.5) is called if the message cannot be delivered right away because the receiver is not ready for it. It is called from `k_ipc` and from `ipc_long`. Upon entry, `v1` contains the senders TID or, in the case of a *deceiving send*, the virtual sender ID specified by the caller in `a6`. Reordering by the assembler is enabled in this code.

Lines 0–6 check the send-timeout specified in the syscall. If it is zero the operation is aborted with an error status.

```
 0          andi    t2, timeout, L4_SND_EXP_MASK
 1          dsrl    t2, t2, 4
 2          beq     t2, zero, 1f
 3          send_timeout(timeout, t2, s8)
 4          bne     s8, zero, 1f
 5          dli     v0, L4_IPC_SETIMEOUT
 6          syscall_ret
 7  1:      bne     dthrd, v1, 1f
 8          dli     v0,  L4_IPC_ENOT_EXISTENT
 9          syscall_ret
10  1:      dli     t0, FS_POLL
11          lui     t1, KERNEL_BASE
12          daddiu  t3, stcb, -TCBO
13          beq     t2, zero, 2f
14          li      t0, FS_POLL+FS_WAKEUP
15          ins_wakeup(s8, t3, t1)
16  2:      sw      t0,  T_FINE_STATE-TCBO(stcb)
17          ins_sendq_end(t3, dtcb)
18          daddiu  sp,sp,-144
19          sd      sdesc, 8(sp)
20          sd      rdesc, 16(sp)
21          sd      timeout, 24(sp)
22          sd      dthrd, 32(sp)
23          sd      wfor, 40(sp)
24          sd      vsend, 48(sp)
25          sd      dtcb, 56(sp)
26          sd      dtcb, T_COMM_PARTNER-TCBO(stcb)
27          sd      s0, 64(sp)
28          sd      s1, 72(sp)
29          sd      s2, 80(sp)
30          sd      s3, 88(sp)
31          sd      s4, 96(sp)
32          sd      s5, 104(sp)
33          sd      s6, 112(sp)
34          sd      s7, 120(sp)
35          sd      v1, 128(sp)
36          sd      v0, 136(sp)
37          dla     t0, pending_restart
38          sd      t0, (sp)
39          lui     t0, KERNEL_BASE
40          to_next_thread(t0)
```

Listing 5.5: Blocking send code: `pending`.

IPC with zero timeout effectively polls the communication partner, as it can only succeed it the destination is already waiting.

Note that Lines 1 and 2 are logically inverted. The above order was probably chosen to prevent the assembler from inserting a `nop` in the branch delay slot.

Lines 7–9 abort with a status of *non-existent destination or source* (which here really means "invalid destination"). While receives from invalid partners are legal (and used to implement timed sleeps), send operations are only considered legal if a message can actually be delivered. It is therefore legal to attempt to receive from oneself, but illegal to attempt to send to oneself. This includes deceiving sends pretending to come from the receiver.

Lines 10–16 insert the thread into the wakeup queue if the send timeout is finite, and set the thread state accordingly (POLL indicating pending send). The sender is inserted into the receiver's *send queue*, the list of pending send operations.

Bug/Restriction 5: Send queue not prioritised.
The send queue should be in priority order rather than FIFO.

The sender's state is stacked (Lines 18–36). Temporary registers and inputs are not saved. The `stcb` register is not stacked as it can be recomputed from the stack pointer using the `tcbtop` macro. The receiver TCB is also recorded in the `comm_partner` field of the sender's TCB.

In Lines 37–38 the address of `pending_restart` is pushed as the restart address. Lines 39–40 then use the `to_next_thread` macro to switch to another thread.

5.3.3 Short IPC send: odds & ends

Unblocking sender: `pending_restart`

The `pending_restart` code (Listing 5.6) first restores all the registers stacked by `pending` (not shown). The thread then removes itself from the destination TCB's send queue (Lines 1–2) and marks itself BUSY (Lines 3–4). Delivery is then performed by invoking `deliver` or `ipc_long_deliver` as appropriate.

Implementation criticism: A comment in `pending_restart` indicates that the code does not properly deal with the case where the receiver has been killed in the meantime. I think this comment is obsolete.

```
0          ...     /* pop registers */
1          daddiu  t3, stcb, -TCBO
2          rem_sendq(t3, dtcb, t0)
3          dli     t0, FS_BUSY
4          sw      t0, T_FINE_STATE-TCBO(stcb)
5          bne     sdesc, zero, ipc_long_deliver
6          b       deliver
```

Listing 5.6: Unblocking sender: `pending_restart`

Find redirection target: `to_chief`

The `to_chief` routine is called from `k_ipc` (Listing 5.1 Line 8) if it was found that the sender and destination had different chiefs, and redirection was therefore required. The routine is to determine the actual destination of the IPC message, i.e., the appropriate chief. It is called with the sender's TID in `v1` and the receiver's TID in `t0`.

```
0          dsll    t1, t0, 32
1          xor     t1, t1, v1
2          dsll    t1, t1, 4
3          dsrl    t1, t1, 53
4          beq     t1, zero, return_to_chief1
5          dsll    t1, v1, 32
6          xor     t1, t1, t0
7          dsll    t1, t1, 4
8          dsrl    t1, t1, 53
9          beq     t1, zero, return_to_chief1
10         move    s8, v1
11         beq     t0, zero, invalid_dest
12         move    t2, v1
13         jal     ipc_nchief
14         move    v1, t0
15         tid2tcb(v1,dtcb)
16         lw      t3, T_FINE_STATE(dtcb)
17         xori    v0, v0, L4_IPC_SRC_MASK /* invert inner/outer */
18         b       return_to_chief2
19         move    v1, s8
```

Listing 5.7: Determining real destination: `to_chief`.

Lines 0–3 check whether the task of the intended receiver is the sender's chief (by comparing the task number in the receiver TID with the chief number in the sender TID). If so, delivery can go ahead to the intended receiver, and the routine returns (Line 4) to where it was called from (the `return_to_chief1` label in `k_ipc`). Similarly, Lines 5–9 return without changing the destination if the sender is the receiver's chief.

At Line 10 we know that proper redirection is required, and `ipc_nchief` is called in Line 13 to determine the actual destination. `ipc_nchief` is actually an entrypoint in `nchief`, the routine which implements most of the `id_nearest` system call (Listing 6.2). It skips the tests which were already done in `k_ipc` or `to_chief` (the code cannot be shared due to inconsistent register assignments, see Table 6.1).

The `ipc_nchief` code expects the (intended) destination TID in `v1` (Line 14) and the sender TID in `t2` (Line 12). It returns the actual destination (*nearest*) in `v1` and the *direction* in `v0`, exactly as the `id_nearest` system call. Prior to invocation the sender TID is saved in `s8` (Line 10).

Bug/Restriction 6: `a7` not initialised when `ipc_nchief` is called..
The entry point `ipc_nchief` also expects the caller to set up `a7` to contain the source task number in the `chief` position (see Listing 6.2, Line 14). The invoking code in `to_chief` does not initialise `a7` at all.

Upon return from `ipc_nchief`, `k_ipc`'s register conventions are re-established (Lines 15, 16) to match the new destination. Line 17 sets up the IPC return code in `v0` **for the receiver** with the direction (type) field, which for the receiver is the inverse as for the sender (for whom it was evaluated by the `ipc_nchief` call). The sender TID is restored (Line 19) and the code returns to `k_ipc`.

Note that `ipc_nchief` will return *outer* as type and the caller's chief as *nearest* if invoked with an invalid destination ID. While this makes sense in the context of the `id_nearest` system call, it does not make sense to redirect a message destined to an invalid thread to the chief; the IPC is to be aborted in this case. Therefore the validity of the destination TID is checked prior to the call (Line 11). As the same check is performed by Line 11 of `k_ipc` after the redirection test, this part of `to_chief` returns to the `return_to_chief2` label of `k_ipc`, skipping the redundant test.

```
0         move    s0, kernel_base
1         move    a0, kernel_base
2         jal     get_next_thread
3         tcbtop(t0)
4         thread_switch_fast(t0, v0, s0)
5         ld      ra, (sp)
6         jr      ra
```

Listing 5.8: The `to_next_thread` macro.

Context switch on blocking: `to_next_thread`

The `to_next_thread` macro expands into the code shown in Listing 5.8.

Line 2 invokes the scheduler, `schedule.c:get_next_thread`, in order to select a runnable thread; its TID is returned in `v0`. The familiar `thread_switch_fast` macro is used to perform a minimal context switch. The new thread is restarted by invoking the restart routine which had been stacked earlier (Lines 34–35 of `k_ipc`, Listing 5.2) and which is responsible for restoring any required context.

5.3.4 Discussion

The advantage of having `to_next_thread` invoke a restart routine whose address was pushed to the stack prior to suspension is obvious: it allows reducing the context that needs to be saved and restored to the minimum required in the particular situation.

5.4 Short IPC Receive

5.4.1 `receive_only`

Sections 5.2 and 5.3 have presented the code for the send side of basic short IPC. The receive code, activated by `sender_restart_receiving` after the context switch to the receiver, is contained in the `receive_only` function. This function is also invoked from the `k_ipc` if the system call does not contain a send phase (Line 0 of Listing 5.1).

As pointed out on page 49, the receive code does not perform any validity checking of the source TID, as an attempted receive from a non-existing thread is allowed and has a well-defined semantics.

However, there is more, as user-level interrupt handling in L4 is implemented via IPC receive operations from virtual hardware threads. This makes the receive code more complicated (and worth looking at).

The code of `receive_only` is shown in Listings 5.9–5.11. At the time the function is invoked (by `k_ipc` or `sender_restart_receiving`) only the IPC arguments provided by the user and `stcb` are set up, all other registers in Table 5.1 are undefined.

Lines 0–1 check whether a send operation is already pending for the caller, if so, execution is diverted to `pending_receive_only` (Listing 5.12) which will deliver the message if possible, and otherwise return to Line 2 (if the wait-for partner is not in the queue).

We now know that no IPC message can be delivered immediately. If the receive timeout is finite (Lines 7–8) we go straight to the epilogue (Line 112) to set up the wakeup and block the caller. If the timeout is infinite (Lines 5–6) wakeup processing is skipped by jumping to Line 115 in the epilogue.

Interrupt association

At this stage in the receive operation we know that we have a zero timeout, and that the partner is not ready to receive the message. This would normally imply that the IPC has timed out. However, L4 uses IPC also for invoking user-level interrupt handlers, and a zero-timeout receive from an *interrupt thread* is used to associate the caller thread with this interrupt, i.e., register the caller as the interrupt handler. Note that we will not get here as long as an interrupt message is pending, so (dis)association is only effective when there are no pending interrupts.

The MIPS supports 8 different interrupts, numbered 0–7, each corresponding to a bit in the interrupt mask `IM` in the status register. Of these, interrupts 2–6 are available to user code, the others are used by the kernel itself (e.g., timer interrupt). ***Check!***

L4 models these user-visible interrupts as virtual threads, each having a TID and a TCB. The five user-visible interrupts are mapped to TIDs 1–6.

The code first checks whether the specified TID refers to an interrupt (TID¡8, Lines 9–10), if not, the IPC is aborted with a timeout (Lines 110–111).

We now know that the caller attempts interrupt association. First, any previous interrupt association of the caller is removed by zeroing its interrupt mask in the TCB (Line 11), and removing the caller from the list of iterrupt handlers in `kernel_vars` (Lines 12–31).

Implementation criticism: Any attempt to associate with an interrupt, whether successful or not will always remove any previous association. This is in conflict with the reference manual [EHL99]. It is probably better to consider the behaviour of the code correct and document it in the manual.

If the destination TID specified by the caller is zero, then interrupt dissociation is all the caller wanted, and the call returns with a timeout status (Lines 32, 110–111). Otherwise the caller wants to associate with a new interrupt. This is possible if the interrupt is presently free (unassociated).

Each interrupt is now checked by the same procedure. The wait-for value is decremented, and, if zero, would refer to user-visible interrupt number 0, which is hardware interrupt number 2 (Lines 33–34). Lines 35–36 check whether this interrupt is free, which is indicated by a zero TID having been recorded in the `int0_thread` field of `kernel_vars`. If not, the operation is aborted with a status *non-existing partner* (Lines 108–109).

We now know that the requested interrupt is free and can proceed with associating the caller with it. The interrupt mask corresponding to hardware interrupt number 2 is recorded in that caller's TCB (Lines 37–38).

Line 39 loads the address of the TCB of the virtual interrupt thread. Note that the constant `INT0_TCB_BASE` (defined in `include/kernel/kernel.h`) points to the top of the TCB, the bottom of the interrupt stack, rather than the top of the TCB, which is the reason for masking out the least significant bits in the address.

Lines 40–45 copy the scheduling parameters (priority and time slice length) of the caller to the interrupt TCB. This allows the kernel to deal with interrupts according to the priorities of their handlers; the virtual interrupt thread

```
0          ld       t0, T_SNDQ_START-TCBO(stcb)
1          bne      t0, zero, pending_recv_only
2  leave_waiting:
3          dli      t0, FS_WAIT
4          lui      t1, KERNEL_BASE
5          andi     t3, timeout, L4_RCV_EXP_MASK
6          beq      t3, zero, 2f
7          receive_timeout(timeout, t3, t9)
8          bne      t9, zero, 1f
```

Listing 5.9: Prologue of `receive_only`.

effectively inherits the handler's scheduling parameters. The caller's TID is recorded as the handler of user-visible interrupt 0 (hardware interrupt 2) in `kernel_vars.int0_thread` (Line 46) and the syscall returns with a timeout status (Lines 47, 110–111).

Lines 48–107 perform the corresponding actions for the other interrupts. Note that specification of a non-existent interrupt (TID=7) will correctly lead to a timeout abort (Line 107).

```
  9          dsrl    t0, wfor, 3
 10          bne     t0, zero, 3f
 11          sd      zero, T_INTERRUPT_MASK-TCBO(stcb)
 12          ld      t0, T_MYSELF-TCBO(stcb)
 13          ld      t2, K_INT0_THREAD(t1)
 14          bne     t2, t0, 5f
 15          sd      zero,  K_INT0_THREAD(t1)
 16          b       4f
 17  5:      ld      t2, K_INT1_THREAD(t1)
 18          bne     t2, t0, 5f
 19          sd      zero,  K_INT1_THREAD(t1)
 20          b       4f
 21  5:      ld      t2, K_INT2_THREAD(t1)
             ...
 29  5:      ld      t2, K_INT4_THREAD(t1)
 30          bne     t2, t0, 4f
 31          sd      zero,  K_INT4_THREAD(t1)
 32  4:      beq     wfor, zero, 3f
 33          daddiu  s0, wfor, -1
 34          bne     s0, zero, 5f
 35          ld      s1, K_INT0_THREAD(t1)
 36          bne     s1, zero, 6f
 37          dli     t3, ST_IM2
 38          sd      t3, T_INTERRUPT_MASK-TCBO(stcb)
 39          dli     t2, INT0_TCB_BASE & (~(TCB_SIZE-1))
 40          lbu     t3, T_TSP-TCBO(stcb)
 41          sb      t3, T_TSP(t2)
 42          sb      t3, T_CTSP(t2)
 43          lhu     t3, T_TIMESLICE-TCBO(stcb)
 44          sh      t3, T_TIMESLICE(t2)
 45          sh      t3, T_REM_TIMESLICE(t2)
 46          sd      t0, K_INT0_THREAD(t1)
 47          b       3f
 48  5:      daddiu  s0, s0, -1
 49          bne     s0, zero, 5f
             ...
106          sd      t0, K_INT4_THREAD(t1)
107          b       3f
108  6:      dli     v0, L4_IPC_ENOT_EXISTENT
109          syscall_ret
110  3:      dli     v0, L4_IPC_RETIMEOUT
111          syscall_ret
```

Listing 5.10: Interrupt association part of `receive_only`.

```
112 1:      li       t0, FS_WAIT+FS_WAKEUP
113         daddiu   t3, stcb, -TCBO
114         ins_wakeup(t9, t3, t1)
115 2:      ld       t2, T_INTERRUPT_MASK-TCBO(stcb)
116         mfc0     t3, C0_STATUS
117         or       t3, t3, t2
118         mtc0     t3, C0_STATUS
119         sw       t0, T_FINE_STATE-TCBO(stcb)
120         sd       rdesc,  T_RECV_DESC-TCBO(stcb)
121         sw       timeout, T_TIMEOUT-TCBO(stcb)
122         sd       wfor, T_WFOR-TCBO(stcb)
123         to_next_thread(t1)
```

Listing 5.11: Epilogue of `receive_only`.

Epilogue

The epilogue of `receive_only` blocks the caller, as there is no pending message, and performs a context switch.

Line 112 is jumped to by the prologue code if the receive has a finite timeout. The caller's state is set to WAIT+WAKEUP (Lines 112, 119), indicating it is blocked for a finite time. The thread is inserted into the wakeup queue according to the timeout value (Lines 113–114). Lines 115–118 perform interrupt acknowledgement as explained below. The receive descriptor, timeout and wait-for TID are stored in the caller's TCB (Lines 120–122) prior to scheduling a new thread (Line 123).

Remember, all the short IPC code is executed with interrupts disabled. The interrupt mask in the status register is left unmodified, so when the `syscall_ret` macro turns off the interrupt-disable bit in the status word, the interrupt status will be as before the system call.

An interrupt, once raised, is disabled until it is received its handler. This implies that an IPC receive operation may change the interrupt mask: When a handler receives an interrupt IPC, that interrupt must be enabled again.

This may have occurred when we reach Line 115. Therefore the interrupt mask stored in the thread's TCB (zero for threads not associated with an interrupt) is or-ed to the interrupt mask in the status register (Lines 115–118). If the caller is an interrupt handler this will enable the corresponding interrupt, in all other cases it has no effect.

5.4.2 `pending_receive_only`

If the caller of a receive IPC has a non-empty pending list, the `receive_only` code in the prologue (Section 5.4.1 diverts to `pending_receive_only`, shown in Listing 5.12.

Line 0 test for an open wait, if so, the message can be delivered (Lines 6–11). Otherwise we loop through the send queue (list of pending sends to this thread) to see if one of them matches the caller's TID (Lines 1–4). If there is no match, the receive will have to block, and Line 5 returns to Line 2 of `receive_only`.

Implementation criticism: The comment "FIXME: check if wfor interrupt" in the source (between Lines 0 and 1) is no longer relevant and should be removed.

Bug/Restriction 7: Wait-for checks real instead of virtual sender.
Lines 1–2 check the send queue for a sender's TID matching the receiver's wait-for specification. This will produce incorrect results in the case of deceiving sends. The code should instead check the sender's `v1` register, which contains the sender TID for non-deceiving sends and the virtual sender otherwise.

Lines 6–7 store the deceive descriptor and timeout word in the caller's TCB. These are needed during delivery of long messages.

A limited context switch is now performed to the sender (Line 9). Remember, all IPC handling is performed in the sender's context. The context switch will restart the blocked sender which had, prior to context switching, pushed the address of `pending_restart` (Line 37 of `pending`, Listing 5.5). That function will now be executed in the sender's context, restoring as much of the sender's context as required, in particular the message registers `s0`–`s7`. It will then invoke the `deliver` part of `k_ipc` (Listing 5.2) or the corresponding long IPC delivery code, as appropriate. This will switch back to the receiver's context, thereby delivering the message.

```
 0          beq     wfor, zero, 1f
 1  3:      ld      t1, T_MYSELF(t0)
 2          beq     t1, wfor, 1f
 3          ld      t0, T_SNDQ_NEXT(t0)
 4          bne     t0, zero, 3b
 5          j       leave_waiting
 6  1:      sd      rdesc,  T_RECV_DESC-TCBO(stcb)
 7          sw      timeout,  T_TIMEOUT-TCBO(stcb)
 8          lui     t1, KERNEL_BASE
 9          thread_switch_fast(stcb, t0, t1)
10          ld      ra, (sp)
11          jr      ra
```

Listing 5.12: Receiver finds sender ready: `pending_receive_only`.

5.4.3 Discussion

Abstracting interrupts as virtual threads, which occasionally send messages to their handlers, binds interrupt handling nicely into L4 IPC, with a minimum of API constructs and little extra kernel code. It also provides an elegant means of prioritising interrupt processing.

Using zero-timeout receive operations, to change association of handlers with interrupts, moves these operations off the critical IPC path, as this processing is done only in cases where the IPC would block (and is therefore more expensive anyway).

Delivery of a pending send involves two limited context switches, the second one taking the message with it. This is all done in the receiver's time slice, without any scheduler invocation. This is an instance of *lazy scheduling* in L4.

5.5 Long IPC

Line 8 of `k_ipc` (Listing 5.1) branches to `ipc_long` when the *source descriptor* of the operation is not empty. A non-empty source descriptor indicates that the operation is not of the simplest kind, it may involve deceiving, memory messages or mappings.

Similarly, the `pending_restart` code, which is (eventually) executed by a receive-only operation, diverts to `ipc_long_deliver` if the source descriptor (describing the receive operation) is non-zero.

In both cases, mostly the same code is executed, as `ipc_long_deliver` is an entrypoint in `ipc_long`. Most of the actual long IPC code is in C functions, `ipc_long` itself contains mostly the clans&chiefs and deception code.

```
 0          andi    t0, sdesc, L4_IPC_DECEIT_MASK
 1          beq     t0, zero, 2f
 2          tid2tcb(vsend,t0)
 3          ld      t0, T_MYSELF(t0)
 4          beq     t0, zero, invalid_dest
 5          move    t3, v0
 6          move    s8, v1
 7          move    t2, v1
 8          move    v1, vsend
 9          jal     long_ipc_nchief
10          move    a3, v0
11          move    t2, s8
12          move    v1, dthrd
13          jal     long_ipc_nchief
14          xor     t0, a3, v0
15          andi    t0, t0, L4_IPC_SRC_MASK
16          beq     t0, zero, 3f
17          move    v1, vsend
18          ori     v0, t3, L4_IPC_DECEIT_MASK
19          b       2f
20  3:      move    v0, t3
21          move    v1, s8
```

Listing 5.13: Long IPC code: `ipc_long`, first part.

5.5.1 Clans & Chiefs and Deception: `ipc_long`

Lines 0–1 test the deception bit in the source descriptor and continue at Line 22 if the operation is not deceiving.

Hence, we are now looking at a deceiving send operation. We have to check whether the deceit is legal, i.e., direction preserving. To this end, the direction of the deceiving operation is compared with that of the operation would actually be performed. Informally speaking, the deceit is legal if the actual send can form part of a (most direct) message chain from the virtual sender to the intended receiver. This means that the virtual sender and the intended receiver must lie at different sides of the actual sender's clan boundary, one *inside* and the other *outside*.

Lines 2–6 prepare for calls to `long_ipc_nchief`, another entry point inside the `nchief`. Lines 2–3 load the real TID of the virtual sender (note that this may cause a page fault on the TCB access). A zero TID (indicating an Invalid TCB) leads to returning with an *non-existing partner* result (Line 4). Lines 5 and 6 save the values of registers `v0` and `v1`, which are used for output by `long_ipc_nchief`, in registers `t3` and `s8`, which are not used by that routine.

In Lines 7–9 `long_ipc_nchief` is invoked with the sender's TID as the source and the virtual sender's TID as the destination. This will deliver the redirection target of a message sent from the caller to the virtual sender, which is the opposite direction of what is logically to be tested. This will be taken into account later.

Line 10 saves the *type* result in `a3`. Lines 11-13 invoke `long_ipc_nchief` again, this time with the sender's TID as the source and the caller-supplied receiver TID as the destination.

Lines 14–16 compare the *direction* bits returned by the two calls. If they differ, the directions (from the actual sender) to the virtual sender and the intended receiver are the different, and hence the attempted deceit is legal. It is performed by loading (Line 17) the virtual sender TID into `v1` (which will later return the "sender TID" to the receiver) and turning on the *deceit* bit in the result word `v0` (Line 18).

An attempted illegal deceit is ignored, and Lines 20–21 simply restore the previous values of `v0` and `v1`.

Implementation criticism: Note that this logic allows deceiving between an inner task and another task of the same clan, but not between an outer task and another task of the same clan. This asymmetry is justified by the fact that a sibling task can send to the same outer destinations as the caller, and no deceit is necessary to achieve the communication. Still, this behaviour required by the present L4 specification increases the overhead of implementing multi-threaded servers.

If the deceit is direction preserving, the sender TID is set to the virtual sender (Line 17) and the deceit flag is set in the return value (Line 18), otherwise the real source is used and no deceit happens (Lines 20–21).

Lines 22–27 check whether a send is pending. If not, the `pending` routine is invoked (see Listing 5.5) is invoked, which, on restart via `pending_restart`, will eventually return to `ipc_long_deliver` (Line 28), which is exactly where the code continues if there is an appropriate pending send operation.

Lines 29–46 stack all registers whose contents are still needed and also save in the TCB the timeout value, which may be needed for setting page-fault timeouts. Lines 47–51 set up the arguments to the function `do_long_ipc`, called at Line 52. The address of the stacked `s` registers is passed to the function in `a2`).

After return Lines 53–63 restore registers. Lines 64–67 merge the result value returned from the C function with what had been accumulated before. If the result indicates an error, `send_only_short` is invoked to finish quickly (ignoring any receive part of the IPC), otherwise the `k_ipc` entrypoint `deliver` is used to finish the send and process the receive part of the IPC.

```
22 2:       lw      t3, T_FINE_STATE(dtcb)
23          andi    t3, t3, FS_WAIT
24          beq     t3, zero, pending
25          ld      t2, T_WFOR(dtcb)
26          beq     t2, zero, ipc_long_deliver
27          bne     v1, t2, pending
28 ipc_long_deliver:
29          daddiu  sp, sp, -128
30          sd      rdesc,   (sp)
31          sd      timeout,  8(sp)
32          sw      timeout, T_TIMEOUT-TCBO(stcb)
33          sd      dthrd, 16(sp)
            ...
46          sd      s7, 120(sp)
47          ld      a1, T_RECV_DESC(dtcb)
48          daddiu  a2, sp, 64
49          daddiu  a3, stcb, -TCBO
50          move    a4, dtcb
51          move    a5, v0
52          jal     do_long_ipc
53          move    a0, v0
54          ld      rdesc,   (sp)
            ...
61          ld      v0, 56(sp)
62          daddiu  sp,sp,128
63          tcbtop(stcb)
64          or      v0, v0, a0
65          andi    t0, a0,  L4_IPC_ERROR_MASK
66          beq     t0, zero, deliver
67          b       send_only_short
```

Listing 5.14: Long IPC: `ipc_long`, second part.

Implementation criticism: This code calls C completely unnecessarily if the IPC is deceiving but otherwise short (register-only and no mappings). After Line 28 we should return to `deliver` if there is no other "long" IPC operation to perform.

5.5.2 Performing long IPC operations: `do_long_ipc`

All the remaining IPC code, i.e., processing mappings and memory copies, is in the function `do_long_ipc` (in `ipc.c`).

Prologue

The prologue is shown in Listing 5.15. The state of the communicating threads is marked as LOCKED, indicating in the process of performing long IPC. The sender is also marked BUSY (remember, IPC processing is all done in the sender's context). The communication partners are recorded in the TCBs so it is possible to find out who is locking whom.

```
 0 dword_t do_long_ipc(dword_t sdesc,
 1                     dword_t rdesc,
 2                     dword_t *sregs,
 3                     tcb_t *stcb,
 4                     tcb_t *dtcb,
 5                     dword_t status) {
 6   l4_msgdope_t r;
 7   dword_t      window_addr;
 8   r.msgdope    = status;
 9   window_addr  = 0;
10
11   stcb->fine_state = FS_LOCKS | FS_BUSY;
12   dtcb->fine_state = FS_LOCKR ;
13   stcb->comm_partner =  dtcb;
14   dtcb->comm_partner =  stcb;
```

Listing 5.15: Prologue of `do_long_ipc`

The next section of the function is concerned with processing mappings (fpage specifications). The first part of this, locating the receive fpage, is shown in Listing 5.16.

Receive fpage

Line 15 tests for the mapping bit in the send descriptor (if it is unset there will be no mappings). We then look for the receive fpage. This can either specified as part of the receive descriptor, if the map-bit is set in the descriptor (Lines 19–22). Otherwise, the first word of the message header pointed to by the receive descriptor is expected to contain the receive fpage.

As the IPC is performed in the sender's context, the receiver's memory, including a potential receive fpage, is not currently accessible. Unless we want to bear the overhead of copying things twice (which we don't) we need to set up a mapping of the receiver's buffer(s) in some free address-space region.

L4/MIPS uses the supervisor address region `XKSSEG` as the *temporary mapping area*. There is a slot in this region for each thread in the sender's task, and it can map a 16MB window in the receiver's address space. Note that long IPC can block on a page fault, so it is inherently preemptible. Care must be taken that the temporary mappings

```
15  if (sdesc & L4_IPC_FPAGE_MASK)
16  {
17    dword_t    recv_fpage;
18    recv_fpage = 0;
19    if (rdesc & L4_IPC_FPAGE_MASK)
20    {
21      recv_fpage = rdesc;
22    }
23    else if (rdesc & (~(dword_t)(L4_IPC_FPAGE_MASK|L4_IPC_DECEIT_MASK)))
24    {
25      if (rdesc <  USER_ADDR_TOP)
26      {
27        window_addr = RECV_WINDOW_BASE +
28          (((stcb->myself & TID_THREAD_MASK) >> TID_THREAD_SHIFT)
29           * RECV_WINDOW_SIZE) +
30          (rdesc & (4 * 1024 * 1024 - 1 - 7));
31        stcb->wdw_map_addr = rdesc & (~(dword_t)(4*1024*1024 -1));
32        recv_fpage = *(dword_t *) window_addr;
33      }
34    }
```

Listing 5.16: Locating the receive fpage in `do_long_ipc`.

used in long IPC do not overlap. Mappings from different tasks are no problem as the TLB entries are tagged with the sender's ASID. (ASID tags are active even in kernel mode unless a TLB entry has the global-bit set.) But within a task mapping windows must be kept disjoint, which is why each local thread has its own slot.[1]

Lines 27–29 determine the address of the mapping slot, and Line 30 shifts the receive buffer address (specified in the receive descriptor) into that slot. The resulting pointer, `window_addr`, will later be used to access the receiver's buffer. The base address of the window in the receiver's address space is recorded as `wdw_map_addr` the sender's TCB.

Line 32 reads the receive fpage from the reveiver's address-space window. This will trigger a page fault (called *window fault*) which will be handled by translating the address back into the receiver's address space, looking up the mapping in the receiver's page table, translating it into the correct mapping area address, and loading the TLB. These mappings are truly "temporary" in that they are never entered into any page table — TLB entries for them are created on-the-fly. For details see the `window_fault` code, Listings 4.12–4.15.

Implementation choice: Note that in the case of intra-task messages the temporary mapping area is not necessary. The long IPC code could be optimised to make use of the fact that the sender and receiver buffers are both directly accessible within the sender's address space. However, sending memory messages intra-task is silly, as the same effect could be achieved by user-level memory copy operations, without any help from the kernel. It therefore makes sense not to complicate kernel code in an attempt to optimise a case which should not be used in the first place.

[1]Obviously, this would be a bit more difficult to manage if the number of threads per task wasn't fixed.

Processing mappings

Next is processing the sender's fpages, this is shown in Listing 5.17. The mapping descriptors (2 words each) are processed in turn by passing them to `gpt.c:vm_map` (in the `vm` directory) which does the actual work. The eight message registers can hold up to four mapping descriptors, and further descriptors should be located in the direct string. The operation stops when a descriptor containing a null fpage is encountered.

Bug/Restriction 8: Four fpages only.
The direct string is presently not searched for mapping descriptors, limiting the number of mappings to four.

Bug/Restriction 9: Fpage processing terminated too late.
The `vm_map` function does not have a return value and therefore cannot indicate whether it has set up the mapping successfully. This means that mapping processing is not terminated immediately when an invalid fpage is found (e.g., one specifying an invalid size), which is contrary to [EHL99].

```
35    if (recv_fpage != 0)
36    { int i;
37      for (i = 0; i < 4; i++)
38      {
39        if (sregs[i*2+1] != 0)
40        {
41          vm_map(stcb, sregs[i*2+1], sregs[i*2],
42                 dtcb, recv_fpage);
43          r.md.fpage_received = 1;
44        }
45        else
46        {
47          recv_fpage = 0;
48          break;
49        }
50      }
51
52      if ((sdesc & (~(dword_t)(L4_IPC_FPAGE_MASK|L4_IPC_DECEIT_MASK)))
53          && (recv_fpage != 0))
54      {
55        /* FIXME: implement fpages from memory */
56      }
57    }
58  }
```

Listing 5.17: Processing mappings in `do_long_ipc`.

Memory messages

Line 59 checks whether there is anything left to do, i.e., whether a send buffer is supplied. The buffer address is checked to be in the valid user address range (Line 61), the three-word message header is extracted (Line 64) and checked whether it specifies any direct or indirect strings to be copied (Lines 65–66). If not we are done and can return, after first flushing any temporary mappings from the TLB.

Implementation criticism: This seems to be purely defensive, as I do not think any harm could come from leaving them in. Better safe than sorry!

```
59   if (sdesc & (~(dword_t)(L4_IPC_FPAGE_MASK | L4_IPC_DECEIT_MASK)))
60   {
61     if ((sdesc + sizeof(l4_msghdr_t)) < USER_ADDR_TOP )
62     {
63       l4_msghdr_t *snd_hdr;
64       snd_hdr = (l4_msghdr_t *) (sdesc & (~(dword_t) 7));
65       if (snd_hdr -> snd_dope.md.dwords == 0 &&
66           snd_hdr -> snd_dope.md.strings == 0)
67       {
68         if (window_addr != 0)
69         {
70           tlb_flush_window(window_addr);
71         }
72         return r.msgdope;
73       }
74
75       if ( ((rdesc & L4_IPC_FPAGE_MASK) == 0) &&
76            (rdesc & (~(dword_t)3)) &&
77            ((rdesc + sizeof(l4_msghdr_t)) < USER_ADDR_TOP))
78       {
79         l4_msghdr_t *rcv_hdr;
80         window_addr = RECV_WINDOW_BASE +
81           (((stcb->myself & TID_THREAD_MASK) >> TID_THREAD_SHIFT)
82            * RECV_WINDOW_SIZE) +
83           (rdesc & (4 * 1024 * 1024 - 1 - 7));
84         stcb->wdw_map_addr = rdesc & (~(dword_t)(4*1024*1024 -1));
85         rcv_hdr = (l4_msghdr_t *) window_addr;
86
87         if ( ((sdesc + sizeof(l4_msghdr_t) +
88                8 * snd_hdr->size_dope.md.dwords +
89                sizeof(l4_strdope_t)*snd_hdr->size_dope.md.strings)
90               >=  USER_ADDR_TOP) ||
91              ((rdesc + sizeof(l4_msghdr_t) +
92                8 * rcv_hdr->size_dope.md.dwords +
93                sizeof(l4_strdope_t)*rcv_hdr->size_dope.md.strings)
94               >=  USER_ADDR_TOP) ||
95              (snd_hdr->snd_dope.md.dwords >
96               snd_hdr->size_dope.md.dwords)
97             )
98         {
99           r.msgdope |= L4_IPC_REMSGCUT;
100          tlb_flush_window(window_addr);
101          return r.msgdope;
102        }
```

Listing 5.18: Processing memory messages in `do_long_ipc`.

The receive descriptor is similarly checked for a valid pointer to the receiver's message buffer (Lines 75–77). The temporary window is set up for the receiver's address space as above (Lines 27–31) to make the receive buffer accessible (Lines 80–84) and the receive message header is extracted (Line 85).

Lines 87–102 check whether one of the buffers extends outside the valid user address range or the sender's string dope specifications are nconsistent. If so we return with an error status indicating “message truncated”. Nothing is copied at all in this case.

Note that the address of the mapping window is always maintained up-to-date in the TCB. This is necessary to allow the TLB miss handler to inverse the mapping if a TLB miss occurs during access (see Line 8 of `window_fault`, Listing 4.12).

Direct strings

We are finally ready to process the direct string (if any), the code is shown in Listing 5.19. If the sender’s string exceeds the receiver’s buffer we again return with a “truncated message” error status. Otherwise a straight-forward memory copy is performed, and the actual number of words copied is recorded in the result word.

During the memory copy (Lines 119–122) interrupts are enabled, by calling `ints_on` (in `likern/ints.S`) before and `ints_off` immediately after. This creates a *preemption point* in long IPC, which is necessary as interrupts could easily be lost in the time it takes to copy 4MB. Note that this is the first point in the IPC path where interrupts are enabled.

Implementation criticism: The simple copying algorithm is inefficient for long strings, the copy loop should be unrolled.

```
103       if (snd_hdr -> snd_dope.md.dwords != 0)
104       {
105         dword_t *sp,*rp;
106         int i;
107
108         if (rcv_hdr->size_dope.md.dwords <
109             snd_hdr -> snd_dope.md.dwords)
110         {
111           r.msgdope |= L4_IPC_REMSGCUT;
112           tlb_flush_window(window_addr);
113           return r.msgdope;
114         }
115
116         sp = (dword_t *)((char *) snd_hdr + sizeof(l4_msghdr_t));
117         rp = (dword_t *)((char *) rcv_hdr + sizeof(l4_msghdr_t));
118         ints_on();
119         for (i = 0; i < snd_hdr->snd_dope.md.dwords; i++)
120         {
121           rp[i] = sp[i];
122         }
123         ints_off();
124         r.md.dwords = snd_hdr->snd_dope.md.dwords;
125       }
```

Listing 5.19: Processing direct strings in `do_long_ipc`.

Indirect strings

After checking that string copying is requested and doing the obvious sanity checking (Lines 126–141) we are ready to start processing the indirect strings. Pointers are set up to the string dopes in the sender’s and receiver’s buffers.

```
126         if (snd_hdr->snd_dope.md.strings != 0)
127         {
128           int i;
129           dword_t direct_map_addr;
130
131           l4_strdope_t *snd_strings, *rcv_strings;
132           if ( (snd_hdr->snd_dope.md.strings >
133                 snd_hdr->size_dope.md.strings)   ||
134                (rcv_hdr->size_dope.md.strings <
135                 snd_hdr->snd_dope.md.strings)
136               )
137           {
138             r.msgdope |= L4_IPC_REMSGCUT;
139             tlb_flush_window(window_addr);
140             return r.msgdope;
141           }
142           snd_strings = (l4_strdope_t *)
143                         ((char *) snd_hdr + sizeof(l4_msghdr_t) +
144                          8 * snd_hdr->size_dope.md.dwords);
145           rcv_strings = (l4_strdope_t *)
146                         ((char *) rcv_hdr + sizeof(l4_msghdr_t) +
147                          8 * rcv_hdr->size_dope.md.dwords);
148           direct_map_addr = stcb->wdw_map_addr;
149           for (i = 0; i < snd_hdr->snd_dope.md.strings; i++)
150           {
151             char *rp, *sp;
152             int  j;
153
154             if ((snd_strings[i].snd_size > L4_MAX_STRING_SIZE) ||
155                 (rcv_strings[i].rcv_size > L4_MAX_STRING_SIZE) ||
156                 (snd_strings[i].snd_size>rcv_strings[i].rcv_size)||
157                 ((snd_strings[i].snd_str +
158                   snd_strings[i].snd_size) >= USER_ADDR_TOP) ||
159                 ((rcv_strings[i].rcv_str +
160                   rcv_strings[i].rcv_size) >= USER_ADDR_TOP)
161                 )
162             {
163                r.msgdope |= L4_IPC_REMSGCUT;
164                tlb_flush_window(window_addr);
165                return r.msgdope;
166             }
```

Listing 5.20: Processing indirect strings in `do_long_ipc`, first part.

Line 148 copies the temporary mapping window address from the TCB into a local variable, essentially saving it on the sender's kernel stack. This is necessary as it contains the description of the window mapping for the receiver's message buffer. The receiver's strings may lie in completely different parts of the receiver's address space, and accessing them may result in conflicting mappings.

Lines 149–185 then process the strings one at a time, starting with the usual sanity checks (Lines 158–166).

```
167             sp = (char *) snd_strings[i].snd_str;
168             rp = (char *) (RECV_WINDOW_BASE +
169               (((stcb->myself & TID_THREAD_MASK)
170                   >> TID_THREAD_SHIFT)
171                * RECV_WINDOW_SIZE) +
172               ( rcv_strings[i].rcv_str & (4 * 1024 * 1024 - 1)));
173             j = (int) snd_strings[i].snd_size - 1;
174             stcb->wdw_map_addr =  rcv_strings[i].rcv_str &
175               (~(dword_t)(4*1024*1024 -1));
176             tlb_flush_window(window_addr);
177             ints_on();
178             for (;j >= 0; j--)
179             {
180               rp[j] = sp[j];
181             }
182             ints_off();
183             stcb->wdw_map_addr =  direct_map_addr;
184             tlb_flush_window(window_addr);
185           }
186           r.md.strings = snd_hdr->snd_dope.md.strings;
187         }
188       }
189       else
190       {
191         r.msgdope |= L4_IPC_REMSGCUT;
192         return r.msgdope;
193       }
194     }
195     else
196     {
197       r.msgdope |= L4_IPC_REMSGCUT;
198     }
199   }
200   if (window_addr != 0)
201   {
202     tlb_flush_window(window_addr);
203   }
204   return r.msgdope;
205 }
```

Listing 5.21: Processing indirect strings in `do_long_ipc`, final part.

In Lines 167–175 the mapping window is set up for the receiver's string. The TLB must then be flushed of window mappings for the receiver's message buffer (or for strings mapped in the previous loop iteration), as these may overlap with the mapping window for the string. The string is then copied, again with interrupts enabled.

Line 183 restores the mapping window address in the TCB to the correct value for the message buffer so that the next loop iteration can access the receiver's string dopes.

Line 186 sets the string count in the result word. Note that, while the result word is maintained in the *sender's* context, it will eventually be returned to the *receiver* via the context switch in the `deliver` code. The number of direct string words and indirect strings copied is **not** returned to the caller. The caller already has this information in its message header.

The rest of the code (Lines 189–205) is cleanup.

Implementation criticism: The byte-wise copy algorithm is highly inefficient. Word-wise copying should be used as much as possible, and the loop should be unrolled.

Chapter 6

Other System Calls

No other L4 system call comes close to IPC in terms of complexity, although some display a fair degree of messiness. They are discussed in this chapter one after the other.

6.1 `id_nearest`

6.1.1 Introduction

The `id_nearest` system call serves two purposes. With a zero argument it returns the caller's thread ID (alias `myself`). Otherwise the argument is the TID of an intended destination thread, and `id_nearest` returns the ID of the *nearest* thread, which would actually receive a message sent to from the caller (source) to the argument (destination) thread. This allows the caller to determine if redirection (via the clans-and-chiefs mechanism, see Section 2.1.2 page 8) takes place. It also allows the determination of the actual sender, if a message has been received with *deception*.

The algorithm is based on the fact that tasks form a tree structure, and the task ID contains the nesting depth as well as the task number of the chief. If the two tasks in question are at the same depth, then comparing their chief numbers indicates whether they are in the *same* clan or not. If they are, the *nearest* ID is the same as the destination ID.

If source and destination are not in the same clan, then either one is inside the other's hierarchy or not. If the source is deeper down than the destination, the destination is certain to be *outside* and the *nearest* task is the source's chief.

If the destination is deeper in the hierarchy than the source, the destination can either be (directly or indirectly) inside the source's clan, or can be outside. This can be determined by following the chain of chiefs from the *destination*, until the (direct or indirect) chief is at the same depth as the source. The two latter tasks may then share a chief, which means that the destination is *inside* the source's clan, and the *nearest* task is the one in the destination's hierarchy whose chief is the source. Otherwise the destination is *outside*, and the *nearest* task is the same as the source's.

The system call consists of two sections of code: the main syscall entry point `syscalls.S:k_id_nearest` handles the `myself` case and invokes `syscalls.S:nchief` to determine the redirection target. The `nchief` routine also contains internal entry points used by the IPC code.

Register usage for the `id_nearest` code is summarised in Table 6.1. Register `a0` contains the user-supplied argument value, while `t8` has been loaded with the pointer to the caller's TCB by the general exception handler.

register	*type*			*conf*	*usage*
	IDN	*LINC*	*INC*		
`a0`	I	-	-	y	destination TID
`a7`	-	-	I	n	source as `chief` in TID
`t0`	T	T	T	n	temporary
`t1`	T	T	T	n	temporary
`t2`	T	I	I	n	source TID
`t8`	I	-	-	s	source TCB pointer (+TCBO)
`t9`	T	-	-	s	destination TCB
`v0`	O	O	O	n	direction
`v1`	O	I/O	I/O	y	destination/nearest TID

Table 6.1: Register usage in `id_nearest` and `nchief` code. The columns *IDN, LINC, INC* refer to usage for the `id_nearest`, `long_ipc_nchief` and `ipc_nchief`, respectively; "-" indicates that the register is not used. The *conf* column indicates conflicts with IPC code: "y" = conflict, "n" = no conflict, "s" = same use. Registers `a1`–`a6`, `t3`, `s0`–`s8` are not used, and should be left alone to avoid conflicts with IPC code.

As in all system calls, `k0` still contains the base address of the kernel miscellaneous data.

6.1.2 `id_nearest`

```
0          bne     a0, zero, 1f
1          ld      v1, T_MYSELF-TCBO(t8)
2          syscall_ret
3  1:      jal     nchief
4          syscall_ret
```

Listing 6.1: `id_nearest`.

The `id_nearest` code is shown in Listing 6.1. It checks the argument and, if zero, returns the caller's TID through `v1` (Lines 0–2). Otherwise the `nchief` function is invoked to determine the real destination (Lines 3–4). A function call is required here as `nchief` is also invoked by IPC code.

Note that the `myself` variant of the system call executes only two instructions (with all load and branch delay slots filled). It does not cause TLB misses (the caller's kernel stack has already been touched in the general exception handler, see Listing 4.3, page 28). Hence it is two cycles short of a true "null system call", which makes it an obvious operation to benchmark.

Implementation criticism: The `myself` operation leaves `v0` undefined, which is supposed to return the *direction* (*inner*, *outer* or *same*). However, instead of wasting a cycle, the reference manual [EHL99] should be amended to specify `v0` as undefined in this case.

6.1.3 `nchief`

The `nchief` function is shown in Listing 6.2. The labels `long_ipc_nchief` and `ipc_nchief` are entry points used by the IPC code (see Listing 5.7 and Listing 5.13). This function implements the algorithm sketched in Section 6.1.1.

The function starts off (Lines 0–3) by loading `t8` with the pointer to the caller's TCB, `t2` with the caller's TID, `t9` with the destination's TCB pointer, and `v1` with the destination's real TID (as supposed to the caller-supplied one, where the `nest` and `chief` fields cannot be trusted). Line 3 may fault the destination's TCB into existence.

```
        tcbtop(t8)
        ld      t2, T_MYSELF-TCBO(t8)
        tid2tcb(a0, t9)
        ld      v1, T_MYSELF(t9)
long_ipc_nchief:
        xor     t0, v1, t2
        dsll    t0, t0, 4
        dsrl    t0, t0, 53
        beq     t0, zero, same_clan
        dsll    t0, v1, 32
        xor     t0, t0, t2
        dsll    t0, t0, 4
        dsrl    t0, t0, 53
        beq     t0, zero, outer_clan
        dsll    a7, t2, 32
1:      xor     t0, v1, a7
        dsll    t0, t0, 4
        dsrl    t0, t0, 53
        beq     t0, zero, inner_clan
        xor     t0, v1, t2
        dsll    t0, t0, 4
        dsrl    t0, t0, 53
        beq     t0, zero, inner_clan
ipc_nchief:
        dsrl    t0, v1, 60
        dsrl    t1, t2, 60
        dsubu   t1, t0, t1
        blez    t1, outer_clan
        dli     t1, TID_TASK_MASK
        dsrl    t0, v1, 32
        and     t0, t1, t0
        tid2tcb(t0, t1)
        b       1b
        ld      v1, T_MYSELF(t1)
outer_clan:
        dli     t1, TID_TASK_MASK
        dsrl    t0, t2, 32
        and     t0, t1, t0
        tid2tcb(t0, t1)
        dli     v0, L4_NC_OUTER_CLAN
        jr      ra
        ld      v1, T_MYSELF(t1)
inner_clan:
        jr      ra
        li      v0, L4_NC_INNER_CLAN
same_clan:
        jr      ra
        li      v0, L4_NC_SAME_CLAN
```

Listing 6.2: `nchief`.

Implementation criticism: Line 0 is redundant as this operation has already been done by the general exception handler. Line 1 could be done cheaper by using the value loaded into `v1` by Line 1 of Listing 6.1.

Line 4 is the entry point used by the IPC code when deception is attempted (Listing 5.13). It verifies that the deceit is legal ("direction preserving") by comparing the direction (*inner* vs. *outer*) of the deceiving send operation with that of the operation that would actually be performed. The entry point expects `t2`, `t9` and `v1` to be set up appropriately.

Lines 5–8 compare the `chief` fields in the source and destination TID. If they match, the direction *same* is returned (Lines 45–47). Note that in this case the *nearest* thread is the intended destination, which is already in the result register `v1`.

Lines 9–13 compare the source TID's `chief` field with the destination TID's `task` field. If they match, the direction *outer* must be returned (Lines 34–41).

Implementation criticism: This case is handled sub-optimally. The code following the `outer_clan` label (Line 34) constructs the chief TID, which is unnecessary in this case, as the destination *is* the chief and can be returned as the *nearest* thread. Given that sending to one's chief occurs frequently this would be worthwhile to optimise.

The following code checks whether the destination is within the source's clan. Register `a7` is set up to contain the source's task number in the `chief` position, to match against the destination's chief in the following loop.

Lines 15–18 match the source's task number against the `chief` field in the destination. If successful, the source is the destination's chief, and the latter is *inside* the source's clan. Lines 42–44 return the corresponding direction through `v0`, `v1` is already set up properly with the destination TID as *nearest*.

Lines 19–22 test whether source or destination have the same chief. This test is irrelevant during the first iteration of the loop, as it has already been performed at Lines 5–8. However, during further iterations of the loop this test will catch the case where the destination's (indirect) chief is in the same clan as the source, without being inside the source's clan. Unlike Lines 5–8, this is not a case of direction *same*, but of direction *inner*. The *nearest* thread in this case is the destination's (indirect) chief which is in the same clan as the source.

Lines 24–27 compare the nesting depths of source and destination by subtracting the values of the `nest` fields in the TIDs. If the destination's depth is less than the source's, the direction *outer* must be returned, which is done by branching to Line 34. Otherwise, Lines 28–33 construct the destination's chief TID by shifting the `chief` field into the task number location in the TID, and obtaining the corresponding task ID from its TCB. The result replaces the destination ID in `v1`. Line 32 branches back to the beginning of the loop.

The *outer* case is handled at Lines 34–41. The sender's chief's TCB is located and the TID loaded as the *nearest* ID to be returned to the caller. *Outer* is returned as the *direction* value.

Line 23 is the entry point which is used by the short IPC code (Listing 5.7). Note that it is inside the loop, just before the destination is replaced by its chief. The location of the entry point is **after** all cases have been handled where the destination is the same as *nearest*. In the context of the IPC code this means that all cases where no redirection is required are taken care off, and the reminder of the code establishes the actual redirection target. The cases of no redirection have been checked by the IPC code beforehand, to keep the fastest IPC path as short as possible.

Bug/Restriction 10: Exiting `id_nearest` may cause fatal TLB miss..
The implementation of `id_nearest` and `nchief` violates the rule that the TCB must be touched prior to executing the `syscall_ret` code (Listing 4.4) to return to the user. The last VM references prior to the `jr` instructions of Lines 40, 43 and 46 of `nchief` are, in general, to the TCB of the *nearest* thread. Hence a TLB miss is possible in `syscall_ret`, with disastrous (and difficult-to-reproduce) results. A safe way to fix this bug is by inserting an instruction such as

```
sw      zero, -8(sp)
```

between Lines 3 and 4 of Listing 6.1.

6.2 `lthread_ex_regs`

6.2.1 Introduction

The `lthread_ex_regs()` system call serves to inquire and modify a local thread's program counter, stack pointer, pager and excepter. It creates the target thread and its TCB as necessary. Register usage throughout the code is summarised in Table 6.2. Registers marked as input (I) or input/output (I/O) are defined at the time the code is invoked by the general exception handler. Register `k0` is also defined (with the base address of `kernel_vars`) at invocation, all others are uninitialised.

register	*type*	*usage*
`a0`	I	lthread number of target thread
`a1`	I/O	target old/new IP
`a2`	I/O	target old/new SP
`a3`	I/O	target old/new excepter
`a4`	I/O	target old/new pager
`a5`	T	kernel base
`s0`	T	caller TID†/target state
`s1`	T	target TID
`s2`	T	target TCB base
`s3`	T	buddy TCB base†
`s4`	T	target new IP (copy of `a1`)
`s5`	T	target new SP (copy of `a2`)
`s6`	T	target new excepter (copy of `a3`)
`s7`	T	target new pager (copy of `a4`)
`s8`	T	buddy TID†/thread context change indicator
`v0`	T	various temporary
`t0`–`t3`	T	various temporary
`t8`	I	caller TCB pointer (+TCBO)

Table 6.2: Register usage in `lthread_ex_regs`. Entries marked † are used during TCB initialisation only. Registers `a6`, `a7`, `v1` and `t9` are not used.

6.2.2 Prologue

The beginning of the `lthread_ex_regs` code is shown in Listing 6.3. Lines 0–3 copy the new thread attribute values from `a1`–`a4` to `s4`–`s7`, so the former can be overwritten with the original values. Lines 4–9 construct the target TID from the caller's TID and the local thread number argument (`a0`). Line 10 computes the target TCB base address.

Lines 11–13 check the *coarse state* of the TCB. If a valid TCB is found (course state not equal to "invalid") execution continues at Line 62. Note that accessing the TCB will cause a page fault if the TCB (and its buddy) has not been accessed before. That fault is handled by mapping the invalid TCB, see Lines 51–56 of Listing 4.9.

6.2.3 Thread creation

If the target TCB is found to be invalid, a proper TCB must be allocated and initialised, as shown in Listing 6.4. Line 14 calls the function `kmem:tcb_frame_alloc()`. This calls `kmem:k_frame_alloc()` to allocate a new frame in the kernel heap and returns its physical address in `v0`. Lines 15–20 invoke `vm:vm_tcb_insert`,

```
        move    s4, a1
        move    s5, a2
        move    s6, a3
        move    s7, a4
        ld      s0, T_MYSELF-TCBO(t8)
        andi    a0, a0, 0177
        dsll    a0, a0, 10
        dli     s1, ~(0177 << 10)
        and     s1, s0, s1
        or      s1, s1, a0
        tid2tcb(s1, s2)
        lw      t1, T_COARSE_STATE(s2)
        andi    t0, t1, CS_INVALID_TCB
        beq     t0, zero, 1f
```

Listing 6.3: Prologue of `lthread_ex_regs`.

passing it the present task's page table, the TCB base address and its physical address. The function inserts the TCB mapping into the page table.

Each page in the TCB array holds two TCBs. Lines 21–27 determine the TID and TCB base address for the destination thread's buddy. Lines 28–29 determine the stack bases for both threads.

Lines 30–61 initialise the two new TCBs. An exception stack frame containing invalid IP and SP values and an initial status byte is set up in Lines 30–37. The `init_tcb` macro is used (Lines 38–39) to initialise most fields of the TCB (with null values). The *fine states* are set to "inactive" (Lines 41–43) and the TIDs are initialised (Lines 44-45). The page table pointers, pager TIDs, excepter TIDs and ASIDs are initialised with the caller's values (Lines 46–57). Finally, the two new TCBs are inserted into the *present list* behind the caller (Lines 58–61).

When Line 62 (Listing 6.5) is reached we have a valid TCB. Lines 62–64 check whether the thread has already been activated, if yes, execution continues at Line 71. Otherwise the TCB has just been allocated, or has been allocated earlier as the buddy of another thread but not yet activated. This is indicated by the fine state of "inactive".

In this case the thread's scheduling parameters are initialised: the *timeslice* and *remaining timeslice* are both set to the caller's timeslice value (Lines 65–67) and the *priority* and *current priority* are both set to the caller's priority (Lines 68–70). This completes the initialisation of the TCB. (The MCP is not initialised as this is a task attribute and is therefore only used for local thread zero.) All the thread now needs to be able to run is an instruction pointer and a stack pointer value, both of which are arguments to this system call.

Bug/Restriction 11: `ex_regs` incorrectly initialises pager and excepter.
The thread's pager and excepter are initialised from the thread which caused the TCB to be initialised (which could be the one which started the target thread's buddy), not necessarily the one which actually activated the thread.

```
        jal     tcb_frame_alloc
        tcbtop(s3)
        ld      a0, T_GPT_POINTER-TCBO(s3)
        dli     t0, ~(L4_PAGESIZE-1)
        and     a1, s2, t0
        move    a2, v0
        jal     vm_tcb_insert
        andi    t0, s2, TCB_SIZE
        bne     t0, zero, 2f
        daddiu  s3, s2, TCB_SIZE
        daddiu  s8, s1, 1 << 10
        b       3f
2:      daddiu  s3, s2, -TCB_SIZE
        daddiu  s8, s1, -(1 << 10)
3:      daddiu  t3, s2, TCB_SIZE
        daddiu  t2, s3, TCB_SIZE
        dli     t0, -1
        sd      t0, -8(t3)
        sd      t0, -8(t2)
        sd      t0, -16(t3)
        sd      t0, -16(t2)
        li      t1, INITIAL_THREAD_ST
        sb      t1, -24(t3)
        sb      t1, -24(t2)
        init_tcb(s2)
        init_tcb(s3)
        tcbtop(t8)
        li      t0, FS_INACTIVE
        sw      t0, T_FINE_STATE(s2)
        sw      t0, T_FINE_STATE(s3)
        sd      s1, T_MYSELF(s2)
        sd      s8, T_MYSELF(s3)
        ld      t0, T_GPT_POINTER-TCBO(t8)
        sd      t0, T_GPT_POINTER(s2)
        sd      t0, T_GPT_POINTER(s3)
        ld      t0, T_PAGER_TID-TCBO(t8)
        sd      t0, T_PAGER_TID(s2)
        sd      t0, T_PAGER_TID(s3)
        ld      t0, T_EXCPT_TID-TCBO(t8)
        sd      t0, T_EXCPT_TID(s2)
        sd      t0, T_EXCPT_TID(s3)
        ld      t0, T_ASID-TCBO(t8)
        sd      t0, T_ASID(s2)
        sd      t0, T_ASID(s3)
        ld      t3, T_PRESENT_NEXT-TCBO(t8)
        sd      t3, T_PRESENT_NEXT(s3)
        sd      s3, T_PRESENT_NEXT(s2)
        sd      s2, T_PRESENT_NEXT-TCBO(t8)
```

Listing 6.4: Allocation and initialisation of new TCBs.

```
1:      lw      t0, T_FINE_STATE(s2)
        andi    t0, t0, FS_INACTIVE
        beq     t0, zero, 1f
        lhu     t0, T_TIMESLICE-TCBO(t8)
        sh      t0, T_REM_TIMESLICE(s2)
        sh      t0, T_TIMESLICE(s2)
        lbu     t0, T_TSP-TCBO(t8)
        sb      t0, T_CTSP(s2)
        sb      t0, T_TSP(s2)
```

Listing 6.5: Initialisation of new thread's scheduling parameters.

6.2.4 Exchanging register values

```
71 1:       daddiu  t3, s2, TCB_SIZE
72          dli     t0, -1
73          move    s8, zero
74          beq     s7, t0, 1f
75          ld      a4, T_PAGER_TID(s2)
76          sd      s7, T_PAGER_TID(s2)
77 1:       beq     s6, t0, 1f
78          ld      a3, T_EXCPT_TID(s2)
79          sd      s6, T_EXCPT_TID(s2)
80 1:       beq     s5, t0, 1f
81          ld      a2, -8(t3)
82          sd      s5, -8(t3)
83          move    s8, t0
84 1:       beq     s4, t0, 4f
85          ld      a1, -16(t3)
86          sd      s4, -16(t3)
87          move    s8, t0
```

Listing 6.6: Exchanging thread attribute values.

When Line 71 (Listing 6.6) is reached we are ready to do the proper exchange of the thread attributes specified as parameters to the system call. Lines 71–73 set up `t3` as a pointer to the target thread's kernel stack, `t0` with the value of -1, indicating an *invalid* address, and `s8` to zero, to indicate that the thread's context has not been changed yet.

Line 75 loads the present pager value into `a4`, the register used to return this value to the caller, and Lines 74 and 76 set the thread's pager to the one specified in the system call, if the latter is not invalid. Note that Line 75 is executed in Line 74's branch delay slot, and thus comes logically before Lines 74. Lines 77–79 perform the same operation with the excepter, Lines 80–82 with the stack pointer and Lines 84–86 with the instruction pointer. Lines 83 and 87 set `s8` to indicate that the SP or IP were changed.

6.2.5 Cleanup: Terminating pending or running IPCs

```
88 4:       ld      t0, T_SNDQ_START(s2)
89          lui     a5, KERNEL_BASE
90          beq     t0, zero, 3f
91          rem_sendq(t0, s2, t1)
92          dli     v0, L4_IPC_SECANCELED
93          make_busy(t0, v0)
94          ins_busy_list(t0, a5, t2)
95          b       4b
```

Listing 6.7: Cancelling pending IPCs.

Lines 88–95 check whether there are any send operations pending to the target thread. Line 91 removes such a sender from the doubly-linked pending queue. This is done using the `rem_sendq` macro, which is straightforward and does not need further examination. The `make_busy` macro (Listing 7.1) is used to make the formerly pending thread runnable, forcing its IPC completion code to `SECANCELLED` to indicate that the send operation

```
96  3:      lui     a5, KERNEL_BASE
97          lw      s0, T_FINE_STATE(s2)
98          andi    t1, s0, FS_LOCKS
99          beq     t1, zero, 4f
100         ld      t1, T_COMM_PARTNER(s2)
101         lw      t2, T_FINE_STATE(t1)
102         andi    a1, t2, FS_POLL
103         beq     a1, zero, 5f
104         ld      t3, T_COMM_PARTNER(t1)
105         rem_sendq(t1, t3, a2)
106 5:      li      a2, L4_IPC_REABORTED
107         make_busy(t1, a2)
108         ins_busy_list(t1, a5, t0)
109         sw      zero, T_STACKED_FINE_STATE(t1)
110         li      a2, L4_IPC_SEABORTED
111         make_busy(s2, a2)
112         ins_busy_list(s2, a5, t0)
113         b       1f
114 4:      andi    t1, s0, FS_LOCKR
115         beq     t1, zero, 4f
```

Listing 6.8: Target thread state was `LOCKS`.

failed (Lines 92–93). The `ins_busy_list` macro (Listing 7.3) is used in Line 94 to insert the thread into the appropriate scheduling queue and give it a time slice.

When reaching Line 96 (Listing 6.8) we know that no sends are pending to the target. What remains to be done is to examine the target thread's state (Lines 96–97) for any pending or on-going IPC, which needs to be terminated.

Lines 98–113 deal with the state `LOCKS`, which means that the target is in the middle of a send operation. The partner state must be `LOCKR` (pre-empted during long IPC) or blocked on a receive page fault. Lines 100–103 examine the state of the thread's communication partner (i.e., the thread the target thread is presently sending a message to). If it is `POLL` the partner is presently blocked on an IPC to its pager. Lines 104–105 cancel this page fault IPC by removing the partner from its pager's send queue.

Independent of what the partner's state was, it is now made runnable, and its completion code set to indicate that the receive part of the IPC was aborted (Lines 106–108). Line 109 resets the thread's `stacked_fine_state`. This gets set when a nested IPC call is performed, such as on a page fault during long IPC (see Listing 4.10). As the `ex_regs` operation aborts the original IPC, any nested IPC is implicitly terminated too. Resetting the stacked state is essential as some code (such as that in Listing 6.9) uses that thread attribute to check for a nested IPC.

The target thread is then made runnable with a completion code indicating an aborted send operation (Lines 110–113).

Bug/Restriction 12: `ex_regs` return values trashed if terminating IPC.
Line 102 trashes `a1`, which contains the old instruction pointer that is to be returned to the caller. Lines 105, 106, 110 and 111 trash `a1`, returning the old stack pointer. Other code up to Line 162 does the same. Registers `a6` and `a7` could safely be used.

Lines 114–129 deal with the situation of the target thread state being `LOCKR` (preempted long IPC receive); the partner must then be in state `LOCKS` or again blocked on a receive page fault. The code is completely analogous to what has just been discussed, except that the completion codes are reversed.

```
129           b        1f
130  4:       lw       t0, T_STACKED_FINE_STATE(s2)
131           andi     t1, t0, FS_LOCKS
132           beq      t1, zero, 4f
133           ld       t1, T_STACKED_COMM_PRTNR(s2)
134           lw       t2, T_FINE_STATE(t1)
135           li       a2, L4_IPC_REABORTED
136           make_busy(t1, a2)
137           ins_busy_list(t1, a5, t0)
138           sw       zero, T_STACKED_FINE_STATE(t1)
139           andi     t0, s0, FS_POLL
140           beq      zero, t0, 5f
141           ld       t1, T_COMM_PARTNER(s2)
142           rem_sendq(s2, t1, t2)
143  5:       li       a2, L4_IPC_SEABORTED
144           make_busy(s2, a2)
145           ins_busy_list(s2, a5, t0)
146           b        1f
147  4:       andi     t1, t0, FS_LOCKR
148           beq      t1, zero, 4f
```

Listing 6.9: Target thread's *stacked* state was `LOCKS`.

Alternatively, the target thread may itself be blocked on a page fault during long IPC. This means that the thread is in the middle of a nested IPC and the state of the primary IPC is saved in the TCB field `stacked_fine_state`. If this state is `LOCKS`, the (stacked) partner's would once more be `LOCKR` or `POLL`.

The handling of this case is shown in Lines 130–146 (Listing 6.9). Lines 130–132 determine that the target state is indeed `LOCKS`. Lines 133–138 make the stacked partner busy with a completion code of `REABORTED`, indicating that its receive operation was terminated half way through.

Lines 139–142 check whether the target's own state is `POLL`, which indicates it is waiting to send to its pager. If so, the target from the current partner's send queue (Lines 143–146). Remember that the case where the present IPC in progress has been handled above. Hence the present partner's state remains unaffected.

Lines 143–145 make the target busy with a completion code of `SEABORTED`, indicating that its send operation was terminated half way through.

Bug/Restriction 13: Stacked state not reset if in recursive IPC.
The target's stacked state should be reset to zero. This omission can lead to misbehaviour of future `ex_regs` operations on the same target.

Note that the above code is very similar to Lines 97–113. The main difference is that here the polling target is being unblocked, while above it is the partner's partner.

Lines 147–162 treat the equivalent case of the target's stacked state being `LOCKR`.

At this point any pending sends have been cancelled and any on-going IPC has been aborted. We now need to check whether the target thread is blocked on a send or receive operation (which hasn't commenced yet).

Line 163 test for the `WAIT` state, indicating that the target is blocked on a receive operation. If so, it is simply unblocked with a completion code indicating a cancelled receive (Lines 164–167). Line 169 tests for the `POLL` state, indicating that the target is blocked on a send. If so, it is removed from the recipient's send-queue (Lines 170–172) and is unblocked with a completion code indicating a cancelled send (Lines 173–175).

Line 177 tests for the `INACTIVE` state, meaning that the thread has never been activated before. Line 179 checks the value of `s8` for an indication of the thread's SP or IP having been set by the call (c.f. Lines 83 and 87), in which case the thread is to be activated. This is done in Lines 180–182.

Implementation criticism: Explicitly setting the completion code (which becomes the target's initial `v0` value) to zero in Line 180 is presumably to prevent a covert channel. However, the same could be achieved by using `s8`, which is known to contain a value of -1, in the macro invocation of Line 181, saving Line 180 (and one cycle).

Implementation criticism: The source contains a comment *"thread was waiting"* between Lines 178 and 179. This is obviously a cut-and-paste error. The comment should be removed.

The final test in Line 184 is for the `BUSY` state, indicating the target thread was ready to run. Nothing more needs to be done for this case. Any other state represents a kernel bug and leads to a kernel panic (Lines 185–187). Note that the only legal state not explicitly tested is `DYING`. This state can only occur while a task is being deleted (see Listing 6.17), and no `lthread_ex_regs` call is possible in that case due to the non-preemptability of task deletion (which should be fixed!)

```
162         b       1f
163 4:      andi    t0, s0, FS_WAIT
164         beq     zero, t0, 2f
165         dli     a0, L4_IPC_RECANCELED
166         make_busy(s2, a0)
167         ins_busy_list(s2, a5, t0)
168         b       1f
169 2:      andi    t0, s0, FS_POLL
170         beq     zero, t0, 2f
171         ld      t1, T_COMM_PARTNER(s2)
172         rem_sendq(s2, t1, t2)
173         dli     a0, L4_IPC_SECANCELED
174         make_busy(s2, a0)
175         ins_busy_list(s2, a5, t0)
176         b       1f
177 2:      andi    t0, s0, FS_INACTIVE
178         beq     zero, t0, 2f
179         beq     s8, zero, 1f
180         move    a0, zero
181         make_busy(s2, a0)
182         ins_busy_list(s2, a5, t0)
183         b       1f
184 2:      andi    t0, s0, FS_BUSY
185         bne     zero, t0, 1f
186         dla     a0, msg_tcb_state
187         j       panic
188 1:      syscall_ret
```

Listing 6.10: Target thread was blocked or ready.

Implementation criticism: None of the code of Lines 88–187 should be executed if the thread's IP and SP were not changed (in contrast to only checking at Line 170). This would enable a non-destructive check of a state's state. Note that this is not a bug, as the implemented behaviour is required by the then L4 specification (which has since changed).

6.3 task_new

The implementation of `task_new` consists of the largest single piece of assembly language code in the kernel (600 lines of sparsely commented source code, expanding into more than 700 instructions). It is (for historical reasons) called `create_thread`.

Most of this code is only moderately time-critical (evidenced e.g. by many redundant recalculations of caller's the TCB address), and should really be written in C. Rather than going through it line-by-line, we list it "as-is", including most of the original comments, and only give a rough description. The attentive reader will be able to follow it easily $\ddot{\smile}$. Table 6.3 lists register allocations used in the code.

register	*type*	*usage*
`a0`	I/T	initial IP for l_0 of new task/temporary
`a1`	I/T	pager of l_0 of new task/caller TID
`a2`	I	initial SP value for l_0 of new task
`a3`	I	task ID
`a4`	I	MCP/new chief of task
`a5`	I	excepter of l_0 of new task
`a6`	T	temporary
`s0`	T	copy of initial `a0` value
`s1`	T	temporary/TCB pointer for new task
`s2`	T	copy of initial `a4` value
`s3`	T	various temporary
`s4`	T	copy of initial `a1` value
`s5`	T	copy of initial `a2` value
`s6`	T	copy of initial `a3` value/target TCB adr
`s7`	T	new TID/temporary
`s8`	T	temporary TCP pointer
`t0`	T	chief pointer/various temporary
`t1`–`t3`	T	various temporary
`t8`	I	caller TCB pointer (+TCBO)
`t9`	T	destination TCB pointer (+TCBO)
`v0`	T/O	various temporary/new task ID
`v1`	T	temporary
`gp`	T	copy of initial `a5` value

Table 6.3: Register usage in `task_create`. Registers `a7`, `v1` are not

The prologue code is shown in Listing 6.11. As the comment indicates, the registers containing the system call arguments (`a0`–`a5`) are copied to "callee-saved" register so they will not get lost during later invocations of C code. Line 7 shows a macro used for tracing kernel code (for kernel debugging).

```
0           /* put args in save registers so we can call C */
1           move     s0, a0
2           move     s4, a1      /* a1 has pager id */
3           move     s5, a2
4           move     s6, a3
5           move     s2, a4
6           move     gp, a5
7           trace(crth)
```

Listing 6.11: `task_new` (`create_thread`) part 0.

```
 8         /* first check validity of task */
 9         tid2ttable(s6, t0)
10         lw      t1, (t0)
11         li      t2, TT_INACTIVE_MASK
12         and     t2, t2, t1
13         beq     zero, t2, active_task
14
15 the_after_life:
16         /* we have invalid task, check chief okay */
17         li      t2, TT_CHIEF_MASK
18         and     t3, t2, t1
19         tcbtop(a0)
20         ld      a1, T_MYSELF-TCBO(a0)
21         beq     t3, zero, 1f
22         xor     a2, t1, a1
23         and     a2, a2, t2
24         beq     a2, zero, 1f
25         /* chief mismatch -> permision denied */
26         move    v0, zero
27         b       ct_ret
28
29 1:      /* chief okay, check pager */
30         bne     s4, zero, 1f
31         /* pager invalid, change chief and return */
32         li      a3, TID_TASK_MASK
33         and     a6, s2, a3 /* a6 has new chief */
34         li      a4, ~TT_CHIEF_MASK
35         and     a5, t1, a4
36         or      a5, a5, a6
37         sw      a5, (t0)   /* new chief stored in task table */
38         dsll    v0, a6, 32
39         and     v1, s6, a3
40         or      v0, v0, v1
41         b       ct_ret
```

Listing 6.12: `task_new` (`create_thread`) part 1.

Part 1 is shown in Listing 6.12. First we determine whether the target task is presently *active* or not. Lines 9–13 do this using the kernel's *task ID table* (TID table, see Section 3.2.4, page 22). The `tid2ttable` macro extracts the task number from the caller-specified TID and returns the address of the corresponding entry in the TID table. If the task is found active, we continue at label `active_task` (Line 182, Listing 6.17, to kill the task first. If successful, execution will return to label `the_after_life` here, which deals with inactive tasks.

Lines 17–27 perform the permission check: A task can only be deactivated by its chief, unless the task has never been assigned a chief (indicated by an inactive task with a zero chief), in which case anyone can claim it. If the permission check fails, a zero task ID is returned to indicate failure. (The `ct_ret` label only contains the *syscall_ret*acro, see Line 180.)

Lines 30–41 handle the case where a zero pager TID was supplied (by the caller in `a1`), which means that the task is left inactive. This operation has the side effect of donating the task to a new chief, specified in the *MCP/new chief* argument (originally in `a4`). The new chief is recorded in the TID table and the new task ID is constructed and returned to the user.

Lines 43–48 increment the task version number. The kernel panics if the version number overflows. The *invalid*

```
42 1:      /* pager valid, inc version number*/
43         addiu   t2, t1, 1
44         andi    t3, t2, TT_OVRFLW_MASK
45         beq     zero, t3, 1f
46         /* ran out of versions */
47         dla     a0, nov_msg
48         j       panic
49
50 1:      /* mask out invalid bit in t table */
51         li      t1, 0x7fffffff
52         and     t2, t2, t1
53         sw      t2, (t0)
54
55         andi    t1, t2, 01777
56         srl     t0, t2, 10
57         andi    t0, t0, 017
58         sll     t0, t0, 28
59         or      t2, t1, t0    /* new version number in right place */
60         /* combine with task id */
61         dli     t0, TID_TASK_MASK
62         and     s7, t0, s6
63         or      s7, s7, t2      /* s7 has new thread id */
64
65         and     s6, a1, t0      /* combine with chief */
66         dsll    s6, s6, 32
67         or      s7, s7, s6
68
69         dsrl    s6, a1, 60      /* set depth FIXME: > 15 */
70         daddiu  s6, s6, 1
71         andi    t2, s6, 020
72         beq     t2, zero, 1f
73         dla     a0, dp_msg
74         j       panic
75
76 1:      dsll    s6, s6, 60
77         or      s7, s7, s6
78         /* s7 has new thread id (inc chief and depth) */
```

Listing 6.13: `task_new` (`create_thread`) part 2.

bit in the TID array is turned off (Lines 51–53). The new version number is extracted from the TID array entry (and split into the two parts as required by the somewhat bizarre L4/MIPS TID format and inserted into the new task ID (Lines 55–63). The caller's task number is inserted as the chief (Lines 65–67) and the caller's nesting depth is incremented and inserted to leave the completed new TID in `s7`. Again, the kernel panics if the depth overflows. Note that the *site* field is undefined in the present L4 specification.

Bug/Restriction 14: Kernel panics on task version or nesting depth overflow.
The kernel should not panic in this case, but return a zero task ID to the user.

```
79          tid2tcb(s7, s6) /* s6 has new tcb vaddress */
80          jal     tcb_frame_alloc /* alloc a new frame for tcb */
81          tcbtop(t9)
82          ld      a0, T_GPT_POINTER-TCBO(t9)
83          move    a1, s6
84          move    a2, v0
85          jal     vm_tcb_insert
86
87          daddiu  s1, s6, TCB_SIZE /* s1 now contains top of new stack */
88          move    s8, s1          /* base of second TCB in pair */
89          daddiu  t2, s8, TCB_SIZE
90
91          /* now build a stack to switch to */
92          dli     t0, -1
93          sd      s5, -8(s1)      /* new thread sp */
94          sd      t0, -8(t2)
95          sd      s0, -16(s1)     /* new thread start address */
96          sd      t0, -16(t2)
97          li      t1, INITIAL_THREAD_ST
98          sb      t1, -24(s1)
99          sb      t1, -24(t2)
100         daddiu  s1,s1,-24
101
102         /* initialise most tcb vars */
103         init_tcb(s6)
104         init_tcb(s8)
105
106         sd      s7, T_MYSELF(s6)
107         daddiu  t0, s7, 1 << 10
108         sd      t0, T_MYSELF(s8)
109
110         /* a4 contains syscall mcp, a1 will contain creator mcp */
111         tcbtop(t9)
112         lbu     a1, T_MCP-TCBO(t9)
113         sub     t2, a1, s2
114         blez    t2, 1f          /* if (creator.mcp > call.mcp) */
115         move    t2, s2          /*      new.mcp = call.mcp */
116         b       2f
117 1:      move    t2, a1          /* else new.mcp = creator.mcp */
118 2:      sb      t2, T_MCP(s6)
```

Listing 6.14: task_new (create_thread) part 3.

A new TCB frame is allocated and mapped for the destination task (Lines 79–85). Registers s1 and t2 are set up as the stack pointers for the two TCBs (of the new task's l_0 and l_1) in Lines 87–89. An exception frame (see Figure 4.1) is set up in the two TCBs, with the caller-supplied IP and SP for l_0 and invalid values for l_1 (Lines 92–99), l_0's stack pointer is set up in s1, and the generic TCB variables (and TIDs) and are initialised (Lines 103–108). The new task's MCP is set up as the lesser of caller's MCP and caller-supplied value.

```
119         lbu     t2, T_TSP-TCBO(t9)
120         sb      t2, T_CTSP(s6)  /* dest.ctsp = src.tsp */
121         sb      t2, T_TSP(s6)   /* dest.tsp = src.tsp */
122         lhu     t2, T_TIMESLICE-TCBO(t9)
123         sh      t2, T_TIMESLICE(s6)     /* timeslice = creator's */
124         sh      t2, T_REM_TIMESLICE(s6) /* rem_timeslice = timeslice */
125
126         /* init the gpt */
127         move    a0, s6
128         jal     vm_new_as
129         ld      v0, T_GPT_POINTER(s6)
130         sd      v0, T_GPT_POINTER(s8)
131         sd      s4, T_PAGER_TID(s6)
132         sd      s4, T_PAGER_TID(s8)
133         sd      gp, T_EXCPT_TID(s6)
134         sd      gp, T_EXCPT_TID(s8)
135
136         /* allocate an asid */
137         move    a0, s7
138         jal     asid_alloc /* ASID alloc uses t0, v0, AT, ra */
139         sd      v0, T_ASID(s6)
140         sd      v0, T_ASID(s8)
141
142         /* add new thread to run queue */
143         sd      s1, T_STACK_POINTER(s6)
144         lui     a2, KERNEL_BASE
145         ins_busy_list(s6, a2, t0)
146
147         /* init new present list for this task */
148         tcbtop(t9)
149         daddiu  s3, t9, -TCBO
150         sd      s8, T_PRESENT_NEXT(s6)
151         sd      zero, T_PRESENT_NEXT(s8)
152         /* add new task as child of this task,
153            and move current child to sister of new task */
154         tcbtop(t0)
155         ld      t0, T_MYSELF-TCBO(t0)
156         dli     t1, TID_TASK_MASK
157         and     t0, t0, t1
158         tid2tcb(t0, t1)
159         ld      t0, T_CHILD_TASK(t1)
160         sd      t0, T_SISTER_TASK(s6)
161         sd      s6, T_CHILD_TASK(t1)
```

Listing 6.15: `task_new` (`create_thread`) part 4.

The scheduling parameters (priority, current priority, time slice and remaining time slice) are initialised from the caller's TCB (Lines 119-124), while the pager and excepter of the destination's l_0 and l_1 are set to the caller-supplied values (Lines 131–134). A new page table is set up and recorded in both thread's TCBs (Lines 127–130). The function `vm_new_as`, which is implemented in assembler, in spite of using the C calling convention, is explained later (Listing 6.23).

An ASID is allocated for the new task and recorded in both thread's TCBs (Lines 137–140). Thread l_0 is linked into the busy list (Lines 143–145). The present list is initialised with the two initial TCBs (Lines 150–151) and the new task is linked into the child task list of the caller (Lines 154–161). Note that this list is only maintained once per task, so the code must first find the TCB of the calling task's l_0.

```
162          /* set running state */
163          li      t2, FS_BUSY
164          sw      t2, T_FINE_STATE(s6)
165          li      t2, FS_INACTIVE
166          sw      t2,  T_FINE_STATE(s8)
167
168          /* stack state for parent return */
169          daddiu  sp, sp, -16
170          dla     t0, parent_thread_restart
171          sd      s7, 8(sp)
172          sd      t0, (sp)
173
174          /* make sure parent is in busy list */
175          ins_busy_list(s3, a2, t0)
176
177          tcbtop(t9)
178          thread_switch_fast(t9, s6, a2)
179          trace(ecrt)
180  ct_ret:
181  1:      syscall_ret
```

Listing 6.16: task_new (create_thread) part 5.

The new threads' fine state is set as appropriate (Lines 164–166). The new task is now ready to run, and is about to be dispatched. In order to set up the context switch, a restart record, consisting of a restart address and the return value (new task ID) is pushed onto the caller's kernel stack (Lines 169–172). The restart routine `parent_thread_restart` will simply pop the restart value off the stack and return.

The `ins_busy_list` macro is used to insert the caller into the busy list (Line 175). Remember, due to lazy scheduling during IPC it is possible that the caller is executing on a donated time slice not actually in the busy list. A fast context switch is performed by the `thread_switch_fast` macro (which leaves registers unchanged) is performed to the new task's l_0 and the `syscall_ret` macro returns to the new task, which is now alive.

```
182 active_task:
183         /* test if chief */
184         dli     t3, TID_TASK_MASK
185         and     t1, s6, t3
186         tid2tcb(t1,s1)
187         ld      t1, T_MYSELF(s1)
188         tcbtop(a0)
189         ld      a1, T_MYSELF-TCBO(a0)
190         dsrl    t2, t1, 32
191         xor     t2, t2, a1
192         and     t2, t2, t3
193         beq     t2, zero, 1f
194         /* not chief, return */
195         move    v0, zero
196         b       ct_ret
197
198 1:      /* chief okay, check if task_new already running */
199         lw      t1, T_FINE_STATE(s1)
200         andi    t2, t1, FS_DYING
201         beq     t2, zero, 1f
202         /* task already dying */
203         move    v0, zero
204         b       ct_ret
205
206 1:      /* okay lets kill the task */
207         /* mark as dying */
208         li      t0, ~(FS_BUSY | FS_WAKEUP)
209         and     t1, t1, t0
210         ori     t1, t1, FS_DYING
211         sw      t1, T_FINE_STATE(s1)
212         sd      zero, T_MYSELF(s1)
```

Listing 6.17: `task_new` (`create_thread`) part 6.

Part 6 heads the code dealing with killing an existing task. Lines 184–196 verify that the caller is the chief of the destination task and thus has the right to kill it. If not, the call returns with a zero TID value. The same happens if the task is already marked as being killed (Lines 199–204). This could be the result of two user threads concurrently attempting to kill the task (or its parent).

Now we have established that we can go ahead and kill the task. First its TCB is marked `DYING` to prevent a concurrent `task_new` call from interfering (although that cannot happen with the present non-preemptable implementation).

```
213         /* now null FINE_STATE and MYSELF of threads in task
214         to prevent struggling while killing */
215         ld      t1, T_PRESENT_NEXT(s1)
216         beq     t1, zero, 1f
217         li      t2, ~(FS_BUSY | FS_WAKEUP)
218 2:      lw      t0, T_FINE_STATE(t1)
219         and     t0, t0, t2
220         ori     t0, t0, FS_DYING
221         sw      t0, T_FINE_STATE(t1)
222         sd      zero, T_MYSELF(t1)
223         ld      t1, T_PRESENT_NEXT(t1)
224         bne     t1, zero, 2b
225         /* now task and thread are unrunnable and invalid */
226         /* remove non-busy tcb’s from busy_list and
227         non-wake tcbs from wake lists */
228 1:      jal     process_lists
229
230 /* BEGIN BIG THREAD CLEANUP LOOP */
231         move    a0, s1
232 2:      lw      t1, T_FINE_STATE(a0)    /* remove if polling */
233         andi    t1, t1, FS_POLL
234         beq     t1, zero, 1f
235         ld      t0, T_COMM_PARTNER(a0)
236         /* we are polling */
237         rem_sendq(a0, t0, t1)
238
239 1:      /* now break off threads pending for this thread */
240         ld      t0, T_SNDQ_START(a0)
241         beq     t0, zero, 1f
242         rem_sendq(t0, a0, t1)   /* pending thread */
243         /* restart them if they are not dying as well */
244         lw      t3, T_FINE_STATE(t0)
245         andi    t3, t3, FS_DYING
246         bne     t3, zero, 1b
247         dla     t1, pending_recv_killed
248         ld      t2, T_STACK_POINTER(t0)
249         sd      t1, (t2)
250         li      t2, FS_BUSY
251         sw      t2, T_FINE_STATE(t0)
252         lui     t3, KERNEL_BASE
253         ins_busy_list(t0, t3, t2)
254         b       1b            /* do any remaining in queue */
```

Listing 6.18: task_new (create_thread) part 7.

The destination’s threads are now made non-runnable by setting their state to DYING. The present list is used to find all allocated TCBs. The C function process_lists is then called to remove any of these threads (and possible other blocked ones which were not removed due to lazy scheduling) from the busy lists and wait queues.

Line 232 is the beginning of a large loop, extending up to Line 433, which cleans up the destination’s threads by traversing the present list. Lines 233–237 check whether the thread is in state POLLING, in which case it is removed from the send queue of the intended receiver. Next, any threads blocked on a send to the target thread are removed and their IPC cancelled by processing the target’s send queue (Lines 240–242). The threads are then made runnable (unless marked DYING) by fixing up their state, replacing their restart procedure by pending_recv_killed, and inserting them into the busy list (Lines 244–253). That restart procedure (defined in exc.S) returns with a status of *non-existent destination thread*.

```
255 1:      /* check if LOCK (in fine state ) */
256         lw      t0, T_FINE_STATE(a0)
257         andi    t1, t0, FS_LOCKS
258         beq     t1, zero, 4f
259
260         ld      t1, T_COMM_PARTNER(a0)
261         lw      t2, T_FINE_STATE(t1)
262         andi    a1, t2, FS_POLL
263         beq     a1, zero, 5f /* check if partner POLL (rcv pf) */
264
265         ld      t3, T_COMM_PARTNER(t1)
266         rem_sendq(t1, t3, a2)
267
268         /* assume partner is LOCKR and make busy if not dying */
269 5:      andi    a1, t2, FS_DYING
270         bne     a1, zero, 1f
271         li      a2, L4_IPC_REABORTED
272         make_busy(t1, a2)
273         lui     t3, KERNEL_BASE
274         ins_busy_list(t1, t3, t2)
275         sw      zero, T_STACKED_FINE_STATE(t1)
276         b       1f
            ...
299 4:      /* test stacked state */
300         lw      t0, T_STACKED_FINE_STATE(a0)
301         andi    t1, t0, FS_LOCKS
            ...
315 4:      andi    t1, t0, FS_LOCKR
316         beq     t1, zero, 1f
317
318         ld      t1, T_STACKED_COMM_PRTNR(a0)
319         lw      t2, T_FINE_STATE(t1)
320         andi    a1, t2, FS_DYING
321         bne     a1, zero, 1f
322         li      a2, L4_IPC_SEABORTED
323         make_busy(t1, a2)
324         lui     t3, KERNEL_BASE
325         ins_busy_list(t1, t3, t2)
326         sw      zero, T_STACKED_FINE_STATE(t1)
327
328 1:      ld      a0, T_PRESENT_NEXT(a0)
329         bne     a0, zero, 2b
330 /* END BIG THREAD CLEANUP LOOP */
```

Listing 6.19: task_new (create_thread) part 8.

Next the target thread's state is checked for the value LOCKS , which indicates that it is preempted during long IPC. Processing of this condition (Lines 256–276) is almost identical to the corresponding IPC abort processing during the lthread_ex_regs system call (see Listing 6.8). The only differences are an additional check for

`DYING` in the partner's state. As in `lthread_ex_regs` the corresponding processing is then performed for the target thread state `LOCKR` (not shown).

The case of the thread being blocked on a page fault during long IPC, indicated by a *stacked* state of `LOCKS` or `LOCKR`, the latter case is shown in Lines 315–326. Again, the handling is equivalent to that in `lthread_ex_regs` (Listing 6.9), except that the case needs to be considered when the target is `DYING`, and that fixing up the target state is not necessary. Lines 328, 329 terminate the loop over the destination task's threads.

```
331           /* now clean up task specific stuff */
332
333           /* unmap gpt */
334           move    a0, s1
335           dli     a1, 63 << 2
336           dli     a2, -1
337           jal     vm_fpage_unmap
338
339           move    a0, s1
340           jal     vm_delete_as
341
342           /* flush asid from tlb (remove window mappings) */
343           ld      a0, T_ASID(s1)
344           bltz    a0, 7f
345           jal     tlb_flush_asid
346
347           /* don't need to flush the STLB as
348                   - user pages are removed via the fpage_unmap
349                   - window pages are never placed in the STLB
350
351           ld      a0, T_ASID(s1)
352           jal     asid_free
```

Listing 6.20: `task_new` (`create_thread`) part 9.

Listing 6.20 shows the cleanup of the task resources. Lines 334–337 call the C function `vm_fpage_unmap` to unmap (actually, flush) the whole address space. As this function is essentially the same as the `fpage_unmap` system call, it is not discussed here. See Section 6.6 for details.

The function `vm_delete_as` is then called to deallocate the page table. This is because unmapping only invalidates page table (and TLB) entries, without removing the entries from the page table, to speed up re-insertion. Next `tlb_flush_asid` is called, which probes all TLB entries and invalidates them if they match the specified ASID (the destination task's). This is necessary, as *window mappings*, used in the temporary mapping area in long IPC, are not entered into a page table, and are therefore not removed by the previous functions. Finally, the destination's ASID is inserted into the free ASID list.

```
353         /* remove from current process hierarchy */
354 7:      tcbtop(t0)
355         ld      t1, T_SISTER_TASK(s1)
356         sd      t1, T_CHILD_TASK-TCBO(t0)
357 /************************************************************
358  * loop through children nailing them
359  */
360 #define start s1
361 #define x     s7
362         move    x, start
363 1:      bne     x, start, 2f
364         ld      t0, T_CHILD_TASK(x)
365         beq     t0, zero, 3f
366
367 2:      ld      t0, T_CHILD_TASK(x)
368         beq     t0, zero, 4f
369         move    x, t0
370         b       2b
371
372 4:      ld      t0, T_MYSELF(x)
373         dsrl    t0, t0, 32
374         dli     t1, TID_TASK_MASK
375         and     t0, t0, t1
376         tid2tcb(t0, s8) /* s8 now has parent */
377         ld      t1, T_SISTER_TASK(x)
378         sd      t1, T_CHILD_TASK(s8)
```

Listing 6.21: `task_new` (`create_thread`) part 10.

Now the children of the destination task must be killed and cleaned up. Since it is necessary to clean up all TCBs belonging to those tasks, and since the task hierarchy is represented by the TCB fields `child_task` and `sister_task`, the cleanup must start at the leafes of the task tree. The algorithm is for depth-first processing the task hierarchy is given as C code in comments, it is here extracted for clarity:

```
x = start;                             // Line  362
while (x != start || x->child != 0) { // Lines 363--365
    while (x->child != 0) {            // Lines 367--368
        x = x->child;                  // Line  369
    }                                  // Line  370
    x->parent->child = x->sister;      // Lines 372--378
    cleanup(x);                        // Lines 331--430
    x = x->parent;                     // Line  432
}                                      // Line  433
```

Note that the TCB contains no explicit parent pointer, so the chief field in the task ID is used to retrieve the parent in Lines 372–378.

```
414 7:      /* set new chief to initial killer in process hierarchy */
415         tid2ttable(s3, t0)
416         lw      t1, (t0)
417         tcbtop(a0)
418         ld      a1, T_MYSELF-TCBO(a0)
419         li      t2, ~TT_CHIEF_MASK
420         and     t1, t2, t1
421         li      t2, TT_INACTIVE_MASK
422         or      t1, t2, t1
423         li      t2,  TID_TASK_MASK
424         and     a1, a1, t2
425         or      t1, a1, t1
426         sw      t1, (t0)
427
428         ld      a0, T_GPT_POINTER-TCBO(a0)
429         move    a1, x
430         jal     vm_tcb_unmap
431
432         move    x, s8
433         b       1b
434 3:      /* END WHILE */
435 /**************************************************************/
436
437         tcbtop(t0)
438         ld      a0, T_GPT_POINTER-TCBO(t0)
439         move    a1, s1
440         jal     vm_tcb_unmap
441         tid2ttable(s6, t0)
442         lw      t1, (t0)
443         li      t2, TT_INACTIVE_MASK
444         or      t1, t2, t1
445         sw      t1, (t0)
446         j       the_after_life
```

Listing 6.22: `task_new` (`create_thread`) final part.

Most of the cleanup code is identical to the cleanup of the original target task's threads. It is simplified by the fact that, after killing the parent, none of the threads can be in communication with a thread surviving the slaughter, a consequence of the clans and chiefs mechanism. Also, the child address spaces do not need to be unmapped, as they can only have received any mapping they have via their parent (directly or indirectly) — also a consequence of clans and chiefs. This task cleanup code is not shown.

The remaining cleanup is shown in Listing 6.22. Lines 415–426 update the task's TID table entry, showing the caller as the new chief, and setting the i-bit to show that the task is inactive.

Finally, `vm_tcb_unmap` is called to finish up cleaning up all remaining task memory. ***FIXME: Need to have a good look at what this code really does, and what `vm_delete_as` does.***

Outside the loop the cleanup of the target task is completed by calling `vm_tcb_unmap` and by marking its TID table entry as inactive (Lines 437–445). Execution then continues with the task creation part (label `the_after_life`, Listing 6.12).

Do `vm_new_as`.

Listing 6.23: `vm_new_as`.

Discussion

`task_new()`, specifically deleting a task, is the most expensive operation in L4, as can be appreciated from the above discussion. The potentially costliest part, however, hasn't even been discussed yet, it is the unmapping of the address space, which will be discussed in Section 6.6. The total time taken for task deletion is essentially unbounded due to the potentially very complex patterns of address-space mappings. A real-time system built on top of L4 would need to understand this, and use tasks and mappings wisely.

Implementation criticism: All the `task_new` code is run with interrupts disabled and is thus non preemptable. While this is ok for pure task creation, it is unacceptable for task deletion. This should be obvious from the discussion of the above code, even without a detailed look at the (potentially very time-consuming) unmapping code. This clearly needs fixing. At the very least, preemption points should be introduced into the unmap code, although that is probably not enough to make real-time guarantees.

Preemption in task deletion will require reconsideration of the `lthread_ex_regs` code, see Listing 6.2.5 on page 86.

6.4 thread_schedule

Listing 6.24: thread_schedule.

6.5 thread_switch

The thread_switch system call performs an explicit time-slice donation to a designated target thread. With a null argument it performs a *yield* operation, i.e., the remainder of the present time slice is forfeit and the scheduler is invoked to pick a new thread to run.

register	*type*	*usage*
a0	I	target TID
t0–t3	T	various temporary
t8	I	caller TCB pointer (+TCBO)

Table 6.4: Register usage in thread_switch. No other registers are used (other than ra, sp).

This is, by far, the simplest system call. Its register usage is shown in Table 6.4, the code is shown in Listing 6.25.

```
0          daddiu  t1, t8, -TCBO
1          daddiu  sp, sp , -8
2          dla     t0, k_thread_switch_restart
3          sd      t0, (sp)
4          lui     t2, KERNEL_BASE
5          ins_busy_list(t1,t2,t3)
6          beq     a0, zero, 1f
7          tid2tcb(a0, a1)
8          lw      t0, T_FINE_STATE(a1)
9          andi    t0, t0, FS_BUSY
10         beq     t0, zero, 1f
11         thread_switch_fast(t8, a1, t2)
12         ld      ra,(sp)
13         jr      ra
14 1:      to_next_thread(t2)
```

Listing 6.25: thread_switch.

Line 0 adjusts the TCB pointer to point to the beginning of the target TCB, as required by ins_busy_list. Lines 1–3 push the restart address on the stack. Lines 4 and 5 invoke ins_busy_list to ensure that the caller is in the busy list (it might have been executing on a donated time slice since being unblocked).

Lines 6–13 perform the switch to the designated target thread (if any). Lines 8–10 verify that the destination is ready. If so, a *fast thread switch* (see Section 5.2.2) is performed to the target. Remember that this leaves all registers as they are. The destination's restart routine, which is invoked at Lines 12–13, is responsible for restoring the destination's context and returning to the user.

In the case of a yield operation (user-supplied destination TID is invalid, or the destination not ready to run), Line 14 invokes the scheduler via the to_next_thread macro (Listing 5.8).

The restart code for the donating thread is correspondingly trivial: It pops its own address off the stack and invokes `syscall_ret`. No user-level register context needs to be restored.

Implementation criticism: It could be argued that not restoring (or clearing) registers opens a covert channel. This would be trivial to close, of course, at the expense of a few cycles.

6.6 `fpage_unmap`

Chapter 7

Other Stuff (Provisional)

7.1 Scheduling

Discuss wakeup queue structure.

Blah blah blah...

7.1.1 `make_busy`

The macro `make_busy` is used to restart a thread blocked on an IPC which is being terminated due to the partner being killed (or ex_reg-ed). It is only used within the `task_new` and `lthread_ex_regs` system calls and is inevitably followed by an invocation of the `ins_busy_list` macro.

The code is shown in Listing 7.1. The `tcb` parameter is a (R/O) register pointing to the target thread's TCB. The `return_code` parameter is an input register which contains the error code which is to be returned from the IPC system call which the target is presently performing.

```
0         daddiu  AT, tcb, TCB_SIZE + ST_EX_V0
1         sd      return_code, (AT)
2         daddiu  AT, tcb, TCB_SIZE - ST_EX_SIZE -8
3         sd      AT, T_STACK_POINTER(tcb)
4         dla     return_code, preempt_ret
5         sd      return_code, (AT)
6         li      AT, FS_BUSY
7         sw      AT, T_FINE_STATE(tcb)
```

Listing 7.1: The code generated by the macro invocation `make_busy(tcb, return_code)`.

Lines 0 and 1 overwrite the stacked value of `v0` with the specified return code. This will force `other_excpt_ret` to return to the user with the specified return code.

Bug/Restriction 15: Terminating nested IPC trashes `v0`.
This way of forcing the system call to return an error code is incorrect. If the operation terminated is a page-fault or exception IPC (which is performed by the kernel, transparently to the user, see Section 4.2.4) all user-visible registers must be conserved, including `v0`. The present implementation overwrites the user thread's `v0` with the IPC error code.
The return code should go into the page-fault/exception IPC stack frame rather then original kernel stack frame. One way to do this is not to stack `v0`, but to store it in a fixed location in the TCB. Only on a nested IPC (Listing 4.9) would the previously saved value be stacked. The `make_busy` macro would then overwrite the `v0` value saved at the fixed TCB location, which would have the desired effect for user-initiated IPCs while conserving the user state for kernel-initiated IPCs.

Lines 2 and 3 unwinds the stack to the original exception stack (discarding any frames belonging to nested IPCs). The restart address is set to `preempt_ret` (Lines 4, 5), and finally the thread is marked `BUSY` (Lines 6, 7).

```
0         daddiu  sp, sp, 8
1         j       other_excpt_ret
```

Listing 7.2: The `preempt_ret` routine.

The `preempt_ret` routine is trivial: It pops the restart address off the stack and jumps to the general exception return code `other_excpt_ret` (Listing 7.2).

Implementation criticism: Rather than performing another jump, and polluting the instruction cache, Line 0 of Listing 7.2 should be prepended to the code of `other_excpt_ret`.

7.1.2 `ins_busy_list`

The `ins_busy_list` macro is invoked to insert a thread into the busy list at the correct priority. This is generally necessary when a thread has been created or has become unblocked (usually via an explicit `make_busy`). It is also required during message delivery (see Listing 5.2), or during preemption, when a thread may be running on a donated time slice (lazy scheduling). The macro also is invoked if a thread's scheduling parameters change, as happens during a `thread_schedule` system call (Section 6.4).

The code is shown in Listing 7.3. The `tcb` parameter is a (R/O) register pointing to the (beginning of the) target thread's TCB. The `kern_base` input register points to the kernel miscellaneous data, while the `temp_reg` parameter designates a scratch register.

Lines 0–1 test the `busy_link` to see whether the thread is already in the busy list (although not necessarily at the correct priority). If so execution continues at Line 16.

Otherwise, the thread's timeslice value is obtained from the TCB (Line 2). If it is zero, the thread is not schedulable (Line 3). Lines 4–8 use the thread's priority value, `TCB.tsp`, to locate the appropriate ready queue, and make it point at the target TCB. Each of the per-priority queues is circular. Hence, if the queue was previously empty, the TCB is linked to itself (Lines 9–11). Otherwise the target TCB is linked into the existing list, behind the TCB previously pointed to by the busy-list pointer, which is then redirected to point to the target TCB (Lines 13–15). This inserts the TCB at the tail of the list (remember that the busy list array points to the tails of the lists).

The "remaining time slice" and "current priority" values of the target thread are then reset to the thread's "normal" time slice length and priority; they might have changed if the thread was preempted while executing on a donated time slice. This ensures that the thread gets a new time slice next time it is scheduled.

```
        ld      AT, T_BUSY_LINK(tcb)
        bne     AT, zero, 254f
        lhu     AT, T_TIMESLICE(tcb)
        beq     AT, zero, 255f
        lbu     temp_reg, T_TSP(tcb)
        sll     temp_reg, 3
        daddu   temp_reg, temp_reg, kern_base
        ld      AT, K_PRIO_BUSY_LIST(temp_reg)
        sd      tcb, K_PRIO_BUSY_LIST(temp_reg)
        bne     AT, zero, 253f
        sd      tcb, T_BUSY_LINK(tcb)
        b       254f
        nop
253:    ld      temp_reg, T_BUSY_LINK(AT)
        sd      temp_reg, T_BUSY_LINK(tcb)
        sd      tcb, T_BUSY_LINK(AT)
254:    lhu     temp_reg, T_TIMESLICE(tcb)
        sh      temp_reg, T_REM_TIMESLICE(tcb)
        lbu     temp_reg, T_TSP(tcb)
        sb      temp_reg, T_CTSP(tcb)
255:
```

Listing 7.3: The code generated by the macro invocation `ins_busy_list(tcb, kern_base, temp_reg)`.

7.1.3 `get_next_thread`

The scheduler is the `get_next_thread` routine, which selects the next runnable thread, based on priority and the round-robin policy. It is shown in Listing 7.4.

The code first checks whether there is a thread which was preempted by an interrupt, if so, this one is chosen (Lines 3–7). Otherwise it examines the busy list in decreasing priority order, skipping empty lists. Any non-`BUSY` entries at the head of the list are removed from the list (Lines 13–18). A whole list may become empty this way, in which case it is marked as such by inserting a null pointer (Lines 19–23). Finally, the thread at the *head* of the first non-empty list is returned as the next one to run. Note that this is the successor of the one pointed to by the busy-list array (which points to the tail). It becomes the new tail (Line 26).

The existence of the idle thread guarantees that there is always a runnable thread to choose.

7.2 Interrupts

7.3 Initialisation

7.4 Sigma Zero

```
tcb_t *get_next_thread(kernel_vars *k) {
  tcb_t *t1,*t2;
  short int i;
  if (k->int_list != END_LIST) {
    t2 = k->int_list;
    k->int_list = t2->int_link;
    return t2;
  }
  for(i = MAX_PRIORITY; i >= 0; i--) {
    t1 = k->prio_busy_list[i];
    if (t1 == 0)
      continue;
    t2 = t1->busy_link;
    while((t2 != (tcb_t *)0) &&
          ((t2->fine_state & FS_BUSY) == 0)) {
     t1->busy_link = t2->busy_link;
     t2->busy_link = 0;
     t2 = t1->busy_link;
    }
    if (t2 == 0)
    {
      k->prio_busy_list[i] = 0;
      continue;
    }
    break;
  }
  k->prio_busy_list[i] = t2;
  return t2;
}
```

Listing 7.4: The scheduler function `get_next_thread`.

Bibliography

[AH98] Alan Au and Gernot Heiser. *L4 User Manual*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, January 1998. UNSW-CSE-TR-9801. Latest version available from `http://www.cse.unsw.edu.au/~disy/L4/`. 5

[EHL99] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00, Version 1.11, Kernel Version 79*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, May 1999. Available from `http://www.cse.unsw.edu.au/~disy/L4/`. 3, 5, 60, 68, 76

[Elp99a] Kevin Elphinstone. Scheduling stuff. Working paper. Saw Mill Linux Project, IBM TJ Watson Research Lab, October 1999. 55

[Elp99b] Kevin Elphinstone. *Virtual Memory in a 64-bit Microkernel*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 1999. Available from `http://www.cse.unsw.edu.au/~disy/papers/`. 23, 45

[Hei93] Joseph Heinrich. *MIPS R4000 User's Manual*. Prentice Hall, 1993. 10

[HEV$^+$98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998. 1

[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 (SOSP)*, pages 66–77, St. Malo, France, October 1997. ACM. 1

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992. 1

[L4M99] L4/MIPS source code, kernel version 79. Available from `http://www.cse.unsw.edu.au/~disy/L4/`, February 1999. 1

[LE95] Jochen Liedtke and Kevin Elphinstone. Guarded page tables on MIPS R4600 or an exercise in architecture-dependent micro optimization. Technical Report UNSW-CSE-TR-9503, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1995. 27

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles (SOSP)*, pages 175–88, Asheville, NC, USA, December 1993. 1, 53

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles (SOSP)*, pages 237–250, Copper Mountain, CO, USA, December 1995. 1, 5

[Lie96a] Jochen Liedtke. *L4 Reference Manual — 486/Pentium/PentiumPro, Version 2.0*. GMD, Schloß Birlighofen, Germany, September 1996. Working Paper 1021. 1, 7

[Lie96b] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996. 5

[Lio77] John Lions. A commentary on the UNIX operating system. Technical report, Department of Computer Science, The University of New South Wales, Sydney, Australia, November 1977. 2

[Lio96] John Lions. *Lions' Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications, P.O. Box 640218, San Jose, CA 95164-0218, USA, 1996. 2

[R4695] Integrated Device Technology. *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, April 1995. 1

Index