THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A Secure Microkernel

Philip Geoffrey Derrin

Thesis submitted as a requirement for the degree
Bachelor of Computer Science (Honours)

Submitted: June 7, 2005

Supervisor: Dr. Kevin Elphinstone
Assessor: Prof. Gernot Heiser

Abstract

Specification of a new kernel API is a challenging task. If the specification is initially developed at an abstract level, it is easy to overlook important implementation issues; on the other hand, writing a real kernel to test a specification involves low-level programming and debugging, which takes a lot of time and effort and which may be difficult to formally verify. This project is an attempt to develop a technique for prototyping a kernel in a high-level functional language, with the goal of being able to rapidly evaluate design decisions and specify the behaviour of the kernel more precisely than an abstract model.

Acknowledgements

I would like to thank all the people who helped to make it possible for me to complete this thesis: my supervisor, Kevin Elphinstone, for the guidance he has given me over the last twelve months; the many people from the ERTOS and Formal Methods groups who provided helpful advice and suggestions, particularly Harvey Tuch, Rafal Kolanski and Adam Wiggins; Sara Falamaki, for her invaluable friendship and support; and my family, for making it possible for me to be here at all.

Contents

1. Introduction **8**
- 1.1. The L4 Microkernel . . . 8
- 1.2. Goal . . . 8
- 1.3. Motivation . . . 9
  - 1.3.1. Rapid Prototyping . . . 9
  - 1.3.2. Formalisation . . . 9
- 1.4. About This Document . . . 10

2. Background **11**
- 2.1. The L4Ka::Pistachio Microkernel . . . 11
- 2.2. L4 Security Issues . . . 13
  - 2.2.1. IPC Security . . . 13
  - 2.2.2. Resource Management . . . 14
  - 2.2.3. Previous Solutions . . . 14
- 2.3. Capabilities . . . 16
  - 2.3.1. EROS . . . 16
  - 2.3.2. Mach and Chorus . . . 16
- 2.4. Haskell . . . 17
  - 2.4.1. Overview . . . 17
  - 2.4.2. State Monad . . . 18
  - 2.4.3. ErrorT Monad Transformer . . . 19

3. The seL4 Kernel **20**
- 3.1. Overview . . . 20
  - 3.1.1. User-level Management of Kernel Memory . . . 20
  - 3.1.2. Endpoint-Oriented IPC . . . 21
  - 3.1.3. API Specification . . . 21
- 3.2. Capabilities . . . 22
  - 3.2.1. User-level Object Allocation . . . 22
  - 3.2.2. Retyping Objects . . . 22
  - 3.2.3. Object Types . . . 23
- 3.3. Endpoints and Inter-Process Communication . . . 23
  - 3.3.1. Identification . . . 24
- 3.4. Threads . . . 24
  - 3.4.1. *ThreadControl* . . . 25
  - 3.4.2. *ExchangeRegisters* . . . 25

3.4.3. Scheduling . . . 26
3.5. The Capability Space . . . 26
3.5.1. Capability Table Entries . . . 26
3.5.2. The *CapCopy* Operation . . . 27
3.5.3. The *CapRevoke* Operation . . . 28
3.5.4. The *Map* Operation . . . 28
3.5.5. Capability Table Structures . . . 29
3.5.6. The Data Space . . . 29
3.6. Errors and Faults . . . 30
3.6.1. System Call Errors . . . 30
3.6.2. Fault IPC . . . 30
3.6.3. Per-Region Fault Handling . . . 31
3.7. Summary . . . 31

4. Modelling seL4 in Haskell **32**
4.1. Overview . . . 32
4.2. The Event Stream . . . 33
4.2.1. Event Generation . . . 34
4.3. System State . . . 34
4.3.1. Kernel Objects . . . 34
4.3.2. Physical Memory . . . 35
4.3.3. Object Types . . . 36
4.4. State Transformations . . . 36
4.5. Faults and Errors . . . 38
4.5.1. Recoverable and Fatal Errors . . . 38
4.5.2. Fault Handling . . . 39
4.6. Event Handling . . . 39
4.7. User-Level Simulator . . . 40
4.7.1. User Context . . . 41
4.7.2. User State Transition Functions . . . 41
4.7.3. An Assembler-like Language . . . 41
4.8. The Main Loop . . . 42
4.9. Summary . . . 44

5. Evaluation **45**
5.1. Usability and Completeness . . . 45
5.2. Rapid Prototyping . . . 45
5.3. A Specification Document . . . 46
5.4. Formal Verification . . . 46

6. Conclusion **48**
6.1. Future Work . . . 48
6.1.1. Further Development of the Kernel . . . 48
6.1.2. A New User Level Simulator . . . 49
6.1.3. Running on Real Hardware . . . 50

Bibliography **51**

A. Annotated Haskell Code **53**
A.1. Kernel API . . . 53
A.1.1. Pointer Types . . . 53
A.1.2. Object Types . . . 54
A.1.3. Events . . . 54
A.1.4. Exceptions . . . 55
A.1.5. Miscellaneous Constants . . . 55
A.2. Kernel State . . . 56
A.2.1. Data Types . . . 56
A.2.2. Public Functions . . . 56
A.3. System Calls . . . 57
A.3.1. Public Functions . . . 57
A.4. Capability Space . . . 63
A.4.1. Types . . . 63
A.4.2. Physical Storage . . . 64
A.4.3. Public Functions . . . 65
A.4.4. Private Functions . . . 71
A.5. Threads . . . 75
A.5.1. Data Types . . . 75
A.5.2. Type Class Instance . . . 76
A.5.3. System Call Implementations . . . 76
A.5.4. Public Functions . . . 79
A.5.5. Constants . . . 81
A.5.6. Private Functions . . . 82
A.6. IPC and Endpoints . . . 83
A.6.1. Data Types . . . 84
A.6.2. Type Class Instance . . . 84
A.6.3. Public Functions . . . 85
A.7. Kernel Objects . . . 88
A.7.1. Data Types . . . 88
A.7.2. Public Functions . . . 89
A.7.3. Type Class . . . 89
A.7.4. Type Class Instances . . . 89
A.8. Physical Address Space . . . 91
A.8.1. Data Types . . . 91
A.8.2. Public Functions . . . 91
A.9. Object Types . . . 92
A.9.1. Public Functions . . . 93
A.9.2. Private Functions . . . 93
A.10. Bootstrapping . . . 94
A.10.1. Public Functions . . . 95
A.11. User Level Machine Model . . . 97
A.11.1. Instructions and Registers . . . 97

A.11.2. User Level State 98
A.11.3. CPU Simulation 99
A.11.4. Public Functions 99
A.11.5. Internal Functions 100

Chapter 1.

Introduction

1.1. The L4 Microkernel

L4 is a high-performance, lightweight operating system kernel. It is a microkernel, which means that it provides only the minimal set of low-level abstractions required for building a complete system.

Unfortunately, L4 suffers from a number of security issues which make it unsuitable for building secure systems. While several attempts have been made to solve those problems, all such attempts have had other limitations. A useable solution appears to require the development of a significantly revised kernel design.

1.2. Goal

The goal of this thesis was to build a prototype for a new microkernel. The new kernel is similar to the existing L4 microkernel, but attempts to solve some of the known security issues by taking a capability-based approach.

I had originally planned to achieve this by modifying the existing L4Ka::Pistachio implementation. However, there is a very great deal of effort involved in understanding an existing kernel implementation and making extensive modifications to it. Trying to do so while the eventual goal is not yet clearly defined, and experimenting with different design alternatives at the same time, is clearly impractical.

The focus of this thesis has therefore changed. Instead of a stand-alone kernel implementation, I have developed an executable model of the kernel. It is written in Haskell, a functional programming language that I believe is quite well suited to this task.

The main focus of this report is the advantages and limitations of building an executable model of the kernel using Haskell. However, in order to provide some context for the discussion of the model, this report also describes the current state of the kernel interface, and some of the issues that motivated the present design.

1.3. Motivation

1.3.1. Rapid Prototyping

A stand-alone operating system kernel runs under performance and implementation constraints that cannot easily be included in an abstract model of a kernel API — whether that model is expressed in prose or using formal methods. When a design is implemented under those constraints, certain aspects of the design may have an adverse effect on performance — or even affect the feasibility of the implementation, depending on the target architecture. Therefore, I believe that starting with a relatively abstract definition of a kernel API risks making decisions that will cause problems later. This implies that it is better to build a low-level prototype early on, so that such problems are more likely to be noticed.

Implementing any large system is challenging; operating system kernels, even microkernels, are no exception. The challenge is made even greater when developing a new kernel because such systems must interact with their host's hardware at a very low level. It is usually necessary to write significant amounts of assembly code; also, incorrect use of privileged hardware operations may cause problems that are very difficult to locate. Building a prototype to test new design features in such an environment is difficult and time consuming, and slows down the design process.

This thesis investigates the possibility that simulating the system's behaviour using an unambiguous, high-level language — for example, Haskell — may be an effective compromise. Such a prototype will expose low-level implementation details which might influence the design of the kernel. At the same time, being written in a high-level programming language and run at user level makes it far easier to write and debug than a standalone kernel implementation, which is a major advantage during the early stages of development.

1.3.2. Formalisation

In order to formally verify an operating system, it is necessary to generate a formal specification of the kernel's API. Previous attempts at deriving such a formal specification from an existing natural-language informal description have found many aspects of the design to be incompletely or inconsistently specified [6, 16].

Developing the new kernel's informal specification as a prototype in a functional programming language may ease this transition considerably. This thesis attempts to show that a prototype implementation developed in Haskell can be made relatively easy to formalise. Such a model can specify the kernel's behaviour in a more readable and concise manner than the more typical C, C++ and assembler languages, but at the same time can be more complete and precise than a natural-language specification.

Also, by using only an abstract model of the hardware, it can avoid being cluttered with complicated architecture-specific implementation details.

1.4. About This Document

The remainder of this report is structured as follows:

- Chapter 2 on the following page provides some context for the work described in this thesis. It gives a brief overview of the existing L4Ka::Pistachio microkernel, and of the security issues present in L4 and some attempts to solve them. It also describes the Haskell functional language, and discusses some of the issues involved in using it to model operating systems kernels.
- Chapter 3 on page 20 describes the present state of the new kernel design, as implemented in the Haskell prototype. It discusses some of the design issues that have been encountered so far, and the solutions chosen.
- Chapter 4 on page 32 documents the design and implementation of the Haskell prototype.
- Chapter 5 on page 45 evaluates the results of the project and discusses its successes and limitations.
- Chapter 6 on page 48 sums up the outcome of the project, and presents some avenues of future work.
- Appendix A on page 53 presents annotated Haskell source code for the executable specification.

Chapter 2.

Background

This chapter provides some context for this thesis, and discusses related work. Specifically, it discusses:

- the existing L4Ka::Pistachio implementation of L4, in section 2.1;
- the security issues in the L4 API, in section 2.2;
- capability-based access control, particularly of IPC operations, in section 2.3; and
- the Haskell programming language used to develop the prototype, in section 2.4.

2.1. The L4Ka::Pistachio Microkernel

L4Ka::Pistachio [7] is an implementation of the L4 microkernel API, version X.2. It is a lightweight, high performance microkernel, which provides user tasks with three basic services: thread management, virtual address spaces, and a communication operation (called *IPC*, an abbreviation of Inter-Process Communication).

L4 provides only a small number of system calls, which are used for performing operations related to the three services it provides. There are also several special IPC protocols, which are used for mapping virtual memory, and for handling interrupts, exceptions and page faults. All other operating system services — such as file systems and device drivers — must be implemented by user-level server tasks.

Threads

Every thread in the system has a *global thread ID* that is unique throughout the system, and a *local thread ID* which is unique among a group of threads sharing the same address space. These IDs are used to identify the thread when performing system calls that modify its state, or when communicating with it.

Every running thread also requires two areas of memory to store its state: a *thread control block* (or TCB) in the kernel address space, and a *user thread control block* (or UTCB) in its own address space.

Scheduling of threads is via a multiple priority round robin scheduler, though there is a mechanism defined for replacing this with a user-level scheduler thread. When the default scheduler is used, threads are scheduled in round robin order from the highest-priority queue of runnable threads; runnable threads at lower priorities are never scheduled.

Two system calls are used to control the creation and execution of threads. *ThreadControl* is used to activate or deactivate a thread, and set its priority, UTCB location, address space, and user-level scheduler and page fault handler. *ExchangeRegisters* is used to modify the state of a running thread.

Some threads are *privileged*, meaning that they are allowed to perform certain operations such as creation of new threads. The initial threads — σ_0 and the root task — are both privileged.

IPC

The IPC operation allows messages to be exchanged between two user-level threads, or in some cases between the microkernel and a user level thread. The messages consist of up to 64 *message registers*, containing a combination of untyped data, *map objects* (discussed in the next section), and *string objects* (which are only available on some implementations).

All messages are synchronous; both the sender and the recipient will block until the message has been transferred. The sender and recipient may each specify a *timeout*, which limits the time that the thread will wait for a message to be transferred. This timeout may have a specific value, including zero; or it may be infinite.

Messages are addressed using local or global thread identifiers. Local identifiers may be used to specify threads that share the caller's address space; all other threads must be identified using their global identifiers. Send operations must specify a recipient, but receive operations may either specify a sender or allow any sender.

Address Spaces

Virtual address spaces in L4 are shared by a group of one or more threads. When a thread is created, its address space must be configured using the *SpaceControl* system call; that call may be used to specify that the thread either has its own new, initially empty address space, or alternatively that it shares the address space of an existing thread.

Virtual memory management in L4 is performed by user-level pager threads. At boot time, the kernel maps all unallocated memory in the system into the address space of the privileged thread σ_0. From then on, all memory mappings are created by sending IPC with map items in the message registers; these specify a region of memory in the sender's address space which is to be mapped into the recipient's address space. This is the *Map* operation. There is also a variant in which the sender transfers its mapping, removing it from its original location in the sender's address space; this is *Grant*. The recipient of a mapping may choose to limit the possible destinations for it.

The kernel keeps records of the ancestry of each mapping. Any thread may call the *Unmap* system call and specify a mapping; the kernel will then revoke any other mappings that have been created with it via *Map* operations. The caller may request that the given mapping itself be removed as well.

When a thread causes a page fault, a message is sent to the *pager* thread that was specified for it using *ThreadControl*, and the faulting thread waits for a reply. The message is in a format defined by the kernel API; it states the nature of the fault and the address at which it occurred. The pager is expected to either use a *Map* operation to resolve the fault and restart the thread, or deny the access and take some other action (such as killing the thread).

2.2. L4 Security Issues

There are some security issues with the L4 API which are a concern from the perspective of designers of secure systems [2, 10, 14]. One of these is the global namespace used for specifying IPC partners, along with the related problem of imposing restrictions on which pairs of threads can engage in IPC; another problem is related to the management of kernel resources, particularly dynamically allocated memory.

2.2.1. IPC Security

There are two security issues with the present IPC model.

First, the namespace for IPC outside the current address space is global, and visible to every thread in the system. This exposes the structure of the system to all threads which need to perform such communication. This causes a few problems. First, it restricts the design of OS personalities somewhat — for example, if a server is multi-threaded, the client threads must be aware of that fact. More importantly, it opens a possible covert channel between areas of the system that should be isolated.

The second issue is that there is no efficient method available for restricting overt communication channels between threads. In fact, until recently, L4Ka::Pistachio did

not apply any restriction at all. There have been several attempts to solve this problem, but every solution has its own issues, most of them related to performance. Refer to section 2.2.3 for a summary of these attempts.

2.2.2. Resource Management

Several of the services provided by L4 need storage space in kernel memory. For example, virtual address spaces require page tables in which the mappings from virtual address to physical frame can be stored, and entries in a mapping database which record the hierarchy of map operations which have been performed on each physical page.

Kernel resources can be managed, to some extent, by user level tasks: system calls exist to map and unmap pages, and to create and destroy threads and address spaces. However, in the L4Ka::Pistachio kernel, the space available to store the in-kernel metadata representing these resources must be allocated from a fixed pool of memory, the size of which is determined when the kernel is compiled. This is obviously not an ideal solution, as the kernel's memory pool can very easily be exhausted — either due to high system load, or a deliberate denial of service attack [10].

2.2.3. Previous Solutions

Several approaches have been proposed or implemented to solve these issues in L4; however, none of them are really satisfactory [2]. Some of the previous attempts and their limitations are briefly summarised below.

IPC Security

Clans and Chiefs were part of version 2 of the L4 API [8]. In this model, each thread is a member of a group of threads known as a *clan*, and each clan has a single thread nominated as the *chief*. Any message passing across a clan's boundary — either from a thread outside the clan directed to a thread inside it, or vice versa — is redirected to the chief, which may forward or discard it.

A major limitation of this approach is its poor performance, due to the many extra IPC operations it causes. An extra IPC operation is required for every clan boundary that a message has to cross. For example, in the common situation in which two communicating threads are members of separate clans contained in a single enclosing clan, three IPC operations are required for each message.

IPC Redirection was developed as a replacement for the Clans and Chiefs model [5]. In this system, tasks are grouped into *redirection sets* which each have a *redirection monitor*. The monitor is responsible for providing the kernel with mappings between (sender, destination) pairs and recipients (which may or may not be the same as the nominal destinations). The kernel is able to cache these redirections, so if two threads are to be permitted to communicate without restrictions, only one extra context switch to the redirection monitor is required for as long as the redirection remains in the cache.

This approach performs better than the Clans and Chiefs model in most cases; however, it increases the penalty for the check when a thread attempts to impersonate another, which is necessary when messages are to be transparently monitored.

IPC Redirectors are used in L4Ka::Pistachio [7]. They are a simplification of the redirection model, in which the kernel does not cache redirections but instead forward *all* messages passing outside the redirection set to a *redirector thread*. This model combines the worst features of the previous two: it suffers from performance problems due to excess IPC operations and must perform a potentially slow search when a redirector impersonates a message's sender.

Virtual Threads replace L4's global thread identifiers with a local address space for each thread, which is managed in a similar way to the virtual memory space. This solves the global IPC namespace covert channel problem, and also enforces mandatory control of the ability to use overt channels. However, to be able to efficiently inform an IPC recipient of the sender's identity, the implementation requires that every sender know and specify its address in the recipient's thread space; if the sender does not know it or specifies it incorrectly, the result is a long-running search to locate the sender in the recipient's space.

Resource Management

Donation: Liedtke et al. have proposed [10] that the privileged σ_0 thread be able to *donate* memory to the kernel to be used to satisfy requests from a specific thread. The donated memory is returned to σ_0 when the specified thread's address space is destroyed, which leads to a major limitation of this solution: if an active thread's requirements decrease over time, there is no way to reclaim the unneeded kernel memory.

Kernel Pagers: Haeberlen and Elphinstone [4] implemented an extension of the existing L4 memory management model, in which the pager thread used for handling virtual memory page faults is joined by a *kpager* thread, which handles kernel memory exhaustion faults. The kpager is able to donate a page to the

kernel on behalf of the faulting client thread; it may also revoke donated memory, for example to store it to disk. The kernel either zeros the contents of reclaimed memory, or converts them to a form that can be validated if the page is returned to the kernel.

This approach solves the problem of reclamation of resources, but introduces a new issue: there is no way for the kpager to tell what a kernel page is being used for, so revoking kernel pages risks severe performance degradation. For example, accidentally revoking the page containing the client's root page table node would unmap the entire virtual address space of the client, and rebuilding it could take a long time.

2.3. Capabilities

A *capability* is an object that represents the right to perform a specific operation on a specific object. In a capability-based system, a task may only perform certain operations if it possesses a capability allowing it to do so; to perform those operations, the task must provide a reference to an appropriate capability when calling a *capability invocation* system call.

Several microkernels other than L4 have used capabilities to restrict tasks' ability to communicate, including Mach, Chorus and EROS.

2.3.1. EROS

EROS [15] provides *entry capabilities*, which allow a process to send requests to another specific process. No distinction is made between processes and threads, so there is exactly one possible recipient of any message sent using a given entry capability.

Invocation of an entry capability can optionally create a *reply capability*, which is a single-use object that can be used to reply to the sender of the original message. Once it is used, it becomes invalid, including all copies of it that may have been sent to other threads.

2.3.2. Mach and Chorus

The Mach [12] and Chorus [13] systems also use capabilities to restrict communication. Unlike EROS, they do not provide capabilities for sending messages to specific threads; instead they allow send or receive operations on intermediate objects called *ports*.

Ports are not tied to a specific destination, so they can be shared by multiple server threads without exposing this fact to clients. They cannot be used concurrently by

multiple server *tasks* (which are groups of threads sharing an address space and a set of capabilities). However, Chorus provides a *port group* object which allows a send operation to address multiple ports (and therefore multiple tasks) at once. Also, both systems allow send and receive capabilities for ports to be sent to other tasks, so migration of a port from one task to another can be performed in a manner transparent to the clients.

The Mach model also includes single-use ports, similar to the reply capabilities in EROS. They become invalid after a single message is send to them, and their capabilities cannot be duplicated. While these are convenient, they are not essential, and they add extra complexity to the IPC path.

Discussion

2.4. Haskell

The executable specification is intended to be the middle ground between an English-language specification, a formal specification, and an implementation. It is important, therefore, to develop it in a language that allows an easy transition into each of those three forms. Haskell one such language.

The following is a summary of the features of Haskell that are relevant to its use in this thesis.

2.4.1. Overview

Haskell is a general purpose functional programming language [11]. It is in widespread use in the research and education communities; several universities, including UNSW, use it in introductory programming courses.

The features of Haskell that are most relevant to this thesis are:

- The type system performs strict checks at compile time; it must be possible to determine the type of every expression. Invalid values such as null pointers are impossible, as are unsafe or implicit type conversions. This simplifies debugging, as most incorrect code will not compile.
- There is a formal definition of the language's sematics [3]. There is only one case in which it is ambiguous; that case is unlikely and easily avoided. The pure functional semantics and strict typing make Haskell quite similar to HOL, which is being used in the ongoing L4 verification project [16].

- By constructs called *monads*, it is possible to write functions that process state changes in a manner that superficially resembles an imperative language. These are easier to read than non-monadic functional programs when performing complex and inherently imperative state transformations, especially for developers who have no experience with functional languages. This is discussed further in sections 2.4.2 and 2.4.3.

The following two sections describe the use of the two monads used in the implementation of this thesis. For further information about monadic programming and its use in Haskell, please refer to the tutorial at `http://www.nomaware.com/monads/html/index.html`.

2.4.2. State Monad

Haskell is a *pure* functional language, meaning that expressions in the language must neither depend on the global state of the system, nor change it as a side effect[1]. This makes it fundamentally very different to imperative languages such as C, which are typically used for kernel implementations.

One problem with pure functional programming is that systems that operate in a complex state space — for example, models of operating system kernels — must pass their entire state around in function parameters and return values. This clutters the code and results in programs that are difficult to read, especially for people unfamiliar with functional programming. As a trivial example, this is a function for which the state data is an integer; it adds a given value to the state, and returns the new value converted to a string:

```
updateAndShow :: Integer → Integer → (Integer, String)
updateAndShow step state = (new_state, show new_state)
        where new_state = state + step
```

Note that the function must both accept and return an extra INTEGER value representing its state. Also, the entire result must be constructed in one expression. If the computation conceptually involves a sequence of imperative steps, representing it this way can be quite difficult.

However, Haskell includes support for *monadic* programming [17], and provides a *monad* called STATE. This monad provides a means to hide the explicit state parameters, and express sequences of state transitions in a form that superficially resembles an imperative program. To repeat the previous example using STATE:

```
updateAndShow :: Integer → State Integer String
updateAndShow step = do
```

[1]There are a few exceptions to this rule, including functions in the IO monad and the foreign function interface. These are special cases, and can only be used under specific conditions.

```
        old_value ← get
        let new_value = old_value + step
        put new_value
        return (show new_value)
```

The get and put functions used in this example are for fetching and setting the current state. Note that due to Haskell's strict typing requirements, transitions between functions that are not in the STATE monad and those that are must be explicit — using the runState function to evaluate expressions in STATE from outside the monad, and the let statement to evaluate non-monadic expressions from inside the monad.

For complex state transformations, writing programs in monadic style provides a significant improvement in readability over traditional functional programming. This technique has been used extensively in this project.

2.4.3. ErrorT Monad Transformer

One limitation of the STATE monad is that there is no straightforward way to halt processing of a sequence of operations when an error occurs. The ERRORT monad transformer provides a mechanism for doing so.

Using ERRORT allows sequences of computations in a monad to fail and return an error value. Further statements in the monad will not be evaluated; the error will be passed up the call stack in a manner that resembles a C++ or Java exception, until it reaches a call to catchError. For example, to extend the updateAndShow function defined above so it returns an error message if the step is not positive:

```
increaseAndShow :: Integer → ErrorT String (State Integer) String
increaseAndShow step = do
        unless (step > 0) $ throwError ``step must be postitive''
        lift $ updateAndShow step
```

Again, Haskell's strict typing requires all transitions in and out of the ERRORT monad transformer to be explicit. The runErrorT function enters the monad; let evaluates a non-monadic expression; and the function lift adds the ERRORT transformer to the type of a function that is already in STATE.

Chapter 3.

The seL4 Kernel

The Haskell code, which is presented in appendix A on page 53, should be considered the authoritative specification of the present kernel interface. This chapter serves as a high-level overview of the interface, as well as explaining the reasoning behind some of the design decisions. The design and implementation of the model itself is described in chapter 4 on page 32.

The design described in this chapter is preliminary, and is not yet ready for serious use. There are still several areas which have not yet been properly investigated, and some things — particularly the interrupt handling mechanism — have not been specified at all. The kernel is presently known as seL4, an abbreviation of "secure embedded L4".

3.1. Overview

This design was loosely based on that of the existing L4Ka::Pistachio kernel, as described in section 2.1 on page 11. It is similar in that the kernel provides the same three basic services: thread management, virtual address spaces, and communication between threads. In many cases, these services are provided in a manner that is at least superficially similar, if not actually the same. There are, however, some major changes, which are intended to solve the problems mentioned in section 2.2.

3.1.1. User-level Management of Kernel Memory

With the exception of a small amount of statically allocated memory used for kernel code and the kernel's stack, *all* memory resources in the system are managed by user-level tasks. This is achieved by giving user threads *capabilities* to access areas of memory, and requiring that capabilities to unused memory be provided when creating objects.

This change is intended to solve the kernel resource allocation problem. The requirement that threads explicitly identify the memory to be used when creating a new kernel object provides a mechanism for user-level accounting of kernel resources,

and allows OS personalities to prevent denial of service attacks by non-privileged threads.

In this respect, the design resembles that of EROS [15]. Section 3.2 discusses kernel resource capabilities in more detail.

3.1.2. Endpoint-Oriented IPC

The other significant change is that IPC operations no longer specify the partner thread explicitly. Instead, there is a new kernel object called an *IPC endpoint*, which is similar in spirit to a Mach port [12]. All send and receive operations must specify exactly one endpoint. Endpoint objects, like all other kernel objects, are managed using capabilities; the references used to invoke them are specific to the invoking thread's address space.

There are several advantages to this approach, compared to that of L4:

- Most importantly, user-level tasks can only communicate through channels that have been explicitly authorised by the OS personality.[1] It is therefore possible to prevent information leakage or denial of service attacks by untrusted tasks, without adding any extra complexity to the IPC path.
- Because there is no longer a global namespace for specifying IPC partners, communication channels are not tied to a particular client or server thread. Both client and server processes are free to delegate communication to other threads inside their address space (or elsewhere, if they are allowed to share their endpoint capabilities). Server processes may even become multithreaded without changing their external interface.
- IPC can also be transparently redirected or monitored by privileged threads, by replacing the sender's endpoint capability with one that a monitor is listening to. The monitor may choose to forward the message to the original recipient.

See section 3.3 on page 23 for more information about endpoints.

3.1.3. API Specification

Of the following four sections of this report, the first discusses the capability system; this is followed by one section for each of the three services provided by the kernel: threads, address spaces, and IPC. These sections provide an overview of the features of the new kernel interface; for a more detailed description, refer to the Haskell prototype in appendix A.

[1]Covert channels are considered to be outside the scope of this thesis.

3.2. Capabilities

One of the fundamental design principles for this kernel is that all access to kernel operations and resources by user-level tasks is via invocation of capabilities. Each task has a set of capabilities to access kernel objects; each object supports one or more kernel operations, which the task may or may not have permission to perform. In this context, kernel operations include system calls, and also accesses to the contents of a page via a virtual memory mapping. No such operation can be performed without an appropriate capability, because the capability itself specifies the object that is to be operated on.

3.2.1. User-level Object Allocation

All access to kernel resources is via capabilities; this includes allocation of new resources for kernel objects. This is achieved by giving the initial user-level server capabilities to use *all* unallocated memory in the system — there is no separate reserved space for dynamically allocated kernel memory. If a new kernel object, such as a thread, is required, the user-level task that wishes to create it must allocate some unused memory to which it has a capability.

3.2.2. Retyping Objects

Once a user-level task has allocated some unused memory, it invokes the *Retype* system call. This system call, given a capability, transforms the object it points to into the requested type.

When performing the *Retype* system call, it is necessary to specify one of the several types of data that can be stored in that memory. These types include ordinary user-accessible memory, and various objects that are used by the kernel to provide resources to user level — including threads, page table entries, and IPC endpoints.

Object types are global; once an area of memory has been retyped, *all* capabilities to that area of memory can be used to access the new object — as long as their permissions are sufficient.

Alternative Approaches

There were several alternative approaches considered for the storage of type data, other than global typing. They involved storing a type tag in each capability mapping, and preventing multiply-typed pages by either:

- traversing the entire tree of mappings for the object and changing all their types, or

- traversing only the descendants of the retyped mapping, and enforcing that only one thread can have the ability to retype a capability at any time, or
- limiting the use of retype to a trusted and privileged set of user-level threads.

There are two major motivations for the use of global types for kernel objects, rather than having a type tag attached to each capability: efficiency and reliability. The global type tag prevents any situation in which a kernel bug or misbehaving user-level thread causes a page to be interpreted as containing an object of a type other than that which it actually contains. This could otherwise be avoided by a sufficiently careful memory allocation server, or by spending a long time retyping every reference to an object; but unlike those solutions, a single global type tag for each object is both efficient and reliable.

3.2.3. Object Types

The following types of object have been defined:

- Untyped memory. The only valid operation for this type is the *Retype* system call.
- Data memory. Depending on the set of permissions it has, a user-level task may be able to read, write and execute the contents of pages with this type.
- Communication endpoints, described in section 3.3.
- Thread control blocks, described in section 3.4 on the following page.
- Capability table entries, described in section 3.5.1 on page 26.

3.3. Endpoints and Inter-Process Communication

Communication between user-level tasks in seL4 uses the *Send IPC* and *Receive IPC* system calls. These are invoked on a kernel object that exists specifically for this purpose — an *IPC endpoint*. When one thread invokes *Send IPC* and another invokes *Receive IPC* on the same endpoint, a message is transferred from the former thread to the latter. All messages are synchronous; the thread which performs the first invocation will block until the message is transferred.

If multiple threads are waiting to send a message through the same endpoint, and another thread attempts to receive from an endpoint, *one* of the send operations will succeed at that point; the other sending threads will continue to wait. The opposite situation, when multiple threads are waiting to receive from the same endpoint, is similar.

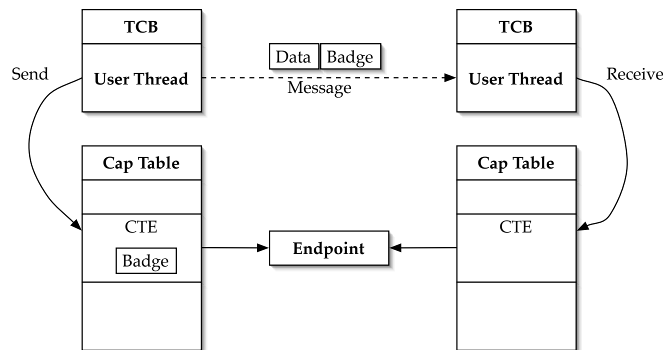

Figure 3.1.: An IPC operation. The badge in the message is copied from the sender's endpoint capability.

The order in which the send operations proceed in this case is defined by the implementation. In the prototype, the order is last in first out; however, first in first out is likely to be more useful in practice.

3.3.1. Identification

It is only possible to wait for messages to a single endpoint at a time. Therefore, if a single server thread is providing a service to multiple client threads, it must provide every client with the same endpoint. So when a server thread receives a message, the kernel must provide some information about which client thread sent it.

This is achieved by tagging every endpoint capability with an integer, called a *badge*, which is provided to the receiver. This tag can be set to an arbitrary value by the user-level server that provides the thread with the capability to the endpoint. Figure 3.1 shows an example IPC operation.

3.4. Threads

Each user-level thread of execution has a corresponding kernel object, a *thread control block* (or *TCB*), which is used to store its configuration and saved context. The creation of a new thread is accomplished simply by retyping memory to become a TCB, and destruction by retyping a TCB to something else.

However, a thread requires configuration to be able to execute. Therefore there are two additional system calls that are used for initialisation and control of existing threads. These system calls — *ThreadControl* and *ExchangeRegisters* — are similar in purpose to the system calls of the same names in L4Ka::Pistachio.

3.4.1. *ThreadControl*

ThreadControl is generally used once for each thread, before it runs for the first time. It sets various configuration parameters for the thread, including:

- the thread's priority, which is used as discussed in section 3.4.3;
- a capability to the root of the structure that maps the thread's capability space (see section 3.5 on the next page);
- a capability to the thread's default page fault handler endpoint (see section 3.6.2 on page 30); and
- optionally, a capability to a table of region-specific page fault handlers (section 3.6.3).

For the purposes of the *CapRevoke* system call, the three capabilities are treated as if they had been copied into the TCB; i.e. a subsequent call to *CapRevoke* on any of the original capabilities will remove the thread's access to that capability, reversing the effects of the *ThreadControl* call.

It is possible to call *ThreadControl* on a running thread; this would most commonly be done to change the thread's priority. To allow such a call to avoid re-copying the three capabilities, *ThreadControl* has a bitmask argument which is used to specify which of the configuration parameters should be set.

3.4.2. *ExchangeRegisters*

The primary purpose of *ExchangeRegisters* is to allow a user-level server thread to save and restore the state of its clients. It provides the caller with direct access to the execution contexts of other threads, by copying all or part of the context between two specified threads.

ExchangeRegisters expects four parameters. The first two are capabilities to the TCBs of the source and destination for the copy operation; the caller must have read permission for the source and write permission for the destination. The third is a pointer, in the caller's address space, to an area of physical memory large enough to save an architecture-dependent subset of a thread's state; it is required only if either the source or the destination is the current thread. The fourth is a bitmask that specifies the operation to be performed.

The meaning of the bits in the mask is partly architecture-dependent. The bits that are always expected to be present have the following meanings when set:

- Copy an architecture-dependent subset of the register set which cannot be copied directly to or from the current thread's context. This includes the stack pointer, the program counter, and any registers required to perform a call to *ExchangeRegisters*. If the source or destination is the current thread, a block of memory specified by the caller will be used to copy these registers, instead of the current thread's context.
- Copy the integer register set, except for the registers in the subset mentioned above.

There will be additional architecture-specific bits that allow control of the threads' use of the floating point and vector units (if they are present), including copying the contents of their register sets. Depending on the architecture and on the circumstances at the time of the call, it may be possible to perform these operations without any copying of registers to or from memory.

3.4.3. Scheduling

The scheduler is essentially identical to that of L4Ka::Pistachio. It is a multiple-priority round robin scheduler, with a separate queue of runnable threads for each priority. Threads are chosen from the highest priority non-empty ready queue.

There is a *Yield* system call, which may be invoked at any time without a capability. This allows a thread to give up the remainder of its time allocation. This system call will be extended in future to allow the time to be donated to a specific thread instead.

3.5. The Capability Space

It was noted in section 3.2 that virtual memory accesses are considered a kernel operation invoking a capability, similar to a system call. It follows that there is not necessarily any distinction between the virtual memory address space and the capability address space. Capabilities, either to kernel objects or to pages of ordinary memory, are mapped to the physical memory used to back them by an implementation-defined page table structure.

3.5.1. Capability Table Entries

While the implementation defines the structure of a capability table, all cap tables are constructed out of arrays of *capability table entry* objects, or *CTEs*. Each CTE maps an address or region of addresses in a user task's address space to a capability to a kernel object; the capability itself is stored in the CTE, where it is not directly accessible from

user level. The contents and layout of the entries are left up to the implementation, but they are expected to contain at least:

- a physical pointer to a kernel object;
- a set of permissions;
- space to store a single machine word, the meaning of which depends on the object's type; and
- meta-data to track copies made of the capability, to be used by the revoke operation.

3.5.2. The *CapCopy* Operation

The *CapCopy* system call is intended for use by the servers responsible for memory allocation and virtual memory paging. It directly accesses a specified capability table entry, copying the capability that is mapped to a given address in the caller's address space. If the destination CTE already contains a valid capability, it is revoked, as if the *CapRevoke* system call had been used on the destination.

Servers can use this system call, along with *Retype*, to construct capability spaces for their clients. The sequence of operations used to do this depends on the capability table structure in use. For example, with an implementation using a two-level table, a server might do the following to create an address space with a data page mapped at address x:

1. Allocate an unused page of memory and call *Retype* to convert it to an array of CTEs.
2. Make the newly created array of CTEs the client thread's page table, using *ThreadControl* (see section 3.4.1 on page 25).
3. Allocate another page, and *Retype* it to an array of CTEs. This is the second level of the table.
4. Use *CapCopy* to place a capability to read the second-level table in the root table. The offset into the table is determined by multiplying the n most significant bits of x by the size of a CTE, where n and the CTE size are implementation-defined.
5. Allocate a page and *Retype* it to a data page.
6. Use *CapCopy* to place a capability to access the new data page into the second-level table.

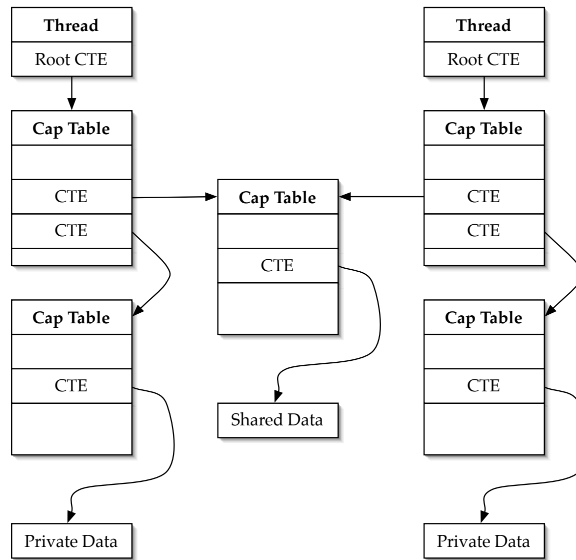

Figure 3.2.: Sharing using capability tables

As shown in figure 3.2, it is possible for capabilities to a single second-level table to appear in two different root-level CTEs — in this case, in the capability tables of two different user-level tasks. This provides a simple mechanism for sharing regions of the address space between multiple tasks. In a system using a variable radix page table, such as a guarded page table, regions of arbitrary size could be shared.

3.5.3. The *CapRevoke* Operation

The *CapRevoke* system call is similar to L4Ka::Pistachio's *Unmap* system call. It locates all capabilities that have been created by copying a specified capability, and invalidates them so they can no longer be invoked. At the caller's request, it may also invalidate the specified capability.

3.5.4. The *Map* Operation

CapCopy is useful for constructing address spaces, but may not be appropriate if a client task wishes to share a kernel resource it possesses. Using *CapCopy* to control sharing requires either that at least one client is given direct access to part of the other client's address space, or that all sharing requests are mediated by a third party with access to both clients' address spaces.

While these limitations may be adequate in some situations, it is often desirable for a client thread to quickly send a capability to another thread — while neither possessing capabilities to the recipient's address space, nor knowing the details of the kernel's capability table implementation. For example, a client may wish to nominate an endpoint for a server to use to send notification that a request has been completed.

The new kernel therefore has a second mechanism for sharing capabilities: the *Map* operation. This presently works by invoking a system call on a thread, and providing two user-level addresses — one in the caller's address space, and one in the recipient's. A capability is then copied from the address in the sender's space to that in the recipient's space.

There are several major problems with *Map* as it is presently defined, so its semantics will most likely change significantly in the near future. Most importantly, the recipient has no control over the new location of the capability, and no way to detect that a *Map* has occurred.

3.5.5. Capability Table Structures

The present kernel prototype implements its capability space using a simple two-level page table. It would, however, be possible for other versions of the kernel to use different structures, depending on the requirements of the system. For example, on a 64-bit architecture, it may be more appropriate to use a guarded page table [9].

In the present API, changing the page table structure would alter the semantics of the *CapCopy* and *Map* operation. This is potentially a problem, especially since *Map* is intended to be used by client applications. Any application using this call would depend on a specific page table structure, and would require modification to work with kernels using a different structure.

It is likely that this problem will motivate a change in the semantics of *Map* in the near future. On the other hand, *CapCopy* is intended to be used by memory allocation servers and other components of the OS personality which are system-specific in any case, so I believe it would not be such a major problem for it to expose the page table structure.

3.5.6. The Data Space

Many hardware architectures specify a page table structure that is used by the MMU to attempt to resolve page faults without entering kernel code. For example, most 32-bit PowerPC CPUs support a hashed page table structure that acts as a cache for the main page table maintained by the kernel.

It may be desirable for the kernel to support access to such a structure from user level. We envisage adding a new kernel object type which is used to store the hardware

page table structure for the current thread, and allowing user-level code to request that a capability to access data memory be copied to the corresponding hardware page table entry (or to simply do this automatically when necessary). The prototype kernel does not do this, however.

3.6. Errors and Faults

When the kernel detects an error caused by an operation at user level, it must report the nature and cause of the error so it can be handled. Errors that are reported to user level fall into three broad categories:

- Invocation of a capability which has an insufficient set of permissions;
- Invocation of a capability that does not exist, or is of the wrong type for the requested operation; and
- Errors caused by invalid system call arguments (other than capabilities).

These errors are reported to user level using two different mechanisms, depending on what sort of event that led to the error.

3.6.1. System Call Errors

If an error is encountered during a system call, the kernel can simply cancel processing of the system call. When this happens, the failed call returns an error code corresponding to one of the three classes of error defined above. If the error involved a capability, the address of the capability will also be returned; this allows the caller to determine which capability caused the error in cases where more than one is being used.

Wherever possible , the kernel should detect any errors and cancel the operation before any changes are made to the kernel's state.

3.6.2. Fault IPC

If a memory load or store operation fails, it is not possible to simply cancel the operation and return an error code. Instead, the user-level task responsible must somehow be suspended while the fault is handled, and then resumed later if appropriate.

The mechanism used to do this is similar to that used by L4Ka::Pistachio. In that kernel, each thread has another thread nominated as being responsible for handling page faults. An IPC is sent to the fault handler thread by the kernel whenever a page fault is detected.

The new kernel retains that mechanism, with only one major modification. Since IPC operations now use endpoints, it is no longer appropriate to nominate a thread as a fault handler; instead, there is a fault handler IPC endpoint. One or more pager threads may listen to this endpoint, waiting for fault messages sent by the kernel. Also, multiple user-level tasks may share a fault endpoint — as for any other request to a server, they can be distinguished by setting a different badge on each thread's capability (see section 3.3.1).

3.6.3. Per-Region Fault Handling

If a region is being shared between multiple threads, it may not be appropriate for fault messages for that region to be passed to different fault handlers depending on which thread caused them. Instead, the messages could be passed to a dedicated fault handler for the shared region, or to the fault handler for the thread that owns the region.

The present implementation achieves this by using a table of fault handlers, which shadows the top level of the capability table. Every time a fault occurs at a specific address, the kernel looks up the address in the fault handler table; if there is a capability to an endpoint present, it sends the fault IPC to that endpoint. Otherwise, the fault is sent to the thread's normal fault handler.

A more general approach would be to allow the entire capability table structure to be mirrored in the fault handler table. This could work with more complex structures, such as guarded tables.

Since the default fault handler will be used if the fault handler table is not set, this feature may be ignored by the user if it is not required.

3.7. Summary

This chapter has presented a preliminary design for an L4-like microkernel that uses capabilities for access control and resource management. There are still many unresolved issues with the design; however, it is sufficiently complete to be used to demonstrate an executable specification.

Refer to section 6.1.1 on page 48 for a list of known problems with the present design.

Chapter 4.

Modelling seL4 in Haskell

This chapter describes the construction of a model implementation of the seL4 kernel, using the functional language Haskell. The purpose of this model is to provide a framework in which to rapidly prototype the kernel design as it is developed; in future it may also function as an executable specification of the kernel.

The present state of the kernel design is described in chapter 3 on page 20; a discussion of the relevant features of Haskell itself can be found in section 2.4 on page 17. The source code for the model is in appendix A on page 53.

4.1. Overview

When developing this model, it was necessary to find a compromise between being close enough to a full implementation to uncover possible implementation issues and allow execution of realistic user-level systems, and being abstract enough to avoid unnecessary difficulty in formalisation. I chose to achieve this by explicitly modelling all elements of the kernel state that could affect the execution of user level programs, but to avoid modelling the architecture-specific hardware features that a complete implementation must support. I also took advantage of Haskell's strict type system in places that would not have been possible in a stand-alone kernel.

The modelled system therefore has:

- no data cache — read and write accesses all access physical memory directly;
- no instruction cache, nor a binary representation for instructions, nor any storage of program text in memory — programs are stored in a data structure which is opaque to the kernel code, in each thread's TCB;
- no cache for virtual address translations — every memory access triggers a page table lookup;
- no current register set — all user-level accesses to registers directly access the saved-context area in the current thread's TCB;
- data stored in physical memory is strongly typed, so bugs caused by type errors are always caught at run time, and often at compile time.

The kernel model works by processing a stream of data objects that correspond to the events that a real kernel implementation would need to respond to. It determines how to respond to each event by referring to a data structure that contains the entire state of the system, and then modifies the state appropriately. The generation of new events depends on a subset of the kernel state data that represents the state of the currently running user-level task.

The following sections describe each of these aspects of the kernel model:

- the generation and contents of the event stream, in section 4.2;
- the structure of the system state data, in section 4.3 on the following page;
- the transformations applied to the state data while handling events, in section 4.4 on page 36;
- the handling of faults and error conditions, in section 4.5 on page 38;
- the event handling functions, in section 4.6 on page 39;
- the user-level simulation, in section 4.7 on page 40; and
- the main loop, which ties all the other components together, in section 4.8 on page 42.

4.2. The Event Stream

The model of the kernel processes a stream of events, each of which corresponds to something that would cause kernel code to execute on a real system:

- invocation of a system call by a user-level task;
- an interrupt from a timer (or some other hardware device);
- an access to a virtual memory region that cannot be translated by the MMU; or
- some illegal action taken by a user-level task.

Additionally there is an event that causes the kernel model to print some debugging output. This does not correspond to a real system event, but it is useful to know the current state of the simulation. This set of possible events is represented by the Haskell type EVENT (see section A.1.3 on page 54).

4.2.1. Event Generation

In a simplistic model, events could be generated by hand and stored in a list, having the kernel simply process the contents of the list in order. This has limitations, however. It is difficult to generate a list of events by hand; writing tests this way would take a long time. Also, in a real system, the order of future events may depend on actions taken by the kernel — such as scheduling a particular thread, or changing the contents of a thread's address space. Therefore, generating events this way would not provide a very realistic simulation of the kernel's behaviour; it might be suitable for a high-level model, but not for one detailed enough to uncover implementation issues.

To overcome these problems, the model described by this report incorporates simulation of user-level tasks as well as of the kernel itself. Events are generated by a user-level CPU simulator, which executes program code associated with the thread that has been scheduled by the kernel; one or more instructions from the program are executed until an event is generated. The actions of each instruction depend on, and may change, the user-level state of the current thread.

Once an event is generated, it is processed by the kernel. This may change the current thread's user-level state; also, the kernel may decide to schedule a new thread. This technique overcomes the limitations of a hard-coded event list: events may be generated by a program interacting with the kernel, and the actions of the kernel may influence the generation of future events.

The user-level CPU simulator is discussed further in section 4.7 on page 40.

4.3. System State

The system's collection of state data is represented by the Haskell type KERNELSTATE. It contains an array of data structures that each represent one frame of physical memory, that may contain state data belonging to the kernel or to a user-level task; it also contains a small amount of statically allocated kernel data, most importantly a physical pointer to the current user thread's TCB.

Figure 4.1 on the following page shows the composition of the state data structure.

4.3.1. Kernel Objects

There are several options for the representation of physical memory. It could be realistically represented as a large array of integers — one for every word of data. However, accessing the contents of kernel data structures would then involve calculation of offsets into the object for each component of the data, which would most likely

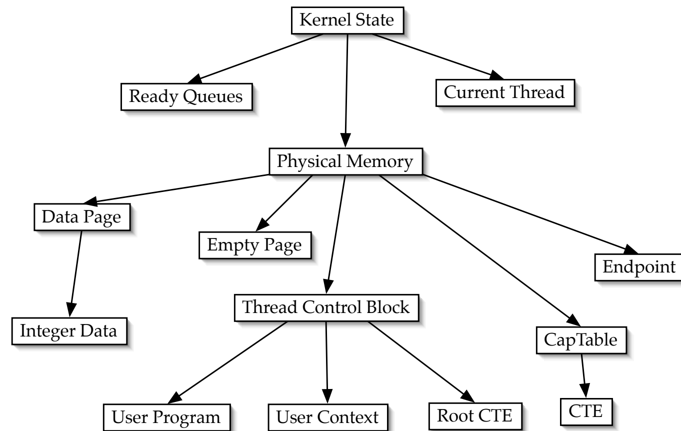

Figure 4.1.: Composition of the kernel state data

result in nearly incomprehensible code. So instead, I make use of Haskell's algebraic data types.

Complex data objects in Haskell are usually represented using an algebraic data type, defined by a set of one or more *data constructors*. Each constructor is a function from zero or more data elements to an object of the defined type. So, for example, the definition of a data type representing an IPC endpoint might look like this:

```
data Endpoint = IdleEP | SendEP ThreadPtr | RecvEP ThreadPtr
```

The | symbol is read "or". So an Endpoint can either be idle, when constructed with IdleEP; or waiting for a send (SendEP) or receive (RecvEP) operation to complete. In the latter two cases the constructor takes a thread control block pointer as a parameter; it points to the first thread in the queue.

The kernel model uses the definition above to represent endpoints, and there are similar structures for thread control blocks and capability table entries. Refer to chapter 3 on page 20 for a discussion of the purpose of each of these object types.

4.3.2. Physical Memory

There are two alternatives for using a kernel object, as defined above, in a physical memory model. One possibility is to convert it into a sequence of integers when storing it in the modelled memory, and then back again before using it in the kernel; this approximates real physical memory while still allowing reasonably clear Haskell code in the functions that use the structure. However, doing this would remove the

ability to check the type of the object at runtime, causing possible kernel bugs if an object of the wrong type is accessed.

We therefore store every kernel object as is, wrapped in an algebraic data type called PhysicalPage. This type acts as a tagged union between all the possible types of contents of a frame of physical memory. It has one constructor for each of the three types of kernel object, one for untyped integer data, and two containing nothing. The latter two are Untyped, which represents an untyped frame which contains no data and is available for use by user tasks, and KernelPage, which represents a frame containing kernel code or statically allocated data that should not be visible outside the kernel. PhysicalPage is defined in section A.7.1 on page 88.

The physical memory space is represented by an array of PhysicalPage objects. Functions are provided in the PSpace module that get or set data in this array; the data may be of any type that is a member of the Haskell type class Object. If higher-level kernel code attempts to get or set a kernel object which is of a different type to that which is already present in the specified page, the kernel will halt with an error message indicating undefined behaviour.

4.3.3. Object Types

The PageType module contains a function that queries the type of the object stored in a given frame. This is module is used by the system call handlers to explicitly check that a capability provided by the user refers to an object of the expected type. After this check is done, the kernel assumes that the object is the right type. An equivalent check in a real-world implementation would most likely use a global table of frame types; the Haskell model needs no such table, because it can find the type by determining which constructor was used to create the PhysicalPage.

There is also a function that changes the type of a frame's contents. It is responsible for cleaning up any kernel objects that previously existed in the frame. For example, in the case of a thread control block, it must remove the thread from the scheduler's queue, cancel any IPC operation involving the thread, and revoke the capabilities that point to the thread's capability table and fault IPC endpoint. It then creates a new kernel object of the requested type, wraps it in a PhysicalPage, and stores it in the physical memory array.

4.4. State Transformations

At the top level, the kernel consists of a state transition function; given an event and an initial state structure, it returns an appropriately updated state structure. The updates might include changes to the contents of kernel structures such as thread control blocks, capability tables, or the current thread pointer; or changes to user-level

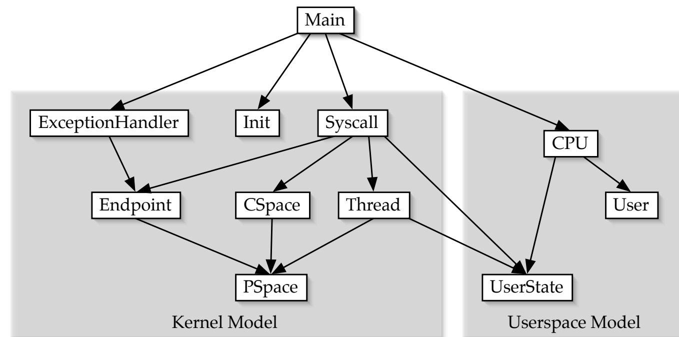

Figure 4.2.: Flow of control between Haskell modules. An arrow between two modules, A → B, indicates that module A modifies the state using functions in module B.

state data such as a thread's register state. Note that these changes might influence the generation of the next event, which depends on the user-level state of the current thread.

The modifications to the kernel state in response to a specific event are potentially quite complex and also depend to a great extent on the current state. Therefore it is most convenient to write them in Haskell's `STATE` monad. This allows the state transforms to be chained together in a manner that resembles an ordinary procedural program, and therefore appears more familiar and readable to a kernel developer. The `STATE` monad is described in more detail in section 2.4.2 on page 18.

Most of the functions in the Haskell model's kernel code, therefore, are of the type `STATE KERNELSTATE a`; where `a` is the type of the result of the function. This is equivalent to a state transition function of type `KERNELSTATE` → (`KERNELSTATE`, `a`). Typically there are also some input parameters. For example, there is a function in the physical memory model:

```
getPage :: POINTER → STATE KERNELSTATE PHYSICALPAGE
```

Given a physical pointer and the current system state, this function returns a new state (which in this case is unmodified) and the contents of the specified area of physical memory.

For each major component the kernel, there is a Haskell module containing functions that perform all of the relevant state transformations. The relationship between these modules is shown in figure 4.2.

4.5. Faults and Errors

Whenever an attempt to process an event encounters a recoverable error — for example, because an invalid capability was invoked or an unmapped memory address was accessed — the kernel must immediately stop processing the event and take some other appropriate action. In the Haskell implementation, this is achieved by adding the ErrorT monad transformer to the existing State KernelState monad. See section 2.4.3 on page 19 for a detailed description of ErrorT.

There are three different classes of error, which are enumerated by the type ExceptionType:

```
data ExceptionType = PermissionFault | CapFault | InvalidArgument
        deriving (Enum, Show, Eq)
```

The values correspond to an insufficient set of permissions, an invalid capability, or a bad system call argument, respectively.

The error values which are thrown and caught in the ErrorT monad are of a user-defined type; a the type Exception. An Exception structure contains a collection of information about the nature and cause of the error, including an ExceptionType value:

```
data Exception = Ex { exType :: ExceptionType,
                      exDesc :: String,
                      exCTLevel :: Int,
                      exAddress :: CapPtr }
```

4.5.1. Recoverable and Fatal Errors

Early versions of the kernel model consisted almost entirely of functions in the ErrorT Exception monad transformer. All failures — including both recoverable failures caused by the actions of user-level tasks, and fatal errors caused by bugs in the kernel — led to an Exception object being thrown using throwError. However, I found that this made it difficult to distinguish between errors caused by misbehaving user-level code, and errors caused by bugs in the kernel.

In the interests of making debugging and formal modelling of the kernel easier, the functions in the kernel have now been split into two distinct groups: those which can fail, and those which must not.

Functions that must be able to recover from failures are in the ErrorT Exception monad; they include all the top-level system call handlers, which explicitly check for all error conditions that could be caused by the user. When an Exception is thrown by one of these functions, the kernel typically handles it as specified in section 3.6 on page 30.

The lower-level kernel functions, in which all failures are a result of kernel bugs, are not in the ErrorT monad. The model's behaviour after detecting a failure in these functions is undefined. This is expressed in Haskell by evaluating $\perp$ (which is pronounced as "bottom", and in Haskell source code is written as "undefined"). The interpreter prints an error message and exits when this symbol is evaluated. In order to assist debugging, the `error` function is often used instead; it is identical to $\perp$ except that it allows the error message to be specified.

Haskell's strict typing requires an explicit transition between functions that are in the ErrorT monad and those that are not. This is achieved using the standard `lift` function, which can be seen as marking the point of no return when handling an event. After the lift, any failures will halt the system, so all checks for errors that might be caused by user-level code must come before it.

4.5.2. Fault Handling

The default response to any error is to set an error code and return. This is done by the function `handleException`, which is used to catch any errors that are not caught before reaching the main loop. It sets one user space register to the error code corresponding to the exception's ExceptionType field; also, if there is a capability associated with the exception, it stores its user-level address in a second register.

However, this is not appropriate for errors that are caused by memory accesses at user-level (see section 3.6.2 on page 30). Therefore, in the event handlers for memory access events, an additional call to `catchError` is made. This will pass any detected errors to `handleFault`. The latter function locates the appropriate fault handler endpoint, sends an IPC to it containing information about the fault, and suspends the thread that caused the fault.

4.6. Event Handling

The top-level kernel function for event handling is:

```
handleEvent :: ThreadPtr → Event →
        ErrorT Exception (State KernelState) (Maybe String, Bool)
```

This function performs all the necessary modifications to the kernel state to process an event, given the event and a pointer to the current thread. The result is a tuple consisting of a boolean value, and maybe a string; the boolean value indicates whether the model should continue processing events, and if a string is returned, it will be printed to the model's standard output.

Event handling functions for system calls all follow the same basic form, shown in the following Haskell-like pseudo-code.

```
handleEvent thread SomeEvent = do
```

To handle an event of type `SomeEvent`, which was generated by a `thread`, perform the following steps:

```
cap ← asUser thread $ getRegister AR0
param ← asUser thread $ getRegister AR1
```

Fetch the capability pointers and other parameters from the thread's registers. `asUser thread` executes a given function, in this case `getRegister`, in the context of a user-level thread.

```
(object, capdata) ←
    capLookup thread cap capAllowTheOperation (=ExpectedType)
```

Find the object being accessed. The last two arguments to `capLookup` are used to check permissions and object type, respectively.

```
when (...) $
    throwError $ Ex { ... }
```

Check for any additional error conditions that would not be noticed by `capLookup`. When an error has occurred, construct and throw an `Exception` that describes the error.

```
lift $ performTheOperation object capdata param
```

Call low-level kernel code to perform the requested operation. This is the point of no return; errors in `performTheOperation` will print an error message and halt the kernel.

```
return (Nothing, True)
```

The first value indicates that nothing should be printed as a result of this event's processing; an alternative would be `Just errorString`. The second value states that the model should continue running; it is only `False` when processing a `Halt` system call. Section 4.8 on page 42 explains the use of these values.

4.7. User-Level Simulator

The user-level simulator is used as a source of events for the kernel to process. The event list could simply be constructed by hand; however, I believe that the ability to run arbitrary complex programs whose behaviour depends on the actions of the kernel is a significant advantage over hard-coded event streams. This way, the event sequences can be less artificial, and testing is easier because incorrect responses from the kernel will cause the simulated program to misbehave.

4.7.1. User Context

The user-level simulator, like the kernel model, consists of a sequence of operations that each makes some modifications to a collection of state data and then returns a result. However, it processes a much smaller state data structure, which contains only the state of a simulated CPU. In the Haskell code, it is known as UserContext.

In the current version of the model, the user context consists of an instruction pointer, a program (in a format which is discussed in the next two sections), and a mapping from register names to integer values. The simulated CPU has 32 general purpose integer registers, 8 registers set aside for system call parameters, and a stack pointer.

4.7.2. User State Transition Functions

Programs for the simulator were originally a sequence of Haskell functions in the State UserContext monad, which could be arbitrarily complex. The only restrictions on them were that the state that each function could communicate to the next was that which could be contained in UserContext; that each function had to generate exactly one event; and that no input or output could be performed[1].

After developing some test programs in this form, it was observed that the functions in each user-level programs were quite similar. They tended to consist only of sequences of relatively simple operations similar to assembly language instructions; they were also quite tedious to write. On this basis I decided to try another approach, and replaced the state transition functions with an interpreter for a simple assembly-like language.

4.7.3. An Assembler-like Language

This is the system used in the current version of the model. It consists of a set of instructions resembling the user-level instruction set of a very simple RISC CPU. The entire instruction set is:

```
data Instruction =
        Arithmetic Register (Int → Int → Int) Register Register |
        ArithmeticI Register (Int → Int) Register |
        Compare Register (Int → Int → Bool) Register Register |
        CompareI Register (Int → Bool) Register |
        LoadImmediate Int Register |
        Load Int Register Register |
        Store Register Int Register |
        Push Register |
        Pop Register |
```

[1]Input and output in Haskell are only possible for functions executing in the IO monad.

```
            Move Register Register |
            Branch Int |
            BranchLinked Int Register |
            BranchIf Register Int |
            Syscall Int |
            DebugPrintf String [Register]
```

The most significant differences between this and a real CPU are the presence of an instruction that prints a formatted string, and the use of Haskell functions to specify arithmetic and comparison operations. In the following example, the function (+), and a partial application of it to a single integer (+3), are used to define add and immediate-add instructions respectively:

```
program = [
    LoadImmediate 1 R0,
    LoadImmediate 2 R1,
    Arithmetic R0 (+) R1 R2,
    ArithmeticI R2 (+3) R3,
    DebugPrintf ``This should print 6: %'' [R3],
    Syscall (fromEnum SysHalt)
]
```

Instructions are executed until either an instruction causes an event, or the thread's time-slice expires. Instructions that cause events are `Load`, `Store`, `Push`, `Pop`, `Syscall` and `DebugPrintf`. If the thread runs out of time, a `TimerInterrupt` event is generated.

Several programs have been written in this language to confirm that the kernel model is working correctly. However, it is not easy to develop complex programs this way, so the existing tests are all short and somewhat artificial. This is discussed further in section 5.1 on page 45.

4.8. The Main Loop

The main loop of the kernel model is shown below, as annotated Haskell code.

```
mainLoop :: State KernelState [Maybe String]
```

The loop is in the form of a state transition function, which returns a list of items which may contain text strings. These strings are generated when a `DebugPrint` event is processed; they are intended to be used to indicate the progress of the user-level program running in the model. At present, `DebugPrint` is the only way for the model to interact with the outside world, because no peripheral devices have been implemented yet.

```
mainLoop = do
        thread ← getCurThread
```

First, a pointer to the current thread's TCB is fetched from the kernel state.

```
        ev ← runThread thread
```

The user-level simulator then runs until an event occurs.

```
        Right (output, continue) ← runErrorT $
                handleEvent thread ev `catchError` handleException thread
```

The kernel's top level event handler function is evaluated inside the ErrorT monad, given the current thread and the event as arguments. The catchError function applied to handle any errors encountered by the event handler.

Note that runErrorT returns a value of type Either a b, where b is the type of a successful result and a is the type of an error value. In this case, all errors will be captured by handleException, so the result will always indicate success; it will therefore be accessible using the data constructor Right — this indicates the right-hand argument of the Either type, and also that the function has returned the "right" (i.e. correct) result.

The result itself consists of a tuple of a Boolean value, continue, and output, which is a Maybe String. continue is True if the model should proceed to process the next event, or False if it should halt.

```
        rest ← if continue then mainLoop else return []
```

If continue is True, mainLoop calls itself recursively to continue processing events. Otherwise, execution is complete, and the result list is terminated with the empty list []. Note that the use of return here is potentially confusing: it is being used to place the value [] into the State KernelState monad, so that it has the same type as the recursive call to mainLoop. This does *not* exit the present function.
rest

now contains a list of the outputs of all future kernel events. This list is potentially infinitely long, but since it is evaluated lazily, that is not a problem.

```
        return $ output:rest
```

The final result is a list consisting of the output for the current event, followed by the outputs of the rest of the events.

The loop function is called by main, the top-level function of any compiled Haskell program. In this case, the main function creates an initial KernelState structure, uses it to evaluate mainLoop in the State KernelState monad, and prints any strings returned to standard output. Note that because evaluation of Haskell programs is

lazy, the model does not actually process each event until the main loop attempts to print the string it may generate.

4.9. Summary

A realistic low-level simulation of a microkernel and its user-level clients has been constructed. It functions by:

- modelling the state as an array of physical data frames and using indices into that array as physical pointers;
- simulating user-level processes to generate a series of events; and
- processing the events and performing appropriate modifications to the system's state.

Chapter 5.

Evaluation

5.1. Usability and Completeness

It is clearly important to know whether the new kernel interface is useful for development of efficient and secure systems, and whether the model of the interface is complete enough to allow this. The best way to determine this is to actually develop such a system, and run it inside the model.

Several short programs have been written in the assembly-like language described in section 4.7.3. These programs check that specific parts of the kernel interface are functioning correctly. However, they were mostly intended to be debugging aids. Each program is short, focuses on testing only one or two elements of the kernel interface, and often uses the interface in ways that are unlikely for a more complete system.

Due to time constraints, combined with the difficulty of writing non-trivial programs in the present low-level user level simulator, no such extensive tests have been written yet. To properly evaluate the usability of the kernel interface, it will be necessary to produce a new user level simulator that is capable of supporting development of more complete applications. Possible approaches to this are described in section 6.1.2.

5.2. Rapid Prototyping

Writing an executable specification in a high-level language allows rapid prototyping of new kernel features, without tedious and difficult low-level programming. Features can be implemented one at a time — even those which would be essential in a stand-alone kernel, such as memory management.

At each stage of the model's development, it was possible to write and run simple user-level test programs, despite being too incomplete to function as a stand-alone kernel. For example, virtual memory was implemented after most of the core features of the kernel; up until that point, the tests were running at user level without ever accessing data memory. Similarly, IPC was implemented before there was any

user-level address space at all; at that time the user level programs were using opaque CAPABILITY objects rather than pointers. Such incremental development would not have been possible with a prototype running as a stand-alone kernel; using an executable model instead allowed specific areas of the kernel design to be designed, implemented and tested one by one.

I recently attempted to replace the *Map* operation with one that more closely resembles L4's, transferring capabilities during an IPC operation. However, I discovered during the attempt that there are several possible error cases to be considered when transferring the message[1]. In situations such as this, the ability to quickly implement and test subsets of the kernel's functionality as they are specified is very helpful; it uncovers implementation issues that would not be so obvious when writing a specification in English.

5.3. A Specification Document

A reference manual for the kernel's behaviour could be based on an annotated version of the Haskell source code itself. An example of the contents of such a specification is shown in appendix A, which was generated directly from the source code of the kernel model. Note that in this instance, the document is concerned as much with documenting the Haskell implementation as the technical details of the kernel interface; a kernel reference manual would most likely contain more discussion of the semantics of the interface. Also, time constraints have limited the extent to which the code was annotated; the most complete documentation can be found in the CSPACE module, in section A.4 on page 63.

Since the source code used to generate the reference manual is also an executable model of the kernel, the manual provides an unambiguous description of how other implementations should behave. Any apparent ambiguity in the prose can be resolved by either reading the embedded Haskell code, or executing the specification and observing its behaviour.

5.4. Formal Verification

Some preliminary work to formalise the new kernel specification, starting with the IPC path, has been done by Harvey Tuch. He has reported that a formal specification of the IPC path was developed with significantly less effort than would be required to formalise a specification in an imperative language, and estimates it to be at least an order of magnitude faster than doing the same for the C++ L4Ka::Pistachio implementation of IPC.

[1]Due to time constraints, solving this problem has been deferred until after the completion of this thesis.

It should be noted that the L4Ka::Pistachio IPC path is inherently more complex than that of the present version of seL4; the former includes transfers of virtual memory mapping and string objects, while the latter transfers only untyped data. However, much of the difference is due to the relative simplicity of the executable specification compared to a full implementation, and the similarity of pure functional languages like Haskell to formal specification languages such as HOL.

Further experience is required before we can properly evaluate the relative advantages and disadvantages of formalising an executable specification in Haskell, compared to the combination of English language manual and C++ and assembler source code of L4Ka::Pistachio. The positive early experiences are promising.

Chapter 6.

Conclusion

The goal of this thesis was to build an executable specification for a new L4-like microkernel. Though many aspects of the kernel design have not yet been defined, an executable model of a partial specification has been successfully constructed. This model will continue to be developed in the future as the kernel design process continues. I believe that it will be useful both as a tool for rapidly prototyping new kernel features, and as an executable specification that can easily be formalised.

6.1. Future Work

There are two important areas of future work on this project. First, there are many issues with the kernel design that have yet to be solved; second, some modifications to the model are necessary to allow effective evaluation of the kernel interface.

6.1.1. Further Development of the Kernel

Many aspects of the kernel design need a significant amount of further development. These include (but are not limited to):

Page table structures: The seL4 API is claimed to be independent of page table structure, but has only been tested with a two-level page table. The interface should be tested with other structures.

***Map* semantics:** It may be desirable to make the *Map* operation resemble its counterpart in L4 more closely. In particular, it should be part of the IPC operation, and should allow the receiving thread to have more control over the destination of the transferred capability.

Multiple object sizes: The memory efficiency of the present design is poor, because IPC endpoints and thread control blocks are required to occupy an entire page.

Increasing access rights: There is no way to efficiently or transparently grant a client additional access rights to an object for which some rights have already been granted, when those rights may have been passed on to other processes. The present implementation requires revocation of the original rights before granting the increased set, which will also revoke any rights that have been passed on.

Interrupt handling: There is not yet any protocol for dispatching interrupts to user level handlers, nor for acknowledging interrupts from user level.

6.1.2. A New User Level Simulator

As discussed in section 5.1, a conclusive evaluation of the usefulness of the kernel interface will require a new user level simulator capable of supporting the construction of more complex test programs.

There are two potential approaches to this. One is an improvement upon the initial user-level simulator that was based on functions in the STATE monad; it involves building a new Haskell monad to allow user-level programs to be written in a manner similar to the kernel model itself. The other approach improves upon the current user-level assembly interpreter, by replacing it with an emulator that can execute binary code compiled for an existing architecture.

It should be noted that these two approaches are not mutually exclusive. It would be possible, and perhaps a good idea, to encapsulate a binary code emulator inside a user-level state monad.

A User-Level State Monad

The monadic programming technique used in implementing the kernel is very flexible. In particular, a sequence of operations inside a monad may be bound together using an arbitrary function — the *bind* operator, >>= — which is defined by the specific monad. By changing the definition of the bind operator, monads can extend the language in a wide variety of ways.

It should be possible to replace the user-level assembly emulator with a new monad, similar to the existing STATE monad. Rather than simply building a state transition function from a sequence of operations, this monad's bind operator could call the kernel model to handle events. Threads could be modelled by saving the bind operator's right hand argument in the current TCB before calling the kernel, and restoring the value from the (potentially different) current TCB afterwards. This might be achievable with a combination of the existing state and continuation monads (STATE and CONT, or their transformer versions STATET and CONTT), but most likely would require development of a new monad.

This solution would allow relatively complex user-level programs to be developed, using syntax quite similar to that used in the kernel itself. However, it would still require test programs to be developed specifically for the kernel model.

Emulating a Realistic CPU

Rather than developing an entirely new user-level system for testing the kernel, it may be desirable to port an existing system to the new kernel interface. This would require that the model be capable of executing compiled binary code.

The present interpreter for the assembler-like user-level language could be replaced by an emulator for binary executables. Some other minor modifications to the model would be necessary before this could be achieved. In particular, there are many references to specific user-level registers scattered through the kernel. These should be changed to reflect their meaning rather than their implementation; for example, by replacing `setRegister` $\mathtt{AR}_0$ with `setErrorCode`.

The emulator itself might either be a new one written in Haskell, or an existing emulator interfaced with the model through Haskell's foreign function interface.

6.1.3. Running on Real Hardware

The hOp project has demonstrated [1] that it is possible for Haskell code to run independently of any external operating system. It might be possible to adapt the executable specification to do the same; this would provide a reference implementation, running as a stand-alone kernel on real hardware. However, Haskell is not a particularly light-weight or fast language, so this would be of very limited practical use.

Bibliography

[1] Sébastien Carlier and Jérémy Bobbio. hOp. http://www.macs.hw.ac.uk/~sebc/hOp/, 2004.

[2] Kevin Elphinstone. Future directions in the evolution of the L4 microkernel. In Gerwin Klein, editor, *Proceedings of the NICTA workshop on OS verification 2004, Technical Report 0401005T-1*, Sydney, Australia, October 2004. National ICT Australia.

[3] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5):295–357, July 2002.

[4] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, September 2003.

[5] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 191. IEEE Computer Society, 1999.

[6] Rafal Kolanski. A formal model of the L4 μ-kernel API using the B method. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2004.

[7] L4Ka Team. L4Ka::Pistachio kernel. http://l4ka.org/projects/pistachio/, 2003.

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

[9] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, 1994.

[10] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 73–79, Cape Cod, MA, USA, May 1997. IEEE.

[11] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[12] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[13] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.

[14] Jonathan S. Shapiro. Vulnerabilities in synchronous IPC designs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 251–262. IEEE, May 2003.

[15] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[16] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, USA, June 2005. To appear.

[17] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM Press, 1992.

Appendix A.

Annotated Haskell Code

This appendix contains the annotated Haskell source code for the kernel model. The entire code is presented, except for some minor implementation details such as compiler pragmas and lists of imported modules.

A.1. Kernel API

```
module KernelAPI where
```

This module contains the data types and constants that define the interface between the kernel model and the user-level code running on the CPU simulator.

A.1.1. Pointer Types

```
newtype CapPtr = CapPtr Word32
    deriving(Show,Eq,Ord,Num,Enum,Real,Integral,Bits)
```

This is the definition of a new type, based on a 32-bit unsigned integer, which is used to represent addresses in a user level thread's capability space. It is declared with `newtype` so that casts between it and other integer types must be explicit.

```
newtype Pointer = Pointer Word32
    deriving(Show,Eq,Ord,Num,Enum,Real,Integral,Bits)
```

Another new type, this time for physical pointers used by the kernel.

```
nullPointer :: Pointer
nullPointer = 0
```

The value of an invalid physical pointer.

```
type ThreadPtr = Pointer
type EndpointPtr = Pointer
```

```
type CTEPtr = Pointer
type CapTablePtr = Pointer
type DPagePtr = Pointer
```

Aliases for the Pointer type, used for specific types of object. Note that the compiler does not distinguish between these types; they exist only to make type signatures for kernel functions a little clearer.

A.1.2. Object Types

```
data ObjectType = Untyped |
                    IntDataObject |
                    TCBObject |
                    EndpointObject |
                    CapTableObject
        deriving (Enum, Bounded, Eq, Show)
```

This enumeration is used to specify one of the types of object that can be stored in a physical page.

A.1.3. Events

```
data Event = Load CapPtr Register |
               Store Register CapPtr |
               SyscallEvent Syscall |
               UnknownSyscall Int |
               DebugPrint String |
               TimerInterrupt
```

These are the events that can appear in the stream of events processed by the kernel model. See section 4.2 on page 33.

Note that, since the kernel model has no TLB, *every* load or store access to virtual memory generates a kernel event, and the kernel must perform the load or store operation itself. When running on real hardware, the Load and Store events would be replaced by TLB or page table faults.

```
data Syscall = SysSendIPC |
                 SysReceiveIPC |
                 SysThreadControl |
                 SysExchangeRegisters |
                 SysCapCopy |
                 SysCapRevoke |
                 SysCapRetype |
                 SysCapSetData |
```

```
                SysYield |
                SysHalt
        deriving (Ord, Enum, Bounded, Eq)
```

This is the set of all valid system calls that can appear after the EVENT type's SYSCALLEVENT constructor.

A.1.4. Exceptions

```
data ExceptionType = PermissionFault |
                     CapFault |
                     InvalidArgument
        deriving (Enum, Show, Eq)
```

This is an enumeration of every type of fault that the kernel can report to a user-level fault handler.

A.1.5. Miscellaneous Constants

```
timeSlice :: Int
timeSlice = 1000
```

The length of a scheduler timeslice, in CPU cycles. This is quite short, but it should be noted that with no cache, no TLB, and a separate register set for each thread, the overhead of a thread switch in this model is effectively zero. On a real system the timeslice would be much longer.

```
pageBits :: Int
pageBits = 12
```

The number of address bits in a page offset.

```
mask :: Int → Pointer
mask bits = (1 `shiftL` bits) − 1
```

A trivial function that finds the mask for a given number of bits in an address.

```
memSize :: Word
memSize = 1024
```

The size of physical memory in pages.

```
kernelTop :: Word
kernelTop = 1
```

The number of pages occupied by the kernel's code and static data. This must be at least 1 to ensure that accesses to null pointers always fail. In a real kernel implementation it would be significantly larger.

A.2. Kernel State

```
module KernelStateData where
```

This module defines the kernel's global data, and also the data structure used by the Haskell model to store the state of the system.

A.2.1. Data Types

```
type ReadyQueue = [ThreadPtr] — single priority round robin
```

The ready queue for each priority is represented by a linked list of thread pointers. In a complete implementation these would be embedded in the thread control blocks.

```
data KernelState = KState { ksPSpace :: PSpace,
                            ksReadyQueues :: Array Priority ReadyQueue,
                            ksCurThread :: ThreadPtr }
```

This structure is used by the Haskell model to represent the state of the entire system. It contains two items of global kernel data — the current thread pointer and the ready queues — and the state of the modelled physical address space.

A.2.2. Public Functions

The functions in this module are simple accessors for the ready queue and current thread.

```
getQueue :: Priority → State KernelState ReadyQueue
getQueue prio = gets $ λks → ksReadyQueues ks ! prio


setQueue :: Priority → ReadyQueue → State KernelState ()
setQueue prio q = modify $ λks →
        ks { ksReadyQueues = (ksReadyQueues ks)//[(prio,q)] }
```

Given a priority, these two functions get and set the contents of the ready queue for that priority level.

```
getCurThread :: State KernelState ThreadPtr
getCurThread = gets ksCurThread
```

```
setCurThread :: ThreadPtr → State KernelState ()
setCurThread tptr = modify $ λks → ks { ksCurThread = tptr }
```

These functions get and set the current thread pointer.

A.3. System Calls

```
module Syscall where
```

This module contains the `handleEvent` function, which processes an event that occurred while the system was running at user level. It also contains a helper function that implements the *Map* operation.

A.3.1. Public Functions

```
handleEvent :: ThreadPtr → Event →
                    ErrorT Exception (State KernelState) (Maybe String, Bool)
```

This function handles an event that has caused kernel entry. There is a separate definition for each type of event.

Note that most definitions of this function are of a similar form, which is documented (in Haskell-like psuedocode) in section 4.6 on page 39. Descriptions of the definitions will be brief except where they diverge from this pattern.

System Call Handlers

The interface for the system calls is described in chapter 3 on page 20. Please refer to that chapter for an overview of the purpose and semantics of the calls.

The following definition is for the *Receive IPC* system call. It calls `receiveIPC`, which is defined in section A.6.3 on page 85.

```
handleEvent tp (SyscallEvent SysReceiveIPC) = do
        fromcap ← lift $ asUser tp $ getRegister AR0
        (fromptr, _) ←
            capLookup tp fromcap capAllowReceive (=EndpointObject)
        lift $ receiveIPC tp fromptr
        return (Nothing, True)
```

The definition for the *Send IPC* system call is slightly different, because it must determine whether the object being invoked is an endpoint or a thread. In the latter case, it calls `handleMapIPC`, defined later in this module. Note that the *Map IPC* operation is likely to change significantly in future revisions.

```
handleEvent tp (SyscallEvent SysSendIPC) = do
        tocap ← lift $ asUser tp $ getRegister AR0
        (toptr, badge) ←
            capLookup tp tocap capAllowSend
                (λt → t=EndpointObject ∨ t=TCBObject)
        ptype ← lift $ getPageType toptr
        case ptype of
            EndpointObject → lift $ sendIPC tp toptr badge
            TCBObject → handleMapIPC tp toptr
        return (Nothing, True)
```

The definition for *ThreadControl* must inspect a bitmask to determine which of the thread's parameters, given in the other arguments, should be set.

Note that the `textid` argument, and the error that results from giving an invalid value for it, are both specific to the Haskell model. They exist to select a program from the list `userTaskText`, which is defined in the User module.

```
handleEvent tp (SyscallEvent SysThreadControl) = do
        targetcap ← lift $ asUser tp $ getRegister AR0
        flags ← lift $ asUser tp $ getRegister AR1
        root ← lift $ asUser tp $ getRegister AR2
        handler ← lift $ asUser tp $ getRegister AR3
        prio ← lift $ asUser tp $ getRegister AR4
        textid ← lift $ asUser tp $ getRegister AR5
        handlerTable ← lift $ asUser tp $ getRegister AR6

        when (textid < 0 ∨ textid ≥ length userTaskText) $
            throwError $ Ex {exType = InvalidArgument,
                             exAddress = ⊥,
                             exCTLevel = ⊥,
                             exDesc = "Invalid index for user text"}

        root' ← if (flags::Int).&.1≠0
            then liftM Just $ findCTE tp root else return Nothing
        handler' ← if flags.&.2≠0
            then liftM Just $ findCTE tp handler else return Nothing
        let prio' = if flags.&.4≠0
                        then Just prio else Nothing
        let text = if flags.&.8≠0
                        then Just $ userTaskText!!textid else Nothing
        handlerTable' ← if flags.&.16≠0
            then liftM Just $ findCTE tp handlerTable else return Nothing
```

```
(target, _) ←
    capLookup tp targetcap capAllowWrite (=TCBObject)

oldPrio ← lift $
    threadControl tp target text root′ handler′ prio′ handlerTable′
lift $ asUser tp $ setRegister AR1 oldPrio
return (Nothing, True)
```

ExchangeRegisters uses a pointer to a virtual memory page. However, it does so only under a certain set of conditions: when the flag to copy the argument and execution state registers is set, and either the source or destination thread is equal to the current thread.

```
handleEvent tp (SyscallEvent SysExchangeRegisters) = do
        srccap ← lift $ asUser tp $ getRegister AR0
        destcap ← lift $ asUser tp $ getRegister AR1
        flags ← lift $ asUser tp $ getRegister AR2
        saveptr ← lift $ asUser tp $ getRegister AR3

        (src, _) ← capLookup tp srccap capAllowRead (=TCBObject)
        (dest, _) ← capLookup tp destcap capAllowWrite (=TCBObject)
        savepptr ← if (src = tp ∨ dest = tp) ∧ (flags .&. 4 ≠ 0)
            then do
```

If this test succeeds, the `exchangeRegisters` function will be using the save area pointer. Therefore, it must be converted to a physical pointer, and checked to make sure it will not cross the boundary between two pages.

```
let pageSize = 1 `shiftL` pageBits
let intSize = 1 `shiftL` (objBits (⊥ :: Int))
let frameSize = fromIntegral $ intSize ∗ (length [SP .. AR7])
when (saveptr `mod` pageSize + frameSize ≥ pageSize) $ do
    throwError
        Ex {exType = InvalidArgument,
            exAddress = saveptr,
            exCTLevel = ⊥,
            exDesc = ''Cannot save state across page boundary''}
```

Calculate the space required to save the thread state. If writing that amount of data starting at the save area pointer will cross a page boundary, fail with an error code.

```
(pptr,_) ← capLookup tp saveptr
    (λp →
        (src ≠ tp ∨ capAllowRead p) ∧
        (dest ≠ tp ∨ capAllowWrite p))
    (=IntDataObject)
return (pptr + fromIntegral (saveptr `mod` pageSize))
```

Finally, calculate the physical pointer for the save address.

```
        else return nullPointer
```

If the save pointer won't be used, this will just set it to 0.

```
    lift $ exchangeRegisters tp src dest flags savepptr
    lift $ asUser tp $ setRegister AR0 0
    return (Nothing, True)
```

The *CapCopy* event is mostly handled in capCopy, defined in the CSpace module. See section A.4.3 on page 66.

```
handleEvent tp (SyscallEvent SysCapCopy) = do
    srcCap ← lift $ asUser tp $ getRegister AR0
    destCap ← lift $ asUser tp $ getRegister AR1
    permMask ← lift $ asUser tp $ getRegister AR2
    capCopy tp srcCap destCap (permsFromWord permMask)
    return (Nothing, True)
```

The *CapRevoke* event must use a different lookup function — findCTE — because it operates on the CTE itself rather than on the object it points to. Refer to section A.4.3 on page 69.

```
handleEvent tp (SyscallEvent SysCapRevoke) = do
    cap ← lift $ asUser tp $ getRegister AR0
    self ← lift $ asUser tp $ getRegister AR1
    (cteptr, cteperms) ← findCTE tp cap
    unless (capAllowModify cteperms) $
        throwError $ Ex {exType = PermissionFault,
                         exAddress = cap,
                         exCTLevel = 2,
                         exDesc = ``Cannot revoke cap''}
    lift $ cteRevoke (self≠0) cteptr
    return (Nothing, True)
```

The *CapRetype* implementation is straightforward.

```
handleEvent tp (SyscallEvent SysCapRetype) = do
    cap ← lift $ asUser tp $ getRegister AR0
    newType ← lift $ asUser tp $ getRegister AR1
    (page, _) ← capLookup tp cap capAllowModify (λ_ → True)
    when (newType < 0 ∨ newType > fromEnum (maxBound :: ObjectType)) $
        throwError $ Ex {exType = InvalidArgument,
                         exAddress = ⊥,
                         exCTLevel = ⊥,
                         exDesc = ``Invalid new type for retype''}
    lift $ retypePage (toEnum newType) page
    return (Nothing, True)
```

Like *CapRevoke*, *CapSetData* operates on a CTE rather than the object it maps, so it uses `findCTE`.

```
handleEvent tp (SyscallEvent SysCapSetData) = do
        cap ← lift $ asUser tp $ getRegister AR0
        newData ← lift $ asUser tp $ getRegister AR1
        capLookup tp cap capAllowModify (λ_ → True)
        (cteptr , _) ← findCTE tp cap
        lift $ cteSetData cteptr newData
        return (Nothing , True)
```

The *Yield* system call has no parameters, and cannot fail — hence the call to `lift` that wraps the entire function definition. It simply resets the current thread's time allocation, and then calls the scheduler to switch to the next thread in the queue.

```
handleEvent tp (SyscallEvent SysYield) = lift $ do
        threadSet (λtcb → tcb {tcbTimeSlice = timeSlice}) tp
        schedule
        return (Nothing , True)
```

Halt is a system call added for the purposes of testing the Haskell model; it simply stops the simulation. This call would not be necessary, or present, in a complete specification.

```
handleEvent _ (SyscallEvent SysHalt) = return (Nothing , False)
```

Memory Accesses

The following two definitions handle the events generated by a memory access at user level. They call functions in the DSpace model to read or write integer values in physical memory, and also functions in the Thread and UserContext modules to access the appropriate user-level register.

These functions would be defined significantly differently in a complete implementation; they would modify the hardware's virtual memory translation cache, and not modify the user level register set.

```
handleEvent tp (KernelAPI.Load ptr reg) = do
        value ← doLoad tp ptr `catchError` (λex → handleFault tp ex >> return 0)
        lift $ asUser tp $ setRegister reg value
        return (Nothing , True)


handleEvent tp (KernelAPI.Store reg ptr) = do
        value ← lift $ asUser tp $ getRegister reg
        doStore tp ptr value `catchError` handleFault tp
```

```
        return (Nothing, True)
```

Miscellaneous Events

The DebugPrint event is used to produce some output from the executable model. It would not be present in a standalone implementations, except as part of a kernel debugger.

```
handleEvent _ (DebugPrint str) = return (Just str, True)
```

The TimerInterrupt event is generated when a thread's time allocation expires. It is handled identically to the *Yield* system call.

```
handleEvent tp TimerInterrupt = handleEvent tp (SyscallEvent SysYield)
```

An attempt to invoke an unknown system call causes an error.

```
handleEvent tp (UnknownSyscall n) =
        throwError $ Ex {exType = InvalidArgument,
                         exAddress = ⊥,
                         exCTLevel = ⊥,
                         exDesc = "Unknown Syscall " ++ show n}
```

Map IPC

The following function is called when performing a *Map* operation; it is of similar form to the system call handlers. It locates CTEs for the source and destination, checks that the sender has permission to copy the source, and calls cteCopy.

Note that this implementation has an additional argument, representing the level of the capability table that contains the destination CTE. This allows the *Map* operation to resolve faults at any level of the table; however, it also potentially exposes kernel implementation details to the client application. As discussed in section 3.5.5, this is a potential problem, and as a result the semantics of the *Map* operation may change significantly in the near future.

```
handleMapIPC :: ThreadPtr → ThreadPtr →
                ErrorT Exception (State KernelState) ()
handleMapIPC sender dest = do
        srcCap ← lift $ asUser sender $ getRegister AR1
        destCap ← lift $ asUser sender $ getRegister AR2
        destLevel ← lift $ asUser sender $ getRegister AR3
        permMask ← lift $ asUser sender $ getRegister AR4

        when (destLevel < 0 ∧ destLevel > 2) $
```

```
    throwError $ Ex {exType = InvalidArgument,
                     exAddress = ⊥,
                     exCTLevel = ⊥,
                     exDesc = ``Map␣level␣is␣out␣of␣range''}

destCTE ← findCTEForMap dest destCap destLevel

oldCTE ← lift $ getObject destCTE
unless (ctePointer oldCTE = 0) $
    lift $ cteRevoke True destCTE

(srcCTE, srcPerms) ← findCTE sender srcCap
unless (capAllowCopy srcPerms) $
    throwError $ Ex {exType = PermissionFault,
                     exAddress = srcCap,
                     exCTLevel = 2,
                     exDesc = ``Cannot␣copy␣capability''}

lift $ cteCopy (srcPerms `maskCapPerms` permsFromWord permMask)
    srcCTE destCTE
lift $ haltThread dest False
```

A.4. Capability Space

```
module CSpace (CTE, CapPerms(..),
    allPerms, noPerms, maskCapPerms, permsFromWord, capCopy,
    cteCopy, cteSetData, cteRevoke, findCTE, findCTEForMap,
    capLookup, ctePointer, ctePerms, cteCapData, findFaultHandler,
    createInitCap) where
```

The CSpace module defines the types and functions related to the capability space structure. The latter include the system calls used for management of capabilities, and several functions used within the kernel to look up entries in capability tables and to create the initial thread's capability table.

A.4.1. Types

Capability Table Entry

```
data CTE = CTE {
    ctePointer :: Pointer,
    ctePerms :: CapPerms,
    cteCapData :: Int,
```

```
        cteMDBNode :: MDBNode }
```

Entries in the capability table each contain:

- a physical pointer to the kernel object referenced by the capability;
- a set of permissions which determines which system calls can be performed using the capability; and
- a mapping database node.

Permissions

```
data CapPerms = CapPerms {
        capAllowRead, capAllowWrite, capAllowCopy,
        capAllowModify, capAllowSend, capAllowReceive :: Bool }
```

XXX: these may not be sufficient

Mapping Database Node

```
data MDBNode = MDB {
        mdbNext, mdbPrev :: CTEPtr,
        mdbDepth :: Int }
```

The mapping database consists of a tree structure for each physical page that can be mapped at user level. It is used to keep track of all CTEs pointing to each kernel object, so capabilities can be recursively revoked. When the contents of a CTE are copied to another, the new CTE becomes a child of the original in the mapping tree.

The structure is similar to that used in L4Ka::Pistachio [7]. It consists of a doubly linked list that is equivalent to a prefix traversal of the mapping tree; each node records its depth in the mapping tree, so the tree can be reconstructed from the list.

A.4.2. Physical Storage

This is the instance of the Object class for CTE. It is used by PSpace to determine how to store CTEs in the simulated physical address space. It is mostly similar to that for Int, except that there are two special cap table entries — accessible only to the kernel, or indirectly by the user via SysThreadControl — stored inside each thread control block.

```
instance Object CTE where
        objBits _ = 4
        makeObject = CTE 0 noPerms 0 (MDB 0 0 0)
```

```
makeObjectPage obj = CapTablePage $
        listArray (0, (1 `shiftL` (pageBits − objBits obj)) − 1) $
            repeat obj

getFromPage offset (CapTablePage array) =
        if align = 0
            then val
            else alignError (objBits val)
        where
            val = array!index
            index = offset `shiftR` (objBits val)
            align = offset .&. (fromIntegral $ mask (objBits val))
getFromPage offset (ThreadPage tcb)
        | offset = tcbFaultHandlerOffset = tcbFaultHandler tcb
        | offset = tcbCTableOffset = tcbCTable tcb
        | offset = tcbFaultHandlerTableOffset = tcbFaultHandlerTable tcb
        | otherwise = typeError "cap table or tcb page"
getFromPage _ _ = typeError "cap table or tcb page"

setInPage val offset (CapTablePage array) =
        if align = 0
            then CapTablePage $ array//[(index, val)]
            else alignError (objBits val)
        where
            index = offset `shiftR` (objBits val)
            align = offset .&. (fromIntegral $ mask (objBits val))
setInPage val offset (ThreadPage tcb)
        | offset = tcbFaultHandlerOffset =
                ThreadPage $ tcb {tcbFaultHandler = val}
        | offset = tcbCTableOffset =
                ThreadPage $ tcb {tcbCTable = val}
        | offset = tcbFaultHandlerTableOffset =
                ThreadPage $ tcb {tcbFaultHandlerTable = val}
        | otherwise = typeError "cap table or tcb page"
setInPage _ _ _ = typeError "cap table or tcb page"
```

A.4.3. Public Functions

Permissions

```
allPerms :: CapPerms
allPerms = CapPerms True True True True True True
```

```
noPerms :: CapPerms
noPerms = CapPerms False False False False False False
```

These are the default values for permissions and permission masks, with all or none of the bits set.

```
maskCapPerms :: CapPerms → CapPerms → CapPerms
maskCapPerms (CapPerms a1 a2 a3 a4 a5 a6) (CapPerms b1 b2 b3 b4 b5 b6) =
        CapPerms (a1∧b1) (a2∧b2) (a3∧b3) (a4∧b4) (a5∧b5) (a6∧b6)
```

Finds the intersection of two sets of capability permissions.

```
permsFromWord :: Word → CapPerms
permsFromWord p =
        CapPerms (p.&.1≠0) (p.&.2≠0) (p.&.4≠0)
            (p.&.8≠0) (p.&.16≠0) (p.&.32≠0)
```

Converts a word to a set of capability permissions.

Copying Capabilities

```
capCopy :: ThreadPtr → CapPtr → CapPtr → CapPerms →
            ErrorT Exception (State KernelState) ()
cteCopy :: CapPerms → CTEPtr → CTEPtr →
            State KernelState ()
```

These two functions are used to copy a capability. The first, capCopy, is a system call implementation; its arguments include the calling thread and the source and destination capability pointers. cteCopy is intended for use by kernel code; it requires physical pointers to the source and destination CTEs, and does not check for sufficient permissions and valid pointers.

```
capCopy thread src dest permMask = do
        destCTEPtr ← capLookupForCopy thread dest
        oldCTE ← lift $ getObject destCTEPtr
        unless (ctePointer oldCTE = 0) $
            lift $ cteRevoke True destCTEPtr
```

Find the destination CTE, and revoke the capability in it if there is one. Note that this may cause the source capability to be either revoked or otherwise removed from the address space (eg by revoking part of the page table), though it is unlikely for that situation to ever occur. If it does, the source lookup will fault. For this reason, the destination lookup and revocation *must* happen before the source lookup.

```
        capLookup thread src capAllowCopy (λ_→True)
        (srcCTEPtr, srcPermMask) ← findCTE thread src
        let newPermMask = srcPermMask `maskCapPerms` permMask
```

Find the source CTE. Note that the source capability pointer is expected to be the capability being copied, while the destination is a capability to a CTE. The actual location of the new capability will *not* be equal to dest; it will be the address mapped by the destination CTE, which may or may not be in the caller's address space.

```
        lift $ cteCopy newPermMask srcCTEPtr destCTEPtr
```

Finally, call cteCopy to perform the copy operation.

```
cteCopy newPermMask srcCTEPtr destCTEPtr = do
        srcCTE ← getObject srcCTEPtr
        let newPerms = ctePerms srcCTE `maskCapPerms` newPermMask
```

Load the source capability table entry from physical memory, and calculate the set of permissions for the new capability. The new permissions are the intersection of the given permissions mask and the permissions on the old capability.

```
        let parentMDB = cteMDBNode srcCTE
        let newMDB = MDB { mdbNext = mdbNext parentMDB,
                           mdbPrev = srcCTEPtr,
                           mdbDepth = 1 + mdbDepth parentMDB }
        let newCTE = srcCTE { cteMDBNode = newMDB,
                              ctePerms = newPerms }
        setObject destCTEPtr newCTE
```

Create the new capability table entry and store it in physical memory. The new entry has the same pointer and additional data as the source entry. Its MDBNode is inserted in the mapping database as a child of the source entry's MDBNode.

```
        let parentMDB' = parentMDB { mdbNext = destCTEPtr }
        let srcCTE' = srcCTE { cteMDBNode = parentMDB' }
        setObject srcCTEPtr srcCTE'
```

Update the forward link in the source entry's MDBNode, which should now point to the new entry.

```
        nextCTE ← getObject $ mdbNext newMDB
        let nextMDB' = (cteMDBNode srcCTE) { mdbPrev = destCTEPtr }
        setObject (mdbNext newMDB) $ nextCTE { cteMDBNode = nextMDB' }
```

Finally, update the next MDBNode's reverse link to point to the new entry.

Additional Capability Data

```
cteSetData :: CTEPtr → Int → State KernelState ()
cteSetData ctePtr val = do
        cte ← getObject ctePtr
        setObject ctePtr $ cte {cteCapData = val}
```

Capability Revocation

```
cteRevoke :: Bool → CTEPtr → State KernelState ()
```

This function revokes the capability stored in a given CTE. If the boolean parameter is false, only capabilities that are copies of the one specified will be revoked; if it is true, the specified capability will also be revoked.

```
cteRevoke revokeSelf ctePtr = do
        cte ← getObject ctePtr
        let mdb = cteMDBNode cte
```

Load the CTE and extract its mapping database node.

```
        when (revokeSelf ∧ mdbDepth mdb = 0) $
            error ``Tried␣to␣revoke␣initial␣cap''
```

If an attempt is being made to revoke the top level cap (that is, the original capability given by the kernel to the initial user level thread), then fail.

```
        let prevPtr = if revokeSelf then mdbPrev mdb else ctePtr
```

Find the MDB node immediately to the left of the revoked nodes in the MDB.

```
        let nextPtr = mdbNext mdb
        nextPtr′ ← revokeWithMinDepth (1 + mdbDepth mdb) prevPtr nextPtr
```

Call a helper function, defined below, to revoke all copies of the given capability, which are immediately to the right of it in the MDB. This function will also find the MDB node to the right of those being removed from the MDB, and will set that node's leftwards pointer appropriately.

```
        prevCTE ← getObject prevPtr
        let prevMDB′ = (cteMDBNode prevCTE) { mdbNext = nextPtr′ }
        setObject prevPtr $ prevCTE { cteMDBNode = prevMDB′ }
```

Update the rightwards pointer of the node to the left of those deleted.

```
        when revokeSelf $ setObject ctePtr (makeObject :: CTE)
```

Finally, if the given capability itself is being revoked, invalidate its CTE.

Capability Space Lookups

```
capLookup :: ThreadPtr → CapPtr → (CapPerms → Bool) →
        (ObjectType → Bool) →
        ErrorT Exception (State KernelState) (Pointer, Int)
```

This function is used by the system call implementations to find the object pointed to by a given capability. Its parameters are the current thread, a capability, and two functions that return True if the permissions and object type, respectively, have the required values.

```
capLookup tp cap checkPerms checkType = do
        ctePtr ← findCTEForCall tp cap checkPerms
        cte ← lift $ getObject ctePtr
        checkCap cap (ctePointer cte) 2 (ctePerms cte) checkPerms checkType
        return (ctePointer cte, cteCapData cte)
```

The implementation simply locates the capability's CTE, reads its contents, calls a helper function to check that the permissions are sufficient and the object type is correct, and then returns a pointer to the object and the value of the capability-specific data.

```
findCTE :: ThreadPtr → CapPtr →
        ErrorT Exception (State KernelState) (CTEPtr, CapPerms)
findCTE thread addr = findCTEAtLevel thread addr 2 capAllowRead
```

This trivial function is used by system calls that need to locate the CTE for a given capability.

```
findCTEForMap t a l = liftM fst $ findCTEAtLevel t a l capAllowWrite
findCTEForMap :: ThreadPtr → CapPtr → Int →
        ErrorT Exception (State KernelState) CTEPtr
```

This function is similar to `findCTE`, except that it is used specifically to find the destination CTE for a map IPC operation. It therefore checks that the CTE is writable, rather than readable.

Fault Handler Lookups

```
findFaultHandler :: ThreadPtr → CapPtr → Int → State KernelState CTE
```

This function is used by the fault handling code to determine whether there is a region-specific fault handler for the region containing the given capability pointer. It returns the contents of the CTE for the appropriate fault handler.

```
findFaultHandler tptr addr level = do
        defaultHandler ←
            getObject (tptr + fromIntegral tcbFaultHandlerOffset)
```

First, locate the default handler, which is used whenever a valid per-region handler cannot be found.

```
        handlerTable ←
            getObject (tptr + fromIntegral tcbFaultHandlerTableOffset)
        let offset = ((fromIntegral addr `shiftR` (pageBits + levelBits)) .&.
                      mask levelBits) ∗ cteSize
```

Locate the table of fault handlers that shadows the top level of the capability table, and calculate the required offset into it.

```
        let tablePermMask = fromIntegral $ cteCapData handlerTable
        if (level = 2) ∧ (capAllowRead $ ctePerms handlerTable) ∧
                (capAllowSend $ permsFromWord tablePermMask)
```

Check the basic conditions for use of a region-specific fault handler: the fault is at the bottom level of the page table, the handler table is readable, and the permissions mask on the handler table allows messages to be sent to endpoints in it.

```
            then do
                let table = ctePointer handlerTable
                ptype ← getPageType table
                if ptype = CapTableObject
```

Check that the fault handler table is, in fact, a capability table.

```
                    then do
                        handler ← getObject $ table + offset
                        ptype ← getPageType (ctePointer handler)
                        if ptype = EndpointObject ∧
                                (capAllowSend $ ctePerms handler)
```

Check that the appropriate entry in the table contains a valid capability to an endpoint with permission to send an IPC to it.

```
                                then return handler else return defaultHandler
                        else return defaultHandler
                else return defaultHandler
```

If all of the above conditions are true, return the handler found in the table. Otherwise, return the default fault handler.

Initial Capability Creation

```
createInitCap :: Pointer → CTEPtr → State KernelState ()
createInitCap ptr cteptr = do
        let cte = CTE { ctePointer = ptr,
                        ctePerms = allPerms,
                        cteCapData = 0,
                        cteMDBNode = MDB { mdbNext = cteptr,
                                           mdbPrev = cteptr,
                                           mdbDepth = 0 } }
        setObject cteptr cte
```

This function is used during the bootstrap sequence. It creates a new capability to a page of user-managed memory, given a pointer to the memory and a pointer to a CTE to store the capability in.

A.4.4. Private Functions

Capability Verification

```
checkCap :: CapPtr → Pointer → Int → CapPerms → (CapPerms → Bool) →
            (ObjectType → Bool) →
            ErrorT Exception (State KernelState) ()
```

This function is used by the capability table lookup code to determine whether the permissions and type of the lookup's result are the same as those expected by the caller. The caller is expected to provide two functions that, given a set of permissions and an object type respectively, will return True if these are correct.

```
checkCap cap ptr level perms checkPerms checkType = do
        unless (checkPerms perms) $
            throwError $ Ex {exType = PermissionFault,
                             exAddress = cap,
                             exCTLevel = level,
                             exDesc = ``Insufficient permissions for cap''}
```

If the permissions check returns False, throw an exception.

```
        ptype ← lift $ getPageType ptr
        unless (checkType ptype) $
            throwError $ Ex {exType = CapFault,
                             exAddress = cap,
                             exCTLevel = level,
                             exDesc = ``Object was not of the expected type''}
```

Fetch the object type and provide it to the type-check function; throw an exception if it returns FALSE.

```
checkMask :: CapPtr → Int → CapPerms → (CapPerms → Bool) →
            ErrorT Exception (State KernelState) ()
checkMask cap level mask checkMask = do
        unless (checkMask mask) $
            throwError $ Ex {exType = PermissionFault,
                             exAddress = cap,
                             exCTLevel = level,
                             exDesc = ``Operation prohibited in this region''}
```

This is a simplifed version of checkCap that checks only the permissions. It is used during capability lookup, to check that permissions masks on the capability tables do not prohibit the requested operation.

Capability Space Lookups

```
findCTEAtLevel :: ThreadPtr → CapPtr → Int → (CapPerms → Bool) →
            ErrorT Exception (State KernelState) (Pointer, CapPerms)
```

This function is the basis of all other capability lookup functions in this module. It locates a CTE, given a capability address, a pointer to a TCB containing the root of a capability table, and a level in the table. It uses the given function to check the permissions on the CTE itself; that function should check for write permissions when performing a map operation, and read permissions at all other times.

The returned values are a pointer to the requested CTE (*not* to the object that it maps), and the product of all the permissions masks that affect access to the capability stored in the CTE.

```
findCTEAtLevel thread _ 0 _ =
        return (thread + fromIntegral tcbCTableOffset, allPerms)
```

The level 0 (root) CTE is simply retrieved from the TCB. There are no effective permissions masks, so all permissions bits are set in the result.

```
findCTEAtLevel thread addr 1 checkPerms = do
        rootCTE ← lift $ getObject (thread + fromIntegral tcbCTableOffset)
        let lperms1 = ctePerms rootCTE
```

```
        let lpermmask1 = permsFromWord $ fromIntegral $ cteCapData rootCTE
        let lroot1 = ctePointer rootCTE
        let loffset1 = (fromIntegral addr `shiftR` (pageBits + levelBits)) .&.
                       mask levelBits
        let lptr1 = lroot1 + loffset1 * cteSize


        checkCap addr lptr1 0 lperms1 checkPerms (=CapTableObject)
        return (lptr1, lpermmask1)
```

A level 1 CTE is located by applying a mask to the capability address, and using the result as an index into the root capability directory. The permissions mask is stored as an integer in the thread's capability to the root of the table.

```
findCTEAtLevel thread addr 2 checkPerms = do
        (lptr1, lpermmask1) ← findCTEAtLevel thread addr 1 capAllowRead
        lcte1 ← lift $ getObject lptr1


        let lperms2 = ctePerms lcte1
        let lpermmask2 = permsFromWord $ fromIntegral $ cteCapData lcte1
        let lroot2 = ctePointer lcte1
        let loffset2 = (fromIntegral addr `shiftR` pageBits) .&.
                       mask levelBits
        let lptr2 = lroot2 + loffset2 * cteSize


        checkCap addr lptr2 1 lperms2 checkPerms (=CapTableObject)
        return (lptr2, lpermmask1 `maskCapPerms` lpermmask2)
```

This is similar to the level 1 definition, except that the level 2 cap directory is obtained from the level 1 CTE and used instead of the root. Also, the address mask is not shifted as far to the right, and the returned permissions mask is the product of those for the two levels of the table.

```
findCTEAtLevel _ _ level _ = error ("Illegal CT level " ++ show level)
```

An attempt to access any other level of the page table will fail.

```
capLookupForCopy :: ThreadPtr → CapPtr →
          ErrorT Exception (State KernelState) CTEPtr
```

This is a helper function for capCopy, used to look up the destination CTE.

```
capLookupForCopy thread addr = do
        (table, _) ←
            capLookup thread addr capAllowWrite (=CapTableObject)
        return $ table + (fromIntegral addr .&. mask pageBits)
```

It calls capLookup with the appropriate parameters, and then adjusts the returned address to locate the required CTE in the page of them that capLookup returns.

```
findCTEForCall :: ThreadPtr → CapPtr → (CapPerms → Bool) →
            ErrorT Exception (State KernelState) CTEPtr
```

This function is used by capLookup to locate the CTE for the requested capability.

```
findCTEForCall t a checkPerms = do
        (_, lmask₁) ← findCTEAtLevel t a 1 capAllowRead
        checkMask a 1 lmask₁ checkPerms
        (cteptr, lmask₂) ← findCTEAtLevel t a 2 capAllowRead
        checkMask a 2 lmask₂ checkPerms
        return cteptr
```

It looks up the level 1 and level 2 entries separately; this is so checkMask will be able to set the capability table level correctly if it encounters a permissions error.

Revocation

```
revokeWithMinDepth :: Int → CTEPtr → CTEPtr →
            State KernelState CTEPtr
revokeWithMinDepth minDepth prevPtr ctePtr = do
        cte ← getObject ctePtr
        let mdb = cteMDBNode cte
        if mdbDepth mdb < minDepth
            then do
                let mdb' = mdb { mdbPrev = prevPtr }
                setObject ctePtr $ cte { cteMDBNode = mdb' }
                return ctePtr
            else do
                setObject ctePtr (makeObject :: CTE)
                revokeWithMinDepth minDepth prevPtr $ mdbNext mdb
```

This is a helper function used by cteRevoke. It traverses the MDB, invalidating each CTE it reaches, until it finds a node that is above the minimum depth in the tree. That node's leftwards pointer is set to the given value, and a pointer to the node is returned.

Constants

```
levelBits :: Int
levelBits = pageBits − objBits (⊥ :: CTE)
```

The number of bits of the address mapped by each level of the capability table.

```
cteSize :: POINTER
cteSize = 1 `shiftL` (objBits (⊥ :: CTE))
```

The size of a CTE.

A.5. Threads

```
module

 THREAD where
```

This module defines thread control blocks, and operations that act on them.

A.5.1. Data Types

```
type PRIORITY = WORD8
```

Threads have an 8 bit unsigned integer priority.

```
data THREADSTATE = TS { tsHalted, tsBlocked :: BOOL }
```

The thread may be halted (using the *ExchangeRegisters* system call), and/or blocked (waiting for completion of an IPC operation). Threads are runnable only if both of these conditions are false.

```
data TCB = THREAD { tcbText :: USERTEXT,
                    tcbContext :: USERCONTEXT,
                    tcbState :: THREADSTATE,
                    tcbPriority :: PRIORITY,
                    tcbFaultHandler :: CTE,
                    tcbException :: MAYBE EXCEPTION,
                    tcbCTable :: CTE,
                    tcbFaultHandlerTable :: CTE,
                    tcbTimeSlice :: INT,
                    tcbIPCNext :: THREADPTR,
                    tcbIPCEndpoint :: ENDPOINTPTR,
                    tcbIPCBadge :: INT }
```

This is the thread control block structure. It contains various data about the state of a thread, including its user-level context, runnable state, priority, CTEs for the fault handlers and capability table, and its remaining time allocation. There is also an EXCEPTION field, which is set while the thread is blocked sending a fault IPC; and three fields which are used to store the IPC state when the thread is in an IPC send or recieve queue.

The UserText value in the TCB contains a program that can be executed by the user-level simulator. This is clearly specific to the Haskell model; in a standalone implementation the executed code is stored in the thread's virtual address spoce.

A.5.2. Type Class Instance

The following is the instance of Object for thread control blocks. This defines how TCBs are stored in the physical memory model. Refer to section A.7.3 on page 89 for the definition of this type class.

```
instance Object TCB where
    objBits _ = pageBits

    makeObject = Thread [] newContext
                    (TS {tsBlocked = False, tsHalted = True})
                    100 makeObject Nothing makeObject makeObject timeSlice
                    nullPointer nullPointer 0
    makeObjectPage = ThreadPage

    getFromPage 0 (ThreadPage tcb) = tcb
    getFromPage _ _ = typeError ``cap␣table␣or␣tcb␣page''

    setInPage tcb 0 (ThreadPage _) = ThreadPage tcb
    setInPage _ _ _ = typeError ``cap␣table␣or␣tcb␣page''
```

A.5.3. System Call Implementations

```
threadControl :: ThreadPtr → ThreadPtr → Maybe UserText →
                 Maybe (CTEPtr, CapPerms) → Maybe (CTEPtr, CapPerms) →
                 Maybe Priority → Maybe (CTEPtr, CapPerms) →
                 State KernelState ()
```

This function implements the *ThreadControl* system call.

```
threadControl cur target text capTable faultHandler prio handlerTable = do
        case text of
            Just text → threadSet (λtcb → tcb {tcbText = text}) target
            _ → return ()
```

If a new value has been supplied for the TCB's tcbText field, set it; otherwise do nothing.

```
        case prio of
            Just prio → setPriority target prio
            _ → return ()
```

Do the same for the thread's priority.

```
        case capTable of
            Just (src, srcpmask) → do
                let dest = target + fromIntegral tcbCTableOffset
                oldCTE ← getObject dest
                unless (ctePointer oldCTE = 0) $
                    cteRevoke True dest
                cteCopy srcpmask src dest
            _ → return ()
```

If a new cap table root has been supplied, call `capCopy` to copy the capability into the `TCB`'s root `CTE`. If there was previously a valid root `CTE`, it is revoked first.

```
        case faultHandler of
            Just (src, srcpmask) → do
                let dest = target + fromIntegral tcbFaultHandlerOffset
                oldCTE ← getObject dest
                unless (ctePointer oldCTE = 0) $
                    cteRevoke True dest
                cteCopy srcpmask src dest
            _ → return ()
        case handlerTable of
            Just (src, srcpmask) → do
                let dest = target + fromIntegral tcbFaultHandlerTableOffset
                oldCTE ← getObject dest
                unless (ctePointer oldCTE = 0) $
                    cteRevoke True dest
                cteCopy srcpmask src dest
            _ → return ()
```

Do the same for the default fault handler and the fault handler table.

```
exchangeRegisters :: ThreadPtr → ThreadPtr → ThreadPtr → Int → Pointer →
                State KernelState ()
```

This function implements the *ExchangeRegisters* system call.

```
exchangeRegisters cur src dest flags saveptr = do
        haltThread src $ flags .&. 1 ≠ 0
        haltThread dest $ flags .&. 2 ≠ 0
```

The first two bits of the bitfield argument determine whether the source and destination threads, respectively, should be halted after the call returns.

```
        when (flags .&. 4 ≠ 0) $ do
```

If the third bit is set, we must copy the part of the integer register set that cannot be directly transferred. The action to be taken depends on whether the current thread is the source, the destination, or neither.

```
if cur = src
    then do — copy from save area
        mapM (λr → do
                v ← getObject $ saveptr + intSize ∗
                        (fromIntegral $ fromEnum r + 1)
                asUser dest $ setRegister r (v :: Int))
              [SP .. AR7]
        ip ← getObject saveptr
        asUser dest $ setIP ip
```

If the current thread is the source, then the instruction pointer, stack pointer and argument registers are copied from the save area into the destination thread's state.

```
    else if cur = dest
        then do — copy to save area
            mapM (λr → do
                    v ← asUser src $ getRegister r
                    setObject (saveptr + intSize ∗
                        (fromIntegral $ fromEnum r + 1)) (v :: Int))
                  [SP .. AR7]
            ip ← asUser src $ getIP
            setObject saveptr ip
```

Otherwise, if the current thread is the destination, the registers are copied from the source thread's state into the save area.

```
        else do — copy directly
            mapM (λr → do
                    v ← asUser src $ getRegister r
                    asUser dest $ setRegister r v)
                  [SP .. AR7]
            ip ← asUser src $ getIP
            asUser dest $ setIP ip
```

If neither source or destination is the current thread, then the registers are copied directly from the source thread's state to the destination's.

```
when (flags .&. 8 ≠ 0) $ do
    mapM (λr → do
            v ← asUser src $ getRegister r
            asUser dest $ setRegister r v)
          [R0 ..]
    return ()
where intSize = 1 `shiftL` objBits (⊥ :: Int)
```

If the fourth bit is set, the remaining integer registers are to be copied. This is always done directly.

A.5.4. Public Functions

Bootstrapping

```
configureInitialThread :: ThreadPtr → UserText → CTEPtr → Priority →
                 State KernelState ()
```

This function is similar to *ThreadControl*, but is called from the bootstrap routine to configure the initial thread.

```
configureInitialThread threadPtr text root prio = do
        threadSet (λtcb → tcb {tcbText = text,
                                tcbPriority = prio}) threadPtr
        cteCopy allPerms root
                (threadPtr + fromIntegral tcbCTableOffset)
```

It sets the text and priority of the initial thread, and copies the root CTE.

```
activateInitialThread :: ThreadPtr → Pointer → State KernelState ()
```

This function is called from the bootstrap routine to activate the initial thread. The Pointer parameter is a structure stored in a physical memory page, containing information about the kernel's configuration.

```
activateInitialThread threadPtr infoPtr = do
        asUser threadPtr $ setRegister R0 $ fromIntegral infoPtr
```

The `infoPtr` should be passed to the user-level thread in some architecture-defined way. In the Haskell model, it is placed in the first general purpose register.

```
        oldSt ← getThreadState threadPtr
        setThreadState threadPtr $ oldSt { tsHalted = False }
        schedule
```

The thread state is then set to make the thread runnable, and the scheduler is called.

Thread State

The following two functions change the blocked or halted state of the thread, placing them in or removing them from the ready queue as necessary.

```
blockThread :: ThreadPtr → Bool → State KernelState ()
blockThread tptr isBlocked = do
        oldSt ← getThreadState tptr
        setThreadState tptr $ oldSt { tsBlocked = isBlocked }
```

```
haltThread :: ThreadPtr → Bool → State KernelState ()
haltThread tptr isHalted = do
        oldSt ← getThreadState tptr
        setThreadState tptr $ oldSt { tsHalted = isHalted }
```

The following functions extract or set any data item in a TCB. They are used elsewhere in the kernel to access data in the TCB which does not require any additional processing (as opposed to, for example, the priority, which might require changes to the ready queues).

```
threadGet :: (TCB → a) → ThreadPtr → State KernelState a
threadGet f tptr = liftM f $ getObject tptr
```

```
threadSet :: (TCB → TCB) → ThreadPtr →
                State KernelState ()
threadSet f tptr = do
        tcb ← getObject tptr
        setObject tptr $ f tcb
```

IPC Transfers

```
setException :: Exception → ThreadPtr → State KernelState ()
setException ex = threadSet (λtcb → tcb { tcbException = Just ex })
```

Set the Exception structure being sent by a fault IPC.

```
doIPCTransfer :: ThreadPtr → ThreadPtr → Int → State KernelState ()
doIPCTransfer sender receiver badge = do
        ex ← threadGet tcbException sender
        case ex of
            Just _ → setExceptionMRs badge sender receiver
            Nothing → doMRTransfer badge sender receiver
```

Select between ordinary and fault IPC, and call the appropriate handler function.

User-level Execution

```
runThread :: ThreadPtr → State KernelState Event
runThread thread = do
        text ← threadGet tcbText thread
        time ← threadGet tcbTimeSlice thread
        (time′, ev) ← asUser thread $ executeInstructions text time
        threadSet (λtcb → tcb {tcbTimeSlice = time′}) thread
        return ev
```

Execute a thread at user level until an event occurs.

```
asUser :: ThreadPtr → State UserContext a → State KernelState a
asUser tptr f = do
        uc ← threadGet tcbContext tptr
        let (a, uc′) = runState f uc
        threadSet (λtcb → tcb { tcbContext = uc′ }) tptr
        return a
```

Evaluate a function in the user-level context of a thread. Functions that can be used as arguments to this function are defined in the `UserState` module.

A.5.5. Constants

These are the physical memory offsets into the `TCB` at which the three `CTE`s may be found. These values are obviously artificial; in a real implementation they would depend on the layout of the `TCB` structure.

```
tcbFaultHandlerOffset :: Word
tcbFaultHandlerOffset = 1
```

```
tcbCTableOffset :: Word
tcbCTableOffset = 2
```

```
tcbFaultHandlerTableOffset :: Word
tcbFaultHandlerTableOffset = 3
```

A.5.6. Private Functions

Thread State

```
getThreadState :: ThreadPtr → State KernelState ThreadState
getThreadState = threadGet tcbState
```

Trivial function to get the current thread state.

```
setThreadState :: ThreadPtr → ThreadState → State KernelState ()
setThreadState tptr st = do
        tcb ← getObject tptr
        setObject tptr $ tcb { tcbState = st }
        q ← getQueue $ tcbPriority tcb
        if not $ tsHalted st ∨ tsBlocked st
            then setQueue (tcbPriority tcb) $ tptr : q
            else do
                setQueue (tcbPriority tcb) $ filter (≠ tptr) q
                curThread ← getCurThread
                when (tptr = curThread) $ schedule
```

Set the state of a thread. Depending on the current and new values of the state, the thread may need to be removed from or added to the ready queue.

Scheduler

```
setPriority :: ThreadPtr → Priority → State KernelState ()
setPriority tptr prio = do
        tcb ← getObject tptr
        let oldPrio = tcbPriority tcb
        setObject tptr $ tcb { tcbPriority = prio }
        let st = tcbState tcb
        if not $ tsHalted st ∨ tsBlocked st
            then do
                q1 ← getQueue oldPrio
                setQueue oldPrio $ filter (≠ tptr) q1
                q2 ← getQueue prio
                setQueue prio $ tptr : q2
            else return ()
```

Set the priority of a thread. If the thread is presently runnable, it must be moved from the old priority's queue to the new priority's queue.

```
schedule :: State KernelState ()
schedule = schedule' (maxBound :: Priority)
        where
```

```
schedule' prio = do
    q ← getQueue prio
    case (q, prio) of
        (first:rest, _) → do
            setQueue prio (rest ++ [first])
            setCurThread first
        ([], 0) → error ''No runnable threads''
        ([], _) → schedule' $ prio−1
```

Choose the next thread to run. This simply finds the first non-empty queue, and returns the first item in it after moving that item to the end of the queue.

IPC Transfers

```
doMRTransfer :: Int → ThreadPtr → ThreadPtr → State KernelState ()
doMRTransfer badge sender receiver = do
        mapM (λr → do
              value ← asUser sender $ getRegister r
              asUser receiver $ setRegister r value)
             [AR1 .. AR7]
        asUser receiver $ setRegister AR0 badge
```

Transfer the architecture-specific set of IPC message registers from the sender to the receiver thread, and set the sender's badge in one of the receiver's other registers.

```
setExceptionMRs :: Int → ThreadPtr → ThreadPtr → State KernelState ()
setExceptionMRs badge sender receiver = do
        Just ex ← threadGet tcbException sender
        threadSet (λtcb → tcb {tcbException = Nothing}) sender
        asUser receiver $ setRegister AR1 $ shiftL (fromEnum $ exType ex) 6
        ip ← asUser sender getIP
        asUser receiver $ setRegister AR2 $ ip
        unless (exType ex = InvalidArgument) $ do
            asUser receiver $ setRegister AR3 $ fromIntegral $ exAddress ex
            asUser receiver $ setRegister AR4 $ fromIntegral $ exCTLevel ex
        asUser receiver $ setRegister AR0 badge
```

Transfer a fault IPC. This reads the Exception value presently stored in the TCB, and places appropriate parts of it into the receiver's message registers. The exception is then set to Nothing, so the next IPC will not re-send the fault.

A.6. IPC and Endpoints

```
module
```

```
ENDPOINT where
```

This module defines IPC endpoints, and operations that act on them.

A.6.1. Data Types

```
data ENDPOINT = IDLEEP |
                SENDEP THREADPTR |
                RECVEP THREADPTR
```

This type represents an IPC endpoint. Endpoints may be idle, or have a queue of threads that are either all waiting to send, or all waiting to receive. In the latter two cases the ENDPOINT stores a physical pointer to the first thread in the queue.

A.6.2. Type Class Instance

The following is the instance of OBJECT for IPC endpoints. This defines how ENDPOINTS are stored in the physical memory model. Refer to section A.7.3 on page 89 for the definition of this type class.

```
instance OBJECT ENDPOINT where
    objBits _ = pageBits

    makeObject = IDLEEP
    makeObjectPage = ENDPOINTPAGE

    getFromPage 0 (ENDPOINTPAGE ep) = ep
    getFromPage _ _ = typeError ``endpoint page''

    setInPage ep 0 (ENDPOINTPAGE _) = ENDPOINTPAGE ep
    setInPage _ _ _ = typeError ``endpoint page''
```

A.6.3. Public Functions

```
sendIPC :: ThreadPtr → EndpointPtr → Int → State KernelState ()
```

This function sends an IPC.

```
sendIPC thread epptr badge = do
        ep ← getObject epptr
        case ep of
```

There are three possible cases, depending on the state of the endpoint.

```
                    IdleEP → do
                        blockThread thread True
                        threadSet (λtcb → tcb { tcbIPCBadge = badge,
                                                 tcbIPCEndpoint = epptr})
                                  thread
                        setObject epptr $ SendEP thread
```

If the endpoint is idle, it becomes a sending endpoint, with only the current thread in the queue. The badge and endpoint are set in the thread's TCB, to be used in the message transfer and when cancelling the IPC operation, respectively. The current thread is blocked while waiting in the queue.

```
                    SendEP next → do
                        blockThread thread True
                        threadSet (λtcb → tcb { tcbIPCNext = next,
                                                 tcbIPCEndpoint = epptr,
                                                 tcbIPCBadge = badge})
                                  thread
                        setObject epptr $ SendEP thread
```

The send endpoint case is similar to the idle endpoint case, except that the current thread is added to the existing queue.

```
                    RecvEP dest → do
                        next ← threadGet tcbIPCNext dest
                        threadSet (λtcb → tcb { tcbIPCNext = nullPointer,
                                                 tcbIPCEndpoint = nullPointer})
                                  dest
                        setObject epptr $ if next = 0
                                then IdleEP
                                else RecvEP next
                        doIPCTransfer thread dest badge
                        blockThread dest False
                        schedule
```

If the endpoint is a receive endpoint, sending to it will immediately complete. To do this, the kernel removes the first thread from the endpoint's queue, unblocks it, and transfers the message to it. The scheduler is then called to switch to the recipient.

```
receiveIPC :: ThreadPtr → EndpointPtr → State KernelState ()
```

Receives an IPC.

```
receiveIPC thread epptr = do
        ep ← getObject epptr
        case ep of
            IdleEP → do
                blockThread thread True
                threadSet (λtcb → tcb { tcbIPCEndpoint = epptr}) thread
                setObject epptr $ RecvEP thread
            RecvEP next → do
                blockThread thread True
                threadSet (λtcb → tcb { tcbIPCNext = next ,
                                         tcbIPCEndpoint = epptr})
                          thread
                setObject epptr $ RecvEP thread
            SendEP sender → do
                next ← threadGet tcbIPCNext sender
                threadSet (λtcb → tcb { tcbIPCNext = nullPointer ,
                                         tcbIPCEndpoint = nullPointer})
                          sender
                setObject epptr $ if next = 0
                        then IdleEP
                        else SendEP next
                badge ← threadGet tcbIPCBadge sender
                doIPCTransfer sender thread badge
                blockThread sender False
```

The recieve and send functions are almost identical, other than the endpoint receive and send states being swapped. Also, in this case the scheduler is not called after an IPC operation completes, because the recipient is already running.

```
ipcCancel :: EndpointPtr → ThreadPtr → State KernelState ()
```

Attempt to cancel an operation on the given endpoint, being performed by the given thread.

```
ipcCancel epptr tptr = do
        ep ← getObject epptr
        ep′ ← case ep of
            IdleEP → return IdleEP
            SendEP first → do
                next ← ipcCancel′ first
```

```
            case next of
                0 → return IdleEP
                _ → return $ SendEP next
        RecvEP first → do
            next ← ipcCancel′ first
            case next of
                0 → return IdleEP
                _ → return $ RecvEP next
    setObject epptr ep
```

Call ipcCancel′ (defined below) to find the new first item in the queue, and set the endpoint appropriately.

```
    asUser tptr $ setRegister AR0 (−1)
    blockThread tptr False
```

Indicate to the thread that the IPC operation failed, and unblock it.

```
    where
        ipcCancel′ this = do
            next ← threadGet tcbIPCNext tptr
            if this = tptr
                then do
                    threadSet (λtcb → tcb
                                  { tcbIPCNext = nullPointer,
                                    tcbIPCEndpoint = nullPointer})
                              this
                    return next
                else do
                    next′ ← ipcCancel′ next
                    when (next′ ≠ next) $
                        threadSet (λtcb → tcb {tcbIPCNext = next′}) tptr
                    return this
```

Iterate through all threads in the endpoint's queue until the thread in question is found; then remove it frome the queue.

```
epCancelAll :: EndpointPtr → State KernelState ()
```

Cancel all IPC being performed on a given endpoint.

```
epCancelAll ptr = do
    ep ← getObject ptr
    case ep of
        IdleEP → return ()
        SendEP thread → do
            ipcCancel ptr thread
            epCancelAll ptr
```

```
        RecvEP thread → do
            ipcCancel ptr thread
            epCancelAll ptr
```

Repeatedly call ipcCancel until the endpoint is idle.

A.7. Kernel Objects

```
module Object where
```

This module contains the definition of the Object type class, which is used by the physical address space model to define the characteristics of different data types when stored in physical memory.

It also defines the PhysicalPage type, which is used for storing the contents of a single physical frame.

A.7.1. Data Types

```
data PhysicalPage = KernelPage |
                    EmptyPage |
                    ThreadPage { pageTCB :: TCB } |
                    EndpointPage { pageEndpoint :: Endpoint } |
                    CapTablePage { pageCapTable :: Array Word CTE } |
                    DataPage { pageData :: Array Word Int }
```

This type defines the contents of a frame of physical memory. It is discussed in detail in section 4.3.2 on page 35.

```
data KernelData = KernelData
```

```
data UntypedData = UntypedData
```

These two data types define the contents of physical memory that is used by the kernel, or currently unused, respectively. Neither of these types have any accessible contents.

A.7.2. Public Functions

The following functions are convenience functions that are called when a type or alignment error occurs in the kernel. They evaluate the `error` function; doing this immediately exits the Haskell model with an error code.

```
typeError :: String → a
typeError t = error (''Wrong␣page␣type␣-␣expected␣'' ++ t)
```

```
alignError :: Int → a
alignError n = error (''Unaligned␣access␣-␣lowest␣'' ++
                      (show n) ++ ''␣bits␣must␣be␣0'')
```

A.7.3. Type Class

The OBJECT type class is used for any type which may be stored in the modelled physical memory. It defines a functions that are used for checking address alignment, creating new objects and frames, and accessing the contents of a frame.

```
class Object a where
    objBits :: a → Int
    makeObject :: a
    makeObjectPage :: a → PhysicalPage

    getFromPage :: Word → PhysicalPage → a
    setInPage :: a → Word → PhysicalPage → PhysicalPage
```

A.7.4. Type Class Instances

The following are instances for the OBJECT type class for the two different "empty" frame types. Accesses to these types always fail with a type error.

```
instance Object UntypedData where
    objBits _ = pageBits
    makeObject = UntypedData
    makeObjectPage _ = EmptyPage

    getFromPage _ EmptyPage = UntypedData
    getFromPage _ _ = typeError ''empty␣page''
```

```
    setInPage _ _ EmptyPage = EmptyPage
    setInPage _ _ _ = typeError ''empty page''


instance Object KernelData where
    objBits _ = pageBits
    makeObject = KernelData
    makeObjectPage _ = KernelPage


    getFromPage _ KernelPage = KernelData
    getFromPage _ _ = typeError ''kernel page''


    setInPage _ _ KernelPage = KernelPage
    setInPage _ _ _ = typeError ''kernel page''
```

This defines the instance of Object for the basic integer type, Int. Integers are defined to be 4 bytes long, and are stored in an array inside their page. The following functions create and access that array.

```
instance Object Int where
    objBits _ = 2
    makeObject = 0
    makeObjectPage obj = DataPage $
            listArray (0, (1 `shiftL` (pageBits − objBits obj)) − 1) $
                repeat obj


    getFromPage offset (DataPage array) =
            if align = 0
                then val
                else alignError (objBits val)
            where
                val = array!index
                index = offset `shiftR` (objBits val)
                align = offset .&. (fromIntegral $ mask (objBits val))
    getFromPage _ _ = typeError ''int data page''


    setInPage val offset (DataPage array) =
            if align = 0
                then DataPage $ array//[(index, val)]
                else alignError (objBits val)
            where
                index = offset `shiftR` (objBits val)
                align = offset .&. (fromIntegral $ mask (objBits val))
    setInPage _ _ _ = typeError ''int data page''
```

A.8. Physical Address Space

```
module
 PSpace where
```

This module defines the physical memory model.

A.8.1. Data Types

```
type PSpace = Array Word PhysicalPage
```

The physical address space is simply an array of physical frames. The latter are represented by the type PhysicalPage, which is discussed in section 4.3.2 on page 35.

A.8.2. Public Functions

Initialisation

```
newPSpace :: Word → PSpace
newPSpace size = listArray (0 , size − 1)
                               $ KernelPage : repeat EmptyPage
```

Creates a new physical memory model, containing a given number of frames. Note that the first frame is a KernelPage; this must always be the case to ensure that accesses to null pointers fail immediately.

Accessing Objects

```
getObject :: Object a ⇒ Pointer → State KernelState a
getObject ptr = do
        page ← getPage ptr
        let offset = fromIntegral $ (ptr .&. mask pageBits)
        return $ getFromPage offset page
```

Retrieves an object from physical memory. This is done by fetching the frame containing it, and returning the result of getFromPage applied to that frame. Note that this function is polymorphic. Type checking is performed at run time by getFromPage,

which is defined by the Object type class; that function will fail if the frame does not contain an object of the requested type.

```
setObject :: Object a ⇒ Pointer → a → State KernelState ()
setObject ptr obj = do
        page ← getPage ptr
        let offset = fromIntegral $ (ptr .&. mask pageBits)
        let page′ = setInPage obj offset page
        setPage ptr page′
```

Stores an object in physical memory. The operation of this function is analagous to that of `getObject`.

Accessing Frames

The following two functions get and set physical frames in the frame array.

```
setPage :: Pointer → PhysicalPage → State KernelState ()
setPage ptr page = do
        ps ← liftM ksPSpace get
        let pageNum = fromIntegral $ ptr `shiftR` pageBits
        if inRange (bounds ps) pageNum
            then modify $ λks → ks { ksPSpace = ps//[(pageNum, page)] }
            else error (``setPage␣with␣bad␣pointer:␣'' ++ show ptr)
```

```
getPage :: Pointer → State KernelState PhysicalPage
getPage ptr = do
        ps ← liftM ksPSpace get
        let pageNum = fromIntegral $ ptr `shiftR` pageBits
        if inRange (bounds ps) pageNum
            then return $ ps ! pageNum
            else error (``getPage␣with␣bad␣pointer:␣'' ++ show ptr)
```

A.9. Object Types

```
module PageType (retypePage, getPageType) where
```

This module contains functions that determine or change the type of object being stored in a physical frame.

Note that this module refers to physical frames as “pages”. This is simply an error in terminology that has not been corrected yet; it has no particular meaning.

A.9.1. Public Functions

```
retypePage :: ObjectType → Pointer → State KernelState ()
```

Change the type of object stored in a physical frame.

```
retypePage otype ptr = do
        detypePage ptr
```

First, prepare the frame to have its type changed, by cleaning up any existing object stored in it.

```
        let page = case otype of
                Untyped → makeObjectPage (makeObject :: UntypedData)
                IntDataObject → makeObjectPage (makeObject :: Int)
                TCBObject → makeObjectPage (makeObject :: TCB)
                EndpointObject → makeObjectPage (makeObject :: Endpoint)
                CapTableObject → makeObjectPage (makeObject :: CTE)
        setPage ptr page
```

Create a frame containing a new initialised kernel object of the appropriate type, and store it at the given physical address.

```
getPageType :: Pointer → State KernelState ObjectType
getPageType ptr = do
        page ← getPage ptr
        case page of
            EmptyPage → return Untyped
            DataPage {} → return IntDataObject
            ThreadPage {} → return TCBObject
            EndpointPage {} → return EndpointObject
            CapTablePage {} → return CapTableObject
            _ → error ''User␣accessed␣a␣kernel␣page''
```

Fetch the type of the given frame, and return it. If a page reserved by the kernel is accessed, fail with an error — user-level threads should never possess a valid capability pointing to such an address.

A.9.2. Private Functions

```
detypePage :: Pointer → State KernelState ()
```

Clean up the existing contents of a frame.

```
detypePage ptr = do
        page ← getPageType ptr
```

```
case page of
    Untyped → return ()
    IntDataObject → return ()
```

The contents of untyped and integer data frames have no special meaning to the kernel, so nothing needs to be done in these cases.

```
        TCBObject → do
            haltThread ptr True
            ep ← threadGet tcbIPCEndpoint ptr
            when (ep ≠ nullPointer) $
                ipcCancel ep ptr
            cteRevoke True
                (ptr + fromIntegral tcbCTableOffset)
            cteRevoke True
                (ptr + fromIntegral tcbFaultHandlerOffset)
            cteRevoke True
                (ptr + fromIntegral tcbFaultHandlerTableOffset)
```

If the frame contains a thread, it must be halted to remove it from the ready queue, have its IPC operations cancelled, and have all capabilities revoked.

```
        EndpointObject → epCancelAll ptr
```

If the frame contains an endpoint, all outstanding IPC operations using it must be cancelled.

```
        CapTableObject → mapM_ (cteRevoke True)
                              [ptr,ptr+cteSize..ptr+pageSize−cteSize]
```

If the frame contains a capability table node, every CTE in it must be revoked.

```
    where
        cteSize = 1 `shiftL` objBits (⊥::CTE)
        pageSize = 1 `shiftL` pageBits
```

These local constants are used when calculating addresses of CTEs to revoke.

A.10. Bootstrapping

```
module Init where
```

This module defines functions that configure the initial state of the kernel.

A.10.1. Public Functions

```
initKernel :: UserText → State KernelState ()
```

This function bootstraps the system; it creates the initial thread and its capability space.

```
initKernel text = do
        retypePage TCBObject threadPtr
        retypePage CapTableObject rootPtr
        mapM (retypePage CapTableObject) lptrs2
```

Create the initial thread's TCB, and first and second level capability table nodes.

```
        zipWithM (λptr cteptr →
                    createInitCap ptr cteptr >> cteSetData cteptr (−1))
                lptrs2 $ map (+rootPtr) [0, cteSize..]
```

Create capabilities for the second level capability table nodes, and place them in the first level node.

```
        zipWithM createInitCap otherptrs $
                map (+head lptrs2) $ map ptrToCap otherptrs
        cteSetData (ptrToCap rootPtr + head lptrs2) (−1)
```

Create capabilites for all remaining memory in the second level nodes.

```
        zipWithM (cteCopy allPerms)
                (map (+rootPtr) $
                     [0, cteSize .. cteSize∗(fromIntegral lcount2−1)])
                (map (+head lptrs2) $ map ptrToCap lptrs2)
```

Copy capabilities for the second level nodes into the address space.

```
        configureInitialThread threadPtr userRootTask
                (ptrToCap rootPtr + head lptrs2) 255
```

Configure the initial thread.

```
        retypePage IntDataObject infoPtr
        zipWithM setObject
            [infoPtr,infoPtr+intSize..]
            ([fromIntegral memSize, fromIntegral kernelTop,
                 fromIntegral threadPtr, fromIntegral rootPtr,
                 fromIntegral $ head lptrs2, fromIntegral lcount2] :: [Int])
```

Create and fill a data page with information about the initial state of the system.

```
        activateInitialThread threadPtr infoPtr
```

Activate the initial thread, and give it a pointer to the information.

```
        where
```

Local variable definitions follow:

```
                pageSize :: Pointer
                pageSize = 1 `shiftL` pageBits
                intSize :: Pointer
                intSize = 1 `shiftL` (objBits (⊥ :: Int))
                cteSize :: Pointer
                cteSize = 1 `shiftL` (objBits (⊥ :: CTE))
```

Constants for the sizes of various object types.

```
                lcount2 = ((memSize-1) `shiftR`
                        (pageBits - objBits (⊥::CTE)))+1
```

The number of level 2 page table nodes needed to map all physical memory in the system.

```
                lptrs2 = map (fromIntegral . (`shiftL` pageBits))
                        [memSize-lcount2 .. memSize-1]
```

A list of pointers to the second level page table nodes.

```
                otherptrs = map (fromIntegral . (`shiftL` pageBits))
                        [kernelTop .. memSize-lcount2-1]
```

A list of pointers to every frame that isn't used by either the capability table or the kernel.

```
                (rootPtr:threadPtr:infoPtr:_) = reverse otherptrs
```

Place the CT root, TCB, and information page in the last three available frames.

```
                ptrToCap :: Pointer → CTEPtr
                ptrToCap = (flip shiftR) (pageBits - objBits (⊥::CTE))
```

A local function that converts physical pointers to offsets into the second level capability table nodes.

```
    newKernelState :: KernelState
    newKernelState = KState { ksPSpace = newPSpace memSize,
                              ksReadyQueues = listArray (minBound,maxBound)
                                                    (repeat []),
                              ksCurThread = ⊥ }
```

A function that creates a new system state structure.

A.11. User Level Machine Model

A.11.1. Instructions and Registers

```
module Instruction where
```

```
data Register =
          SP |
          AR0 | AR1 | AR2 | AR3 | AR4 | AR5 | AR6 | AR7 |
          R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
          R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
          R16 | R17 | R18 | R19 | R20 | R21 | R22 | R23 |
          R24 | R25 | R26 | R27 | R28 | R29 | R30 | R31
     deriving (Enum, Ord, Eq, Show)
```

This is the set of registers accessible at user level. SP is the stack pointer; AR_0 through to AR_7 are the system call argument registers; the remainder are general purpose integer registers.

This register set is abnormally large, and an odd size. That has no effect on the behaviour of the model, however. On a typical real architecture, the stack pointer and argument registers would be general purpose registers assigned to those tasks by the ABI.

```
data Instruction = Arithmetic Register (Int → Int → Int) Register Register |
            ArithmeticI Register (Int → Int) Register |
            Compare Register (Int → Int → Bool) Register Register |
            CompareI Register (Int → Bool) Register |
            LoadImmediate Int Register |
            Load Int Register Register |
            Store Register Int Register |
            Push Register |
            Pop Register |
            Move Register Register |
            Branch Int |
            BranchLinked Int Register |
            BranchIf Register Int |
            Syscall Int |
            DebugPrintf String [Register]
```

This is the set of instructions supported by the simple simulated CPU. See section 4.7 on page 40.

```
type UserText = [Instruction]
```

This is the type used to represent a user-level program. It is simply a list of instructions.

```
nullText :: UserText
nullText = []
```

An empty program.

A.11.2. User Level State

```
data UserContext = UC { ucRegisters :: Map Register Int,
                        ucIP :: Int }
```

This data type represents the state of a user-level thread. It contains an instruction pointer (which is actually an index into the list of instructions, not a pointer to memory), and also a mapping from register names to integer values.

```
newContext :: UserContext
newContext = UC { ucRegisters = empty,
                  ucIP = 0 }
```

The initial instruction pointer is 0, and the register map is empty. The latter effectively sets all registers to the default value in `getRegister`, which is also 0.

```
getIP :: State UserContext Int
getIP = gets $ ucIP
```

Get the current instruction pointer.

```
setIP :: Int → State UserContext ()
setIP ip = modify $ λuc → uc { ucIP = ip }
```

Set the instruction pointer.

```
advanceIP :: State UserContext Int
advanceIP = do
        ip ← getIP
        setIP (ip+1)
        return ip
```

Advance the instruction pointer by one instruction.

```
rewindIP :: State UserContext ()
rewindIP = do
        ip ← getIP
```

```
        setIP (ip-1)
```

Move the instruction pointer backwards by one instruction.

```
getRegister :: Num a ⇒ Register → State UserContext a
getRegister r = do
        uc ← get
        let mvalue = Map.lookup r (ucRegisters uc)
        case mvalue of
            Just value → return $ fromIntegral value
            Nothing → return 0
```

Fetch the value of the given register. If no value exists yet, return the default value 0.

```
setRegister :: Integral a ⇒ Register → a → State UserContext ()
setRegister r v = modify $
        λuc → uc { ucRegisters = insert r (fromIntegral v) (ucRegisters uc) }
```

Set the value of the given register.

A.11.3. CPU Simulation

```
module CPU (executeInstructions) where
```

This module contains the interpreter for the simple assembler-like language. It is documented in section 4.7 on page 40.

A.11.4. Public Functions

```
executeInstructions :: UserText → Int → State UserContext (Int, Event)
```

This function, given a program and a time limit, executes the program until an event occurs or it runs out of time. It returns the amount of remaining time, and the event that occurred.

```
executeInstructions _ 0 = return $ (0, TimerInterrupt)
```

If there is no time left, generate a timer interrupt.

```
executeInstructions text time = do
        ip ← advanceIP
        result ← executeInstruction (text!!ip)
        case result of
            Just event → return (time-1, event)
```

```
NOTHING → executeInstructions text (time−1)
```

Otherwise, recursively execute instructions until one of them generates an event.

A.11.5. Internal Functions

```
executeInstruction :: INSTRUCTION → STATE USERCONTEXT (MAYBE EVENT)
```

This function interprets a single instruction, and may return an event. There is one definition below for each possible instruction type.

```
executeInstruction (ARITHMETIC ra f rb rd) = do
        a ← getRegister ra
        b ← getRegister rb
        setRegister rd $ f a b
        return NOTHING


executeInstruction (ARITHMETICI ra f rd) = do
        a ← getRegister ra
        setRegister rd $ f a
        return NOTHING


executeInstruction (COMPARE ra f rb rd) =
        executeInstruction $ ARITHMETIC ra (λa b → fromEnum $ f a b) rb rd


executeInstruction (COMPAREI ra f rd) =
        executeInstruction $ ARITHMETICI ra (λa → fromEnum $ f a) rd


executeInstruction (LOADIMMEDIATE i rd) = setRegister rd i >> return NOTHING


executeInstruction (INSTRUCTION.LOAD i ra rb) = do
        a ← getRegister ra
        return $ JUST $ KERNELAPI.LOAD (fromIntegral $ i+a) rb


executeInstruction (INSTRUCTION.STORE ra i rb) = do
        b ← getRegister rb
        return $ JUST $ KERNELAPI.STORE ra (fromIntegral $ i+b)


executeInstruction (PUSH ra) = do
        sp ← getRegister SP
```

```
        setRegister SP $ sp−intSize
        return $ Just $ KernelAPI.Store ra (fromIntegral $ sp−intSize)


executeInstruction (Pop ra) = do
        sp ← getRegister SP
        setRegister SP $ sp+intSize
        return $ Just $ KernelAPI.Load (fromIntegral sp) ra


executeInstruction (Move ra rb) = do
        a ← getRegister ra
        setRegister rb a
        return Nothing


executeInstruction (Branch a) = do
        ip ← getIP
        setIP $ ip + a − 1
        return Nothing


executeInstruction (BranchIf ra b) = do
        a ← getRegister ra
        if a≠0
            then executeInstruction $ Branch b
            else return Nothing


executeInstruction (BranchLinked a rl) = do
        ip ← getIP
        setRegister rl ip
        executeInstruction $ Branch a


executeInstruction (Syscall n)
    | n≥0 ∧ n ≤ fromEnum (maxBound :: Syscall) =
        return $ Just $ SyscallEvent $ toEnum n
    | otherwise = return $ Just $ UnknownSyscall n


executeInstruction (DebugPrintf msg regs) = do
        values ← mapM getRegister regs
        let string = formatError msg values
        return $ Just $ DebugPrint string


formatError :: String → [Int] → String
formatError ‘‘’’ _ = ‘‘’’
```

```
formatError ('%':chars) (i:values) = (show i) ++ (formatError chars values)
formatError (x:chars) values = x : (formatError chars values)


intSize :: Pointer
intSize = 1 `shiftL` objBits (⊥::Int)
```