

# **seL4 Reference Manual**

Philip Derrin    Dhammika Elkaduwe    Kevin Elphinstone

`sel4@nicta.com.au-devel/l4cap-haskell-0.4-patch-304`

# Acknowledgements

The seL4 project team would like to acknowledge the following people and teams (in no particular order) who have contributed to seL4 in some way. Apologies if we missed you.

seL4 is a descendant of L4 itself, and thus owes a lot to Jochen Liedtke's work on microkernels.

The L4Ka team at the University of Karlsruhe and Hermann Härtig's operating system group at TU Dresden have provided many fruitful discussions that have influenced seL4's design.

Jonathon Shapiro has also provided insightful interaction on many occasions, and his work on EROS and its successor Coyotos has also influenced seL4's design.

We would also like to thank Gerwin Klein and his *l4.verified* project team at NICTA for their insights and continual feedback on our work. David Cock deserves special mention for his work on bringing the ARM simulator into being.

Finally, we would like to thank members of the ERTOS program here at NICTA for their daily interaction and input.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
<b>2</b>	<b>seL4 Overview</b>	<b>9</b>
2.1	Introduction . . . . .	9
2.2	Capabilities, Endpoints, and System Calls . . . . .	9
2.2.1	Capability Invocation . . . . .	9
2.2.2	Synchronous Endpoints and IPC . . . . .	10
2.2.3	System Calls . . . . .	13
2.2.4	System Call Monitors . . . . .	15
2.3	Untyped Memory and Kernel Memory Allocation . . . . .	15
2.4	Threads . . . . .	16
2.5	Capability Address Spaces . . . . .	18
2.5.1	The Mapping Database . . . . .	18
2.5.2	CNode Operations . . . . .	19
2.5.3	Capability Rights . . . . .	20
2.5.4	Capability Faults . . . . .	20
2.6	Virtual Address Spaces . . . . .	21
2.6.1	Software Loaded TLBs . . . . .	21
2.6.2	Hardware Loaded TLBs . . . . .	21
2.7	Asynchronous IPC . . . . .	21
<b>3</b>	<b>Haskell Model Overview</b>	<b>23</b>
3.1	Introduction to Haskell . . . . .	23
3.2	Modules . . . . .	24
3.3	System State and Monads . . . . .	24
3.3.1	The State Monad . . . . .	24
3.3.2	Errors and Faults . . . . .	25
3.3.3	System State . . . . .	26
3.3.4	Physical Memory Model . . . . .	26
3.4	The Simulator . . . . .	27
3.4.1	Events . . . . .	27
3.4.2	Hardware Parameters . . . . .	27
3.4.3	Simulator Interface . . . . .	27

<b>4</b>	<b>Kernel API</b>	<b>28</b>
4.1	Kernel Object Types . . . . .	28
4.1.1	Object Types . . . . .	29
4.1.2	Capability Rights . . . . .	29
4.1.3	Thread Priority . . . . .	30
4.1.4	Capability References . . . . .	30
4.1.5	Message Parameters . . . . .	31
4.1.6	Capability Transfers . . . . .	31
4.1.7	Initial Address Space . . . . .	32
4.1.8	Constants . . . . .	34
4.2	Machine-Independent Object Types . . . . .	34
4.2.1	Types . . . . .	35
4.3	Kernel Object Invocations . . . . .	35
4.3.1	Invocation Type . . . . .	36
4.4	System Calls . . . . .	38
4.4.1	Types . . . . .	38
4.4.2	Handling Events . . . . .	39
4.4.3	System Calls . . . . .	40
4.4.4	Capability Invocations . . . . .	44
4.5	Faults and Errors . . . . .	45
4.5.1	Types . . . . .	45
4.5.2	Sending Failure Messages . . . . .	47
<b>5</b>	<b>Kernel Operations</b>	<b>49</b>
5.1	Capability Space Lookups . . . . .	49
5.1.1	Capability Lookups . . . . .	50
5.1.2	Locating a Capability Table Entry . . . . .	50
5.2	Threads and Scheduling . . . . .	53
5.2.1	Initial Thread Creation . . . . .	54
5.2.2	Thread Activation . . . . .	54
5.2.3	Thread State . . . . .	55
5.2.4	IPC Transfers . . . . .	55
5.2.5	Scheduling . . . . .	58
5.3	Bootstrapping the Kernel . . . . .	61
5.3.1	Kernel Initialisation . . . . .	62
5.4	Handling Faults . . . . .	68
5.4.1	Handling Faults . . . . .	68
5.4.2	Sending Fault IPC . . . . .	69
5.4.3	Double Faults . . . . .	70
5.5	Virtual Memory . . . . .	70
5.5.1	Implementation-defined Functions . . . . .	71
5.5.2	IPC Buffer Lookups . . . . .	71

<b>6</b>	<b>Kernel Objects</b>	<b>72</b>
6.1	Data Structures . . . . .	72
6.1.1	Capabilities . . . . .	73
6.1.2	Kernel Objects . . . . .	74
6.1.3	Other Types . . . . .	76
6.2	Thread Control Blocks . . . . .	79
6.2.1	TCB Invocations . . . . .	83
6.2.2	Messages . . . . .	86
6.2.3	TCB Accessors . . . . .	87
6.2.4	User-level Context . . . . .	88
6.3	Capability Nodes . . . . .	88
6.3.1	Capability Node Object Invocations . . . . .	89
6.3.2	CNode Operations . . . . .	91
6.3.3	Object Creation . . . . .	95
6.3.4	Helper Functions . . . . .	96
6.3.5	Accessing Capabilities . . . . .	97
6.3.6	Capability Transfers . . . . .	99
6.4	Synchronous Endpoints . . . . .	100
6.4.1	Sending IPC . . . . .	101
6.4.2	Receiving IPC . . . . .	103
6.4.3	The Receive Phase of SendWait . . . . .	104
6.4.4	Kernel Invocation Replies . . . . .	104
6.4.5	Cancelling IPC . . . . .	106
6.4.6	Accessing Endpoints . . . . .	107
6.5	Asynchronous Endpoints . . . . .	107
6.5.1	Sending Messages . . . . .	108
6.5.2	Receiving Messages . . . . .	109
6.5.3	Delete Operation . . . . .	110
6.5.4	Accessing Asynchronous Endpoints . . . . .	110
6.6	Untyped Objects . . . . .	111
6.6.1	Invocation . . . . .	111
6.7	Object Types and Retyping . . . . .	114
6.7.1	Creating and Deleting Capabilities . . . . .	115
6.7.2	Inspecting Capabilities . . . . .	115
6.7.3	Modifying Capabilities . . . . .	116
6.7.4	Creating and Deleting Objects . . . . .	118
6.7.5	Invoking Objects . . . . .	120
6.8	Storing Objects . . . . .	121
6.8.1	Type Class Instances . . . . .	121
<b>7</b>	<b>Haskell Model Details</b>	<b>125</b>
7.1	System State . . . . .	125
7.1.1	Types . . . . .	125
7.1.2	Kernel State Functions . . . . .	127

7.1.3	Performing Machine Operations . . . . .	127
7.1.4	Miscellaneous Monad Functions . . . . .	128
7.2	Physical Address Space Model . . . . .	128
7.2.1	Types . . . . .	129
7.2.2	Physical Address Space Initialisation . . . . .	129
7.2.3	Accessing Objects . . . . .	129
7.2.4	Creating Objects . . . . .	130
7.2.5	Deleting Objects . . . . .	131
7.2.6	Helper Functions . . . . .	131
7.3	Storable Objects . . . . .	132
7.3.1	Imported Modules . . . . .	132
7.3.2	Types . . . . .	132
7.3.3	Public Functions . . . . .	133
7.3.4	Type Classes . . . . .	133
7.3.5	Class Instances . . . . .	134
7.4	System Calls . . . . .	134
7.5	Failures . . . . .	135
7.5.1	Data Types . . . . .	136
7.5.2	Class Instances . . . . .	136
7.5.3	Failure Handling . . . . .	137
7.5.4	Detecting Failures . . . . .	138
7.6	Preemption . . . . .	138
7.6.1	Types . . . . .	139
7.6.2	Functions . . . . .	139
<b>8</b>	<b>Modelling the Hardware</b>	<b>140</b>
8.1	Words and Registers . . . . .	140
8.1.1	Types . . . . .	140
8.1.2	Monads . . . . .	142
8.1.3	Functions and Constants . . . . .	142
8.2	Hardware Functions . . . . .	144
8.2.1	Types . . . . .	144
8.2.2	Hardware Operations . . . . .	144
8.2.3	Constants . . . . .	146
<b>9</b>	<b>Implementation-Specific Features</b>	<b>147</b>
9.1	Generic VSpace Implementation . . . . .	147
9.2	ARM . . . . .	148
9.2.1	Register Set . . . . .	148
9.2.2	Hardware Interface . . . . .	149
9.2.3	API . . . . .	151
9.2.4	Kernel Objects . . . . .	151
9.2.5	Virtual Address Space . . . . .	153
9.3	Simple Haskell-based Simulator . . . . .	153

9.3.1	Register Set . . . . .	153
9.3.2	Hardware Interface . . . . .	154
9.3.3	API . . . . .	155
9.3.4	Kernel Objects . . . . .	156
9.3.5	Virtual Address Space . . . . .	156

# 1 Introduction

This manual documents the seL4 kernel reference implementation, which is written in Haskell. It provides annotated Haskell source code describing the high-level, architecture independent behaviour of the three services that the kernel provides — threads, address spaces, and communication — including the system calls used to control them from user level. This source code is executable — not pseudocode — and forms part of a simulator which can be used to examine the behaviour of a running seL4 kernel.

## 2 seL4 Overview

### 2.1 Introduction

This chapter provides an overview of the *seL4* kernel. It concentrates on the conceptual differences between seL4 and existing L4 specifications.

The seL4 kernel's design focuses on two general issues with the existing L4 API: authorisation of system calls and kernel resource management. seL4 addresses these general areas by using capabilities to authorise system calls and making kernel data structures first class objects whose allocation is performed by user-level operating system personalities.

### 2.2 Capabilities, Endpoints, and System Calls

Each user-space thread in the system possesses a set of *capabilities*. A capability provides the thread that possesses it with the ability to perform one or more kernel operations. A thread may address its capabilities using their locations in that thread's *capability address space* (henceforth abbreviated *Cspace*).

A capability contains a reference to an area of memory that is used to support the kernel services that the capability provides. It also contains data identifying the type of services provided. Capabilities are immutable and opaque from the point of view of the user threads possessing them; once a capability has been created, its contents are only visible to the kernel.

#### 2.2.1 Capability Invocation

Capabilities may be *invoked* by passing their Cspace addresses to one of the kernel's two basic operations — *Send* and *Wait*. The former is used to send a message to the object represented by the capability, and the latter to wait for a reply.

All communication channels in seL4 are one-way; under normal circumstances, a user-level thread will never perform both Send and Wait operations using a single capability. In fact, most capabilities will only allow one of these two operations, and attempting to perform the other will fail.

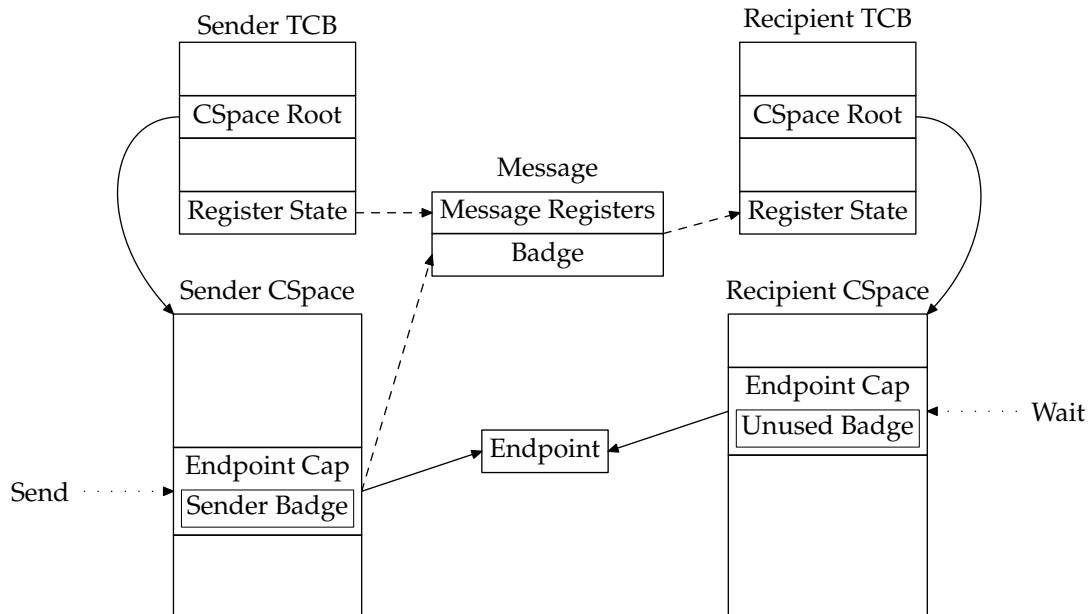


Figure 2.1: IPC operation, without message buffers. The solid lines represent references to kernel objects; the dashed lines represent the transfer of control and data between the threads.

The potential recipients or senders of a message through an invoked capability depend on the type of the kernel object that the capability refers to. For most object types, the kernel itself is the recipient of the message; however, messages may also be sent to other user-level threads.

## 2.2.2 Synchronous Endpoints and IPC

The basic inter-process communication (or *IPC*) operation is coordinated using *endpoint* objects. An endpoint is a small kernel object which contains a queue of threads, and a state flag that indicates whether the threads in the queue are waiting for *Send* operations or *Wait* operations. It acts as a one-way channel for communication between one or more senders and one or more recipients. Unlike previous versions of L4, the global identities of the participants in IPC are not exposed; threads send and receive IPC using only the local addresses of their endpoints.

Multiple threads may perform *Send* or *Wait* operations on the same endpoint. When a pair of threads invoke an endpoint, performing a *Send* operation and a *Wait* operation respectively, the kernel will perform a *message transfer* between them. Threads which cannot immediately be paired are suspended until a partner arrives. If multiple threads are suspended on one endpoint, the kernel's algorithm for choosing a thread to resume is not specified, but typically will be first in, first out order.

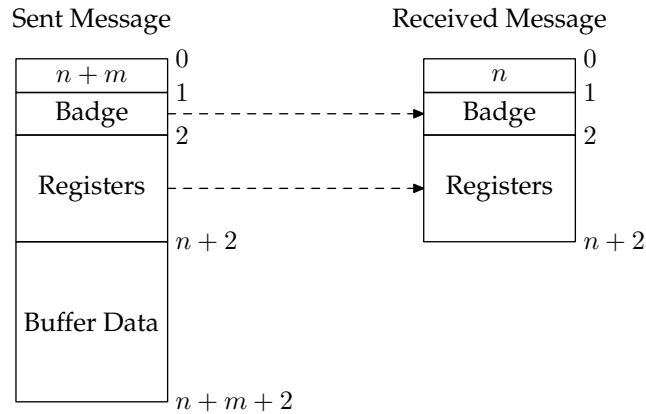


Figure 2.2: Truncation of a message when one of the IPC buffer capabilities is missing.

### Badged Messages

Since it is only possible to wait for messages from one specific endpoint at a time, a server wishing to provide multiple interfaces or to service requests from multiple clients must provide capabilities to the same endpoint for every interface or client. The kernel provides the server with a means to identify the capability used to send a message (and therefore the client or interface): *badged* endpoint capabilities.

Threads holding an endpoint capability with the right to perform the Mint operation on it may create new capabilities to that endpoint which are marked with badges. A badge is a data word with a meaning defined by the user-level server that creates it; typically it will be an index into a table of interfaces provided by the server. The badge on a capability is immutable and (mostly) opaque; it is not possible to read or modify the badge without possessing the mint right.

When a message is sent using a badged endpoint capability, the badge is passed to the recipient along with the rest of the message. See figure 2.1 on the preceding page for an example.

It is sometimes necessary for an interface to provide methods that operate on multiple objects, with privileged access by the server to each of those objects. Each object may be identified by a badged capability; however, only one of the capabilities may be invoked, so the server that receives the invocation must inspect badges the other capabilities to determine which objects they identify.

The kernel provides the *Identify* system call to allow such multiple-object invocations. To use it, a server provides two capability references: one to the server's endpoint capability, which must have the right to mint new badged capabilities; and one to the additional capability provided by the client. If the capability provided by the client is valid and refers to the server's endpoint, then the call will return the badge on the client's capability; otherwise an error code will be returned. This allows servers to

provide badged capabilities to clients which may be identified later, without allowing untrusted clients to read the badges.

## Message Transfers

Message transfers work by transferring the contents of a certain number of *message words* from the sender to the recipient. A small number of these will be transferred in the CPU's general purpose registers — the number of registers used for this purpose depends on the architecture. If there is more data to be transferred, then per-thread *message buffers* are used.

The first word of the message contains parameters that affect the transfer of the message, including the number of words after the first that will be transferred. If the kernel is unable to perform the transfer as requested, the recipient (but *not* the sender) will receive a modified version of the first word that reflects the actual transfer performed.

If the message buffer must be used to transfer the requested number of message words, and either the sender's or the recipient's message buffer capability is missing, the message will be truncated as shown in figure 2.2. Note that the sender cannot tell that this has happened, as informing it would open a covert channel from the recipient to the sender. The recipient is not explicitly notified either, though the message length specified in the first word will be reduced to reflect the amount of data actually sent. The recipient may then use protocol-specific means to detect truncation.

It is the responsibility of each of the two threads involved, and their pagers, to ensure that their message buffers are present when an IPC transfer occurs. Pagers will typically consider pages containing these buffers to be pinned, and never unmap them while a thread is running, unless that thread is expected to be able to handle the resulting truncation of all of its messages.

## Capability Transfers

A message may optionally transfer one capability possessed by the sender into the receiver's capability space. To send a capability, the sender sets a specific bit in the first message word, and places a reference to the capability, a set of capability rights, and optionally a new capability data word in its IPC buffer. To allow reception of sent capabilities, the receiver places in its IPC buffer a CNode capability reference, and an index and depth into the CNode. Together, these form a complete set of arguments to a Mint or Copy operation (see 2.5.2 on page 19). This operation is then performed to transfer the capability.

If the capability transfer is unsuccessful, the kernel clears the capability transfer bit of the first word when transferring it to the recipient. No fault messages are sent, and no indication of an error is returned to the sender.

Transferred capabilities will have the rights of the sender's capabilities, reduced by the sender's provided set of capability rights and by the rights mask of the CNode in which the receiver places them. Also, if the endpoint capability used to receive is read-only, then the received capabilities will also be read-only; this prevents one-way IPC channels being turned into two-way channels by sending a writeable capability to the read-only end of the channel.

## Non-Blocking Invocations

In some instances, a thread may need to send a message through an endpoint provided by another, untrusted, thread. In this situation, the sender must not assume that the endpoint capability is valid, or that there will ever be a partner available to receive the message. To avoid making those assumptions, the sender may set a bit in the first message word that marks the invocation as *non-blocking*.

A non-blocking invocation will not fail or fault if the invoked capability is absent, and will not suspend the sender if no partner is waiting on the endpoint. It is the untrusted recipient's responsibility to provide a valid endpoint capability and to perform a Wait operation on it before the message is sent; otherwise the message will be silently dropped. The kernel does not inform the sender that the non-blocking invocation has been dropped, as doing so would open a covert channel.

### 2.2.3 System Calls

Figure 2.3 on the following page shows a request being made of the kernel by a client thread. The client possesses a capability to a kernel object, and a designated *reply capability*, which is an endpoint capability that the client (or a system thread controlling the client) has nominated to be used for receiving replies to requests. It performs a Send operation to the kernel object, which transfers a message to the kernel along with a reference to the invoked kernel object.

After receiving a message from a client thread, the kernel determines which operation is being requested, based on the invoked object's type and the contents of the message; it then performs the operation if possible, and sends a message to the thread's *reply endpoint* giving the result of the operation. The client is expected to have performed a Wait operation on its reply capability to receive this message — the reply is treated as if the kernel performed a non-blocking Send operation, and will be dropped if neither the client nor any other thread is waiting to receive it. To allow the client to immediately perform the Wait operation after sending the request, the kernel provides an atomic Send and Wait operation called *SendWait*.

Figure 2.4 shows the procedure followed when a client requests a service provided by a user-level server thread. Rather than invoke a kernel object capability, the client invokes an endpoint which represents the service; the server thread receives the message,

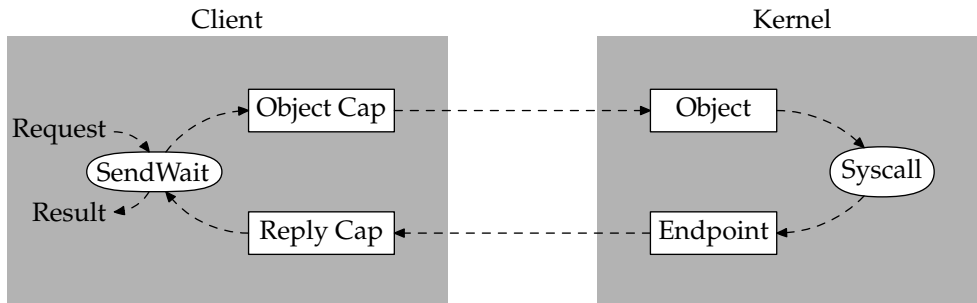


Figure 2.3: A request by a client for a kernel service. The dotted lines represent the conceptual flow of control and data during the request.

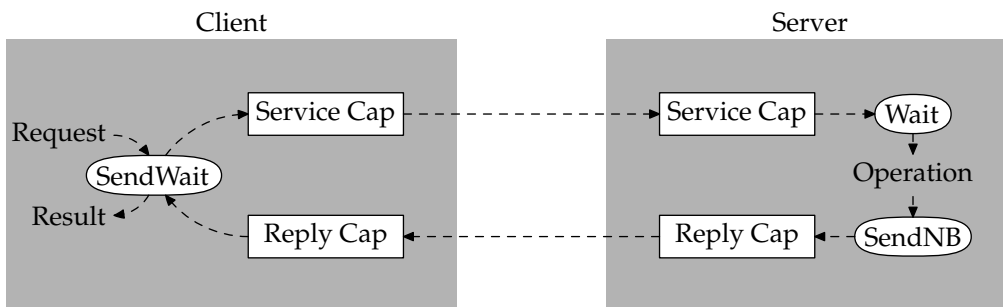


Figure 2.4: A request by a client for a service provided by a user-level server.

determines the identity of the client using the sending capability's badge, performs the operation, and sends the result to the client's reply capability. Generally, a server will possess a single capability used to receive requests, and also one reply capability for each of its clients; a client will have a single reply capability, and one capability for each object or server it is permitted to access.

## 2.2.4 System Call Monitors

Client-server communication and client-kernel communication are quite similar. In fact, if the scheduler can guarantee that the client will never be scheduled while the operation is running, the two are indistinguishable from the client's point of view. This allows kernel operations to be intercepted by a *system call monitor* thread, by replacing the client's kernel object capabilities with capabilities to an endpoint which the monitor can receive from. The monitor may then selectively deny, log or modify each system call before forwarding it to the kernel, possibly logging or forging the kernel's reply as well.

Note that capability references are sometimes passed to the kernel as system call arguments, in which case the monitor will need to replace the references with valid ones in the monitor's address space.

## 2.3 Untyped Memory and Kernel Memory Allocation

The kernel memory management issues with existing L4 kernels are addressed in seL4 by managing *all* dynamic memory allocation at user level. The kernel never allocates memory after user-space code begins running.

The CSpace of the initial user-space thread includes, among other things, a set of capabilities to *untyped memory objects* representing all of the system's unallocated physical memory. Initially most of these capabilities represent large regions of memory — they may represent any power of two size, up to the size of the entire unallocated region. The initial thread is free to allocate these regions for specific purposes, or to hand out the capabilities to other user-space threads that it creates.

To allocate objects in an untyped memory region, the user-level thread that possesses it must send a message to it, specifying the type of object to create, the size of the created objects (for object types with variable sizes), and a capability reference to an area of a CSpace that will be used to store capabilities to the newly allocated objects. This operation is called *Retype*. *Retype*'s effect on CSpace is to fill a contiguous empty CSpace region specified by the caller with new capabilities. If the operation is successful, the kernel returns the number of capabilities created. The construction of CSpace is discussed further in section 2.5 on page 18.

Allocated objects may be *deactivated* by the *Revoke* or *Delete* operations (discussed in section 2.5.2 on page 19), which delete capabilities. If either of these operations causes all existing capabilities to a specific object to be deleted, then the kernel will abort any operations involving that object and remove all references to it as a typed object. It is then no longer possible to use that object, until the memory is reused to create a new object.

If the kernel were to allow a region of memory to be simultaneously used by two different objects, it might be possible for a malicious or buggy thread to gain direct access to the contents of a kernel data structure. Therefore, the *Retype* operation will fail with a `REVOKEFIRST` error if there are already existing objects in the region. The caller should call *Revoke* on the Untyped object's capability first, if there is any possibility that it already contains other objects.

Each *Retype* operation creates kernel objects of one of the possible types. The types present in all implementations are:

**Untyped memory** objects, which can be created with a *Retype* operation in order to split up a large memory region;

**Virtual memory** objects, which can be mapped in the virtual address space (using some implementation-specific process as described in section 2.6 on page 21);

**Thread Control Block** objects, which each provide the resources needed for a single user-level thread (see section 2.4);

**CNode** objects, which are nodes in the guarded page tables used to translate CSpace addresses (see section 2.5 on page 18);

**Endpoint** objects, which contain queues for synchronous IPC (as described in section 2.2.2 on page 10); and

**Asynchronous Endpoint** objects, which contain message buffers for asynchronous IPC (see section 2.7 on page 21).

There may also be implementation-specific object types. The untyped memory, virtual memory and CNode objects may be of variable size (limited to powers of two), though the minimum and maximum sizes are specific to each implementation and type. Other objects are of fixed, implementation-specific size.

## 2.4 Threads

Unlike previous versions of L4, threads in seL4 have no global identifier visible outside the kernel. A thread is identified only by the CSpace address of a capability that refers to its *thread control block*, or *TCB*. A thread may or may not possess a capability to its own TCB; in a typical system, most threads would not.

There are five operations provided by TCB objects, which can be used to manipulate the state of the corresponding thread:

**ThreadControl** allows user-space threads to configure various parameters of a thread — including the root of the thread’s address space, the thread’s scheduler priority, and the endpoint capabilities used by the kernel to send system call replies and fault messages generated by the thread. This is similar in purpose to the system call of the same name in earlier versions of L4.

**ExchangeRegisters** has changed significantly, in comparison to previous versions of L4. In particular, it can now copy or set the entire user-level register state of a nominated thread, rather than only the instruction and stack pointers; the caller can select implementation-specific subsets of the register state to copy. Both source and destination threads may be explicitly specified and may be different to the thread performing the system call.

Since the kernel has direct access to both threads’ user-level contexts and save areas, it can perform this operation with a minimal number of copies of the register set — possibly zero copies, for floating-point and vector register sets which can simply remain untouched by the kernel as it switches from the source thread to the destination.

The new ExchangeRegisters call is intended to allow servers to efficiently save, duplicate or restore the entire register state of a client task — or the subset of it that is in use — without requiring code and shared memory that resides in the client’s address space.

**Resume** will resume execution of a thread that is inactive or waiting for a kernel operation to complete. If the invoked thread is waiting for a kernel operation, Resume will modify the thread’s state so that it will attempt to perform the faulting or aborted operation again. Resuming a thread that is already ready has no effect. Resuming a thread that is in the waiting phase of a SendWait operation may cause the sending phase to be performed again, even if it has previously succeeded.

This system call can resume execution of threads which have encountered a virtual memory fault, and is presently the only mechanism for doing that; it may be replaced in future by a more efficient mechanism.

Note that ExchangeRegisters may perform this operation on its destination thread if requested to do so. In fact, there is no separate Resume operation; it is really an ExchangeRegisters operation that transfers no registers.

**Suspend** will make a thread inactive. The thread will not be scheduled again until a Resume operation is performed on it.

Like Resume, this operation may be performed by ExchangeRegisters, though on the source thread rather than the destination. There is no separate Suspend operation; it is an ExchangeRegisters operation that does not transfer registers.

**YieldTo** donates the caller's remaining timeslice to the invoked thread and then switches to the invoked thread if it is ready to run.

## 2.5 Capability Address Spaces

The capability space is represented by a *guarded capability table*, which is similar to a guarded page table [4]. The nodes of the table are individual kernel objects allocated using the `Retype` system call, and are called *capability nodes*, abbreviated *CNodes*.

Each CNode has a power-of-two number of *slots*; the number is determined at the time the CNode is created. Each slot contains a *capability table entry*, which can store a single capability; the contents of the entry are opaque to user-space code. The format of entries is the same at all levels of the table. There is also a slot in each TCB, which is used to store a capability to the root CNode.

The capability table is constructed by copying a capability to the root CNode into the TCB's root slot, capabilities to lower-level CNodes into the root CNode's slots, and so on. The number of address bits resolved by each CNode depends on the node's size and on the number of *guard bits* of the CNode. A capability will appear in the user thread's CSpace when it can be reached by translating address bits starting at the page table root capability.

Note that all CNodes are required to resolve at least one address bit, to prevent infinite loops in the capability lookup code.

### 2.5.1 The Mapping Database

The kernel keeps a record of the *derivation tree* for capabilities. This tree is stored in the *mapping database*, and is used when revoking a capability to locate the capabilities which must be deleted. This is similar to previous L4 kernels, except that the storage for the mapping database is not dynamically allocated in seL4 — every capability table slot includes some space reserved for a mapping database node.

For implementation reasons, the kernel imposes limitations on the structure of the derivation tree. In particular, the tree may not have unlimited depth. The exact nature of these limitations is implementation-defined.

The mapping database is optional. With some minor modifications to the API, it can be removed from the kernel entirely; this halves the size of a CNode slot, and therefore considerably reduces kernel resource usage. However, it also prevents the kernel from implementing the `Revoke` operation, and makes multiple uses of the `Retype` operation on a single memory region unsafe. Therefore its removal is only appropriate for systems that never need to revoke or reuse system resources, or that completely trust any user-level thread that has access to untyped memory.

This document describes the API for kernels that have a mapping database.

## 2.5.2 CNode Operations

User-level tasks possessing a capability to a CNode may perform any of five operations on the capabilities stored in the tree of which that CNode is the root. Each operation modifies at least one capability slot in the tree; the caller must specify an address within the tree and the number of significant bits in that address.

**Mint** creates a new capability in a specified CNode slot, given a reference to a slot containing an existing capability. The newly created capability may differ from the original — it may have fewer rights, and its type-specific data (such as the IPC badge on an endpoint capability) may be changed — but it will always refer to the same kernel object.

If the source capability is an endpoint, it must not have previously had new capability data set; otherwise, use of this operation will not be permitted.

**Copy** is similar to Mint, but does not change the capability's type-specific data.

**Move** moves a capability between two specified capability slots. It cannot explicitly change the rights on the capability, but they may be decreased by any masks on CNodes used to look up the source capability. It may also change the badge on the capability, if it possesses the Mint right.

**Delete** will remove a capability from the specified slot.

**Revoke** is equivalent to calling *Delete* on each derivation tree child of the specified capability. It has no effect on the capability itself.

Delete and Revoke are potentially very long-running operations, especially if:

- a capability is revoked while it has a large number of children in the derivation tree, or
- the last capability to a CNode is deleted (either by a direct Delete call, or a Revoke call on the Untyped capability controlling the CNode's storage), while that CNode still contains valid capabilities.

Systems with real-time constraints should avoid using Revoke or Delete unless they can ensure that only a small number of capabilities will be deleted. Implementations of the seL4 API may include preemption points in this operation if possible; however, the specification does not require them.

Copy, Mint, Move and Retype operations require their destination slot or slots to be empty; that is, they must not contain valid capabilities. If any of these operations finds a valid capability in a destination slot, it will fail with a `DELETEFIRST` error. As the error

name suggests, the caller should call Delete on destination slots first, if there is any possibility that they might contain valid capabilities.

### 2.5.3 Capability Rights

The set of operations which may be performed on a given capability are determined by the type of the object that the capability refers to, and a set of *rights* specific to the capability. The rights are a small set of bits corresponding to classes of operation:

**Write** allows data to be sent or written to the capability's object. For example, an endpoint capability must have this right if it is to be used to send messages.

**Read** allows data to be received or read from the object.

**Mint** allows creation of new capabilities that refer to the same object but have different properties. For example, to create endpoint capabilities with different badges. It also allows the use of the *Identify* operation to read the badges of other capabilities to the same object.

Newly created capabilities have all rights. They may be reduced when performing Copy, Mint or Move operations, either via a CSpace invocation or a capability transfer IPC.

When a capability is looked up in a CSpace, the effective set of rights is equal to the intersection of its own set and those of all CNode capabilities used in the lookup. This allows rights to be removed from capabilities or pages in a large region of the address space with only one operation.

### 2.5.4 Capability Faults

Whenever a user-level task encounters an error that it cannot handle itself — for example, an access to a missing capability — the kernel will suspend it and send a *fault message* to the thread's *fault handler endpoint*. The latter is a capability reference stored in the thread's TCB, and can be set or changed using the ThreadControl system call. If the kernel is unable to queue or send the fault message, the thread will remain suspended indefinitely, with no other notification sent by the kernel.

The fault message contains information about the nature and cause of the fault, sufficient for the recipient of the fault message to recover from the fault if possible. After taking any necessary action to prevent the fault occurring again — for example, inserting a capability into the faulting thread's CSpace — the recipient may perform a Resume call on the TCB of the faulting thread, which will allow that thread to retry the operation that caused the fault.

## 2.6 Virtual Address Spaces

Many modern hardware architectures dictate specific page table formats to be used by the kernel to implement virtual memory. These usually require the dynamic allocation of memory to be used for page tables; since the seL4 kernel never dynamically allocates its own memory, the page tables must therefore be exposed to user level somehow.

The kernel's interface for virtual memory management is implementation-defined, but will generally fit into one of two categories: architectures with hardware-loaded TLBs and hardware-defined page table structures, and those with software-loaded TLBs.

### 2.6.1 Software Loaded TLBs

On architectures with software-loaded TLBs, the kernel simply uses a structure similar to that of CSpace, constructed from CNodes, to implement the virtual address space. There is a separate root node in each TCB — though it may contain a capability to a CNode that is also present in the CSpace, if the system requires CSpace and VSpace addresses to be consistent.

### 2.6.2 Hardware Loaded TLBs

On architectures with hardware-defined page table structures, such as ARM and x86, a separate address space is used in parallel with CSpace. This address space is known as *VSpace*, an abbreviation of *virtual address space*.

The details of the VSpace interface are architecture-dependent. Typically they include one or more new object types, from which a page table can be constructed in a manner similar to the construction of CSpace. Messages sent to the new objects are used to request operations similar to those provided by CNode objects. As the source parameters of such operations are always capabilities (in the CSpace), it is not possible to copy mappings between different locations in VSpace without possessing an appropriate capability.

## 2.7 Asynchronous IPC

In some situations, it is appropriate to provide an asynchronous notification of an event. The kernel provides an *asynchronous IPC* operation for this purpose. It is implemented using *asynchronous endpoint* objects, which contain a fixed size buffer of two machine words – identifier and message word. Each asynchronous endpoint capability is marked with a badge, which is set when the capability is created. A thread may

perform *Send* or *Wait* operations on an asynchronous endpoint, by invoking the corresponding capability.

When a *Send* operation is performed, the badge of the invoked capability and the user specified message are bitwise ORed with the corresponding values in the endpoint to produce its new contents. This operation is guaranteed not to block a sufficiently authorised sender, even if there is no thread waiting to receive the message. The *Wait* operation, on the other hand, is blocking — if the buffer is empty, the thread is blocked until a message arrives. However, in case of a non-empty buffer, the *Wait* operation will read the contents of the buffer, reset it to zero, and returns the value immediately.

Similar to synchronous endpoints, multiple threads may perform *Send* or *Wait* operations on the same asynchronous endpoint. When there are multiple messages sent to an endpoint before any thread attempts to receive from it, the identifier and the message delivered to the receiver are the bitwise OR of capability badges used by the senders and the messages sent by them, respectively. It is the responsibility of the user level tasks to define a suitable protocol for identifying the individual event. When multiple receivers are waiting on the same endpoint when a message is sent to it, the kernel chooses one of them to deliver the message. The algorithm for choosing a thread is not specified, but typically will be first in, first out order.

## 3 Haskell Model Overview

This chapter describes the construction of the Haskell model presented in the following chapters. It is not necessary to read this chapter to understand the seL4 API, but it will be helpful for those who wish to read the Haskell code.

### 3.1 Introduction to Haskell

Haskell is a general purpose functional programming language [6]. It is in widespread use in the research and education communities; several universities use it in introductory programming courses.

The features of Haskell that are most relevant to this project 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 semantics [1]. 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 [7].
- By using 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 section 3.3.

A more detailed introduction to Haskell is outside the scope of this document. Please refer to the online Haskell tutorials [2,5] for more information.

## 3.2 Modules

The Haskell model is split into a hierarchy of modules. At the top level is `module SEL4`, which contains no real code, but exports the entire external interface of the kernel model. Below this are five modules that serve to separate the high-level and low-level parts of the model; each of these corresponds to one of the following five chapters in this report.

The five modules have the following purposes:

**API** defines the interface between the kernel and the user-level threads running under it (other than the parts relating to specific kernel objects). See chapter 4 on page 28.

**Kernel** defines internal procedures of the kernel that do not relate to a specific kernel object — most importantly, the scheduler and the capability space lookup functions. See chapter 5 on page 49.

**Object** defines the first-class kernel objects exposed by the seL4 API, and the operations that may be performed on them. See chapter 6 on page 72.

**Model** contains the implementation of the low-level parts of the Haskell kernel — the physical memory model and the kernel state monads. See chapter 7 on page 125.

**Machine** contains the kernel's interface to the underlying hardware — specifically the register set, the kernel's uses for each register, the word type, and the virtual page size. See chapter 8 on page 140.

Each of these modules is further divided into smaller modules implementing separate functional areas. For example, `module SEL4.KERNEL` contains a module implementing capability space lookups, `module SEL4.KERNEL.CSPACE`.

## 3.3 System State and Monads

### 3.3.1 The 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<sup>1</sup>. 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

---

<sup>1</sup>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.

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 [8], 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
    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. Monadic style has been used extensively in this specification. Nearly all of the functions in the model are in `KERNELMONAD`, which is an application of the `STATE` monad to the `KERNELSTATE` type. Both are defined in section 7.1 on page 125. The `KERNELSTATE` type is described briefly in section 3.3.3 on the following page.

### 3.3.2 Errors and Faults

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 expressions in the sequence 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 positive'
    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`.

The Haskell kernel model makes use of the `ERRORT` transformer to abort operations that cannot complete successfully. The first type parameter of `ERRORT` is a type that represents the error; there are several such types defined in `sel4`, in section 4.5 on page 45.

### 3.3.3 System State

The state of the simulated kernel is stored in values of type `KERNELSTATE`. Most functions in the model are in the monad `KERNELMONAD`, and are therefore state transformation functions that operate on a value of type `KERNELSTATE`.

The state data structure contains a value of type `PSPACE`, which is used to store data which is dynamically allocated by calls from user level. See section 3.3.4 for details.

The other values present in the `KERNELSTATE` structure are global kernel data, such as a pointer to the current thread.

The `KERNELSTATE` value is added to `KERNELMONAD` by a monad transformer, `STATET`. This adds the state to an existing monad, the *machine monad*, the type of which depends on the machine being simulated. See section 3.4.3 on the next page.

### 3.3.4 Physical Memory Model

The simulated kernel's physical memory is modelled by the Haskell type `PSPACE`, defined in section 7.2 on page 128. The contents of each address in this space are typed; that is, the model keeps a record of whether they are allocated, and whether they contain either integer data, or a specific type of kernel object. Accesses to objects of the wrong type, or with an incorrectly aligned address, are detected by the model and disallowed.

For simulator performance reasons, and to make the operation of the virtual memory related parts of the model more realistic, the `PSpace` contains only typed kernel objects; it does not contain data accessible to user level. Instead, regions occupied by such data are represented by a `USERDATA` object, which has no contents. The actual data is stored separately, and is accessed via the machine monad.

## 3.4 The Simulator

The kernel model functions as one half of a complete system model. The other half is the CPU simulator, responsible for modelling the execution of user-level programs and the operation of hardware devices.

### 3.4.1 Events

When the state of the simulated system changes in a way that the kernel must respond to, the simulator sends the kernel an *event*. These events correspond to situations that would cause kernel code to execute on a real system: specifically hardware exceptions and interrupts, and user-level software traps.

The set of events supported by the Haskell model is defined by the type `EVENT`, found in section 4.1 on the next page.

### 3.4.2 Hardware Parameters

Several aspects of the kernel's API and behaviour vary depending on the architecture of the host machine. The model contains abstract interfaces to the components that change; the interfaces are implemented in separate modules, selected using the preprocessor.

### 3.4.3 Simulator Interface

The state of the simulated machine is accessed via a monad, the type of which depends on which CPU simulator is in use — again, selected using the preprocessor. For example, for a simulator implemented entirely in Haskell, this monad is likely to be `STATE`, with a data structure containing the machine's state data. If the simulator is implemented externally, and communicates with the kernel via the Haskell foreign function interface, then the machine monad will be `I0` (or some transformation of `I0`), allowing functions in it to call out to the external simulator.

## 4 Kernel API

This chapter documents the parts of the kernel that interact directly with user-level code.

### 4.1 Kernel Object Types

This module specifies the user-level interface to the various object types defined by the kernel. Parts of this interface are architecture-specific; they are defined in the other modules in the `SEL4.API.TYPES` branch of the module hierarchy.

We use the C preprocessor to select a target architecture. Also, this file makes use of the GHC extension allowing derivation of arbitrary type classes for types defined with `newtype`, so GHC language extensions are enabled.

```
{-# OPTIONS_GHC -cpp -fglasgow-exts #-}
```

```
module SEL4.API.TYPES (
    module SEL4.API.TYPES,
    module SEL4.API.TYPES.UNIVERSAL,
) where
```

```
import SEL4.MACHINE
```

```
import DATA.BITS
import DATA.WORD(WORD8)
```

The architecture-specific definitions are imported qualified with the `ARCH` prefix.

```
import qualified SEL4.API.TYPES.TARGET as ARCH
```

The types and behaviours that are applicable to all architectures are defined in the `UNIVERSAL` module. Some of this module's definitions are used by the architecture-specific modules, and are hidden here to avoid confusion.

```
import SEL4.API.TYPES.UNIVERSAL hiding (OBJECTTYPE, fromAPIType, toAPIType)
```

### 4.1.1 Object Types

User-allocated memory can contain objects of several kernel-defined types, or be untyped. The set of defined types is partly platform-specific — it includes some universal types, and possibly some additional platform-defined types. The following are type aliases for the platform-specific enumeration of all valid object types, and the enumeration of all universally available object types, respectively.

```
type OBJECTTYPE = ARCH.OBJECTTYPE
```

The following functions are used to convert between the above two types.

```
fromAPIType :: APIOBJECTTYPE → OBJECTTYPE
```

```
fromAPIType = ARCH.fromAPIType
```

```
toAPIType :: OBJECTTYPE → MAYBE APIOBJECTTYPE
```

```
toAPIType = ARCH.toAPIType
```

### 4.1.2 Capability Rights

This is a set of boolean values that specifies the operations that may be performed using a capability.

```
data CAPRIGHTS = CAPRIGHTS {
```

The rights are:

- the right to write or send data to an object, and to retain this right on received capabilities;

```
capAllowWrite,
```

- the right to read or receive data from an object;

```
capAllowRead,
```

- and the right to send capabilities via IPC.

```
capAllowGrant :: BOOL }
```

```
deriving (SHOW, EQ)
```

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

```
allRights :: CAPRIGHTS
allRights = CAPRIGHTS TRUE TRUE TRUE
```

```
noRights :: CAPRIGHTS
noRights = CAPRIGHTS FALSE FALSE FALSE
```

The following function finds the intersection of two sets of capability rights.

```
andCapRights :: CAPRIGHTS → CAPRIGHTS → CAPRIGHTS
andCapRights (CAPRIGHTS  $a_1$   $a_2$   $a_3$ ) (CAPRIGHTS  $b_1$   $b_2$   $b_3$ ) =
  CAPRIGHTS ( $a_1 \wedge b_1$ ) ( $a_2 \wedge b_2$ ) ( $a_3 \wedge b_3$ )
```

A set of capability rights may be converted to or from a machine word.

```
rightsFromWord :: WORD → CAPRIGHTS
rightsFromWord  $p$  =
  CAPRIGHTS ( $p$  `testBit` 0) ( $p$  `testBit` 1) ( $p$  `testBit` 2)
```

```
wordFromRights :: CAPRIGHTS → WORD
wordFromRights (CAPRIGHTS  $r_1$   $r_2$   $r_3$ ) =
  ( $bitIf$   $r_1$  0) .—. ( $bitIf$   $r_2$  1) .—. ( $bitIf$   $r_3$  2)
  where  $bitIf$   $b$   $n$  = if  $b$  then  $bit$   $n$  else 0
```

### 4.1.3 Thread Priority

The priority of a thread is represented by an 8-bit unsigned integer.

```
type PRIORITY = WORD8
```

### 4.1.4 Capability References

The type CPTR is a reference to a capability in a user-level thread's capability space; that is, a *capability pointer*.

```
newtype CPTR = CPTR { fromCPtr :: WORD }
  deriving (SHOW, EQ, NUM, BITS, ORD)
```

### 4.1.5 Message Parameters

The first word of a message contains information about the contents and type of the message, which must be interpreted (and possibly modified) by the kernel.

```
data MESSAGEINFO = MI {
    msgLength :: WORD,
    msgCapTransfer :: BOOL,
    msgBlockingSend :: BOOL,
    msgLabel :: WORD }
deriving SHOW

messageInfoFromWord :: WORD → MESSAGEINFO
messageInfoFromWord w = MI {
    msgLength = w .&. (bit 8 - 1),
    msgCapTransfer = w `testBit` 8,
    msgBlockingSend = w `testBit` 9,
    msgLabel = w `shiftR` 10 }

wordFromMessageInfo :: MESSAGEINFO → WORD
wordFromMessageInfo mi = label .—. blocking .—. capTransfer .—. len
    where
        len = msgLength mi
        capTransfer = if msgCapTransfer mi then bit 8 else 0
        blocking = if msgBlockingSend mi then bit 9 else 0
        label = msgLabel mi `shiftL` 10
```

### 4.1.6 Capability Transfers

Each thread has an IPC buffer, which contains message data that does not fit in the available registers of the host architecture. It also contains information about the source or destination of an IPC capability transfer, defined by the following structure.

```
data CAPTRANSFER = CT {
    ctSendCNode :: CPTR,
    ctSendIndex :: CPTR,
    ctSendDepth :: INT,
    ctSendMask :: CAPRIGHTS,
    ctSendData :: MAYBE WORD,
    ctReceiveCNode :: CPTR,
    ctReceiveIndex :: CPTR,
    ctReceiveDepth :: INT }

capTransferDataSize :: WORD
```

```

capTransferDataSize = 8

capTransferFromWords :: [WORD] → CAPTRANSFER
capTransferFromWords words = CT {
    ctSendCNode = CPTR $ words !! 0,
    ctSendIndex = CPTR $ words !! 1,
    ctSendDepth = fromIntegral $ words !! 2,
    ctSendMask = rightsFromWord $ words !! 3,
    ctSendData = if (words !! 3) `testBit` 8
        then JUST $ words !! 4 else NOTHING,
    ctReceiveCNode = CPTR $ words !! 5,
    ctReceiveIndex = CPTR $ words !! 6,
    ctReceiveDepth = fromIntegral $ words !! 7 }

```

### 4.1.7 Initial Address Space

The following structures are used by the kernel to communicate the initial state of the system to the root task.

The top-level structure is `BOOTINFO`, which contains an IPC buffer pointer, information about the initial untyped capabilities, and an array of virtual address space regions.

```

data BOOTINFO = BOOTINFO {
    biIPCBuffer :: VPTR,
    biFirstSmallUntyped :: CPTR,
    biSmallUntypedCount :: WORD,
    biFirstLargeUntyped :: CPTR,
    biLargeUntypedSizes :: WORD,
    ---_insert (length biRegions) here
    biRegions :: [BOOTREGION] }
deriving SHOW

wordsFromBootInfo :: BOOTINFO → [WORD]
wordsFromBootInfo bi = [
    fromVPtr $ biIPCBuffer bi,
    fromCPtr $ biFirstSmallUntyped bi,
    biSmallUntypedCount bi,
    fromCPtr $ biFirstLargeUntyped bi,
    biLargeUntypedSizes bi,
    fromIntegral $ length $ biRegions bi ]
++ (concat $ map wordsFromBootRegion $ biRegions bi)

```

Each region descriptor has start and end pointers (with the latter pointing to the last address in the region, rather than the first address after it), a type, and some data whose use depends on the type.

```

data BOOTREGION = BOOTREGION {
    brBase :: VPTR,
    brEnd  :: VPTR,
    brType :: BOOTREGIONTYPE,
    brData :: WORD }
deriving SHOW

wordsFromBootRegion :: BOOTREGION → [WORD]
wordsFromBootRegion br = [
    fromVPtr $ brBase br,
    fromVPtr $ brEnd  br,
    fromIntegral $ fromEnum $ brType br,
    brData br ]

```

The possible region types are:

```
data BOOTREGIONTYPE
```

- an empty region that may be used for capabilities or virtual memory;
 

```
      = BREMPTY
```
- a region containing some of the root task's initial capabilities, backed by exactly one CNode and for which the data word points to the first empty capability slot in the CNode;
 

```
      | BRINITCAPS
```
- a region containing the root task's mapped text or data, backed by one or more page-sized frames;
 

```
      | BRROOTTASK
```
- a memory-mapped device, for which the region data word points to an implementation-defined device descriptor;
 

```
      | BRDEVICE
```
- or a region that cannot be used for virtual memory, but is still usable for capabilities. This may be, for example, because the kernel maps its own data in the region, or because the hardware MMU does not implement addressing of the region. Note that regions of this type may overlap BRINITCAPS regions.
 

```
      | BRCAPSONLY
```

```
deriving (SHOW, ENUM)
```

### 4.1.8 Constants

A number of frames at the bottom end of physical memory are assumed to be reserved by the kernel, and used for code and static data. The following constant is the number of frames reserved. Note that it should be at least 1, to be certain that `nullPointer` is an invalid pointer.

```
kernelTop :: WORD
kernelTop = 1
```

The maximum number of message registers transferred between threads by an IPC operation. The kernel object interface requires that this number is at least ten. One or more of these message registers will be implemented using general-purpose integer registers, depending on the architecture; the rest are placed in a region of data memory called the *IPC buffer*. Note that the first register is used to store the number of registers to be transferred.

```
numberOfMRs :: WORD
numberOfMRs = 32
```

The number of timer interrupts between scheduling and preemption.

```
timeSlice :: INT
timeSlice = 5
```

## 4.2 Machine-Independent Object Types

This module defines the set of kernel object types that are available on all implementations.

```
module SEL4.API.TYPES.UNIVERSAL where

import SEL4.MACHINE

import DATA.BITS
import DATA.WORD(WORD8)
```

## 4.2.1 Types

### Object Types

The following is the definition of the five object types that are always available, as well as untyped memory. This enumeration may be extended on some platforms to add platform-specific object types.

```
data APIOBJECTTYPE
  = UNTYPED
  | INTDATAOBJECT
  | TCBOBJECT
  | ENDPOINTOBJECT
  | ASYNCENDPOINTOBJECT
  | CAPTABLEOBJECT
  deriving (ENUM, BOUNDED, Eq, SHOW)
```

```
type OBJECTTYPE = APIOBJECTTYPE
```

The following functions convert between the set of platform-independent object types and the set of all object types. If this module is used as the platform definition, the above object types are the only ones available. Therefore the conversion functions are no-ops.

```
fromAPIType = id
```

```
toAPIType = JUST
```

## 4.3 Kernel Object Invocations

This module defines the interfaces presented to clients by the kernel's objects.

We use the C preprocessor to select a target architecture.

```
{-# OPTIONS_GHC -cpp #-}
```

```
module SEL4.API.INVOCATION where
```

```
import SEL4.MACHINE
```

```
import SEL4.API.TYPES
```

```
import SEL4.OBJECT.STRUCTURES
```

The architecture-specific definitions are imported qualified with the ARCH prefix.

```
import qualified SEL4.OBJECT.OBJECTTYPE.TARGET as ARCH
```

### 4.3.1 Invocation Type

The following type can specify any kernel object invocation. It contains physical pointers to any kernel objects required for the operation, and other arguments decoded from the message registers.

```
data INVOCATION
  = INVOKEUNTYPED UNTYPEDINVOCATION
  | INVOKEENDPOINT (PPTR ENDPOINT) WORD BOOL
  | INVOKEASYNCENDPOINT (PPTR ASYNCENDPOINT) WORD WORD
  | INVOKETCB TCBINVOCATION
  | INVOKECNODE CNODEINVOCATION
  | INVOKEARCHOBJECT ARCH.INVOCATION
```

### TCB Object Invocations

The following data type defines the set of possible TCB invocation operations. The operations are discussed and defined in more detail in section 6.2 on page 79.

All operations modify the state of the invoked TCB. The pointer to this TCB is not stored in the TCBINVOCATION type. See the INVOKETCB constructor for INVOCATION, above.

```
data TCBINVOCATION
  = EXCHANGEREGISTERS {
    exRegsTarget :: PPTR TCB,
    exRegsSource :: MAYBE (PPTR TCB),
    exRegsSuspendSource, exRegsResumeTarget :: BOOL,
    exRegsSourceFrame, exRegsTargetFrame :: MAYBE (PPTR WORD),
    exRegsTransferFrame, exRegsTransferInteger,
    exRegsTransferExtra1, exRegsTransferExtra2 :: BOOL }
  | THREADCONTROL {
    tcThread :: PPTR TCB,
    tcNewFaultEP, tcNewResultEP :: MAYBE CPTR,
    tcNewPriority :: MAYBE PRIORITY,
    tcNewCRoot, tcNewVRoot :: MAYBE (CAPABILITY, PPTR CTE),
    tcNewIPCBuffer :: MAYBE VPTR }
  | YIELDTO (PPTR TCB)
```

The following function decodes a `TCBINVOCATION` structure from the label field of the invocation's first message word.

## CNode Invocations

The following data type defines the set of possible CNode invocation operations. The operations are discussed and defined in more detail in section 6.3 on page 88.

All operations modify the contents of a capability slot; the pointer to this slot is not stored in the `CNODEINVOCATION` type. See the `INVOKECNODE` constructor for `INVOCATION`, above.

The following data type defines the set of possible CNode invocation operations. The operations are discussed in more detail below.

```
data CNODEINVOCATION
  = REVOKE { targetSlot :: PPTR CTE }
  | DELETE { targetSlot :: PPTR CTE }
  | INSERT {
      insertCap :: CAPABILITY,
      sourceSlot, targetSlot :: PPTR CTE }
  | MOVE {
      moveCap :: CAPABILITY,
      sourceSlot, targetSlot :: PPTR CTE }
```

## Untyped Invocations

The following data type defines the parameters expected for invocations of Untyped objects.

```
data UNTYPEDINVOCATION
  = RETYPE {
      retypeSource :: PPTR CTE,
      retypeRegionBase :: PPTR (),
      retypeRegionSizeBits :: INT,
      retypeNewType :: OBJECTTYPE,
      retypeNewSizeBits :: INT,
      retypeSlots :: [PPTR CTE] }
```

## 4.4 System Calls

This module contains the top-level parts of the kernel model: the system call interface, and the interrupt and fault handlers. It exports the kernel entry points used by the simulator.

The system call interface is defined by functions in this module; specifically by `handleEvent`. This interface is distinct from the interface to any specific type of kernel object; the operations that may be performed on those objects are defined in their respective modules.

```
module SEL4.API.SYSCALL(EVENT(..), SYSCALL(..), handleEvent) where
```

```
import SEL4.API.TYPES
import SEL4.API.FAILURES
import {—# SOURCE #—} SEL4.KERNEL.THREAD
import SEL4.KERNEL.CSPACE
import SEL4.KERNEL.VSPACE
import SEL4.KERNEL.FAULTHANDLER
import SEL4.OBJECT
import SEL4.MODEL
import SEL4.MACHINE

import DATA.ARRAY
import DATA.BITS
```

### 4.4.1 Types

#### Events

The kernel model works by processing events caused by sources outside the kernel — either user-level code or hardware devices. The following type defines the events that the kernel can respond to. Other than `TIMERINTERRUPT`, they all include additional information about the nature of the event.

```
data EVENT
  = SYSCALLEVENT SYSCALL
  | UNKNOWNSYSCALL INT
  | USERLEVELFAULT INT
  | TIMERINTERRUPT
  | VMFAULTEVENT VPTR BOOL
```

## System Calls

The `SYSCALLEVENT` constructor defined above requires an additional value which specifies the system call that was made. This value is of the enumerated type `SYSCALL`:

```
data SYSCALL
  = SYSSEND
  | SYSWAIT
  | SYSSENDWAIT
  | SYSIDENTIFY
  | SYSYIELD
  deriving (ENUM, BOUNDED, Eq)
```

### 4.4.2 Handling Events

The `handleEvent` function determines the type of event, checks that any user-supplied inputs are correct, and then calls internal kernel functions to perform the appropriate actions. The parameter is the event being handled.

```
handleEvent :: EVENT → KERNELP ()
```

### System Call Events

System call events are dispatched here to the appropriate system call handlers, defined in the next section.

```
handleEvent (SYSCALLEVENT call) = case call of
  SYSSEND → handleSend
  SYSWAIT → withoutPreemption handleWait
  SYSSENDWAIT → handleSendWait
  SYSIDENTIFY → withoutPreemption handleIdentify
  SYSYIELD → withoutPreemption handleYield
```

### Timer Interrupts

If the event is a timer interrupt, decrement the current thread's remaining time. If it reaches zero, the `schedule` call in `callKernel` will switch threads.

```
handleEvent TIMERINTERRUPT = withoutPreemption $ do
  thread ← getCurThread
  ts ← threadGet tcbTimeSlice thread
  let ts' = ts - 1
  threadSet (λt → t {tcbTimeSlice=ts'}) thread
```

## Unknown System Calls

An unknown system call raises an `UNKNOWN_SYSCALL_EXCEPTION`, which reports the system call number to the thread's fault handler. This may allow the fault handler to emulate system call interfaces other than `seL4`.

```
handleEvent (UNKNOWN_SYSCALL n) = withoutPreemption $ do
  thread ← getCurThread
  handleFault thread $
    UNKNOWN_SYSCALL_EXCEPTION $ fromIntegral n
  return ()
```

## Miscellaneous User-level Faults

The `USER_LEVEL_FAULT` event represents a fault caused directly by user level code. This might be, for example, an illegal instruction, or a floating point exception. A real kernel implementation should provide the handler with more information about the nature of the fault than the following function does; the nature of that information is specific to each architecture.

```
handleEvent (USER_LEVEL_FAULT n) = withoutPreemption $ do
  thread ← getCurThread
  handleFault thread USER_EXCEPTION
  return ()
```

## Virtual Memory Faults

If the simulator reports a VM fault, the appropriate action depends on whether the architecture has a software-loaded TLB. If so, we look up the address, and then insert it into the TLB; otherwise we simply send a fault IPC.

```
handleEvent (VM_FAULT_EVENT vptr isWrite) = withoutPreemption $ do
  thread ← getCurThread
  handleVMFault thread vptr isWrite `catchFailure` handleFault thread
  return ()
```

### 4.4.3 System Calls

#### SendWait System Call

The `sysSendWait` system call invokes a capability, and then waits for a reply on a given endpoint (typically, but not necessarily, the syscall reply endpoint specified in the TCB).

This call is equivalent to a call to `sysSend` followed by `sysWait`. The two operations are atomic. Using this call is the only way for a thread to invoke a real kernel object and receive a reply from the kernel.

```
handleSendWait :: KERNELP ()
handleSendWait = do
    thread ← withoutPreemption getCurThread
    info ← withoutPreemption $ getMessageInfo thread
    toCPtr ← withoutPreemption $ asUser thread $ liftM CPtr $
        getRegister (fst syscallRegisters)
    fromCPtr ← withoutPreemption $ asUser thread $ liftM CPtr $
        getRegister (snd syscallRegisters)
```

The `handleInvocation` function is called to perform the send phase.

```
result ← handleInvocation info toCPtr
```

If the invocation has deleted the current thread, no further action is taken.

```
withoutPreemption $ do
    sa ← getSchedulAction
    when (sa ≠ CURRENTTHREADISINVALID) $ do
```

The current thread listens for a reply on the specified receive endpoint.

```
listenForReply thread fromCPtr
    `catchFailure` handleFault thread
```

At this point, the thread is

- waiting to receive a reply,
- blocked on the reply endpoint,
- halted due to a fault while waiting for the reply,
- blocked on the send endpoint, or
- waiting to transfer the sent message.

The latter two cases are possible only if the invoked object was a synchronous or asynchronous endpoint. In that situation, the reply will be generated by a user-level task, so the kernel need not do anything more.

If the invocation was successful and the invoked object was not an endpoint, then the kernel must generate the reply. In this case, we send a nonblocking IPC to the reply endpoint, indicating that there was no error. If this fails for any reason, including a failed capability lookup, it will do so silently. Also, note that this send is performed even if the wait phase faulted.

```
sendResultIPC thread result
```

## Send System Call

The `SYSEND` system call is used to invoke a capability without waiting for a reply. It is equivalent to the first phase of `SYSENDWAIT`. Therefore, `handleSend` is identical to `handleSendWait` (defined above), except that the receive phase is never started.

Unless another thread is already waiting on the reply endpoint, any reply to an invocation made with this call will probably be lost. It will not be lost if the recipient replies with a blocking send, or if the sender is scheduled again in time to call `SYWAIT` before the recipient replies; however, the sender should *not* rely on either of these things happening. If the sender expects a reply, it should use `SYSENDWAIT` instead.

Note that system call monitors which wish to avoid detection must run at a higher priority than their clients. Otherwise, there is a race condition which allows the client to detect the monitor if it is able to call `SYWAIT` soon after a `SYSEND` call. In this situation, the kernel would never reply (as its replies are always sent during the `SYSEND` call), but a monitor will reply if it is slower than the sender.

```
handleSend :: KERNELP ()
handleSend = do
    thread ← withoutPreemption getCurThread
    info ← withoutPreemption $ getMessageInfo thread
    toCPtr ← withoutPreemption $ asUser thread $ liftM CPtr $
        getRegister (fst syscallRegisters)
    result ← handleInvocation info toCPtr
    sa ← withoutPreemption getSchedulerAction
    when (sa ≠ CURRENTTHREADISINVALID) $
        withoutFailure $ sendResultIPC thread result
```

## Wait System Call

The `SYWAIT` system call is used to wait for a message to be sent to a specified endpoint. It is equivalent to the second phase of `SYSENDWAIT`.

```
handleWait :: KERNEL ()
handleWait = do
    thread ← getCurThread
    epCPtr ← asUser thread $ liftM CPtr $
        getRegister (snd syscallRegisters)
    do
        epCap ← capFaultOnFailure epCPtr TRUE $ lookupCap epCPtr
        case epCap of
            ENDPPOINTCAP { capEPCanReceive = TRUE } →
                withoutFailure $ receiveIPC thread epCap
            ASYNCENDPOINTCAP { capAEPCanReceive = TRUE } →
```

```

        withoutFailure $ receiveAsyncIPC thread epCap
    _ → throw $ CAPFAULT epCPtr TRUE $
        MISSINGCAPABILITY { missingCapBitsLeft = 0 }
    `catchFailure` handleFault thread
return ()

```

## Identify System Call

The `SYSIDENTIFY` system call, given an endpoint capability from which capabilities can be minted with new badges, will read the badge from a second capability to the same endpoint. If the second capability is not valid or refers to a different endpoint, an error code is returned and the badge is not read.

```

handleIdentify :: KERNEL ()
handleIdentify = do
    thread ← getCurThread
    epCPtr ← asUser thread $ liftM CPTR $
        getRegister (fst syscallRegisters)
    idCapCPtr ← asUser thread $ liftM CPTR $
        getRegister (snd syscallRegisters)
    idCap ← nullCapOnFailure $ lookupCap idCapCPtr
    atomicSyscall
        (capFaultOnFailure epCPtr FALSE $ lookupCap epCPtr)
        (handleFault thread)
        (λepCap → case (epCap, idCap) of
            (ENDPOINTCAP { capEPCanIdentify = TRUE }, ENDPOINTCAP {})
            | capEPPtr epCap = capEPPtr idCap →
                return (capEPBadge idCap)
            (ENDPOINTCAP { capEPCanIdentify = TRUE }, _) →
                throw INVALIDCAPABILITY
            _ → throw ILLEGALOPERATION)
        (λerr →
            let msg = msgFromSyscallError err
            in asUser thread $ setRegister (fst syscallRegisters) (fst msg))
        (λbadge → do
            asUser thread $ setRegister (fst syscallRegisters) 0
            asUser thread $ setRegister (snd syscallRegisters) badge)

```

## Yield System Call

The yield system call is trivial; it simply sets the current thread's remaining time to 0. The scheduler will choose a new thread to switch to.

```

handleYield :: KERNEL ()
handleYield = do
    thread ← getCurThread
    threadSet (λtcb → tcb {tcbTimeSlice = 0}) thread

```

#### 4.4.4 Capability Invocations

If a capability is invoked in a `SEND` or `SENDWAIT` system call, the following function is called. It determines the type of invocation, based on the object type; then it calls the appropriate internal kernel function to perform the operation.

```

handleInvocation :: MESSAGEINFO → CPTR → KERNELP (MAYBE (WORD, [WORD]))
handleInvocation info ptr = do
    thread ← withoutPreemption getCurThread
    syscall

```

The destination capability's slot is located, and the capability read from it.

```

(do
    (slot, rights) ← capFaultOnFailure ptr FALSE $
        lookupSlotForCurrentThread ptr
    cap ← withoutFailure $ getSlotCap slot rights
    return (slot, cap))

```

If a fault was encountered looking up the capability, and the invocation is a blocking one, a fault message is sent. If the invocation is non-blocking, the fault is ignored. In either case, no reply is sent.

```

(λfault → do
    when (msgBlockingSend info) $
        handleFault thread fault >> return ()
    return NOTHING)

```

If there was no fault, then the capability, message registers and message label are used to determine the requested operation.

```

(λ(slot, cap) → do
    buffer ← withoutFailure $ lookupIPCBuffer FALSE thread
    args ← withoutFailure $ getMRs thread buffer info
    decodeInvocation (msgLabel info) args ptr slot cap)

```

If a system call error was encountered while decoding the operation, it is converted to an error message.

```

(λerr → return $! JUST (msgFromSyscallError err))

```

Otherwise, the operation is performed. If there is a result, it is converted to a success message (with label 0).

While the system call is running, the thread's state is set to `RESTART`, so any preemption will cause the system call to restart at user level when the thread resumes. If it is still set to `RESTART` when the operation completes, it is reset to `RUNNING` so the thread resumes at the next instruction.

```
(λop → do
  withoutPreemption $ setThreadState RESTART thread
  reply ← invoke (msgBlockingSend info) op
  withoutPreemption $ do
    state ← getThreadState thread
    case state of
      RESTART → setThreadState RUNNING thread
      _ → return ()
  return $! liftM (λa → (0, a)) reply)
```

## 4.5 Faults and Errors

This module specifies the mechanisms used by the `seL4` kernel to handle failures in kernel operations that must be communicated somehow to user-level code.

```
module SEL4.API.FAILURES where
```

```
import SEL4.MACHINE
import SEL4.API.TYPES
```

### 4.5.1 Types

#### Faults

When user-level code causes a kernel event, processing of that event may fail in a manner that the current thread normally cannot or should not handle itself. When that occurs, a *fault IPC* is sent to a designated fault handler. Such events typically include virtual memory or capability lookup failures, exceptions generated by the CPU, and interrupts from external hardware devices.

The procedure for handling faults is defined in section 5.4 on page 68.

```
data FAULT
  = CAPFAULT {
    capFaultAddress :: CPTR,
```

```

        capFaultInReceivePhase :: BOOL,
        capFaultFailure :: LOOKUPFAILURE }
    | VMFAULT {
        vmFaultAddress :: VPTR,
        vmFaultIsWrite :: BOOL,
        vmFaultFailure :: LOOKUPFAILURE }
    | UNKNOWNSYSCALL EXCEPTION {
        unknownSyscallNumber :: WORD }
    | USEREXCEPTION
    | INTERRUPT
deriving SHOW

```

## System Call Errors

Some errors encountered by system calls cannot reasonably be handled by a fault handler thread, and are best returned as error codes to the user. These include operations that can never succeed, because their arguments are out of range or invalid. These errors are typically sent to the calling thread via an IPC reply.

The following data type defines the set of errors that can be returned from a system call.

```

data SYSCALLERROR
    = INVALIDARGUMENT {
        invalidArgumentNumber :: INT }
    | INVALIDCAPABILITY
    | ILLEGALOPERATION
    | RANGEERROR {
        rangeErrorMin, rangeErrorMax :: WORD }
    | ALIGNMENTERROR
    | FAILEDLOOKUP {
        failedLookupWasSource :: BOOL,
        failedLookupDescription :: LOOKUPFAILURE }
    | TRUNCATEDMESSAGE
    | DELETEFIRST
    | REVOKEFIRST
deriving SHOW

```

## Lookup Failures

A capability or virtual address space lookup may fail in several different ways:

```

data LOOKUPFAILURE

```

- the root of the address space is not valid (that is, it is not of the correct type, or is not writable for the destination of a CNode operation, or is not readable for the source of a CNode operation);

= INVALIDROOT

- a slot on the lookup path contains no capability;

```
| MISSINGCAPABILITY {
    missingCapBitsLeft :: INT }
```

- there is no slot at the requested depth, or a page capability was found at the wrong depth;

```
| DEPTHMISMATCH {
    depthMismatchBitsLeft :: INT,
    depthMismatchBitsFound :: INT }
```

- or there is a CNode with a guard making the requested slot unreachable.

```
| GUARDMISMATCH {
    guardMismatchBitsLeft :: INT,
    guardMismatchGuardFound :: WORD,
    guardMismatchGuardSize :: INT }
deriving SHOW
```

## 4.5.2 Sending Failure Messages

When generating a fault or error IPC, the structures defined above must be converted to a list of words to be used to fill the message. The following functions are used to do so. For the functions that return a (WORD, [WORD]) tuple, the first element is to be used as the label of the reply IPC.

### Faults

```
msgFromFault :: FAULT → (WORD, [WORD])

msgFromFault (CAPFAULT cptr rp lf) =
    (1, fromCPtr cptr:fromIntegral (fromEnum rp):msgFromLookupFailure lf)

msgFromFault (VMFAULT vptr wr lf) =
    (2, fromVPtr vptr:fromIntegral (fromEnum wr):msgFromLookupFailure lf)

msgFromFault (UNKNOWNSCALLSEXCEPTION n) = (3, [n])

msgFromFault USEREXCEPTION = (4, [])
```

```
msgFromFault INTERRUPT = (5, [])
```

## System Call Errors

```
msgFromSyscallError :: SYSCALLERROR → (WORD, [WORD])
```

```
msgFromSyscallError (INVALIDARGUMENT n) = (1, [fromIntegral n])
```

```
msgFromSyscallError INVALIDCAPABILITY = (2, [])
```

```
msgFromSyscallError ILLEGALOPERATION = (3, [])
```

```
msgFromSyscallError (RANGEERROR min max) = (4, [min, max])
```

```
msgFromSyscallError ALIGNMENTERROR = (5, [])
```

```
msgFromSyscallError (FAILEDLOOKUP s lf) =  
  (6, (fromIntegral $ fromEnum s):(msgFromLookupFailure lf))
```

```
msgFromSyscallError TRUNCATEDMESSAGE = (7, [])
```

```
msgFromSyscallError DELETEDFIRST = (8, [])
```

```
msgFromSyscallError REVOKEFIRST = (9, [])
```

## Lookup Failures

Note that these are not directly returned to user level; this function is used only by the fault and error functions above. It does not return a `WORD` for the message tag.

```
msgFromLookupFailure :: LOOKUPFAILURE → [WORD]
```

```
msgFromLookupFailure INVALIDROOT = [1]
```

```
msgFromLookupFailure (MISSINGCAPABILITY bl) = [2, fromIntegral bl]
```

```
msgFromLookupFailure (DEPTHMISMATCH bl bf) =  
  [3, fromIntegral bl, fromIntegral bf]
```

```
msgFromLookupFailure (GUARDMISMATCH bl g gs) =  
  [4, fromIntegral bl, g, fromIntegral gs]
```

# 5 Kernel Operations

This chapter documents operations within the kernel that are not directly related to a single kernel object.

## 5.1 Capability Space Lookups

Each thread has a capability space, possibly shared with other threads. This is a mapping between addresses and kernel objects.

The capability space is represented by a guarded page table. Each level of the table contains  $2^n$  capability table entries, which are represented by the Haskell type `CTE`. These entries each contain a physical pointer to a kernel object, a set of permissions, a word with a meaning depending on the object type, and a mapping database entry.

This module implements the functions that look up a capability or virtual address in a user-level thread's capability space, as well as the function that assigns a new address space root for a thread.

```
module SEL4.KERNEL.CSPACE where
```

```
import SEL4.MACHINE
import SEL4.OBJECT
import SEL4.MODEL
import SEL4.API.TYPES
import SEL4.API.FAILURES

import DATA.BITS
```

### 5.1.1 Capability Lookups

These functions are used when invoking a capability, to determine the location and type of the object being invoked, and what rights the caller possesses for it.

These functions simply call the slot lookup functions defined below to locate the slot containing the requested capability, and then load the capability from it.

```
lookupCap :: CPTR → KERNELF LOOKUPFAILURE CAPABILITY
lookupCap cPtr = do
    (slot, rightsMask) ← lookupSlotForCurrentThread cPtr
    withoutFailure $ getSlotCap slot rightsMask

lookupCapForThread :: PPTR TCB → CPTR → KERNELF LOOKUPFAILURE CAPABILITY
lookupCapForThread thread cPtr = do
    (slot, rightsMask) ← lookupSlotForThread thread cPtr
    withoutFailure $ getSlotCap slot rightsMask
```

### 5.1.2 Locating a Capability Table Entry

The following functions are used in the three different situations in which the kernel must locate a capability table entry: invocation of a capability in the current thread's address space, invocation of a capability in a given address space, and modification of a specific entry in an invoked capability table.

When a capability is being invoked, the root can be determined from the identity of the thread making the call, and any failures are reported as faults.

In nearly all cases, this is done by the current thread; for convenience, one of the slot lookup functions uses the current thread's address space, while the other uses a specified address space. They are otherwise identical.

```
lookupSlotForCurrentThread :: CPTR →
    KERNELF LOOKUPFAILURE (PPTR CTE, CAPRIGHTS)
lookupSlotForCurrentThread capPtr = do
    thread ← withoutFailure $ getCurThread
    lookupSlotForThread thread capPtr

lookupSlotForThread :: PPTR TCB → CPTR →
    KERNELF LOOKUPFAILURE (PPTR CTE, CAPRIGHTS)
lookupSlotForThread thread capPtr = do
    threadRootSlot ← withoutFailure $ getThreadCspaceRoot thread
    threadRoot ← withoutFailure $ getSlotCap threadRootSlot allRights
    let bits = bitSize $ fromCPtr capPtr
        (s, _, r) ← resolveAddressBits threadRoot (const TRUE) capPtr bits
```

```
return (s, r)
```

When a pager is manipulating a capability space using a system call, the root and depth have been explicitly specified, and any failures encountered during the lookup should be turned into a system call error. Every CNode traversed during these lookups must possess the appropriate rights, depending on whether the specified slot will be read or modified; a function must be provided to test the rights of a given CNode capability. The specified depth must be correct (with no bits remaining after the lookup).

```
lookupSlotForCNodeOp ::
  BOOL → BOOL → CAPABILITY → CPTR → INT →
  KERNELF SYSCALLERROR (PPTR CTE, CAPRIGHTS)
lookupSlotForCNodeOp isSource willModify root@(CNODECAP {}) capptr depth = do
  rangeCheck depth 1 $ bitSize capptr
  let canUseCNode cap =
        (capCNodeCanRead cap ∨ not isSource) ∧
        (capCNodeCanModify cap ∨ not willModify)
      lookupErrorOnFailure isSource $ do
        unless (canUseCNode root) $ throw INVALIDROOT
        result ← resolveAddressBits root canUseCNode capptr depth
        case result of
          (slot, 0, rights) → return (slot, rights)
          (_, bitsLeft, _) → throw $ DEPTHMISMATCH {
            depthMismatchBitsLeft = bitsLeft,
            depthMismatchBitsFound = 0 }
  lookupSlotForCNodeOp isSource _ _ _ _ =
    throw $ FAILEDLOOKUP isSource INVALIDROOT
```

## Internal Functions

The capability table lookup is performed by the function `resolveAddressBits`.

The arguments to this function are a capability to access a CNode, a function that determines whether a CNode capability is usable for the lookup, the capability space address being looked up, and the number of bits remaining to be resolved in the address. The function on CNode capabilities is used by `lookupSlotForCNodeOp`, above; for normal lookups it should always return `True`.

It returns a pointer to a slot, the number of bits still unresolved in the address when a valid non-CNode capability was found, and a mask to be applied to the rights of the capability.

```
resolveAddressBits :: CAPABILITY → (CAPABILITY → BOOL) → CPTR → INT →
  KERNELF LOOKUPFAILURE (PPTR CTE, INT, CAPRIGHTS)
```

The following definition is used when a CNode capability is encountered in the cap table.

```
resolveAddressBits nodeCap@(CNODECAP {cap}) canUseCNode capptr bits = do
```

Determine the number of bits that this CNode can resolve.

```
let radixBits = capCNodeBits nodeCap
let guardBits = capCNodeGuardSize nodeCap
let levelBits = radixBits + guardBits
assert (levelBits ≠ 0) ‘‘All CNodes must resolve bits’’
```

If the lookup has exceeded the expected depth, then the table is badly formed. In this case, throw a fault.

```
when (levelBits > bits) $ throw $ DEPTHMISMATCH {
  depthMismatchBitsLeft = bits ,
  depthMismatchBitsFound = levelBits }
```

Check that the guard is correct.

```
let guard = (fromCPtr capptr `shiftR` (bits - guardBits)) .&.
             (mask guardBits)
unless (guard = capCNodeGuard nodeCap) $ throw $ GUARDMISMATCH {
  guardMismatchBitsLeft = bits ,
  guardMismatchGuardFound = capCNodeGuard nodeCap ,
  guardMismatchGuardSize = guardBits }
```

If the current CNode is valid, then locate the slot in it that contains the next capability to be visited.

```
let offset = (fromCPtr capptr `shiftR` (bits - levelBits)) .&.
             (mask radixBits)
slot ← withoutFailure $ locateSlot (capCNodePtr nodeCap) offset
```

If all of the remaining bits in the address have been resolved, then *slot* is the final result.

```
let bitsLeft = bits - levelBits
if (bitsLeft = 0)
  then return (slot , 0, capCNodeRightsMask nodeCap)
```

Otherwise, there are more address bits left to resolve. Fetch the next capability.

```
else do
  nextCap ← withoutFailure $
    getSlotCap slot allRights
```

Determine the type of the next capability.

```
case nextCap of
```

If no capability exists, then throw a fault.

```

NULLCAP → throw $ MISSINGCAPABILITY {
    missingCapBitsLeft = bitsLeft }

```

If the next capability is a CNode, then perform a recursive call to look at the next level of the table. Mask the returned rights with the current node's rights mask.

```

CNODECAP {} → do
    unless (canUseCNode nextCap) $
        throw $ MISSINGCAPABILITY {
            missingCapBitsLeft = bitsLeft }
    (cap, endBits, rights) ←
        resolveAddressBits nextCap canUseCNode capptr bitsLeft
    return (cap, endBits,
        rights `andCapRights` capCNodeRightsMask nodeCap)

```

Otherwise, return the new slot, the number of unresolved address bits, and the rights mask on the current CNode.

```

_ → return (slot, bitsLeft, capCNodeRightsMask nodeCap)

```

The following definition will be used if the top level CNode is not valid.

```

resolveAddressBits _ _ capptr bits = throw INVALIDROOT

```

## 5.2 Threads and Scheduling

This module contains the scheduler, and miscellaneous functions that manipulate thread state.

```

module SEL4.KERNEL.THREAD where

import SEL4.API.SYSCALL
import SEL4.API.TYPES
import SEL4.API.FAILURES
import SEL4.MACHINE
import SEL4.MODEL
import SEL4.OBJECT
import SEL4.OBJECT.STRUCTURES(TCB(..), THREADSTATE(..), SCHEDULERACTION(..))
import SEL4.KERNEL.CSPACE
import SEL4.KERNEL.VSPACE
import SEL4.KERNEL.FAULTHANDLER

import DATA.BITS
import DATA.MAYBE

```

```
import DATA.LIST
```

## 5.2.1 Initial Thread Creation

Activation of the initial thread requires the thread's program counter and first argument register to be set explicitly. The Exchange Registers system call can only copy those registers from another thread, so the following function is used instead. It simply sets the first argument register and the program counter, sets the thread's state so it can be scheduled, and then switches to the thread.

```
activateInitialThread :: PPTR TCB → VPTR → VPTR → KERNEL ()
activateInitialThread threadPtr entry infoPtr = do
  asUser threadPtr $ setRegister (fst syscallRegisters) $
    fromVPtr infoPtr
  asUser threadPtr $ setNextPC $ fromVPtr entry
  setThreadState RUNNING threadPtr
  switchToThread threadPtr
```

## 5.2.2 Thread Activation

Before a thread starts running, the kernel must determine whether there are any kernel operations that must be completed beforehand. This includes long-running system call operations that have been interrupted and must be resumed, and IPC transfers that have not been completed.

```
activateThread :: KERNEL ()
activateThread = do
  thread ← getCurThread
  state ← getThreadState thread
  case state of
    RUNNING → return ()
    RESTART → do
      pc ← asUser thread $ getRestartPC
      asUser thread $ setNextPC pc
      setThreadState RUNNING thread
    CONTINUE _ →
      error '‘Unimplemented’' ---_XXX
  WAITINGTOSEND { pendingReceiveCap = NOTHING } →
    doIPCtransfer thread (waitingIPCpartner state)
  WAITINGTORECEIVE {} →
    doIPCtransfer (waitingIPCpartner state) thread
  _ → error $ '‘Current_thread_is_blocked,_state:_’' ++ show state
```

### 5.2.3 Thread State

The following functions are used by the scheduler to determine whether a particular thread is ready to be scheduled, and whether it is ready to run.

```
isBlocked :: PPtr TCB → KERNEL BOOL
isBlocked thread = do
    state ← getThreadState thread
    return $ case state of
        INACTIVE → TRUE
        BLOCKEDONRECEIVE {} → TRUE
        BLOCKEDONSEND {} → TRUE
        BLOCKEDONASYNC EVENT {} → TRUE
        WAITINGTOSEND { pendingReceiveCap = JUST _ } → TRUE
        _ → FALSE

isRunnable :: PPtr TCB → KERNEL BOOL
isRunnable thread = liftM not $ isBlocked thread
```

### Restarting a Blocked Thread

The Restart operation forces a thread that has blocked to retry the operation that caused it to block.

The invoked thread will return to the instruction that caused it to enter the kernel prior to blocking. If an IPC is in progress (including a fault IPC), it will be silently aborted. Beware of doing this to restart an atomic send and receive operation — the thread will retry the send phase, even if it had previously succeeded in sending the message and was waiting for the receive phase to complete.

```
restart :: PPtr TCB → KERNEL ()
restart target = do
    blocked ← isBlocked target
    when blocked $ do
        ipcCancel target
        setThreadState RESTART target
```

### 5.2.4 IPC Transfers

The following function is called before resuming execution of a thread that has a pending IPC transfer. It looks up the sender and receiver's message buffers (in that order, and skipping the send buffer for a fault IPC), and then transfers the message.

If either of the buffers is missing, then the message will be truncated to include only the part not stored in the buffer.

```
doIPCTransfer :: PPTR TCB → PPTR TCB → KERNEL ()
doIPCTransfer sender receiver = do
  senderState ← getThreadState sender
  recvState ← getThreadState receiver
  let fault = waitingIPCFault senderState
      badge = waitingIPCBadge senderState
      receiveBuffer ← lookupIPCBuffer TRUE receiver
      setThreadState RUNNING receiver
  case fault of
    NOTHING → do
      sendBuffer ← lookupIPCBuffer FALSE sender
      doMRTransfer badge
        sender sendBuffer receiver receiveBuffer
        (waitingIPCCanGrant senderState)
        (waitingIPCDiminishCaps recvState)
      setThreadState RUNNING sender
```

For non-fault IPCs, the kernel switches to the thread with higher priority (unless the sender has higher priority and has a pending receive phase).

```
senderPrio ← threadGet tcbPriority sender
receiverPrio ← threadGet tcbPriority receiver
when (receiverPrio > senderPrio) $ switchToThread receiver
when (senderPrio > receiverPrio ∧
      pendingReceiveCap senderState = NOTHING) $
  switchToThread sender
```

If the sent message is a fault IPC, the sender is placed in the blocked state; the kernel then switches to the receiving thread.

```
JUST f → do
  setFaultMRs badge sender f receiver receiveBuffer
  setThreadState INACTIVE sender
  switchToThread receiver
```

If the sender has a pending receive phase of an atomic send and receive operation, that phase is started.

```
case pendingReceiveCap senderState of
  JUST cptr → performReceivePhase sender cptr
  `catchFailure` (λf → do
    handleFault sender f
    return ())
  NOTHING → return ()
```

## Ordinary IPC

Ordinary IPC simply transfers all message registers. It requires pointers to the source and destination threads, and also to their respective IPC buffers.

```
doMRTransfer :: WORD → PPTR TCB → MAYBE (PPTR WORD) → PPTR TCB →
  MAYBE (PPTR WORD) → BOOL → BOOL → KERNEL ()
doMRTransfer badge sender sendBuffer receiver receiveBuffer canGrant diminish
= do
  msgInfo ← getMessageInfo sender
  mrs ← getMRs sender sendBuffer msgInfo
  msgTransferred ← setMRs receiver receiveBuffer mrs
  capTransferred ← if msgCapTransfer msgInfo ∧ canGrant
  then transferCap
    msgInfo sender sendBuffer receiver receiveBuffer diminish
  else return FALSE
let msgInfo' = msgInfo {
  msgLength = msgTransferred,
  msgCapTransfer = capTransferred }
setMessageInfo receiver msgInfo'
asUser receiver $ setRegister badgeRegister badge
```

## Fault IPC

If the message is a fault — either just generated, or loaded from the sender's TCB — then it will be transferred instead of the sender's message registers. In this case, no pointer to the sender's buffer is required.

The recipient's argument registers are filled in with various information about the nature of the fault and the present state of the faulting thread.

```
setFaultMRs :: WORD → PPTR TCB → FAULT → PPTR TCB → MAYBE (PPTR WORD) →
  KERNEL ()
setFaultMRs badge sender f receiver receiverIPCBuffer = do
  pc ← asUser sender $ getRestartPC
  let (faultLabel, faultMsg) = msgFromFault f
  sent ← setMRs receiver receiverIPCBuffer $ pc: faultMsg
  let msgInfo = MI {
    msgLength = sent,
    msgCapTransfer = FALSE,
    msgBlockingSend = TRUE,
    msgLabel = faultLabel }
  setMessageInfo receiver msgInfo
  asUser receiver $ setRegister badgeRegister badge
```

## IPC Capability Transfers

This function is called when an IPC message includes a capability to transfer. It attempts to perform the transfer, and returns TRUE if successful.

```
transferCap :: MESSAGEINFO → PPTR TCB → MAYBE (PPTR WORD) →
             PPTR TCB → MAYBE (PPTR WORD) → BOOL →
             KERNEL BOOL
transferCap info sender sendBuffer receiver receiveBuffer diminish = do
  sendInfo ← getSendCap info sender sendBuffer
  destSlot ← getReceiveSlot receiver receiveBuffer
  case (sendInfo, destSlot) of
    (JUST (sendCap, sendSlot), JUST destSlot) → do
      let sendCap' = if diminish
                    then allRights { capAllowWrite = FALSE }
                      `maskCapRights` sendCap
                    else sendCap
          cteInsert sendCap' sendSlot destSlot
          return TRUE
    _ → return FALSE
```

## Asynchronous IPC

In the case of asynchronous IPC, the message is loaded from the receiver's TCB and into the IPC buffer.

```
doAsyncTransfer :: WORD → WORD → PPTR TCB → KERNEL ()
doAsyncTransfer badge msgWord thread = do
  setThreadState RUNNING thread
  setMRs thread NOTHING [msgWord]
  asUser thread $ setRegister badgeRegister badge
```

## 5.2.5 Scheduling

### The Scheduler

The scheduler will perform one of three actions, depending on the scheduler action field of the global kernel state.

```
schedule :: KERNEL ()
schedule = do
  action ← getSchedAction
  case action of
```

The default action is to choose a new thread whenever the current thread is not runnable, or when its timeslice has expired.

```
SCHEDULENORMALLY → do
  curThread ← getCurThread
  runnable ← isRunnable curThread
  st ← threadGet tcbState curThread
  time ← threadGet tcbTimeSlice curThread
  when (not runnable ∨ time = 0) chooseThread
```

An IPC operation may request that the scheduler switch to a specific thread.

```
SWITCHTOTHREAD t → do
  switchToThread t
  setSchedulerAction SCHEDULENORMALLY
```

If the current thread has been deleted, or if there has never been a current thread, then the scheduler must choose a new thread without accessing the current TCB.

```
CURRENTTHREADISINVALID → do
  chooseThread
  setSchedulerAction SCHEDULENORMALLY
```

Threads are scheduled using a simple multiple-priority round robin algorithm. It iterates through the ready queues, starting with the highest priority queue; when it finds a non-empty ready queue, it selects the first thread in the queue, makes it the current thread, and moves it to the end of the queue.

Note that the ready queues are a separate structure in the kernel model. In a real implementation, to avoid requiring dynamically-allocated kernel memory, these queues would be linked lists using the TCBs themselves as nodes.

```
chooseThread :: KERNEL ()
chooseThread = do
  r ← findM chooseThread' (reverse [minBound .. maxBound])
  assert (isJust r) 'No_⊥runnable_⊥threads''
  where
    chooseThread' :: PRIORITY → KERNEL BOOL
    chooseThread' prio = do
      q ← getQueue prio
      liftM isJust $ findM (chooseThread'' prio) (tails q)
    chooseThread'' :: PRIORITY → READYQUEUE → KERNEL BOOL
    chooseThread'' prio (first:rest) = do
      runnable ← isRunnable first
      if not runnable
      then do
        setQueue prio rest
        threadSet (λt → t {tcbQueued=FALSE}) first
```

```

        return FALSE
    else do
        setQueue prio (rest ++ [first])
        switchToThread first
        return TRUE
chooseThread" prio [] = return FALSE

```

## Switching Threads

When switching to a new thread, we check its current timeslice; if it is zero, then we give the thread a new timeslice. It may not be zero if it has been donated time by another thread, or previously exchanged IPC with another thread before its timeslice expired.

```

switchToThread :: PPTR TCB → KERNEL ()
switchToThread thread = do
    time ← threadGet tcbTimeSlice thread
    when (time = 0) $ threadSet (λt → t {tcbTimeSlice=timeSlice}) thread
    doMachineOp flushCaches
    setCurThread thread

```

The following function is used to alter a thread's priority. If it is changed while the thread is queued, then the thread must be removed from the old priority's queue. If the thread is runnable, it is then added to the queue for the new priority.

```

setPriority :: PPTR TCB → PRIORITY → KERNEL ()
setPriority tptr prio = do
    oldPrio ← threadGet tcbPriority tptr
    runnable ← isRunnable tptr
    queued ← threadGet tcbQueued tptr
    if queued
    then do
        q1 ← getQueue oldPrio
        setQueue oldPrio $ filter (≠ tptr) q1
        if runnable
        then do
            q2 ← getQueue prio
            setQueue prio $ tptr : q2
            threadSet (λt → t { tcbPriority = prio }) tptr
        else threadSet (λt → t {
            tcbQueued = FALSE, tcbPriority = prio }) tptr
    else threadSet (λt → t { tcbPriority = prio }) tptr

```

## The Yield-To Call

The YieldTo operation donates the current thread's timeslice to the destination thread, and then switches to that thread if it is runnable.

```
yieldTo :: PPTR TCB → KERNEL ()
yieldTo target = do
  caller ← getCurThread
  callertime ← threadGet tcbTimeSlice caller
  threadSet (λt → t { tcbTimeSlice = 0 }) caller
  targettime ← threadGet tcbTimeSlice target
  threadSet (λt → t { tcbTimeSlice = callertime + targettime }) target
  st ← getThreadState target
  runnable ← isRunnable target
  when runnable $ switchToThread target
```

## Scheduling Parameters

A trivial function is provided to fetch the current scheduler state of a thread.

```
getThreadState :: PPTR TCB → KERNEL THREADSTATE
getThreadState = threadGet tcbState
```

When setting the scheduler state, the thread is added to the appropriate ready queue if it is not blocked, and isn't already in the queue.

```
setThreadState :: THREADSTATE → PPTR TCB → KERNEL ()
setThreadState st tptr = do
  threadSet (λt → t { tcbState = st }) tptr
  runnable ← isRunnable tptr
  queued ← threadGet tcbQueued tptr
  when (runnable ∧ not queued) $ do
    prio ← threadGet tcbPriority tptr
    q ← getQueue prio
    setQueue prio $ tptr : q
    threadSet (λt → t { tcbQueued = TRUE }) tptr
```

## 5.3 Bootstrapping the Kernel

This module contains functions that create a new kernel state and set up the address space and context of the initial user-level task.

```
module SEL4.KERNEL.INIT where
```

```

import SEL4.API.TYPES
import SEL4.MODEL
import SEL4.OBJECT
import SEL4.MACHINE
import SEL4.KERNEL.THREAD
import SEL4.KERNEL.VSPACE

import DATA.ARRAY
import DATA.BITS
import DATA.LIST(mapAccumL, sort, sortBy)
import DATA.MAYBE(catMaybes)

```

### 5.3.1 Kernel Initialisation

The first thread in the system is created by the kernel; its address space contains capabilities to use all of the machine’s physical resources, minus any physical memory which is used by the kernel’s static data and code.

#### Initial Address Space

The initial address space layout needs to be balanced between using superpages to cover as much memory as possible, to allow greater flexibility for creating large mappings; and providing a reasonable number of small objects, to make the root server’s address space easy to manage reliably.

So, the initial address space will be constructed as follows:

- The kernel will create four CNodes that each occupy one frame of physical memory (equal to the hardware’s smallest page size), and map three of them at the bottom of the initial thread’s CSpace. The first will start at address 0, though the positions of the others are implementation-defined. The fourth CNode will serve as the root of the CSpace.
- The initial task’s thread capability, address space root capability, and system call reply endpoint will be mapped into its address space starting at capability address 1. The address of each of these capabilities will be recorded in the boot information structure.

- The initial task's code and data will be loaded and mapped. The means of doing this is implementation-defined, but all capabilities used in the process will be placed in the empty slots following the initial kernel object capabilities. The CSpace and VSpace addresses of these objects will be recorded in the boot information structure.
- Two extra virtual memory pages will be allocated and mapped. The boot information structure will be written into one; its location will be passed to the initial thread in a register. The second page will become the initial thread's IPC buffer.
- From the remaining physical memory, the kernel will create untyped *small blocks* of size equal to the machine's smallest virtual page size, and insert capabilities to use them into the second initial CNode. The number of small blocks created is limited to the number that will fit in the CNode, and the kernel may reduce it further to ensure that the small pages occupy a block of memory with a power of two size. On architectures with cache aliasing issues, the small blocks may be divided into groups by cache colour.
- After allocation of the small blocks, all remaining physical memory will be divided into *large blocks*, and capabilities to use them will be inserted into the third initial CNode. On architectures with cache aliasing issues, all of the initial large blocks must be large enough for no division by cache colour to be necessary.
- A virtual mapping capability will be created for each memory-mapped IO device in the system. These capabilities will be provided to the initial thread in the same manner as the untyped capabilities. The means of identifying these devices, including specifying any restrictions on the way they are mapped, is implementation-dependent.

Implementations that require separate CSpace and VSpace structures will provide caps in the initial CSpace for all frames used to back the initial VSpace. Otherwise, the initial VSpace root will be a copy of the initial CSpace root. In both cases, the CSpace address of each mapped frame capability will be equal to the virtual address of the page it backs.

## Initialisation Function

The following Haskell function creates the initial thread and configures its address space, given the address of the initial thread's entry point.

```
initKernel :: VPTR → [PPTR ()] → KERNEL ()
initKernel entry initFrames = do
```

First, calculate the size of the small-block area; that is where we will be storing the initial objects. The top of this area is the top of memory or the end of the region that can be mapped by a single page-sized CNode, whichever is smaller.

```

let levelBits = pageBits - objBits (⊥::CTE)
let levelSize = 1 `shiftL` levelBits
memTop ← liftM fromPPtr $ doMachineOp $ getMemoryTop
let smallBlockTop = min memTop $ (levelSize `shiftL` pageBits) - 1

```

Get the addresses and types of all page-sized objects in the small block area.

```

let pageSize = 1 `shiftL` pageBits
let smallObjects = map PPtr $
    [kernelTop*pageSize, (kernelTop+1)*pageSize .. smallBlockTop]

```

Then, allocate physical addresses to the four CNodes, the info pointer, the syscall reply endpoint and the initial thread, and allocate capability addresses to those of them objects that will be mapped.

```

let buffer:bootInfo:rootCN:taskCN:smallCN:superCN:initialTCB:replyEP:~ =
    reverse smallObjects
let initialTCBIndex:rootCNIndex:rootVNIndex:replyEPIndex:freeSlots =
    [1 .. levelSize - 1]

```

Find the set of all typed objects that must be mapped for the initial thread, other than the root CNode. The tuples each contain the object's physical address, the object's CSpace address and size in address bits, and an object type.

```

let initialObjects =
    (bootInfo, fromPPtr bootInfo, INTDATAOBJECT):
    (buffer, fromPPtr buffer, INTDATAOBJECT):
    (initialTCB, initialTCBIndex, TCBOBJECT):
    (replyEP, replyEPIndex, ENDPOINTOBJECT):
    map (λaddr → (addr, fromPPtr addr, INTDATAOBJECT))
        initFrames

```

Create the root CNode.

```

[rootCNCap] ← createNewCaps (fromAPIType CAPTABLEOBJECT)
    rootCN pageBits levelBits

```

Create the three second-level CNodes.

```

[[taskCNCap], [smallCNCap], [superCNCap]] ←
    mapM (λptr → createNewCaps
        (fromAPIType CAPTABLEOBJECT) ptr pageBits levelBits)
        [taskCN, smallCN, superCN]

```

Create the initial objects and map them into the task or root CNode.

```

mapM
    (λ(ptr, cap, otype) → do
        slot ← if otype = INTDATAOBJECT
            then locateSlot rootCN (cap `shiftR` pageBits)

```

```

        else locateSlot taskCN cap
        caps ← createNewCaps (fromAPIType otype)
        ptr pageBits 0
        insertInitCap slot (head caps)
    initialObjects

```

Insert the root CNode cap in the task CNode.

```

let rootCNCap' = rootCNCap {
    capCNodeGuardSize = bitSize entry - pageBits - levelBits }
rootCNSlot ← locateSlot taskCN rootCNIndex
insertInitCap rootCNSlot rootCNCap'

```

Get the set of all page-sized untyped objects remaining in the small block area.

```

let freeBlocks =
    [address | address ← smallObjects,
      not $ elem address $
        map (λ(a, -, -) → a) initialObjects,
      address ≠ smallCN,
      address ≠ superCN,
      address ≠ taskCN]

```

Create the VTable.

```

(usedBlocks, rootVNCap) ← initVSpace freeBlocks rootCNCap'
rootVNSlot ← locateSlot taskCN rootVNIndex
cteInsert rootVNCap rootCNSlot rootVNSlot

```

Find the set of small blocks to be provided to the user.

```

let smallBlocks = [address | address ← freeBlocks,
      not $ elem address usedBlocks]

```

Map all the small untyped objects into their CNode.

```

zipWithM_
  (λptr cnOffset → do
    slot ← locateSlot smallCN cnOffset
    insertInitCap slot (UNTYPEDCAP ptr pageBits))
  smallBlocks [0,1..]

```

Get the set of large untyped objects.

```

let largeBlocks = makeSuperBlockList
  (PPTR $ smallBlockTop + 1) (memTop - smallBlockTop)

```

Map all the large untyped objects into their CNode.

```

zipWithM_
  (λ(ptr, size) cnOffset → do

```

```

    slot ← locateSlot superCN cnOffset
    insertInitCap slot (UNTYPEDCAP ptr size)
    largeBlocks [0,1..]

```

Map the three second-level CNodes into the root CNode.

```

zipWithM_
  (λcap cnOffset → do
    slot ← locateSlot rootCN cnOffset
    let cap' = cap { capCNodeGuardSize = pageBits - levelBits }
    insertInitCap slot cap')
  [taskCNCap, smallCNCap, superCNCap] [0,1,2]

```

The root nodes, reply endpoint and priority are set. The priority is the maximum possible value.

```

threadCRoot ← getThreadCSpaceRoot initialTCB
threadVRoot ← getThreadVSpaceRoot initialTCB
cteInsert rootCNCap' rootCNSlot threadCRoot
cteInsert rootVNCap rootVNSlot threadVRoot
threadSet (λt → t { tcbResultEndpoint = CPTR replyEPIndex,
                    tcbIPCBuffer = VPTR $ fromPPtr buffer })
    initialTCB
setPriority initialTCB maxBound

```

Create region descriptors for the initial capability regions.

```

let firstFreeSlot = case freeSlots of
  (slot:_) → slot
  _ → levelSize
let capRegions = [
  BOOTREGION 0 (VPTR $ pageSize - 1) BRINITCAPS firstFreeSlot,
  BOOTREGION (VPTR pageSize) (VPTR $ 2*pageSize - 1)
  BRINITCAPS
  (pageSize + (fromIntegral $ length smallBlocks)),
  BOOTREGION (VPTR $ 2*pageSize) (VPTR $ 3*pageSize - 1)
  BRINITCAPS
  (2*pageSize + (fromIntegral $ length largeBlocks))]

```

Create region descriptors for the root task's mapped objects.

```

let rootTaskRegions =
  map (λaddr → BOOTREGION addr (addr + 1 `shiftL` pageBits - 1)
    BRROOTTASK 0)
    $ sort $ map (VPTR . fromPPtr)
      (buffer:bootInfo:initFrames)

```

Calculate the empty regions of the address space.

```

let usedRegions = sortBy ( $\lambda a b \rightarrow compare (brBase a) (brBase b)$ ) $
    rootTaskRegions ++ capRegions
let (topUsed, precedingEmptyRegions) = mapAccumL
    ( $\lambda a r \rightarrow$  if  $a < brBase r$ 
     then (brEnd r + 1,
          JUST $ BOOTREGION a (brBase r - 1) BREMPTY 0)
     else (brEnd r + 1, NOTHING)) 0 usedRegions
let emptyRegions = catMaybes precedingEmptyRegions ++
    if (topUsed = 0) then [] else
    [BOOTREGION topUsed maxBound BREMPTY 0]

```

The boot data page is filled in with information about the system that the initial thread needs — in particular, the locations of all the initial objects.

```

let bootInfoData = BOOTINFO {
    biIPCBuffer = VPTR $ fromPPtr buffer,
    biFirstSmallUntyped = CPTR pageSize,
    biSmallUntypedCount = fromIntegral $ length smallBlocks,
    biFirstLargeUntyped = CPTR $ pageSize * 2,
    biLargeUntypedSizes = memTop - smallBlockTop,
    biRegions = usedRegions ++ emptyRegions }
let bootInfoWords = wordsFromBootInfo bootInfoData
let intSize = fromIntegral $ bitSize (⊥::WORD) `div` 8
doMachineOp $ zipWithM_ storeWord
    [bootInfo, bootInfo + (PPTR intSize)..] bootInfoWords

```

Finally, the initial thread's context is set up with the entry point and the location of the boot information page, and then activated.

```

activateInitialThread initialTCB entry
    (VPTR $ fromPPtr bootInfo)

```

## Creating Large Untyped Objects

The following function is used to construct a list of large untyped memory objects, given the base address and size of a contiguous region of unallocated memory.

The sum of the base address and the size must be a power of two.

```

makeSuperBlockList :: PPTR () → WORD → [(PPTR (), INT)]
makeSuperBlockList base size = snd $ mapAccumL makeBlock base blockSizes
  where
    blockSizes = filter (testBit size) [0 .. bitSize base - 1]
    makeBlock b s = (b + (1 `shiftL` s), (b, s))

```

## Initial Kernel State

A new kernel state structure contains an empty physical address space, a set of empty scheduler queues, and an undefined value for the initial thread.

```
newKernelState :: KERNELSTATE
newKernelState = KSTATE {
    ksPSpace = newPSpace,
    ksReadyQueues = listArray (minBound,maxBound) (repeat []),
    ksCurThread = error "No initial thread",
    ksSchedulerAction = CURRENTTHREADISINVALID }
```

## 5.4 Handling Faults

This module contains functions that determine how recoverable faults encountered by user-level threads are propagated to the appropriate fault handlers.

```
module SEL4.KERNEL.FAULTHANDLER (handleFault) where

import SEL4.API.TYPES
import SEL4.API.FAILURES
import SEL4.MACHINE
import SEL4.MODEL
import SEL4.OBJECT
import {-# SOURCE #-} SEL4.KERNEL.THREAD
import SEL4.KERNEL.CSPACE

import DATA.BITS
```

### 5.4.1 Handling Faults

Faults generated by the `handleEvent` function (which is defined in section 4.4 on page 38) are caught and sent to `handleFault`, defined below.

The parameters of this function are the fault and a pointer to the thread which requested the kernel operation that generated the fault.

```
handleFault :: PPTR TCB → FAULT → KERNEL ()

handleFault tptr ex = do
```

The thread is made inactive to prevent it restarting without an explicit operation by the fault handler.

```
setThreadState INACTIVE tptr
```

A fault IPC is sent to the fault handler endpoint.

```
sendFaultIPC tptr ex`catchFailure`handleDoubleFault tptr ex
```

## 5.4.2 Sending Fault IPC

If a thread causes a fault, then an IPC containing details of the fault is sent to a fault handler endpoint specified in the thread's TCB.

```
sendFaultIPC :: PPtr TCB → FAULT → KERNELF FAULT ()
sendFaultIPC tptr fault = do
```

The fault handler endpoint capability is fetched from the TCB.

```
handlerCPtr ← withoutFailure $ threadGet tcbFaultHandler tptr
handlerCap ← capFaultOnFailure handlerCPtr FALSE $
  lookupCapForThread tptr handlerCPtr
```

```
case handlerCap of
```

The kernel stores a copy of the fault in the thread's TCB, and performs an IPC send operation to the fault handler endpoint on behalf of the faulting thread. When the IPC completes, the fault will be retrieved from the TCB and sent instead of the message registers.

```
ENDPOINTCAP { capEPCanSend = TRUE } → withoutFailure $
  sendIPC TRUE (JUST fault) (capEPBadge handlerCap)
  FALSE tptr (capEPPtr handlerCap)
```

If there are insufficient permissions to send to the fault handler, then another fault will be generated.

```
_ → throw $ CAPFAULT handlerCPtr FALSE $
  MISSINGCAPABILITY { missingCapBitsLeft = 0 }
```

### 5.4.3 Double Faults

```
handleDoubleFault :: PPTR TCB → FAULT → FAULT → KERNEL ()
```

If a fault IPC cannot be sent because the fault handler endpoint capability is missing, then we are left with two faults which cannot be reasonably handled. The faults are both printed to the console for debugging purposes. The faulting thread is already blocked, and will remain that way until a user-level thread destroys or restarts it.

```
handleDoubleFault tptr ex1 ex2 = do
  faultPC ← asUser tptr getRestartPC
  let errmsg = ‘‘Caught fault’’ ++ (show ex2)
        ++ ‘‘\nwhile trying to handle fault’’ ++ (show ex1)
        ++ ‘‘\nin thread’’ ++ (show tptr)
        ++ ‘‘\nat address’’ ++ (show faultPC)
  doMachineOp $ debugPrint errmsg
```

## 5.5 Virtual Memory

Each thread has a virtual memory address space, possibly shared with other threads.

The representation of the virtual address space is implementation-defined. On many architectures — those with software-loaded TLBs, for example — it will simply be a second CSpace structure, possibly identical to or a subtree of the thread’s main CSpace. On others, such as those with hardware-defined page table structures, it must be constructed using implementation-defined kernel objects.

We use the C preprocessor to select a target architecture.

```
{-# OPTIONS_GHC -cpp #-}

module SEL4.KERNEL.VSPACE where

import SEL4.MACHINE
import SEL4.MODEL
import SEL4.OBJECT
import SEL4.API.FAILURES
import SEL4.API.TYPES
import qualified SEL4.KERNEL.VSPACE.TARGET as ARCH
```

## 5.5.1 Implementation-defined Functions

This module defines four functions, for:

- creating the initial address space, given a list of free page-sized frames and the root CSpace capability, and returning a list of used page-sized frames and the root VSpace capability;

```
initVSpace :: [PPTR ()] → CAPABILITY → KERNEL ([PPTR ()], CAPABILITY)
initVSpace = ARCH.initVSpace
```

- handling virtual memory faults, given the current thread, the faulting address, and a boolean value that is true for write accesses;

```
handleVMFault :: PPTR TCB → VPTR → BOOL → KERNELF FAULT ()
handleVMFault = ARCH.handleVMFault
```

- translating virtual addresses to physical addresses for in-kernel accesses;

```
lookupVPtr :: PPTR TCB → VPTR → BOOL → KERNELF LOOKUPFAILURE (PPTR WORD)
lookupVPtr = ARCH.lookupVPtr
```

- and determining whether a given capability is a valid VSpace root.

```
isValidVTableRoot :: CAPABILITY → BOOL
isValidVTableRoot = ARCH.isValidVTableRoot
```

## 5.5.2 IPC Buffer Lookups

This function is used during system calls to find a physical pointer to a thread's IPC buffer, which is an area of physical memory used to store any IPC message registers that don't fit in hardware registers.

```
lookupIPCBuffer :: BOOL → PPTR TCB → KERNEL (MAYBE (PPTR WORD))
lookupIPCBuffer isReceiver thread = do
    bufferPtr ← threadGet tcbIPCBuffer thread
    nothingOnFailure $
        liftM JUST $ lookupVPtr thread bufferPtr isReceiver
```

## 6 Kernel Objects

This chapter defines the first-class kernel objects, including their representation in physical memory, and the operations that may be performed on them by user-level threads and by other kernel-level code.

### 6.1 Data Structures

This module defines the structures which represent kernel objects in the modelled physical memory.

This module makes use of the GHC extension allowing derivation of the class `TYPEABLE`, so GHC language extensions are enabled. We also use the C preprocessor to select a target architecture.

```
{-# OPTIONS_GHC -fglasgow-exts -cpp #-}

module SEL4.OBJECT.STRUCTURES (
    module SEL4.OBJECT.STRUCTURES,
    module SEL4.OBJECT.STRUCTURES.TARGET
) where

import SEL4.MACHINE
import SEL4.API.TYPES
import SEL4.API.FAILURES

import SEL4.OBJECT.STRUCTURES.TARGET

import DATA.TYPEABLE
```

## 6.1.1 Capabilities

This is the type used to represent a capability.

```
data CAPABILITY
  = NULLCAP
  | UNTYPEDCAP {
    capPtr :: PPTR (),
    capBlockSize :: INT }
  | ENDPOINTCAP {
    capEPPtr :: PPTR ENDPOINT,
    capEPBadge :: WORD,
    capEPCanSend, capEPCanReceive :: BOOL,
    capEPCanGrant, capEPCanIdentify :: BOOL }
  | ASYNCENDPOINTCAP {
    capAEPPtr :: PPTR ASYNCENDPOINT,
    capAEPBadge :: WORD,
    capAEPCanSend, capAEPCanReceive :: BOOL }
  | CNODECAP {
    capCNodePtr :: PPTR CTE,
    capCNodeBits :: INT,
    capCNodeRightsMask :: CAPRIGHTS,
    capCNodeGuard :: WORD,
    capCNodeGuardSize :: INT,
    capCNodeCanRead, capCNodeCanModify :: BOOL }
  | THREADCAP {
    capTCBPtr :: PPTR TCB,
    capTCBCanRead, capTCBCanWrite, capTCBCanGrant :: BOOL }
  | FRAMECAP {
    capVPBasePtr :: PPTR WORD,
    capVPSizeBits :: INT,
    capVPCanRead, capVPCanWrite :: BOOL }
  | ARCHOBJECTCAP {
    archCap :: ARCHCAPABILITY }
  | ZOMBIE {
    zombieObject :: EITHER (PPTR CTE, INT) (PPTR TCB),
    zombieBackPtr :: EITHER (PPTR CTE) (PPTR TCB) }
  deriving (SHOW, Eq)
```

## 6.1.2 Kernel Objects

### Synchronous Endpoint

Synchronous endpoints are represented in the physical memory model using the `ENDPOINT` data structure.

```
data ENDPOINT
```

There are three possible states for a synchronous endpoint:

- idle;  
= `IDLEEP`
- waiting for one or more send operations to complete, with a list of pointers to waiting threads;  
| `SENDEP { epQueue :: [PPTR TCB] }`
- or waiting for one or more receive operations to complete, with a list of pointers to waiting threads.  
| `RECVEP { epQueue :: [PPTR TCB] }`  
`deriving (TYPEABLE, SHOW)`

### Asynchronous Endpoints

Asynchronous endpoints are represented in the physical memory model using the `ASYNCENDPOINT` data structure.

```
data ASYNCENDPOINT
```

There are three possible states for an asynchronous endpoint:

- idle;  
= `IDLEAEP`
- waiting for one or more send operations to complete, with an array of thread pointers.  
| `WAITING { aepQueue :: [PPTR TCB] }`
- or active — that is, it has received at least one message. An active asynchronous endpoint contains a data field and a message identifier.  
| `ACTIVEAEP { aepData :: WORD, aepMsgIdentifier :: WORD }`  
`deriving (TYPEABLE, SHOW)`

## Capability Table Entry

Entries in a capability table node (CNode) are represented by the type CTE, an abbreviation of *Capability Table Entry*. Each CTE contains a capability and a mapping database node.

```
data CTE = CTE {
    cteCap :: CAPABILITY,
    cteMDBNode :: MDBNODE }
deriving (TYPEABLE, SHOW)
```

## Thread Control Block

Every thread has a thread control block, allocated by a user-level server but directly accessible only by the kernel.

```
data TCB = THREAD {
```

The TCB is used to store various data about the thread's current state:

- a slot for a capability to the root node of this thread's address space — note that this is the first item in the TCB, allowing a thread pointer to be cast to a CNode pointer by *getThreadRoot*;

```
    tcbCTable :: CTE,
```

- a slot for a capability to the root of the thread's page table — on some architectures, this is a CNode capability, while on others it is a machine-specific type;

```
    tcbVTable :: CTE,
```

- the thread's scheduler state and priority;

```
    tcbState :: THREADSTATE,
    tcbPriority :: PRIORITY,
    tcbQueued :: BOOL,
```

- the amount of time remaining in this thread's timeslice;

```
    tcbTimeSlice :: INT,
```

- a capability pointer to an endpoint which is used for sending system call results to the thread;

```
    tcbResultEndpoint :: CPTR,
```

- a capability pointer to the fault handler endpoint, which receives an IPC from the kernel whenever this thread generates a fault;

```
tcbFaultHandler :: CPTR,
```

- the virtual address of the buffer used for IPC message registers that do not fit in machine registers;

```
tcbIPCBuffer :: VPTR,
```

- and the saved user-level context of the thread.

```
tcbContext :: USERCONTEXT }
deriving (TYPEABLE, SHOW)
```

Each TCB contains two CTE entries. The following constants define the slot numbers in which they will be found if the TCB is treated as a CNode.

```
tcbCTableSlot :: WORD
tcbCTableSlot = 0
```

```
tcbVTableSlot :: WORD
tcbVTableSlot = 1
```

### 6.1.3 Other Types

#### Mapping Database Node

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.

```
data MDBNODE = MDB {
  mdbNext, mdbPrev :: PPTR CTE,
  mdbRevocable :: BOOL }
deriving SHOW
```

The basic structure is a double-linked list. The algorithm used to determine the mapping hierarchy from this list is described in section 6.3.4 on page 96.

The MDB node of the initial CTEs contains two null pointers and has the revocable flag set.

```
initMDBNode :: MDBNODE
initMDBNode = MDB {
  mdbNext = nullPointer,
  mdbPrev = nullPointer,
  mdbRevocable = TRUE }
```

## Thread State

A user thread may be in the following states:

```
data THREADSTATE
```

- ready to start executing the next instruction;  
= RUNNING
- ready to start executing at the current instruction (after a fault, an interrupted system call, or an explicitly set program counter);  
| RESTART
- interrupted during a long-running operation that can be continued immediately without returning to user level;  
| CONTINUE { interruptedOperation :: INTERRUPTEDOPERATION }
- waiting to be explicitly started;  
| INACTIVE
- blocked on a synchronous IPC send or receive (which require the presence of additional data about the operation);  
| BLOCKEDONRECEIVE {  
    blockingIPCEndpoint :: PPTR ENDPOINT,  
    blockingIPCDiminishCaps :: BOOL }  
| BLOCKEDONSEND {  
    blockingIPCEndpoint :: PPTR ENDPOINT,  
    blockingIPCBadge :: WORD,  
    blockingIPCFault :: MAYBE FAULT,  
    blockingIPCCanGrant :: BOOL,  
    pendingReceiveCap :: MAYBE CPTR }
- waiting for a synchronous IPC transfer to occur, after finding another thread to communicate with;  
| WAITINGTORECEIVE {  
    waitingIPCPartner :: PPTR TCB,  
    waitingIPCDiminishCaps :: BOOL }  
| WAITINGTOSEND {  
    waitingIPCPartner :: PPTR TCB,  
    waitingIPCBadge :: WORD,  
    waitingIPCFault :: MAYBE FAULT,  
    waitingIPCCanGrant :: BOOL,

```
pendingReceiveCap :: MAYBE CPTR }
```

- or blocked on an asynchronous notification.

```
| BLOCKEDONASYNC EVENT {  
    waitingOnAsyncEP :: PPTR ASYNCENDPOINT }  
deriving SHOW
```

## Interrupted Operations

This type represents a long-running kernel operation that has been preempted. It contains any information necessary to resume the operation.

```
data INTERRUPTEDOPERATION  
= INTERRUPTEDDESTROY (PPTR CTE)  
deriving SHOW
```

## Scheduler State

This type is used to represent the required action, if any, of the scheduler next time it runs.

```
data SCHEDULERACTION
```

- The normal action of the scheduler before returning to user level is to check that the current thread has a non-zero timeslice and is runnable, and to choose a new thread otherwise.

```
= SCHEDULENORMALLY
```

- IPC operations may request that the scheduler switch to a specific thread.

```
| SWITCHTOTHREAD (PPTR TCB)
```

- If no current thread has been chosen yet, or if the current thread dies by deleting its own TCB, the scheduler must pick a new thread without touching the current TCB.

```
| CURRENTTHREADISINVALID  
deriving (Eq, SHOW)
```

## 6.2 Thread Control Blocks

This module contains the thread control block structure, the TCB invocation operation, and various accessors used by both TCB invocations and the kernel.

```
module SEL4.OBJECT.TCB (
    threadGet, threadSet, asUser,
    getThreadCSpaceRoot, getThreadVSpaceRoot,
    getMRs, setMRs, getMessageInfo, setMessageInfo,
    tcbFaultHandler, tcbResultEndpoint, tcbIPCBuffer, tcbTimeSlice,
    decodeTCBInvocation, invokeTCB
) where

import SEL4.API.TYPES
import SEL4.API.FAILURES
import SEL4.API.INVOCATION
import SEL4.MACHINE
import SEL4.MODEL
import SEL4.OBJECT.STRUCTURES
import SEL4.OBJECT.INSTANCES
import SEL4.OBJECT.CNODE
import SEL4.OBJECT.OBJECTTYPE
import {-# SOURCE #-} SEL4.OBJECT.ENDPOINT
import {-# SOURCE #-} SEL4.KERNEL.THREAD
import {-# SOURCE #-} SEL4.KERNEL.CSPACE
import {-# SOURCE #-} SEL4.KERNEL.VSPACE

import DATA.BITS
import DATA.MAYBE
import CONTROL.MONAD.STATE(STATE, runState)

decodeTCBInvocation :: WORD → [WORD] → CAPABILITY →
    KERNELF SYSCALLERROR TCBINVOCATION
decodeTCBInvocation label args cap =
    case (label `shiftR` 8) .&. 3 of
        0 → decodeExchangeRegisters label args cap
        1 → decodeThreadControl label args cap
        2 → return $! YIELDTO (capTCBPtr cap)
        _ → throw ILLEGALOPERATION
```

The `EXCHANGeregisters` call transfers parts of the user-level context between two different threads, and suspends or resumes each thread. The context is divided into two or

more parts, depending on the architecture. The caller is able to select which parts are copied. The implementation of this call is in section 6.2.1 on page 84.

```
decodeExchangeRegisters :: WORD → [WORD] → CAPABILITY →
    KERNELF SYSCALLERROR TCBINVOCATION
decodeExchangeRegisters label (srcCptr:saveptr:_) cap = do
```

The two lowest bits of the label are used to determine whether the source thread should be suspended and the destination thread should be resumed, respectively. If either bit is not set, the corresponding thread's scheduler state is not affected.

```
let suspendSource = label `testBit` 0
    resumeTarget = label `testBit` 1
```

The next two bits are used to indicate whether the source or destination threads, respectively, should use the save area instead of copying registers directly. If the source or destination is the current thread, then the corresponding bit should be set. If both bits are set, the invocation will return an error.

```
sourceIsCaller = label `testBit` 2
targetIsCaller = label `testBit` 3
```

The four remaining bits may be used to select which subsets of the register set will be copied. The first two are used for the integer registers which must be copied using the save area for the current thread, and those which may be copied directly, respectively. The meanings of the remaining two bits are implementation-defined; they may be used for floating point and vector registers, for example.

```
transferFrame = label `testBit` 4
transferInteger = label `testBit` 5
transferExtra1 = label `testBit` 6
transferExtra2 = label `testBit` 7
```

If the source TCB is needed, then look up the provided capability and check permissions.

```
srcTCB ← if suspendSource ∨ transferInteger ∨
    transferExtra1 ∨ transferExtra1 ∨
    (transferFrame ∧ not sourceIsCaller)
then do
    srccap ← capErrorOnFailure $ lookupCap $ CPTR srcCptr
    case srccap of
        THREADCAP { capTCBPtr = ptr, capTCBCanRead = TRUE } →
            return $! JUST ptr
        _ → throw INVALIDCAPABILITY
    else return NOTHING
```

If a save area is needed, then it must be entirely within a page. The following code calculates the size of the save area, ensures that it fits inside a single page, and locates the physical address of that page.

The format of the save area is architecture-specific; it is defined by `frameRegisters`, in the module `SEL4.MACHINE.REGISTERSET` (section 8.1 on page 140).

```

savepptr ← if transferFrame ∧ (sourceIsCaller ∨ targetIsCaller)
then liftM JUST $ do
  let pageSize = 1 `shiftL` pageBits
  let wordSize = fromIntegral $ bitSize (⊥::WORD) `div` 8
  let frameSize = fromIntegral $ wordSize * length frameRegisters
  when (savepptr `mod` pageSize + frameSize ≥ pageSize) $
    throw ALIGNMENTERROR
  curThread ← withoutFailure getCurThread

  capErrorOnFailure $
    lookupVPtr curThread (VPtr savepptr) targetIsCaller
else return NOTHING

return EXCHANGEREGISTERS {
  exRegsTarget = capTCBPtr cap,
  exRegsSource = srcTCB,
  exRegsSuspendSource = suspendSource,
  exRegsResumeTarget = resumeTarget,
  exRegsSourceFrame = if sourceIsCaller then savepptr else NOTHING,
  exRegsTargetFrame = if targetIsCaller then savepptr else NOTHING,
  exRegsTransferFrame = transferFrame,
  exRegsTransferInteger = transferInteger,
  exRegsTransferExtra1 = transferExtra1,
  exRegsTransferExtra2 = transferExtra2 }

decodeExchangeRegisters _ _ _ = throw TRUNCATEDMESSAGE

```

The `THREADCONTROL` call is used for setting a thread's priority, the locations within its capability space of its fault handler and system call reply endpoints, its capability and virtual address space roots, and the virtual memory location of its IPC buffer. Its implementation is in section 6.2.1 on page 84.

```

decodeThreadControl :: WORD → [WORD] → CAPABILITY →
  KERNELF SYSCALLERROR TCBINVOCAION
decodeThreadControl label args cap = do

```

This call has eight arguments, and the lowest eight bits of the label are used to determine whether the corresponding arguments are valid and should be set in the TCB.

This allows the caller to set a subset of the thread's parameters, without knowing or modifying the others.

```

let tcArg n = if label `testBit` n
  then do
    unless (length args > n) $ throw TRUNCATEDMESSAGE
    return $! JUST (args!!n)
  else return NOTHING

let target = capTCBPtr cap

faultEP ← tcArg 0
resultEP ← tcArg 1

```

Setting the thread's priority is only allowed if the new priority is lower than the current thread's. This prevents untrusted clients that hold untyped or TCB capabilities from performing denial of service attacks by creating new maximum-priority threads. This is a temporary solution; there may be significant changes to the scheduler in future versions to provide better partitioning of CPU time.

```

priority ← do
  newPrio ← tcArg 2
  maybe (return NOTHING) (λprio → do
    curThread ← withoutFailure $ getCurThread
    curPriority ← withoutFailure $ threadGet tcbPriority curThread
    when (fromIntegral prio ≥ curPriority) $
      throw ILLEGALOPERATION
    return $! JUST prio
  ) newPrio

```

Setting the capability space or virtual address space root is similar to a CNode Insert operation, except that any previous root is implicitly deleted rather than causing an error, and the new root must be a valid capability of the appropriate type.

```

cRoot ← do
  cRootPtr ← tcArg 3
  maybe (return NOTHING) (λrootcap → do
    rootdata ← tcArg 4
    (srcSlot, srcMask) ← capErrorOnFailure $
      lookupSlotForCurrentThread $ CPTR rootcap
    srcCap ← withoutFailure $ getSlotCap srcSlot srcMask
    newCap ← deriveCap $ case rootdata of
      JUST w → updateCapData w srcCap
      NOTHING → srcCap
    case newCap of
      CNODECAP {} → return $! JUST (newCap, srcSlot)
      _ → throw INVALIDCAPABILITY
  )

```

```

    ) cRootPtr
vRoot ← do
  vRootPtr ← tcArg 5
  maybe (return NOTHING) (λrootcap → do
    rootdata ← tcArg 6
    (srcSlot , srcMask) ← capErrorOnFailure $
      lookupSlotForCurrentThread $ CPTR rootcap
    srcCap ← withoutFailure $ getSlotCap srcSlot srcMask
    newCap ← deriveCap $ case rootdata of
      JUST w → updateCapData w srcCap
      NOTHING → srcCap
    if isValidVTableRoot newCap
      then return $! JUST (newCap , srcSlot)
      else throw INVALIDCAPABILITY
  ) vRootPtr

```

The IPC buffer is a pointer into the thread’s virtual address space. It must fit inside a single virtual address page, assuming the machine’s smallest supported page size.

```

ipcBuffer ← do
  let intSize = fromIntegral $ bitSize (⊥::WORD) `div` 8
      bufferPtr ← tcArg 7
  maybe (return NOTHING) (λptr → do
    let vptr = VPTR ptr
        bSize = VPTR $ (capTransferDataSize+numberOfMRs)*intSize
        when (vptr `shiftR` pageBits ≠ (vptr+bSize) `shiftR` pageBits)
          $ throw ALIGNMENTERROR
    return $ JUST vptr) bufferPtr
return THREADCONTROL {
  tcThread = target ,
  tcNewFaultEP = liftM CPTR faultEP ,
  tcNewResultEP = liftM CPTR resultEP ,
  tcNewPriority = liftM fromIntegral priority ,
  tcNewCRoot = cRoot ,
  tcNewVRoot = vRoot ,
  tcNewIPCBuffer = ipcBuffer }

```

## 6.2.1 TCB Invocations

The following function is called when a user-level thread invokes a TCB object.

There are four types of thread invocation; they are selected using bits 8 and 9 of the first argument. Bits 0 – 7 are used by ThreadControl and ExchangeRegisters calls to select which operations will be performed.

All these operations, require write permission over the TCB object. In addition, *threadSetCRoot* and *threadSetVRoot* operations require grant permission. Decoding the system call, and checking for appropriate permission(s) is done by the caller (see Section 6.7).

```
invokeTCB :: TCBINVOCATION → KERNELP ()
```

```
invokeTCB (YIELDTo thread) = withoutPreemption $ yieldTo thread
```

## The Thread Control Call

```
invokeTCB (THREADCONTROL target faultep replyep priority cRoot vRoot buffer)
= do
  withoutPreemption $ maybe (return ())
    (λep → threadSet (λt → t {tcbFaultHandler = ep}) target)
    faultep
  withoutPreemption $ maybe (return ())
    (λep → threadSet (λt → t {tcbResultEndpoint = ep}) target)
    replyep
  withoutPreemption $ maybe (return ()) (setPriority target) priority
  maybe (return ()) (λ(newCap, srcSlot) → do
    rootSlot ← withoutPreemption $ getThreadCSpaceRoot target
    cteDelete rootSlot
    withoutPreemption $ cteInsert newCap srcSlot rootSlot)
    cRoot
  maybe (return ()) (λ(newCap, srcSlot) → do
    rootSlot ← withoutPreemption $ getThreadVSpaceRoot target
    cteDelete rootSlot
    withoutPreemption $ cteInsert newCap srcSlot rootSlot)
    vRoot
  withoutPreemption $ maybe (return ())
    (λptr → threadSet (λt → t {tcbIPCBuffer = ptr}) target)
    buffer
```

## The Exchange Registers Call

```
invokeTCB (EXCHANGEREGISTERS dest src suspendSource resumeTarget
  sourceFrame targetFrame transferFrame transferInteger _ _)
= withoutPreemption $ do
```

First, calculate the word size.

```
let wordSize = fromIntegral $ bitSize (⊥::WORD) `div` 8
```

The source is suspended and the destination resumed, if requested.

```

when suspendSource $ do
  ipcCancel (fromJust src)
  setThreadState INACTIVE (fromJust src)
when resumeTarget $ restart dest

```

Determine whether the frame registers are to be copied.

```

when transferFrame $ do

```

If so, there are three possible situations:

```

case (sourceFrame, targetFrame) of

```

- The caller is the source. The frame registers must be copied from the save area provided by the caller.

```

  (JUST frame, NOTHING) → do
    zipWithM_ (λoffset r → do
      v ← doMachineOp $ loadWord $
        frame + PPTR (wordSize * offset)
      asUser dest $ setRegister r v)
    [0,1..] frameRegisters

```

- The caller is the destination. The frame registers must be copied to the save area.

```

  (NOTHING, JUST frame) → do
    zipWithM_ (λoffset r → do
      v ← asUser (fromJust src) $ getRegister r
      doMachineOp $ storeWord
        (frame + PPTR (wordSize * offset)) v)
    [0,1..] frameRegisters

```

- The caller is neither the source or the destination; the frame registers may be copied directly.

```

  (NOTHING, NOTHING) → do
    mapM_ (λr → do
      v ← asUser (fromJust src) $ getRegister r
      asUser dest $ setRegister r v)
    frameRegisters

```

If the frame has been transferred, then the target thread's resume address has been modified. Make this new address the next instruction to be executed.

```

pc ← asUser dest $ getRestartPC
asUser dest $ setNextPC pc

```

Finally, if the other integer registers are to be copied, this is done directly between the threads' saved user contexts.

```

when transferInteger $ do
  mapM_ ( $\lambda r \rightarrow$  do
     $v \leftarrow$  asUser (fromJust src) $ getRegister r
    asUser dest $ setRegister r v)
  gpRegisters

```

At this point, implementations may copy any registers indicated by the two implementation-defined transfer flags. This model does not define any registers corresponding to those flags.

## 6.2.2 Messages

### Message Parameters

The following two functions get and set the message information register for the given thread.

```

setMessageInfo :: PPTR TCB  $\rightarrow$  MESSAGEINFO  $\rightarrow$  KERNEL ()
setMessageInfo thread info = do
  let  $x =$  wordFromMessageInfo info
  asUser thread $ setRegister msgInfoRegister  $x$ 

getMessageInfo :: PPTR TCB  $\rightarrow$  KERNEL MESSAGEINFO
getMessageInfo thread = do
   $x \leftarrow$  asUser thread $ getRegister msgInfoRegister
  return $ messageInfoFromWord  $x$ 

```

### Message Data

The following functions get or set a thread's message data, given a pointer to a TCB and a pointer to the start of the thread's IPC buffer.

These functions assume that the buffer is large enough to store all MRs without crossing any page boundaries.

The `setMRs` function returns the number of words of message data successfully transferred.

```

setMRs :: PPTR TCB  $\rightarrow$  MAYBE (PPTR WORD)  $\rightarrow$  [WORD]  $\rightarrow$  KERNEL WORD
setMRs thread buffer messageData = do
  let intSize = fromIntegral $ bitSize ( $\perp ::$  WORD) `div` 8
  let hardwareMRs = msgRegisters
  let bufferMRs = case buffer of
    JUST bufferPtr  $\rightarrow$ 
      map ( $\lambda x \rightarrow$  bufferPtr +

```

```

        PPtr ((x + capTransferDataSize)*intSize))
    [fromIntegral $ length hardwareMRs .. numberOfMRs - 1]
    NOTHING → []
let msgLength = min
    (length messageData)
    (length hardwareMRs + length bufferMRs)
let mrs = take msgLength messageData
asUser thread $ zipWithM_ setRegister hardwareMRs mrs
doMachineOp $
    zipWithM_ storeWord bufferMRs $ drop (length hardwareMRs) mrs
return $ fromIntegral msgLength

getMRs :: PPtr TCB → MAYBE (PPtr WORD) → MESSAGEINFO →
    KERNEL [WORD]
getMRs thread buffer info = do
    let intSize = fromIntegral $ bitSize (⊥::WORD) `div` 8
        hardwareMRs = msgRegisters
        hardwareMRValues ← asUser thread $ mapM getRegister hardwareMRs
        bufferMRValues ← case buffer of
            JUST bufferPtr → do
                let bufferMRs = map (λx → bufferPtr +
                    PPtr ((x + capTransferDataSize)*intSize))
                    [fromIntegral $ length hardwareMRs .. numberOfMRs - 1]
                doMachineOp $ mapM loadWord bufferMRs
            NOTHING → return []
        let values = hardwareMRValues ++ bufferMRValues
    return $ take (fromIntegral $ msgLength info) values

```

## 6.2.3 TCB Accessors

### Address Space Accesses

This function will return a physical pointer to a thread's root capability table entry, given a pointer to its TCB.

```

getThreadCspaceRoot :: PPtr TCB → KERNEL (PPtr CTE)
getThreadCspaceRoot thread = do
    locateSlot (PPtr $ fromPPtr thread) tcbCTableSlot

```

This function will return a physical pointer to a thread's page table root, given a pointer to its TCB.

```

getThreadVSpaceRoot :: PPtr TCB → KERNEL (PPtr CTE)
getThreadVSpaceRoot thread = do

```

```
locateSlot (PPTR $ fromPPtr thread) tcbVTableSlot
```

## Fetching or Modifying TCB Fields

The following two trivial functions will get or set a given field of a TCB, using a pointer to the TCB.

```
threadGet :: (TCB → a) → PPTR TCB → KERNEL a
threadGet f tptr = liftM f $ getObject tptr

threadSet :: (TCB → TCB) → PPTR TCB → KERNEL ()
threadSet f tptr = do
    tcb ← getObject tptr
    setObject tptr $ f tcb
```

### 6.2.4 User-level Context

Actions performed by user-level code, or by the kernel when modifying the user-level context of a thread, access only the `USERCONTEXT` structure in the thread's TCB.

The following function performs a user-level operation as a specified thread. The user-level operation is represented by a function in the `STATE` monad operating on the thread's `USERCONTEXT` structure.

A typical use of this function is `asUser tcbPtr $ setRegister R0 1`, which stores the value 1 in the register `R0` of to the thread identified by `tcbPtr`.

```
asUser :: PPTR TCB → USERMONAD a → KERNEL a
asUser tptr f = do
    uc ← threadGet tcbContext tptr
    let (a, uc') = runState f uc
    threadSet (λ tcb → tcb { tcbContext = uc' }) tptr
    return a
```

## 6.3 Capability Nodes

This module defines the types and functions related to the kernel objects used to represent a capability space. It contains implementations of the operations that user-level threads can request by invoking a capability table node. It is also responsible for creating the `CAPABILITY` objects used at higher levels of the kernel.

```
module SEL4.OBJECT.CNODE (
```

```

        cteRevoke, cteDelete, cteInsert,
        ensureNoChildren, ensureEmptySlot,
        getSlotCap, locateSlot, getSendCap, getReceiveSlot,
        insertNewCaps, insertInitCap, decodeCNodeInvocation, invokeCNode
    ) where

{--# BOOT-IMPORTS: SEL4.MACHINE SEL4.API.TYPES SEL4.API.FAILURES SEL4.MODEL SEL4.OBJECT.STRUCT
{--# BOOT-EXPORTS: ensureNoChildren getSlotCap locateSlot ensureEmptySlot insertNewCaps cteDel

import SEL4.API.TYPES
import SEL4.API.FAILURES
import SEL4.API.INVOCATION
import SEL4.MACHINE
import SEL4.MODEL
import SEL4.OBJECT.STRUCTURES
import SEL4.OBJECT.INSTANCES
import SEL4.OBJECT.OBJECTTYPE
import {--# SOURCE #-} SEL4.KERNEL.CSPACE
import {--# SOURCE #-} SEL4.KERNEL.THREAD

import DATA.MAYBE
import DATA.BITS

```

### 6.3.1 Capability Node Object Invocations

The following function decodes a CNode invocation message, and checks for any error conditions.

```

decodeCNodeInvocation :: WORD → [WORD] → CAPABILITY →
    KERNELF SYSCALLERROR CNODEINVOCATION
decodeCNodeInvocation label (index:bits:args)
    cap@(CNODECAP {capCNodeCanModify=TRUE})
    = do

```

All CNode operations require the caller to specify a slot in the capability space mapped by the tree whose root is the invoked node, by providing a pointer in the space defined by the tree and the depth of the tree.

The lookup can be terminated early, and therefore operate on a capability at one of the intermediate levels, by reducing the specified depth and shifting the address to the right by the corresponding number of bits. However, the depth *must* be that of a

CNode existing in the specified region of the tree; if not, the operation will fail with a `MISSINGCAPABILITYERROR`.

```
(destSlot, _) ← lookupSlotForCNodeOp FALSE TRUE
  cap (CPTR index) (fromIntegral bits)

case args of
```

If bit 2 of the label is set, then the operation is a revoke or delete operation, with no additional arguments.

```
- | label `testBit` 2 ∧ label `testBit` 0 →
  return $! DELETE destSlot
- | label `testBit` 2 →
  return $! REVOKE destSlot
```

Otherwise, the operation creates a new capability, and must specify a source slot, in the same manner as the destination.

```
(srcCNode:srcIndex:srcDepth:args) → do
```

For these operations, the destination slot must be empty, so a new capability can be created in it.

```
ensureEmptySlot destSlot
```

The source capability root is looked up as if it was a capability being invoked. If it is invalid, an invalid capability error will be thrown rather than faulting.

```
srcRootCap ← capErrorOnFailure $ lookupCap $ CPTR srcCNode
(srcSlot, srcMask) ←
  lookupSlotForCNodeOp TRUE FALSE srcRootCap
  (CPTR srcIndex) (fromIntegral srcDepth)
```

The source slot must contain a valid capability.

```
srcCTE ← withoutFailure $ getCTE srcSlot
when (cteCap srcCTE = NULLCAP) $
  throw $ FAILEDLOOKUP TRUE $ MISSINGCAPABILITY {
    missingCapBitsLeft = fromIntegral srcDepth }
```

If bit 1 is set, this is a move operation, and no rights mask is specified; if bit 0 is set for either move or insert operations, then a new capability data word is specified.

```
let isMove = label `testBit` 1
let isMint = label `testBit` 0

(rights, capData) ←
  case (isMove, isMint, args) of
  (TRUE, FALSE, _) →
    return $! (allRights, NOTHING)
  (TRUE, TRUE, newData:_) →
```

```

    return $! (allRights, JUST newData)
(FALSE, FALSE, rights:_) →
    return $! (rightsFromWord rights, NOTHING)
(FALSE, TRUE, rights:newData:_) →
    return $! (rightsFromWord rights, JUST newData)
_ → throw TRUNCATEDMESSAGE

```

The rights and capability data word are applied to the source capability to create a new capability.

```

let srcCap = maskCapRights rights $
    maskCapRights srcMask $ cteCap srcCTE
newCap ← deriveCap $ case capData of
    JUST w → updateCapData w srcCap
    NOTHING → srcCap
when (newCap = NULLCAP) $ throw ILLEGALOPERATION

```

If a capability to untyped memory is being copied, it must not have any existing MDB children. This is to prevent situations in which the same untyped memory can be re-typed into objects of two different types, which would allow user-level tasks direct access to kernel data structures.

```

when (not isMint) $ case newCap of
    UNTYPEDCAP {} → ensureNoChildren srcSlot
    _ → return ()

return $!
    (if isMove then MOVE else INSERT) newCap srcSlot destSlot

```

The function `invokeCNode` dispatches an invocation to one of the handlers defined below, given a `CNODEINVOCATION` object.

```

invokeCNode :: CNODEINVOCATION → KERNELP ()

invokeCNode (REVOKE destSlot) = cteRevoke destSlot

invokeCNode (DELETE destSlot) = cteDelete destSlot

invokeCNode (INSERT cap srcSlot destSlot) =
    withoutPreemption $ cteInsert cap srcSlot destSlot

invokeCNode (MOVE cap srcSlot destSlot) =
    withoutPreemption $ cteMove cap srcSlot destSlot

```

### 6.3.2 CNode Operations

The following functions define the operations that can be performed by a CNode invocation.

## Inserting New Capabilities

Insertion of new capabilities copied from existing capabilities is performed by `cteInsert`. The parameters are the physical addresses of the source and destination slots, and the capability to be inserted in the destination slot. This function requires the destination slot to be empty, and assumes that the provided capability is one that may be derived from the contents of the source slot.

```
cteInsert :: CAPABILITY → PPTR CTE → PPTR CTE → KERNEL ()
cteInsert newCap srcSlot destSlot = do
```

First, fetch the capability table entry for the source.

```
srcCTE ← getCTE srcSlot
let srcMDB = cteMDBNode srcCTE
    let srcCap = cteCap srcCTE
```

The new capability is revocable if, and only if, the new capability is an endpoint capability and isn't the same as the original. This, along with special treatment of endpoint capabilities in `isMDBParentOf`, allows endpoint capabilities with a specific badge to be revoked. This applies to both synchronous and asynchronous endpoints.

```
let newCapIsRevocable = case newCap of
    ENDPOINTCAP {} → newCap ≠ srcCap
    ASYNCENDPOINTCAP {} → newCap ≠ srcCap
    _ → FALSE
```

Create the new capability table entry. Its `MDBNODE` is inserted in the mapping database immediately after the existing capability.

```
let newMDB = srcMDB {
    mdbPrev = srcSlot ,
    mdbRevocable = newCapIsRevocable }
let newCTE = CTE newCap newMDB
```

The destination slot must be empty.

```
oldCTE ← getCTE destSlot
when (cteCap oldCTE ≠ NULLCAP) $
    error 'cteInsert to non-empty destination'
```

Store the new entry in the destination slot and update the mapping database.

```
setCTE destSlot newCTE
updateMDB srcSlot (λm → m { mdbNext = destSlot })
updateMDB (mdbNext newMDB) (λm → m { mdbPrev = destSlot })
```

## Moving Capabilities

```
cteMove :: CAPABILITY → PPTR CTE → PPTR CTE → KERNEL ()
cteMove newCap srcSlot destSlot = do
```

Move the CTE into the new slot and update the mapping database.

```
    cte ← getCTE srcSlot
    let mdb = cteMDBNode cte
        setCTE destSlot $ cte { cteCap = newCap }
        setCTE srcSlot makeObject
        updateMDB (mdbPrev mdb) (λm → m { mdbNext = destSlot })
        updateMDB (mdbNext mdb) (λm → m { mdbPrev = destSlot })
```

## Revoking and Deleting Capabilities

The work for the revoke operation is done by `cteRevoke`, which revokes all capabilities which are children of the invoked capability in the derivation tree.

```
cteRevoke :: PPTR CTE → KERNELP ()
cteRevoke slot = do
    thread ← withoutPreemption getCurThread
```

Load the CTE being revoked.

```
    cte ← withoutPreemption $ getCTE slot
    let mdb = cteMDBNode cte
        let nextPtr = mdbNext $ cteMDBNode cte
```

Determine whether the CTE is valid, and has children to be deleted.

```
    child ← withoutPreemption $ case () of
    - | cteCap cte = NULLCAP → return NOTHING
      | nextPtr = nullPointer → return NOTHING
      | otherwise → do
        nextCTE ← getCTE nextPtr
        return $! if (cte `isMDBParentOf` nextCTE)
            then JUST nextPtr else NOTHING
```

If a child was found, then it is deleted, and we look for another; otherwise the revocation is complete. There is a preemption point immediately after each capability deletion.

```
    case child of
    NOTHING → return ()
    JUST child → do
        cteDelete child
        preemptionPoint
```

```
cteRevoke slot
```

This function deletes the capability in a given slot. If it is the last remaining capability for the given object, the object will be destroyed.

```
cteDelete :: PPTR CTE → KERNELP ()
cteDelete slot = do
```

Load the contents of the slot.

```
cte ← withoutPreemption $ getCTE slot
let mdbNode = cteMDBNode cte
let prev = mdbPrev mdbNode
let next = mdbNext mdbNode
```

Check that the CTE is valid. If it isn't, nothing needs to be done.

```
unless (cteCap cte = NULLCAP) $ do
  detype ← withoutPreemption $ do
```

Perform any object type specific operations required by the capability deletion, such as flushing caches or keeping other kernel structures consistent.

```
deleteCap (cteCap cte) slot
```

Invalidate the slot's contents and remove it from the mapping database.

```
setCTE slot makeObject
updateMDB prev (λmdb → mdb { mdbNext = next })
updateMDB next (λmdb → mdb { mdbPrev = prev })
```

Determine whether the object should be detyped.

```
isFinalCapability cte
```

Detype the object. This destroys any references to it that exist elsewhere in the kernel, and also deletes any capabilities stored in it. This must be done *after* invalidating the contents of the slot, to avoid infinite loops when the object is a CNode that contains the last capability to itself.

```
if detype then detypeObject (cteCap cte) else return ()
```

### 6.3.3 Object Creation

Create a set of new capabilities (and possibly the objects backing them) and insert them in given empty slots. The required parameters are an object type, a pointer to the source capability's slot, a list of pointers to empty slots, and the expected size of the created capabilities (as a power of two). The result is the number of capabilities created.

```
insertNewCaps :: OBJECTTYPE → PPTR CTE → [PPTR CTE] → INT → KERNEL WORD
insertNewCaps newType srcSlot destSlots sizeBits = do
```

Fetch the source CTE and its MDB.

```
srcCTE ← getCTE srcSlot
let srcMDB = cteMDBNode srcCTE
```

Create the new objects.

```
let (regionPtr, regionSize) = case cteCap srcCTE of
    UNTYPEDCAP { capPtr = p, capBlockSize = s } → (p, s)
    _ → error "createCaps: source is not a valid untyped cap"
caps ← createNewCaps newType regionPtr regionSize sizeBits
```

Calculate the pointers for the slots to be used, and their MDB next and previous pointers.

```
let slotPtrs = take (length caps) destSlots
let prevPtrs = srcSlot : destSlots
let nextPtrs = tail slotPtrs ++ [mdbNext srcMDB]
```

Create new capabilities in the destination slots.

```
let ctePtrs = zip3 slotPtrs prevPtrs nextPtrs
zipWithM_
    (λ(slot, prev, next) cap →
        setCTE slot $ CTE cap $ MDB prev next TRUE)
    ctePtrs caps
```

Link the new capabilities to the rest of the MDB.

```
updateMDB srcSlot (λmdb → mdb { mdbNext = head slotPtrs })
updateMDB (mdbNext srcMDB) (λmdb → mdb { mdbPrev = last slotPtrs })
```

Return the number of capabilities inserted.

```
return $ fromIntegral $ length slotPtrs
```

The following function is used by the bootstrap code to create the initial set of capabilities.

```
insertInitCap :: PPTR CTE → CAPABILITY → KERNEL ()
insertInitCap slot cap = setCTE slot $ CTE cap initMDBNode
```

## 6.3.4 Helper Functions

### MDB Operations

The Mapping Database (MDB) is used to keep track of the derivation hierachy of seL4 capabilities, so all existing capabilities to an object can be revoked before that object is reused or deleted. A similar structure is used in L4Ka::Pistachio [3] to support that kernel's Unmap operation.

The MDB is a double-linked list that is equivalent to a prefix traversal of the derivation tree. It is possible to compare two capabilities to determine whether one is an ancestor of the other in the derivation tree.

Given two capabilities  $a$  and  $b$ , which appear in the MDB in that order, the following rules determine whether  $b$  is a child of  $a$  and should be deleted when `REVOKE` is performed on  $a$ :

`isMDBParentOf :: CTE → CTE → BOOL`

- If capability  $a$  is Untyped, then  $b$  is a child of  $a$  if it points to an object in the region covered by  $a$ .

`isMDBParentOf (CTE a@(UNYPEDCAP { }) _) (CTE b _) = a `sameRegionAs` b`

- Otherwise,  $a$  points to a typed object. To be the parent of  $b$ , it must have the `mdbRevocable` bit set and must point to the same object as  $b$ .

`isMDBParentOf (CTE a mdbA) (CTE b _)`  
| `not $ mdbRevocable mdbA = FALSE`  
| `not $ a `sameRegionAs` b = FALSE`

- If  $a$  is an endpoint capability (synchronous or asynchronous) with no badge set, then it is the parent of  $b$ .

`isMDBParentOf (CTE (ENDPOINTCAP { capEPBadge = 0 }) _) _ = TRUE`  
`isMDBParentOf (CTE (ASYNCENDPOINTCAP { capAEPBadge = 0 }) _) _ = TRUE`

- Otherwise, if  $a$  is an endpoint capability (with a badge set), then it is the parent of  $b$  if  $b$  has the same badge and has the `mdbRevocable` bit clear.

`isMDBParentOf (CTE a@(ENDPOINTCAP { }) _) (CTE b mdbB)`  
= `(capEPBadge a = capEPBadge b) ∧ (not $ mdbRevocable mdbB)`  
`isMDBParentOf (CTE a@(ASYNCENDPOINTCAP { }) _) (CTE b mdbB)`  
= `(capAEPBadge a = capAEPBadge b) ∧ (not $ mdbRevocable mdbB)`

- Otherwise, the object is not an endpoint, and  $a$  is the parent of  $b$ .

`isMDBParentOf _ _ = TRUE`

In several of the functions in this module, it is necessary to modify the MDB node in a CTE without changing anything else in the CTE, and to skip the operation if the CTE pointer is null. The following function is a helper function used to do so.

```
updateMDB :: PPtr CTE → (MDBNode → MDBNode) → KERNEL ()
updateMDB 0 f = return ()
updateMDB slot f = do
    cte ← getCTE slot
    let mdb = cteMDBNode cte
        mdb' = f mdb
        let cte' = cte { cteMDBNode = mdb' }
            setCTE slot cte'
```

## Error Checking

Before retyping an untyped memory object, it is necessary to check that the object contains no existing objects.

```
ensureNoChildren :: PPtr CTE → KERNELF SYSCALLERROR ()
ensureNoChildren slot = do
    cte ← withoutFailure $ getCTE slot
    when (mdbNext (cteMDBNode cte) ≠ nullPointer) $ do
        next ← withoutFailure $ getCTE (mdbNext $ cteMDBNode cte)
        when (cte `isMDBParentOf` next) $ throw REVOKEFIRST
```

When creating or deriving new capabilities, the destination slot must be empty.

```
ensureEmptySlot :: PPtr CTE → KERNELF SYSCALLERROR ()
ensureEmptySlot slot = do
    cte ← withoutFailure $ getCTE slot
    when (cteCap cte ≠ NULLCAP) $ throw DELETEFIRST
```

## 6.3.5 Accessing Capabilities

### Locating Slots

This function is used for locating a slot at a given offset in a CNode.

```
locateSlot :: PPtr CTE → WORD → KERNEL (PPtr CTE)
locateSlot cnode offset = do
    let slotSize = 1 `shiftL` objBits (⊥::CTE)
        return $ PPtr $ fromPPtr $ cnode + PPtr (slotSize * offset)
```

## Loading and Storing Entries

The following two functions are specialisations of `getObject` and `setObject` for the capability table entry object and pointer types.

```
getCTE :: PPtr CTE → KERNEL CTE
getCTE = getObject

setCTE :: PPtr CTE → CTE → KERNEL ()
setCTE = setObject
```

## Reading Capabilities

At higher levels of the kernel model (`SEL4.KERNEL` and `SEL4.API`), capabilities are represented by the abstract `CAPABILITY` type, rather than the `CTE` type that is used to store them in memory.

The following functions are used to extract capabilities, in the form of `CAPABILITY` objects, from the CTEs that store them.

```
getSlotCap :: PPtr CTE → CAPRIGHTS → KERNEL CAPABILITY
getSlotCap ptr rightsMask = do
  cte ← getCTE ptr
  return $! rightsMask `maskCapRights` cteCap cte
```

## Testing Capabilities

When deleting a capability, it is necessary to determine whether it is the only existing capability to access a typed object it refers to. When an object's last remaining capability is deleted, the kernel must clean up any internal references it has to the object (in the scheduler's ready queues, the mapping database, and so on). The following function tests a capability to determine whether it is the last.

```
isFinalCapability :: CTE → KERNEL BOOL
isFinalCapability (CTE { cteCap = UNTYPEDCAP {} }) = return FALSE
isFinalCapability cte@(CTE { cteMDBNode = mdb }) = do
  prevIsSameObject ← if mdbPrev mdb = nullPointer
  then return FALSE
  else do
    prev ← getCTE (mdbPrev mdb)
    return $! case cteCap prev of
      UNTYPEDCAP {} → FALSE
      _ → sameRegionAs (cteCap prev) (cteCap cte)
  if prevIsSameObject
```

```

then return FALSE
else if mdbNext mdb = nullPointer
  then return TRUE
  else do
    next ← getCTE (mdbNext mdb)
    return $ not $ sameRegionAs (cteCap cte) (cteCap next)

```

### 6.3.6 Capability Transfers

The following functions locate the send and receive slots for a capability transfer, and ensure that the transfer can be safely completed.

```

getSendCap :: MESSAGEINFO → PPTR TCB → MAYBE (PPTR WORD) →
  KERNEL (MAYBE (CAPABILITY, PPTR CTE))
getSendCap info thread (JUST buffer) = do

```

First, load the send parameters from the sending thread's IPC buffer.

```

let intSize = fromIntegral $ bitSize (⊥::WORD) `div` 8
let wordPtrs = map (λx → buffer + PPTR (x*intSize))
  [0 .. capTransferDataSize - 1]
words ← doMachineOp $ mapM loadWord wordPtrs
let ct = capTransferFromWords words

```

Look up the slot of the capability being sent.

```

nothingOnFailure $ do
  let cptr = ctSendCNode ct
      cnode ← capErrorOnFailure $ lookupCapForThread thread cptr
      (slot, rights) ← lookupSlotForCNodeOp TRUE FALSE
      cnode (ctSendIndex ct) (ctSendDepth ct)

```

If the lookup found a slot, then determine whether the capability in it is valid; if so, determine whether the operation is permitted.

```

cte ← withoutFailure $ getCTE slot
let srcCap = maskCapRights rights $
  maskCapRights (ctSendMask ct) $ cteCap cte
newCap ← deriveCap $ case ctSendData ct of
  JUST w → updateCapData w srcCap
  NOTHING → srcCap
return $ if newCap = NULLCAP
  then NOTHING
  else JUST (newCap, slot)

getSendCap _ _ NOTHING = return NOTHING

```

```

getReceiveSlot :: PPTR TCB → MAYBE (PPTR WORD) → KERNEL (MAYBE (PPTR CTE))
getReceiveSlot thread (JUST buffer) = do

```

Load the receive parameters from the receiving thread's IPC buffer.

```

    let intSize = fromIntegral $ bitSize (⊥::WORD) `div` 8
        wordPtrs = map (λx → buffer + PPTR (x*intSize))
                      [0 .. capTransferDataSize - 1]
        words ← doMachineOp $ mapM loadWord wordPtrs
        let ct = capTransferFromWords words

```

Look up the specified root CNode, and then the slot at the given index and depth in that CNode's tree. Fail if there is already a valid capability in it, to avoid a potentially slow revocation during the IPC operation. Any faults or errors occurring during the lookup operations are caught here.

```

    nothingOnFailure $ do
        let cptr = ctReceiveCNode ct
            cnode ← capErrorOnFailure $ lookupCapForThread thread cptr
            (slot, _) ← lookupSlotForCNodeOp FALSE TRUE
                      cnode (ctReceiveIndex ct) (ctReceiveDepth ct)
            cte ← withoutFailure $ getCTE slot
            when (cteCap cte ≠ NULLCAP) $ throw REVOKEFIRST
            return $ JUST slot

```

```

getReceiveSlot _ NOTHING = return NOTHING

```

## 6.4 Synchronous Endpoints

This module specifies the contents and behaviour of a synchronous IPC endpoint.

```

module SEL4.OBJECT.ENDPOINT (
    sendIPC, receiveIPC, performReceivePhase,
    listenForReply, sendResultIPC,
    ipcCancel, epCancelAll
) where

{--# BOOT-IMPORTS: SEL4.MODEL SEL4.MACHINE SEL4.API.FAILURES SEL4.OBJECT.STRUCTURES #-}
{--# BOOT-EXPORTS: sendIPC ipcCancel epCancelAll #-}

import SEL4.API.TYPES
import SEL4.API.FAILURES
import SEL4.MACHINE
import SEL4.MODEL

```

```

import SEL4.OBJECT.STRUCTURES
import SEL4.OBJECT.INSTANCES
import SEL4.OBJECT.TCB
import SEL4.OBJECT.ASYNCENDPOINT
import {—# SOURCE #—} SEL4.KERNEL.THREAD
import {—# SOURCE #—} SEL4.KERNEL.CSPACE
import {—# SOURCE #—} SEL4.KERNEL.VSPACE
import {—# SOURCE #—} SEL4.KERNEL.FAULTHANDLER

import DATA.BITS
import DATA.LIST

```

### 6.4.1 Sending IPC

This function performs an IPC send operation, given a pointer to the sending thread, a capability to an endpoint, and possibly a fault that should be sent instead of a message from the thread.

```

sendIPC :: BOOL → MAYBE FAULT → WORD → BOOL → PPTR TCB →
         PPTR ENDPOINT → KERNEL ()

```

The normal (blocking) version of the send operation will remove a recipient from the endpoint's queue if one is available, or otherwise add the sender to the queue.

```

sendIPC TRUE fault badge canGrant thread ep_ptr = do
  ep ← getEndpoint ep_ptr
  case ep of

```

If the endpoint is idle, then it becomes a sending endpoint, with the current thread in its queue. The thread is blocked.

```

  IDLEP → do
    setThreadState (BLOCKEDONSEND {
      blockingIPCEndpoint = ep_ptr,
      blockingIPCBadge = badge,
      blockingIPCFault = fault,
      blockingIPCCanGrant = canGrant,
      pendingReceiveCap = NOTHING }) thread
    setEndpoint ep_ptr $ SENDEP [thread]

```

If the endpoint is already a sending endpoint, the current thread is blocked and added to the queue.

```

  SENDEP queue → do
    setThreadState (BLOCKEDONSEND {

```

```

        blockingIPCEndpoint = epptr ,
        blockingIPCBadge = badge ,
        blockingIPCFault = fault ,
        blockingIPCCanGrant = canGrant ,
        pendingReceiveCap = NOTHING }) thread
setEndpoint epptr $ SENDEP $ thread:queue

```

If the endpoint is a receiving endpoint, a thread is removed from its queue, and both that thread and the current thread wait for the transfer to occur. In most cases the transfer will begin when the current thread starts running again, but the threads may remain in the waiting state while other threads run. The transfer will complete as soon as one of the partners is scheduled to run.

If the recipient is the last thread in the endpoint's queue, the endpoint becomes idle.

```

RCVPEP (dest:queue) → do
  setEndpoint epptr $ case queue of
    [] → IDLEEP
    _ → RCVPEP queue
  recvState ← getThreadState dest
  setThreadState (WAITINGTORECEIVE {
    waitingIPCPartner = thread ,
    waitingIPCDiminishCaps =
      blockingIPCDiminishCaps recvState }) dest
  setThreadState (WAITINGTOSEND {
    waitingIPCPartner = dest ,
    waitingIPCBadge = badge ,
    waitingIPCFault = fault ,
    waitingIPCCanGrant = canGrant ,
    pendingReceiveCap = NOTHING }) thread

```

Non-blocking IPC will either transfer the message immediately (without going via the waiting state), or silently drop it. Fault IPC is always blocking, so this assumes there is no fault to be transferred.

```

sendIPC FALSE NOTHING badge canGrant thread epptr = do
  ep ← getEndpoint epptr
  case ep of
    RCVPEP (dest:queue) → do

```

First, look up the sender's IPC buffer. If this lookup fails, the IPC will be aborted.

```

senderIPCBuffer ← lookupIPCBuffer FALSE thread

```

If there are no more threads in the queue, the endpoint becomes idle; otherwise it is waiting for the next thread in the queue to complete its receive operation.

```

setEndpoint epptr $ case queue of

```

```

[] → IDLEEP
_ → RECVEP queue

```

Locate the destination thread's buffer and perform the transfer.

```

destIPCBuffer ← lookupIPCBuffer TRUE dest
destState ← getThreadState dest

```

```

doMRTransfer badge thread senderIPCBuffer dest destIPCBuffer
             canGrant (blockingIPCDiminishCaps destState)

```

The destination thread is unblocked and made the current thread.

```

setThreadState RUNNING dest
setSchedulerAction $ SWITCHTOTHREAD dest
_ → return ()

```

Non-blocking IPC cannot be used to send a fault.

```

sendIPC FALSE (JUST _) _ _ _ = error "Non-blocking IPC used for fault"

```

## 6.4.2 Receiving IPC

The IPC receive operation is essentially the same as the send operation, but with the send and receive states swapped. There are a few other differences: the badge must be retrieved from the TCB when completing an operation, and is not set when adding a TCB to the queue.

```

receiveIPC :: PPTR TCB → CAPABILITY → KERNEL ()

receiveIPC thread cap@(ENDPOINTCAP {}) = do
  let epptr = capEPPtr cap
      ep ← getEndpoint epptr
      let diminish = not $ capEPCanSend cap
      case ep of
        IDLEEP → do
          setThreadState (BLOCKEDONRECEIVE {
            blockingIPCEndpoint = epptr,
            blockingIPCDiminishCaps = diminish }) thread
          setEndpoint epptr $ RECVEP [thread]
        RECVEP queue → do
          setThreadState (BLOCKEDONRECEIVE {
            blockingIPCEndpoint = epptr,
            blockingIPCDiminishCaps = diminish }) thread
          setEndpoint epptr $ RECVEP $ thread:queue
        SENDEP (sender:queue) → do

```

```

senderState ← getThreadState sender
setEndpoint epPtr $ case queue of
  [] → IDLEEP
  _ → SENDEP queue
setThreadState (WAITINGTOSEND {
  waitingIPCPartner = thread,
  waitingIPCBadge = blockingIPCBadge senderState,
  waitingIPCFault = blockingIPCFault senderState,
  waitingIPCCanGrant = blockingIPCCanGrant senderState,
  pendingReceiveCap = pendingReceiveCap senderState }
) sender
setThreadState (WAITINGTORECEIVE {
  waitingIPCPartner = sender,
  waitingIPCDiminishCaps = diminish }) thread

```

### 6.4.3 The Receive Phase of SendWait

The following function is called to complete an atomic send and receive operation after the send phase finishes.

```

performReceivePhase :: PPTR TCB → CPTR → KERNELF FAULT ()
performReceivePhase thread epCPtr = do
  epCap ← capFaultOnFailure epCPtr TRUE $
    lookupCapForThread thread epCPtr
  case epCap of
    ENDPOINTCAP { capEPCanReceive=TRUE } → withoutFailure $
      receiveIPC thread epCap
    ASYNCENDPOINTCAP { capAEPCanReceive=TRUE } → withoutFailure $
      receiveAsyncIPC thread epCap
    _ → throw $ CAPFAULT epCPtr TRUE $
      MISSINGCAPABILITY { missingCapBitsLeft = 0 }

```

### 6.4.4 Kernel Invocation Replies

The following function is used during the atomic send and receive system call, to listen for a reply after invoking an object.

```

listenForReply :: PPTR TCB → CPTR → KERNELF FAULT ()
listenForReply thread cptr = do
  waitState ← withoutFailure $ getThreadState thread
  case waitState of
    RUNNING → performReceivePhase thread cptr
    WAITINGTOSEND { pendingReceiveCap = NOTHING } →

```

```

withoutFailure $ setThreadState
  (waitState { pendingReceiveCap = JUST cptr })
  thread
BLOCKEDONSEND { pendingReceiveCap = NOTHING } →
  withoutFailure $ setThreadState
    (waitState { pendingReceiveCap = JUST cptr })
    thread
_ → error ‘‘Invalid□thread□state□in□listenForReply’’

```

When an invocation is handled by the kernel and must return a result IPC, the following function is called to do so. Any errors cause the send to silently fail.

```

sendResultIPC :: PPTR TCB → (MAYBE (WORD, [WORD])) → KERNEL ()
sendResultIPC _ NOTHING = return ()
sendResultIPC thread (JUST (resultLabel, resultData)) = do
  resultEP ← threadGet tcbResultEndpoint thread
  epCap ← nullCapOnFailure $ lookupCap resultEP
  sendKernelIPC epCap resultLabel resultData

```

An IPC from the kernel is a non-blocking IPC that has no sender thread, and always fails silently if the capability is invalid or there is no thread waiting on the destination endpoint. The message registers are explicitly specified rather than coming from the sender’s context.

```

sendKernelIPC :: CAPABILITY → WORD → [WORD] → KERNEL ()
sendKernelIPC cap@(ENDPOINTCAP {{}}) label mrs = do
  let epptr = capEPPtr cap
      badge = capEPBadge cap
      ep ← getEndpoint epptr
  case ep of
    RECVEP (dest:queue) → do
      setEndpoint epptr $ case queue of
        [] → IDLEEP
        _ → RECVEP queue
      setThreadState RUNNING dest

```

The message is transferred by copying the register values directly into the destination thread’s state.

```

destIPCBuffer ← lookupIPCBuffer TRUE dest
asUser dest $ setRegister badgeRegister badge
len ← setMRs dest destIPCBuffer mrs
let msgInfo = MI {
  msgLength = len,
  msgCapTransfer = FALSE,
  msgBlockingSend = FALSE,
  msgLabel = label }

```

```

        setMessageInfo dest msgInfo
    _ → return ()

sendKernelIPC _ _ _ = return ()

```

## 6.4.5 Cancelling IPC

If a thread is waiting for an IPC operation, it may be necessary to move the thread into a state where it is no longer waiting; for example if the thread is deleted. The following function either cancels or forces immediate completion of an IPC operation, given a pointer to the thread performing it.

```

ipcCancel :: PPtr TCB → KERNEL ()
ipcCancel tptr = do
    state ← getThreadState tptr
    case state of

```

If the affected thread is ready to send, we complete the send immediately, making both the thread and its partner runnable. Any pending receive phase will be removed first, to prevent the thread blocking on it.

```

    WAITINGTOSEND {} → do
        setThreadState (state { pendingReceiveCap = NOTHING }) tptr
        doIPCtransfer tptr (waitingIPCpartner state)

```

If the affected thread is ready to receive, then again, we complete the IPC. This may either put the thread's partner into its receive phase, or make it runnable.

Note that an obvious optimisation here is to not actually transfer the registers, if this call to `ipcCancel` is being caused by a thread deletion.

```

    WAITINGTORECEIVE {} →
        doIPCtransfer (waitingIPCpartner state) tptr

```

Threads blocked waiting for endpoints will simply be removed from the endpoint queue.

```

    BLOCKEDONSEND {} → blockedIPCCancel state
    BLOCKEDONRECEIVE {} → blockedIPCCancel state
    BLOCKEDONASYNC EVENT {} → asyncIPCCancel tptr (waitingOnAsyncEP state)
    _ → return ()
where

```

If the thread is blocking on an endpoint, then the endpoint is fetched and the thread removed from its queue.

```

    blockedIPCCancel state = do
        let ep_ptr = blockingIPCendpoint state

```

```

    ep ← getEndpoint epptr
    let queue' = delete tptr $ epQueue ep
    ep' ← case queue' of
        [] → return IDLEEP
        _ → return $ ep { epQueue = queue' }
    setEndpoint epptr ep'

```

Finally, replace the IPC block with a fault block (which will retry the operation if the thread is resumed).

```

    setThreadState INACTIVE tptr

```

If an endpoint is deleted, then every pending IPC operation using it must be cancelled.

```

epCancelAll :: PPTR ENDPOINT → KERNEL ()
epCancelAll epptr = do
    ep ← getEndpoint epptr
    case ep of
        IDLEEP →
            return ()
        _ → do
            mapM_ (setThreadState INACTIVE) $ epQueue ep
            setEndpoint epptr IDLEEP

```

## 6.4.6 Accessing Endpoints

The following two functions are specialisations of `getObject` and `setObject` for the endpoint object and pointer types.

```

getEndpoint :: PPTR ENDPOINT → KERNEL ENDPOINT
getEndpoint = getObject

setEndpoint :: PPTR ENDPOINT → ENDPOINT → KERNEL ()
setEndpoint = setObject

```

## 6.5 Asynchronous Endpoints

This module specifies the behavior of asynchronous IPC endpoints.

```

module SEL4.OBJECT.ASYNCENDPOINT (
    sendAsyncIPC, receiveAsyncIPC,
    aepCancelAll, asyncIPCCancel

```

) where

```
{-# BOOT-IMPORTS: SEL4.MODEL SEL4.MACHINE SEL4.OBJECT.STRUCTURES #-}  
{-# BOOT-EXPORTS: sendAsyncIPC aepCancelAll #-}
```

```
import SEL4.API.TYPES  
import SEL4.MACHINE  
import SEL4.MODEL  
import SEL4.OBJECT.STRUCTURES  
import SEL4.OBJECT.INSTANCES  
import SEL4.OBJECT.TCB
```

```
import {-# SOURCE #-} SEL4.KERNEL.THREAD  
import {-# SOURCE #-} SEL4.KERNEL.CSPACE
```

```
import DATA.BITS  
import DATA.LIST
```

## 6.5.1 Sending Messages

This function performs an asynchronous IPC send operation, given a capability to an asynchronous endpoint, and a single machine word of message data. This operation will never block the sending thread.

```
sendAsyncIPC :: PPtr ASYNCEMPOINT → WORD → WORD → KERNEL ()
```

```
sendAsyncIPC aeptr badge val = do
```

Fetch the asynchronous endpoint object, and select the operation based on its state.

```
aEP ← getAsyncEP aeptr  
case aEP of
```

If the asynchronous endpoint is idle, store the badge and the value, and then mark the endpoint as active.

```
  IDLEAEP → do  
    setAsyncEP aeptr $ ACTIVEAEP badge val
```

If the asynchronous endpoint is waiting, a thread is removed from its queue and is marked as waiting for the transfer to occur. The transfer will take place only when the receiver is scheduled.

```
  WAITING (dest:queue) → do
```

```

setAsyncEP aepptr $ case queue of
  [] → IDLEAEP
  _ → WAITING queue
doAsyncTransfer badge val dest

```

If the endpoint is active, new values are calculated and stored in the endpoint. The calculation is done by a bitwise OR operation of the currently stored, and the newly sent values.

```

ACTIVEAEP badge' val' → do
  let newVal = val .—. val'
  let newBadge = badge .—. badge'
  setAsyncEP aepptr $ ACTIVEAEP newBadge newVal

```

## 6.5.2 Receiving Messages

This function performs an asynchronous IPC receive operation, given a thread pointer and a capability to an asynchronous endpoint. The receive is blocking – the thread will be blocked on the endpoint till a message arrives.

```

receiveAsyncIPC :: PPTR TCB → CAPABILITY → KERNEL ()

```

```

receiveAsyncIPC thread cap = do

```

Fetch the asynchronous endpoint, and select the operation based on its state.

```

let aepptr = capAEPPtr cap
    aep ← getAsyncEP aepptr
case aep of

```

If the asynchronous endpoint is idle, then it becomes a waiting asynchronous endpoint, with the current thread in its queue. The thread is blocked.

```

IDLEAEP → do
  setThreadState (BLOCKEDONASYNC EVENT {
    waitingOnAsyncEP = aepptr } ) thread
  setAsyncEP aepptr $ WAITING [thread]

```

If the asynchronous endpoint is already waiting, the current thread is blocked and added to the queue.

```

WAITING queue → do
  setThreadState (BLOCKEDONASYNC EVENT {
    waitingOnAsyncEP = aepptr } ) thread
  setAsyncEP aepptr $ WAITING $ thread:queue

```

If the asynchronous endpoint is active, the message will be loaded to the MRs of the thread and the endpoint will be marked as idle.

```

ACTIVEAEP badge currentValue → do
  doAsyncTransfer badge currentValue thread
  setAsyncEP aeptr $ IDLEAEP

```

### 6.5.3 Delete Operation

If an asynchronous endpoint is deleted, then pending receive operations must be cancelled.

```

aepCancelAll :: PPTR ASYNCENDPOINT → KERNEL ()
aepCancelAll aeptr = do
  aep ← getAsyncEP aeptr
  case aep of
    WAITING queue → do
      mapM_ (setThreadState INACTIVE) queue
      setAsyncEP aeptr IDLEAEP
    _ → return ()

```

The following function will remove the given thread from the queue of the asynchronous endpoint, and replace the thread's IPC block with a fault block (which will retry the operation if the thread is resumed).

```

asyncIPCCancel :: PPTR TCB → PPTR ASYNCENDPOINT → KERNEL ()
asyncIPCCancel threadPtr aeptr = do
  aep ← getAsyncEP aeptr
  let queue' = delete threadPtr $ aepQueue aep
      aep' ← case queue' of
        [] → return IDLEAEP
        _ → return $ aep { aepQueue = queue' }
  setAsyncEP aeptr aep'
  setThreadState INACTIVE threadPtr

```

### 6.5.4 Accessing Asynchronous Endpoints

The following functions are specialisations of the *getObject* and *setObject* for the asynchronous endpoint object and pointer type.

```

getAsyncEP :: PPTR ASYNCENDPOINT → KERNEL ASYNCENDPOINT
getAsyncEP = getObject

```

```

setAsyncEP :: PPTR ASYNCENDPOINT → ASYNCENDPOINT → KERNEL ()
setAsyncEP = setObject

```

## 6.6 Untyped Objects

This module defines the behavior of untyped objects.

```
module SEL4.OBJECT.UNTYPED (decodeUntypedInvocation, invokeUntyped) where

import SEL4.API.TYPES
import SEL4.API.FAILURES
import SEL4.API.INVOCATION
import SEL4.MACHINE
import SEL4.MODEL
import SEL4.OBJECT.STRUCTURES
import SEL4.OBJECT.INSTANCES
import {—# SOURCE #—} SEL4.OBJECT.CNODE

import {—# SOURCE #—} SEL4.KERNEL.CSPACE

import DATA.MAYBE
import DATA.BITS
```

### 6.6.1 Invocation

Invocation of an untyped object retypes the memory region, possibly creating new typed kernel objects. As shown in figure 6.1, the retype operation will generate one or more new capabilities, which are inserted in the mapping database as children of the initial capability. These newly created capabilities will have all access rights, and other object specific fields will be initialised to some sensible value.

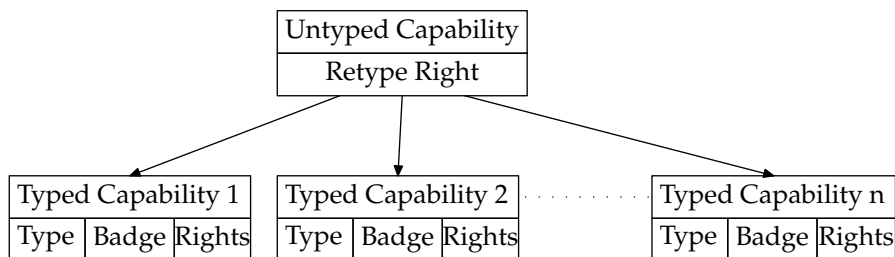


Figure 6.1: Invoking an Untyped Object

The expected parameters are the type of the new objects, the size of the requested objects (for those with variable size), a capability to a capability space, index and depth used to locate the destination for the new capabilities, and the maximum number of

new capabilities to be created. When successful, it returns the number of new objects or regions created.

```
decodeUntypedInvocation :: [WORD] → PPTR CTE → CAPABILITY →
    KERNELF SYSCALLERROR UNTYPEDINVOCATION
decodeUntypedInvocation args slot cap = do
    when (length args < 7) $ throw TRUNCATEDMESSAGE
```

The first argument must be a valid object type.

```
when (fromIntegral (args !! 0) > fromEnum (maxBound :: OBJECTTYPE)) $
    throw $ INVALIDARGUMENT 0
let newType = toEnum (fromIntegral $ args !! 0) :: OBJECTTYPE
```

The second argument specifies the size of the object, for the types for which the object size may vary — untyped memory, data frames, and CNodes, and possibly architecture-defined types. For any other type, it is ignored.

The value of this argument is the base 2 logarithm of the actual required size. The unit depends on the object type: one byte for untyped memory objects, the architecture’s minimum page size for data frames, and one capability slot for CNodes.

```
let userObjSize = fromIntegral $ args !! 1
```

The kernel does not allow creation of CNodes containing only one entry; this is done to avoid non-terminating loops in capability lookup. Note that it is possible for a single entry CNode to translate bits using its guard; this is not allowed, however, to avoid having to check for it during capability lookups.

```
when (newType = fromAPIType CAPTABLEOBJECT ∧ userObjSize = 0) $
    throw $ INVALIDARGUMENT 1
```

It must be possible to derive new capabilities from the invoked capability. If there are any capabilities already derived from it, the operation will fail with a `REVOKEFIRST` error.

```
ensureNoChildren slot
```

The next three arguments specify a CNode to place newly created capabilities in, in a similar manner to the source argument of the Insert and Move operations. However, unlike those operations, the specified slot must contain a CNode capability, and the new capabilities will be placed in *that* CNode.

```
let nodeRoot = CPTR $ args !! 2
    nodeCPtr = CPTR $ args !! 3
    nodeDepth = fromIntegral $ args !! 4

rootCap ← capErrorOnFailure $ lookupCap nodeRoot
nodeCap ← if nodeDepth = 0
    then return rootCap
```

```

else do
  (nodeSlot ,_) ← lookupSlotForCNodeOp FALSE TRUE
    rootCap nodeCPtr nodeDepth
  withoutFailure $ getSlotCap nodeSlot allRights

```

If the destination capability is not a writable CNode, an error is returned.

```

case nodeCap of
  CNODECAP { capCNodeCanModify = TRUE } → return ()
  _ → throw $ FAILEDLOOKUP FALSE $ MISSINGCAPABILITY {
    missingCapBitsLeft = nodeDepth }

```

The next two arguments specify the start and the length of a contiguous block of empty capability slots in the destination CNode.

```

let nodeOffset = args !! 5
let nodeWindow = args !! 6

let nodeSize = 1 `shiftL` (capCNodeBits nodeCap)
rangeCheck nodeWindow 1 $ nodeSize - nodeOffset

slots ← withoutFailure $
  mapM (locateSlot $ capCNodePtr nodeCap)
    [nodeOffset .. nodeOffset + nodeWindow - 1]

```

The destination slots must all be empty. If any of them contains a capability, the operation will fail with a `DELETEFIRST` error.

```

mapM_ ensureEmptySlot slots

return $! RETYPE {
  retypeSource = slot ,
  retypeRegionBase = capPtr cap ,
  retypeRegionSizeBits = capBlockSize cap ,
  retypeNewType = newType ,
  retypeNewSizeBits = userObjSize ,
  retypeSlots = slots }

```

```

invokeUntyped :: UNTYPEDINVOCATION → KERNEL [WORD]
invokeUntyped (RETYPE srcSlot base size newType objSize destSlots) = do

```

The following code removes any existing objects in the physical memory region. This operation is specific to the Haskell physical memory model, in which memory objects are typed; it is not necessary (or possible) when running on real hardware.

```

deleteObjects base size

count ← insertNewCaps newType srcSlot destSlots objSize

```

```
return [count]
```

## 6.7 Object Types and Retyping

This module defines several functions operating on kernel objects. These operations are applicable to all objects but depend partly on their types. Therefore these functions are partly implementation-defined, as they may operate on implementation-specific objects.

We use the C preprocessor to select a target architecture.

```
{-# OPTIONS_GHC -cpp #-}  
  
module SEL4.OBJECT.OBJECTTYPE where  
  
import SEL4.API.TYPES  
import SEL4.API.FAILURES  
import SEL4.API.INVOCATION  
import SEL4.MACHINE  
import SEL4.MODEL  
import SEL4.OBJECT.STRUCTURES  
import SEL4.OBJECT.INSTANCES  
import SEL4.OBJECT.UNTYPED  
import {-# SOURCE #-} SEL4.OBJECT.CNODE  
import {-# SOURCE #-} SEL4.OBJECT.ENDPOINT  
import {-# SOURCE #-} SEL4.OBJECT.ASYNCENDPOINT  
import {-# SOURCE #-} SEL4.OBJECT.TCB  
import {-# SOURCE #-} SEL4.KERNEL.CSPACE  
import {-# SOURCE #-} SEL4.KERNEL.THREAD  
  
import qualified SEL4.OBJECT.OBJECTTYPE.TARGET as ARCH  
  
import DATA.BITS  
import DATA.MAYBE
```

## 6.7.1 Creating and Deleting Capabilities

When copying a capability, it may be necessary to reset or modify data that is specific to each capability (rather than to each object). This is not necessary for any of the universal object types, but may be necessary for architecture-specific types. The following function is used when copying a capability, to allow such changes.

```
deriveCap :: CAPABILITY → KERNELF SYSCALLERROR CAPABILITY
deriveCap (ARCHOBJECTCAP cap) = liftM ARCHOBJECTCAP $ ARCH.deriveCap cap
deriveCap (ZOMBIE {}) = return NULLCAP
deriveCap cap = return cap
```

Similarly, when deleting a capability, it may be necessary to change other parts of the kernel or machine state that refer to that specific capability.

```
deleteCap :: CAPABILITY → PPTR CTE → KERNEL ()
deleteCap (ARCHOBJECTCAP cap) slot = ARCH.deleteCap cap slot
deleteCap (FRAMECAP {}) _ = doMachineOp flushCaches
deleteCap _ _ = return ()
```

## 6.7.2 Inspecting Capabilities

The following function is used to determine whether two capabilities refer to the same object. Note that it is not correct to use `=` for this, as only the object type, pointer and size are relevant.

```
sameRegionAs :: CAPABILITY → CAPABILITY → BOOL

sameRegionAs (a@UNTYPEDCAP {}) (b@UNTYPEDCAP {}) =
  capPtr a = capPtr b ∧ capBlockSize a = capBlockSize b

sameRegionAs (a@ENDPOINTCAP {}) (b@ENDPOINTCAP {}) =
  capEPPtr a = capEPPtr b

sameRegionAs (a@ASYNCENDPOINTCAP {}) (b@ASYNCENDPOINTCAP {}) =
  capAEPPtr a = capAEPPtr b

sameRegionAs (a@CNODECAP {}) (b@CNODECAP {}) =
  capCNodePtr a = capCNodePtr b ∧ capCNodeBits a = capCNodeBits b

sameRegionAs (a@THREADCAP {}) (b@THREADCAP {}) =
  capCNodePtr a = capCNodePtr b

sameRegionAs (a@FRAMECAP {}) (b@FRAMECAP {}) =
  capVPBasePtr a = capVPBasePtr b ∧ capVPSizeBits a = capVPSizeBits b
```

```
sameRegionAs (ARCHOBJECTCAP a) (ARCHOBJECTCAP b) =
  a `ARCH.sameRegionAs` b
```

```
sameRegionAs _ _ = FALSE
```

### 6.7.3 Modifying Capabilities

The `updateCapData` function is used to update a capability when moving or copying it, given a data word provided by the user. It may return `NULLCAP` (given a valid input `cap`) if the user provides an invalid data word. The meaning of the data word depends on the type of the capability; some types do not use it at all.

```
updateCapData :: WORD → CAPABILITY → CAPABILITY
```

Endpoint badges can only be set once; if the existing badge is not zero (the default value), then the update will fail.

```
updateCapData new cap@(ENDPOINTCAP {})
  | capEPBadge cap = 0 = cap { capEPBadge = new , capEPCanIdentify = FALSE }
  | otherwise = NULLCAP
```

```
updateCapData new cap@(ASYNCENDPOINTCAP {})
  | capAEPBadge cap = 0 = cap { capAEPBadge = new }
  | otherwise = NULLCAP
```

Whenever a CSpace node capability is copied or moved, the rights mask in its capability data must be reduced by the effective rights mask on the region containing the original capability. Also, the rights in the CNode's mask itself may only be decreased. This prevents the effective rights on a node's contents being increased.

```
updateCapData w cap@(CNODECAP {}) =
  cap {
    capCNodeGuard = (w `shiftR` (rightsBits + guardSizeBits)) .&.
      mask guardBits ,
    capCNodeRightsMask = capCNodeRightsMask cap
      `andCapRights` rightsFromWord (w .&. mask rightsBits),
    capCNodeGuardSize = fromIntegral $ (w `shiftR` rightsBits) .&.
      mask guardSizeBits }
```

where

```
rightsBits = 3
guardBits = case bitSize w of
  32 → 19
  64 → 49
guardSizeBits = case bitSize w of
  32 → 5
```

```

updateCapData w (ARCHOBJECTCAP cap) = ARCH.updateCapData w cap
updateCapData _ (ZOMBIE {}) = NULLCAP
updateCapData _ cap = cap

```

The `maskCapRights` function restricts the operations that can be performed on a capability, given a set of rights. This function is used when looking up capabilities in the capability space, to apply the masks that are present at each level of the tree; it is also used when copying capabilities, to apply a mask specified by the task invoking the copy operation.

```

maskCapRights :: CAPRIGHTS → CAPABILITY → CAPABILITY

```

```

maskCapRights _ NULLCAP = NULLCAP

```

```

maskCapRights _ c@(UNTYPEDCAP {}) = c

```

```

maskCapRights r c@(ENDPOINTCAP {}) = c {
  capEPCanSend = capEPCanSend c ∧ capAllowWrite r,
  capEPCanReceive = capEPCanReceive c ∧ capAllowRead r,
  capEPCanGrant = capEPCanGrant c ∧ capAllowGrant r }

```

```

maskCapRights r c@(ASYNCENDPOINTCAP {}) = c {
  capAEPCanSend = capAEPCanSend c ∧ capAllowWrite r,
  capAEPCanReceive = capEPCanReceive c ∧ capAllowRead r }

```

```

maskCapRights r c@(CNODECAP {}) = c {
  capCNodeCanModify = capCNodeCanModify c ∧ capAllowWrite r,
  capCNodeCanRead = capCNodeCanRead c ∧ capAllowRead r }

```

```

maskCapRights r c@(THREADCAP {}) = c {
  capTCBCanRead = capTCBCanRead c ∧ capAllowRead r,
  capTCBCanWrite = capTCBCanWrite c ∧ capAllowWrite r,
  capTCBCanGrant = capTCBCanGrant c ∧ capAllowGrant r }

```

```

maskCapRights r c@(FRAMECAP {}) = c {
  capVPCanRead = capVPCanRead c ∧ capAllowRead r,
  capVPCanWrite = capVPCanWrite c ∧ capAllowWrite r }

```

```

maskCapRights r (ARCHOBJECTCAP c) = ARCH.maskCapRights r c

```

```

maskCapRights _ (ZOMBIE {}) = NULLCAP

```

## 6.7.4 Creating and Deleting Objects

The `createNewCaps` function creates an array of new objects in physical memory, and returns a list of capabilities referring to them. Its parameters are an object type, the base address and size of the destination memory region, and the object size requested by the user. The latter is used only for variable-sized objects such as frames or CNodes.

```
createNewCaps :: OBJECTTYPE → PPTR () → INT → INT → KERNEL [CAPABILITY]
createNewCaps t rb rs us =
  let pointerCast = PPTR . fromPPtr
  in case toAPIType t of
    JUST INTDATAOBJECT → do
      let sz = objBits (⊥ :: USERDATA)
          c ← createObjects rb rs (makeObject :: USERDATA)
      return $! map
        (λn → FRAMECAP
          (pointerCast $ rb + n `shiftL` sz) (us + sz) TRUE TRUE)
        [0, 1 `shiftL` us .. 1 `shiftL` c - 1]
    JUST TCBOBJECT → do
      let sz = objBits (⊥ :: TCB)
          c ← createObjects rb rs (makeObject :: TCB)
      return $! map
        (λn → THREADCAP
          (pointerCast $ rb + n `shiftL` sz) TRUE TRUE TRUE)
        [0 .. 1 `shiftL` c - 1]
    JUST ENDPOINTOBJECT → do
      let sz = objBits (⊥ :: ENDPOINT)
          c ← createObjects rb rs (makeObject :: ENDPOINT)
      return $! map
        (λn → ENDPOINTCAP
          (pointerCast $ rb + n `shiftL` sz) 0 TRUE TRUE TRUE TRUE)
        [0 .. 1 `shiftL` c - 1]
    JUST ASYNCENDPOINTOBJECT → do
      let sz = objBits (⊥ :: ASYNCENDPOINT)
          c ← createObjects rb rs (makeObject :: ASYNCENDPOINT)
      return $! map
        (λn → ASYNCENDPOINTCAP
          (pointerCast $ rb + n `shiftL` sz) 0 TRUE TRUE)
        [0 .. 1 `shiftL` c - 1]
    JUST CAPTABLEOBJECT → do
      let sz = objBits (⊥ :: CTE)
          c ← createObjects rb rs (makeObject :: CTE)
      return $! map
        (λn → CNODECAP (pointerCast $ rb + n `shiftL` sz) us
```

```

        allRights 0 0 TRUE TRUE)
    [0, 1 `shiftL` us .. 1 `shiftL` c - 1]
JUST UNTYPED → do
    return $! map
        (λn → UNTYPEDCAP (rb + n) us)
    [0, 1 `shiftL` us, 1 `shiftL` rs - 1]
NOTHING → do
    archCaps ← ARCH.createNewCaps t rb rs us
    return $! map ARCHOBJECTCAP archCaps

```

The `detypeObject` function is called when the kernel deletes a kernel object of a given type.

```

detypeObject :: CAPABILITY → KERNELP ()
detypeObject cap = do
    case cap of
        NULLCAP → error ‘‘tried to detype null cap’’
        ENDPOINTCAP { capEPPtr = ptr } →
            withoutPreemption $ epCancelAll ptr
        ASYNCENDPOINTCAP { capAEPptr = ptr } →
            withoutPreemption $ aepCancelAll ptr
        CNODECAP { capCNodePtr = ptr, capCNodeBits = nodeBits } → do
            let numberOfSlots = 1 `shiftL` nodeBits
                mapM_ (λslotNum → do
                    slotPtr ← withoutPreemption $ locateSlot ptr slotNum
                    cteDelete slotPtr)
                    [0 .. numberOfSlots - 1]
        THREADCAP { capTCBPtr = thread } → do
            curThread ← withoutPreemption getCurThread
            withoutPreemption $ setThreadState INACTIVE thread
            withoutPreemption $ ipcCancel thread
            cRootSlot ← withoutPreemption $ getThreadCSpaceRoot thread
            cteDelete cRootSlot
            vRootSlot ← withoutPreemption $ getThreadVSpaceRoot thread
            cteDelete vRootSlot
            when (thread = curThread) $ withoutPreemption $ do
                setSchedulerAction CURRENTTHREADISINVALID
                setCurThread nullPointer
        FRAMECAP {} → return ()
        UNTYPEDCAP {} → return ()
        ARCHOBJECTCAP c → ARCH.detypeObject c

```

## 6.7.5 Invoking Objects

The following functions are used to handle messages that are sent to kernel objects by user level code using a `SEND` or `SENDWAIT` system call.

The `decodeInvocation` function parses the message, determines the operation that is being performed, and checks for any error conditions. If it returns successfully, the invocation is guaranteed to complete without any errors.

```
decodeInvocation :: WORD → [WORD] → CPTR → PPTR CTE →
    CAPABILITY → KERNELF SYSCALLERROR INVOCATION

decodeInvocation _ _ _ _ cap@(ENDPOINTCAP {capEPCanSend=TRUE}) =
    return $ INVOKEENDPOINT
        (capEPPtr cap) (capEPBadge cap) (capEPCanGrant cap)

decodeInvocation label _ _ _ cap@(ASYNCENDPOINTCAP {capAEPCanSend=TRUE}) =
    return $ INVOKEASYNCENDPOINT (capAEPPtr cap) (capAEPBadge cap) label

decodeInvocation label args _ _ cap@(THREADCAP {capTCBCanWrite=TRUE}) =
    liftM INVOKETCB $ decodeTCBInvocation label args cap

decodeInvocation label args _ _ cap@(CNODECAP {capCNodeCanModify=TRUE}) =
    liftM INVOKECNODE $ decodeCNodeInvocation label args cap

decodeInvocation label args _ slot cap@(UNTYPEDCAP {}) =
    liftM INVOKEUNTYPED $ decodeUntypedInvocation args slot cap

decodeInvocation label args capIndex slot cap@(ARCHOBJECTCAP {}) =
    liftM INVOKEARCHOBJECT $
        ARCH.decodeInvocation label args capIndex slot cap
```

If the capability cannot be invoked, because it does not have a required right, then the operation returns `INVALIDCAPABILITY`.

Note that at present, null capabilities will have caused faults before `decodeInvocation` is called, so this will only be thrown for valid capabilities with insufficient rights to be invoked; for example, a read-only endpoint.

```
decodeInvocation _ _ _ _ _ = throw INVALIDCAPABILITY
```

The `invoke` function performs the operation itself. It cannot throw faults, but it may be pre-empted. If it returns a list of words, they will be sent as a reply message with label 0; this is optional because the kernel does not generate replies from endpoint invocations.

This function just dispatches invocations to the type-specific invocation functions.

```
invoke :: BOOL → INVOCATION → KERNELP (MAYBE [WORD])

invoke _ (INVOKEUNTYPED op) = withoutPreemption $
```

```

liftM JUST $ invokeUntyped op

invoke block (INVOKEENDPOINT ep badge canGrant) = withoutPreemption $ do
  thread ← getCurThread
  sendIPC block NOTHING badge canGrant thread ep
  return NOTHING

invoke _ (INVOKEASYNCENDPOINT ep badge message) = do
  withoutPreemption $ sendAsyncIPC ep badge message
  return NOTHING

invoke _ (INVOKETCB op) = invokeTCB op >> (return $! JUST [])

invoke _ (INVOKECNODE op) = invokeCNode op >> (return $! JUST [])

invoke _ (INVOKEARCHOBJECT op) = ARCH.invoke op

```

## 6.8 Storing Objects

This module defines the instances of `STORABLE` objects.

```

module SEL4.OBJECT.INSTANCES where

import SEL4.API.TYPES
import SEL4.MACHINE
import SEL4.OBJECT.STRUCTURES
import SEL4.MODEL.STORABLE

import DATA.BITS
import DATA.DYNAMIC

```

### 6.8.1 Type Class Instances

The following are the instances of `STORABLE` for the four main types of kernel object: synchronous IPC endpoints, asynchronous IPC endpoints, thread control blocks, and capability table entries.

## Synchronous IPC Endpoint

```
instance STORABLE ENDPOINT where
  objBits _ = case bitSize (⊥::WORD) of
    32 → 4
    64 → 5
  makeObject = IDLEEP
```

## Asynchronous IPC Endpoint

```
instance STORABLE ASYNCENDPOINT where
  objBits _ = case bitSize (⊥::WORD) of
    32 → 4
    64 → 5
  makeObject = IDLEAEP
```

## Capability Table Entry

```
instance STORABLE CTE where
  objBits _ = case bitSize (⊥::WORD) of
    32 → 4
    64 → 5
  makeObject = CTE {
    cteCap = NULLCAP,
    cteMDBNode = MDB {
      mdbNext = error 'uninitialised_MDB',
      mdbPrev = error 'uninitialised_MDB',
      mdbRevocable = FALSE } }
```

As mentioned in the documentation for the type class `STORABLE`, there is one kernel object which needs its own definitions for `loadObject` and `storeObject`; it is the capability table entry. The reason for this is that thread control blocks contain a single capability table entry, for the root of the table; the capability copy and revocation functions need to access that CTE without knowing that it's actually inside a TCB. So the CTE versions of `loadObject` and `storeObject` must be able to handle accesses to TCBs as well.

```
loadObject bits obj = case fromDynamic obj :: MAYBE CTE of
  JUST cte
    | bits = TRUE → alignError $ objBits cte
    | length bits ≠ objBits cte → error 'malformed_CSpace'
    | otherwise → cte
  NOTHING → case fromDynamic obj :: MAYBE TCB of
    JUST tcb
```

```

    | bits = tcbSlotToOffset tcbVTableSlot → tcbVTable tcb
    | bits = tcbSlotToOffset tcbCTableSlot → tcbCTable tcb
    | length bits ≠ objBits tcb → error ‘malformed_CSpace’
    | otherwise → alignError $ objBits tcb
NOTHING → typeError ‘CTE’ (show obj)
where
  toBits x = map (testBit x) [0 .. (bitSize x) - 1]
  tcbSlotToOffset slot = reverse $ take (length bits) $
    replicate (objBits (⊥::CTE)) FALSE ++ toBits slot

updateObject cte bits oldObj = case fromDynamic oldObj :: MAYBE CTE of
  JUST _
    | √ bits = TRUE → alignError $ objBits cte
    | length bits ≠ objBits cte → error ‘malformed_PSpace’
    | otherwise → toDyn cte
  NOTHING → case fromDynamic oldObj :: MAYBE TCB of
    JUST tcb
      | bits = tcbSlotToOffset tcbVTableSlot →
        toDyn $ tcb { tcbVTable = cte }
      | bits = tcbSlotToOffset tcbCTableSlot →
        toDyn $ tcb { tcbCTable = cte }
      | length bits ≠ objBits tcb → error ‘malformed_PSpace’
      | otherwise → alignError $ objBits tcb
    NOTHING → typeError (show $ typeOf cte) (show oldObj)
where
  toBits x = map (testBit x) [0 .. (bitSize x) - 1]
  tcbSlotToOffset slot = reverse $ take (length bits) $
    replicate (objBits cte) FALSE ++ toBits slot

```

## Thread Control Block

The value of `objBits` in this instance is an estimate; the value used in real implementations may vary and may be architecture-dependent.

```

instance STORABLE TCB
where
  objBits _ = 10
  makeObject = THREAD {
    tcbCTable = makeObject,
    tcbVTable = makeObject,
    tcbState = INACTIVE,
    tcbPriority = minBound,
    tcbQueued = FALSE,

```

```
tcbTimeSlice = 0,  
tcbResultEndpoint = CPTR 0,  
tcbFaultHandler = CPTR 0,  
tcbIPCBuffer = VPTR 0,  
tcbContext = newContext }
```

# 7 Haskell Model Details

This chapter describes implementation details of the Haskell kernel. It should not be necessary to read this section to understand the API; however, it may help in understanding the way the model works.

## 7.1 System State

This module defines the high-level structures used to represent the state of the entire system, and the types and functions used to perform basic operations on the state.

```
module SEL4.MODEL.STATEDATA (
    module SEL4.MODEL.STATEDATA,
    module CONTROL.MONAD, get, gets, put, modify,
) where

import SEL4.API.TYPES
import SEL4.MODEL.STORABLE
import {-# SOURCE #-} SEL4.MODEL.PSPACE(PSPACE)
import SEL4.OBJECT.STRUCTURES
import SEL4.MACHINE

import DATA.ARRAY
import CONTROL.MONAD
import CONTROL.MONAD.STATE
```

### 7.1.1 Types

#### Kernel State

The top-level kernel state structure is called `KERNELSTATE`. It contains:

```
data KERNELSTATE = KSTATE {
```

- the physical address space, of type `PSPACE` (defined in section 7.2 on page 128);

```
ksPSPACE :: PSPACE,
```

- an array of ready queues, indexed by thread priority;

```
ksReadyQueues :: ARRAY PRIORITY READYQUEUE,
```

- a pointer to the current thread's control block;

```
ksCurThread :: PPTR TCB,
```

- and the required action of the scheduler next time it runs.

```
ksSchedulerAction :: SCHEDULERACTION }
```

Note that this definition of `KERNELSTATE` assumes a single processor. The behaviour of the kernel on multi-processor systems is not specified by this document.

## Monads

Kernel functions are sequences of operations that transform a `KERNELSTATE` object. They are encapsulated in the monad `KERNEL`, which uses `STATET` to add a `KERNELSTATE` data structure to the monad that encapsulates the simulated machine, `MACHINEMONAD`. This allows functions to read and modify the kernel state.

```
type KERNEL = STATET KERNELSTATE MACHINEMONAD
```

Note that there is no error-signalling mechanism available to functions in `KERNEL`. Therefore, all errors encountered in such functions are fatal, and will halt the kernel. See section 7.5 on page 135 for the definition of monads used for error handling.

## Scheduler Queues

The ready queue is simply a list of threads that are ready to run. Each thread in this list is at the same priority level.

```
type READYQUEUE = [PPTR TCB]
```

This is a standard Haskell singly-linked list independent of the thread control block structures. However, in a real implementation, it would most likely be embedded in the thread control blocks themselves.

### 7.1.2 Kernel State Functions

The following two functions are used to get and set the value of the current thread pointer which is stored in the kernel state.

These functions have the same basic form as many others in the kernel which fetch or set the value of some part of the state data. They make use of `gets` and `modify`, two functions which each apply a given function to the current state — either returning some value extracted from the state, or calculating a new state which replaces the previous one.

```
getCurThread :: KERNEL (PPTR TCB)
getCurThread = gets ksCurThread
```

```
setCurThread :: PPTR TCB → KERNEL ()
setCurThread tptr = modify (λks → ks { ksCurThread = tptr })
```

Similarly, functions are provided that get or set the ready queue for a given priority level, and that get or set the requested action of the scheduler.

```
getQueue :: PRIORITY → KERNEL READYQUEUE
getQueue prio = gets (λks → ksReadyQueues ks ! prio)
```

```
setQueue :: PRIORITY → READYQUEUE → KERNEL ()
setQueue prio q = modify (λks →
    ks { ksReadyQueues = (ksReadyQueues ks)//[(prio, q)] })
```

```
getSchedulerAction :: KERNEL SCHEDULERACTION
getSchedulerAction = gets ksSchedulerAction
```

```
setSchedulerAction :: SCHEDULERACTION → KERNEL ()
setSchedulerAction a = modify (λks → ks { ksSchedulerAction = a })
```

### 7.1.3 Performing Machine Operations

The following function allows the machine monad to be directly accessed from kernel code.

```
doMachineOp :: MACHINEMONAD a → KERNEL a
doMachineOp = lift
```

## 7.1.4 Miscellaneous Monad Functions

### Assertions and Undefined Behaviour

The function `assert` is used to state that a predicate must be true at a given point. If it is not, the behaviour of the kernel is undefined. The Haskell model will not terminate in this case.

```
assert :: MONAD m => BOOL -> STRING -> m ()
assert p e = if p then return () else error $ 'Assertion failed:␣' ++ e
```

### Searching a List

The function `findM` searches a list, returning the first item for which the given function returns `TRUE`. It is the monadic equivalent of `DATA.LIST.find`.

```
findM :: MONAD m => (a -> m BOOL) -> [a] -> m (MAYBE a)
findM _ [] = return NOTHING
findM f (x:xs) = do
    r <- f x
    if r then return $ JUST x else findM f xs
```

## 7.2 Physical Address Space Model

This module contains the data structure and operations for the physical memory model.

```
module SEL4.MODEL.PSPACE (
    PSPACE, newPSPACE,
    getObject, setObject, createObjects, deleteObjects
) where

import SEL4.API.TYPES(kernelTop)
import SEL4.MODEL.STATEDATA
import SEL4.MODEL.STORABLE
import SEL4.MACHINE

import DATA.BINARYTREE as BINARYTREE
import DATA.BITS
import DATA.DYNAMIC
```

## 7.2.1 Types

### Physical Address Space

The physical address space is represented by a binary tree; the key used for lookups is a list of boolean values corresponding to the bits in the physical address. The objects themselves are dynamically typed.

```
newtype PSPACE = PSPACE { psMap :: BINARYTREE DYNAMIC }
```

### 7.2.2 Physical Address Space Initialisation

A new physical address space contains a small number of kernel-reserved frames, followed by empty frames.

```
newPSPACE :: PSPACE
newPSPACE = PSPACE {
  psMap = foldr
    (λptr tree → BINARYTREE.insert
      (ptrBitsForSize frameBits ptr)
      (toDyn (makeObject :: KERNELDATA))
      tree)
    BINARYTREE.empty
    (map (λx → x * frameSize) [0 .. PPTR kernelTop - 1])}
where
  frameBits = objBits (⊥ :: KERNELDATA)
  frameSize = 1 `shiftL` frameBits
```

### 7.2.3 Accessing Objects

Given a pointer into physical memory, an attempt may be made to fetch or update an object of any storable type from the address space. The caller is assumed to have checked that the address is correctly aligned for the requested object type and that it actually contains an object of the requested type.

If the object type or alignment is wrong, the behaviour of the kernel is unpredictable. In Haskell this is modelled by evaluating  $\perp$  (by calling the error function); this is specified to have undefined behaviour, but in practice it halts the model with an error message.

```
getObject :: STORABLE a ⇒ PPTR a → KERNEL a
getObject ptr = do
  map ← gets $ psMap . ksPSPACE
  (bits, dynVal) ← BINARYTREE.lookup (ptrBits ptr) map
```

```

return $ loadObject bits dynVal

setObject :: STORABLE a => PPTR a -> a -> KERNEL ()
setObject ptr val = do
  ps ← gets ksPSpace
  let map = psMap ps
      map' = BINARYTREE.adjust (updateObject val) (ptrBits ptr) map
      ps' = ps { psMap = map' }
      modify (λks → ks { ksPSpace = ps'})

```

## 7.2.4 Creating Objects

Objects are created in regions, which must contain a power of two number of objects. The arguments to `createObjects` are a pointer to the start of the region (which must be aligned to the region's size), the size of the region (as a power of two), and the value used to initialise the created objects. The result is the number of objects created, as a power of two.

```

createObjects :: STORABLE a => PPTR () -> INT -> a -> KERNEL INT
createObjects ptr bits val = do

```

First, check the alignment of the specified region.

```

unless (fromPPtr ptr .&. mask bits = 0) $
  alignError bits

```

Fetch the `PSPACE` structure from the current state, and search the region for existing objects; fail if any are found.

```

ps ← gets ksPSpace
unless (BINARYTREE.isEmpty (ptrBitsForSize bits ptr) $ psMap ps) $
  error "Object creation would destroy an existing object"

```

Now calculate the number of objects that will be created. Note that this may be zero, if the region is too small.

```

let oBits = objBits val
    objectCount = 1 `shiftL` (bits - oBits)

```

Make a list of addresses at which to create objects.

```

let addresses = map
  (λn → ptrBitsForSize oBits (ptr + n `shiftL` oBits))
  [0 .. objectCount - 1]

```

Ensure that the new objects will fit inside the valid physical address space.

```

memoryTop ← liftM pointerCast $ doMachineOp getMemoryTop

```

```
when (ptr + objectCount `shiftL` objBits val > memoryTop) $
  error ‘‘Created an object outside the physical memory space’’
```

Insert the objects into the PSPACE map.

```
let map' = foldr
  (\addr tree → BINARYTREE.insert addr (toDyn val) tree)
  (psMap ps) addresses
```

Update the state with the new PSPACE map.

```
let ps' = ps { psMap = map' }
modify (\ks → ks { ksPSpace = ps' })
```

The return value is the number of objects created, as a power of two; if this is negative, no objects were created.

```
return $ bits - objBits val
```

## 7.2.5 Deleting Objects

No type checks are performed when deleting objects; `deleteObjects` simply deletes every object in the given region. If an object partially overlaps with the given region but is not completely inside it, this function’s behaviour is undefined.

```
deleteObjects :: PPtr a → INT → KERNEL ()
deleteObjects ptr bits = do
  ps ← gets ksPSpace
  let map' = BINARYTREE.delete (ptrBitsForSize bits ptr) $ psMap ps
  let ps' = ps { psMap = map' }
  modify (\ks → ks { ksPSpace = ps' })
```

## 7.2.6 Helper Functions

The following functions are used to generate a list of the bits in the given address, starting with the most significant bit. The second version truncates the result appropriately to specify the entire region occupied by an object of the given size.

```
ptrBits :: PPtr a → [BOOL]
ptrBits ptr = take (bitSize ptr) $
  map (`testBit` (bitSize ptr - 1)) $ iterate (`shiftL` 1) ptr

ptrBitsForSize :: INT → PPtr a → [BOOL]
ptrBitsForSize s ptr = take (bitSize ptr - s) $ ptrBits ptr
```

The following function casts between two types of physical pointers.

```
pointerCast :: PPTR a → PPTR b
pointerCast = PPTR . fromPPtr
```

## 7.3 Storable Objects

This module defines the class of types that may be stored in the simulated machine's physical memory. These types include integer data, and several different kernel objects.

This module makes use of the GHC extension allowing derivation of the class `TYPEABLE`, so GHC language extensions are enabled.

```
{-# OPTIONS_GHC -fglasgow-exts #-}
```

```
module SEL4.MODEL.STORABLE where
```

### 7.3.1 Imported Modules

```
import SEL4.MACHINE.HARDWARE
```

```
import DATA.DYNAMIC
import DATA.BITS
```

### 7.3.2 Types

Values of the type `KERNELDATA` represent kernel data or code; they cannot be retyped or accessed by user-level code. Note that values of this type contain no real data.

```
data KERNELDATA = KERNELDATA
    deriving TYPEABLE
```

Values of the type `USERDATA` represent user-level data or code. For simulator performance reasons, values of this type also contain no real data.

```
data USERDATA = USERDATA
    deriving TYPEABLE
```

### 7.3.3 Public Functions

These two functions are simple convenience functions that halt the kernel with an error message when a memory access is performed with incorrect type or alignment.

```
typeError :: STRING → STRING → a
typeError t1 t2 = error ('Wrong object type - expected' ++ t1 ++ ', found' ++ t2)

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

### 7.3.4 Type Classes

#### Storable Objects

The type class `STORABLE` defines a set of operations that may be performed on any object that is storable in the modelled physical memory.

If an instance of the class `STORABLE a r w` exists, then objects of type `a` are storable on machines with register name type `r` and word type `w`. The latter two types must represent an architecture that can be simulated. The object type `a` must uniquely define both the register name and word types.

Also, all three types must derive the standard Haskell class `TYPEABLE`. This requirement allows the physical address space model to determine at runtime whether an item of data is of the expected type.

```
class TYPEABLE a ⇒ STORABLE a where
```

The size and alignment of the physical region occupied by objects of type `a` is defined by the function `objBits`. This returns the logarithm base 2 of the object's size (i.e., the number of least significant bits of the physical address space that must be zero).

```
objBits :: a → INT
```

The function `makeObject` creates a new object, containing reasonable default values.

```
makeObject :: a
```

The function `loadObject` attempts to load an object from the physical address space, and halts the kernel with an error if it finds the wrong type of object.

A default definition is provided which is sufficient in most cases; there is only one exception, which can be found in section 6.8 on page 121.

```
loadObject :: [BOOL] → DYNAMIC → a
loadObject bits obj = case fromDynamic obj of
```

```

JUST value
  |  $\bigvee$  bits  $\rightarrow$  alignError $ objBits value
  | length bits  $\neq$  objBits value  $\rightarrow$  error ‘malformed_PSpace’
  | otherwise  $\rightarrow$  value
NOTHING  $\rightarrow$  TypeError (show $ typeOf expectedValue) (show obj)
where expectedValue =  $\perp$  :: a

```

The function `updateObject` updates the existing contents of a memory location by replacing them with a new object of the same type.

Like `loadObject`, there is a default definition which is sufficient in all but one case.

```

updateObject :: a  $\rightarrow$  [BOOL]  $\rightarrow$  DYNAMIC  $\rightarrow$  DYNAMIC
updateObject val bits oldObj = case fromDynamic oldObj :: MAYBE a of
  JUST _
    |  $\bigvee$  bits  $\rightarrow$  alignError $ objBits val
    | length bits  $\neq$  objBits val  $\rightarrow$  error ‘malformed_PSpace’
    | otherwise  $\rightarrow$  toDyn val
  NOTHING  $\rightarrow$  TypeError (show $ typeOf val) (show oldObj)

```

### 7.3.5 Class Instances

The following are the class instances of `STORABLE` for the user and kernel data frame types.

```

instance STORABLE USERDATA where
  objBits _ = pageBits
  makeObject = USERDATA

instance STORABLE KERNELDATA where
  objBits _ = pageBits
  makeObject = KERNELDATA

```

## 7.4 System Calls

This module contains utility functions used in system call implementations.

```

module SEL4.MODEL.SYSCALL where

import SEL4.API.FAILURES
import SEL4.MODEL.STATEDATA
import SEL4.MODEL.FAILURES
import SEL4.MODEL.PREEMPTION

```

```
import CONTROL.MONAD.ERROR
```

System calls in seL4 have three stages: one which may fault, one which may encounter system call errors, and one which cannot fail but may be interrupted. Generally the first is used to look up capabilities, the second to decode arguments and check for possible failures, and the third to perform the operation itself. If either of the first two stages fails, a failure handler runs instead of the following stages.

The `syscall` function lifts code into the appropriate monads, and handles faults and errors.

```
syscall :: KERNELF FAULT a → (FAULT → KERNEL c) →
  (a → KERNELF SYSCALLERROR b) → (SYSCALLERROR → KERNEL c) →
  (b → KERNELP c) → KERNELP c
syscall mFault hFault mError hError mFinalise = do
  rFault ← withoutPreemption $ runErrorT mFault
  case rFault of
    LEFT f → withoutPreemption $ hFault f
    RIGHT a → do
      rError ← withoutPreemption $ runErrorT $ mError a
      case rError of
        LEFT e → withoutPreemption $ hError e
        RIGHT b → mFinalise b
```

The `atomicSyscall` function is equivalent, but cannot be interrupted.

```
atomicSyscall :: KERNELF FAULT a → (FAULT → KERNEL c) →
  (a → KERNELF SYSCALLERROR b) → (SYSCALLERROR → KERNEL c) →
  (b → KERNEL c) → KERNEL c
atomicSyscall mFault hFault mError hError mFinalise = do
  rFault ← runErrorT mFault
  case rFault of
    LEFT f → hFault f
    RIGHT a → do
      rError ← runErrorT $ mError a
      case rError of
        LEFT e → hError e
        RIGHT b → mFinalise b
```

## 7.5 Failures

This module defines the kernel's mechanisms for handling failures in kernel code.

```
module SEL4.MODEL.FAILURES where
```

```
import SEL4.API.FAILURES
import SEL4.API.TYPES
import SEL4.MACHINE
import SEL4.OBJECT.STRUCTURES
import SEL4.MODEL.STATEDATA

import CONTROL.MONAD.ERROR
```

## 7.5.1 Data Types

### Monads

The `KERNELF` monad is a transformation of the `KERNEL` monad defined in section 7.1 on page 125. The `ERRORT` monad transformer is applied to `KERNEL` to allow kernel functions to abort with a non-fatal error value. Depending on the type of error, which is indicated here by the type parameter  $f$ , it may be either handled internally by the kernel or propagated to user level.

```
type KERNELF  $f$  = ERRORT  $f$  KERNEL
```

Note that fatal errors, which are caused by kernel bugs or invalid states and should *never* actually occur, are modelled by evaluating  $\perp$ . This typically happens via a call to the Haskell function `error`, or an implicit evaluation of  $\perp$  by the Haskell compiler on a pattern or guard match failure.

## 7.5.2 Class Instances

All four of the failure types must have an instance of the `ERROR` class to be usable with `ERRORT` (and therefore with `KERNELF`).

```
instance ERROR FAULT
instance ERROR SYSCALLERROR
instance ERROR LOOKUPFAILURE
```

## 7.5.3 Failure Handling

### Allowing and Preventing Failure

The use of the `ERRORT` monad transformer to encapsulate code that can fail requires that transitions in and out of such code be explicitly marked. The following functions may be used to do so. Note that these are simply specialisations of existing functions defined on the `ERRORT` transformer or the `MONADTRANS` class.

```
withoutFailure :: ERROR f => KERNEL a -> KERNELF f a
withoutFailure = lift
```

```
throw :: ERROR f => f -> KERNELF f a
throw = throwError
```

The `catchFailure` function is used to call code that may fail, given a function that can handle any failures.

```
catchFailure :: ERROR f => KERNELF f a -> (f -> KERNEL a) -> KERNEL a
catchFailure f h = do
    result <- runErrorT f
    either h return result
```

The `rethrowFailure` function converts one type of failure into another. This is used to convert a `LOOKUPFAILURE` into the appropriate `FAULT` or `SYSCALLERROR`.

```
rethrowFailure :: (ERROR f1, ERROR f2) =>
    (f1 -> f2) -> KERNELF f1 a -> KERNELF f2 a
rethrowFailure t m = do
    result <- lift $ runErrorT m
    either (throw . t) return result
```

### Lookup Failures

Lookup failures are handled by converting them to either faults or system call errors, depending on the type of lookup. The following functions perform this conversion.

```
capFaultOnFailure :: CPTR -> BOOL -> KERNELF LOOKUPFAILURE a ->
    KERNELF FAULT a
capFaultOnFailure cptr rp = rethrowFailure $ CAPFAULT cptr rp
```

```
vmFaultOnFailure :: VPTR -> BOOL -> KERNELF LOOKUPFAILURE a ->
    KERNELF FAULT a
vmFaultOnFailure vptr wr = rethrowFailure $ VMFAULT vptr wr
```

```
lookupErrorOnFailure :: BOOL -> KERNELF LOOKUPFAILURE a ->
```

```

    KERNELF SYSCALLERROR a
lookupErrorOnFailure isSource = rethrowFailure $ FAILEDLOOKUP isSource

capErrorOnFailure :: KERNELF LOOKUPFAILURE a → KERNELF SYSCALLERROR a
capErrorOnFailure = rethrowFailure $ const INVALIDCAPABILITY

```

## Silent Failures

Some failures are silent; the kernel simply aborts the operation.

```

nullCapOnFailure :: ERROR f ⇒ KERNELF f CAPABILITY → KERNEL CAPABILITY
nullCapOnFailure = flip catchFailure $ const $ return NULLCAP

nothingOnFailure :: ERROR f ⇒ KERNELF f (MAYBE a) → KERNEL (MAYBE a)
nothingOnFailure = flip catchFailure $ const $ return NOTHING

```

## 7.5.4 Detecting Failures

This trivial helper function is used to check that an argument is within an acceptable range.

```

rangeCheck :: INTEGRAL a ⇒ a → a → a → KERNELF SYSCALLERROR ()
rangeCheck value min max =
    when (value < min ∨ value > max) $ throw $
        RANGEERROR (fromIntegral min) (fromIntegral max)

```

## 7.6 Preemption

This module defines the types and functions used by the kernel model to implement preemption points and non-preemptible sections in the kernel code.

```

module SEL4.MODEL.PREEMPTION(
    KERNELP, withoutPreemption, preemptionPoint
) where

import SEL4.MACHINE
import SEL4.MODEL.STATEDATA

import CONTROL.MONAD.ERROR

```

## 7.6.1 Types

### Interrupts

Objects of this type are thrown from an `ERRORT` monad transformer when the simulator signals a pending interrupt at a kernel preemption point. The nature of the interrupt is not relevant here, because the simulator will send it to the kernel model as an `EVENT` once the kernel has been preempted.

```
data INTERRUPT = INTERRUPT

instance ERROR INTERRUPT
```

### Monads

The `KERNELP` monad is a transformation of the `KERNEL` monad used for functions which may be preempted. Any function in this monad must not leave the kernel in an inconsistent state when calling other functions in the monad (though the model has no means of effectively enforcing this restriction).

```
type KERNELP a = ERRORT INTERRUPT KERNEL a
```

## 7.6.2 Functions

If an operation must be performed during which the kernel state is temporarily inconsistent, it should be performed in the argument of a `withoutPreemption` call, to ensure that no preemption points are encountered during the operation.

```
withoutPreemption :: KERNEL a → KERNELP a
withoutPreemption = lift
```

In preemptible code, the kernel may explicitly mark a preemption point with the `preemptionPoint` function.

```
preemptionPoint :: KERNELP ()
preemptionPoint = do
  preempt ← lift $ doMachineOp checkForInterrupt
  when preempt $ throwError INTERRUPT
```

## 8 Modelling the Hardware

This chapter defines the interface that is used by the rest of the kernel to represent the underlying hardware.

### 8.1 Words and Registers

This module defines the interface that the kernel model uses to determine the properties of the architecture of the machine being simulated, and particularly the types, names and functions of the general purpose registers.

We use the C preprocessor to select a target architecture. Also, this file makes use of the GHC extension allowing derivation of arbitrary type classes for types defined with `newtype`, so GHC language extensions are enabled.

```
{-# OPTIONS_GHC -cpp -fglasgow-exts #-}

module SEL4.MACHINE.REGISTERSET where

import qualified SEL4.MACHINE.REGISTERSET.TARGET as ARCH

import DATA.BITS
import DATA.ARRAY
import FOREIGN.STORABLE

import CONTROL.MONAD.STATE(STATE, gets, modify)
```

#### 8.1.1 Types

The architecture must define two types: one for the type of the machine's word, and one for the set of valid register names.

## Word Types

The type `WORD` represents the native word size of the modelled machine. It must be an instance of the type classes that allow bitwise arithmetic, use as an integer, storage in dynamically typed objects and conversion to a string.

```
newtype WORD = WORD ARCH.WORD
    deriving (Eq, Ord, Enum, Real, Num, Bits, Integral, Bounded, Storable, Show)
```

## Register Names

The `REGISTER` type defines a bounded, enumerated set of register names.

```
newtype REGISTER = REGISTER ARCH.REGISTER
    deriving (Eq, Ord, Enum, Bounded, Ix, Show)
```

## Pointers

To enforce explicit casts between pointer types, `newtype` is used to declare types for various types of user and kernel pointers. These types derive a number of basic type classes allowing sorting, comparison for equality, pointer arithmetic and printing.

Note that they do not derive `Integral`, despite being integers; this is to avoid allowing them to be cast using `fromIntegral`, which does not indicate that the cast is to or from a pointer type.

Also, these types derive `Num`, which requires a GHC extension.

The types defined here are used for pointers to physical or virtual memory.

```
newtype PPTR a = PPTR { fromPPtr :: WORD }
    deriving (Show, Eq, Num, Bits, Ord, Enum, Bounded)

newtype VPTR = VPTR { fromVPtr :: WORD }
    deriving (Show, Eq, Num, Bits, Ord, Enum, Bounded)
```

## User-level Context

The representation of the user-level context of a thread is an array of machine words, indexed by register name.

```
newtype USERCONTEXT = UC { fromUC :: ARRAY REGISTER WORD }
    deriving Show
```

## 8.1.2 Monads

```
type USERMONAD = STATE USERCONTEXT
```

USERMONAD is a specialisation of CONTROL.MONAD.STATE, used to execute functions that have access only to the user-level context of a single thread.

## 8.1.3 Functions and Constants

### Register Set

The following functions and constants define registers or groups of user-level registers that are used for specific purposes by the kernel.

**The message information register** contains metadata about the contents of an IPC message, such as the length of the message and whether a capability is attached.

```
msgInfoRegister :: REGISTER
msgInfoRegister = REGISTER ARCH.msgInfoRegister
```

**Message registers** are used to hold the message being sent by an object invocation.

This list may be empty, though it should contain as many registers as possible. Message words that do not fit in these registers will be transferred in a buffer in user-accessible memory.

```
msgRegisters :: [REGISTER]
msgRegisters = map REGISTER ARCH.msgRegisters
```

**System call registers** are used when performing a system call to specify the location of the invoked capabilities — the destination of the message in the first register, and the source of the reply in the second register.

```
syscallRegisters :: (REGISTER, REGISTER)
syscallRegisters = (REGISTER  $r_1$ , REGISTER  $r_2$ )
  where ( $r_1$ ,  $r_2$ ) = ARCH.syscallRegisters
```

**The badge register** is used to return the badge of the capability from which a message was received. This is typically the same as one of the system call registers.

```
badgeRegister :: REGISTER
badgeRegister = REGISTER ARCH.badgeRegister
```

**The frame registers** are those that must be stored in the data page argument to the `EXCHANGEREGISTERS` system call, rather than being copied directly to or from the current thread's state. This includes at least the program counter, the stack pointer, the message registers, and the syscall input and output registers.

```
frameRegisters :: [REGISTER]
frameRegisters = map REGISTER ARCH.frameRegisters
```

**The general-purpose registers** may be transferred directly to or from the current thread's state by an `EXCHANGEREGISTERS` system call. It includes all registers that are not in `frameRegisters`, except any kernel-reserved or constant registers (such as the MIPS *zero*,  $k_0$  and  $k_1$  registers).

```
gpRegisters :: [REGISTER]
gpRegisters = map REGISTER ARCH.gpRegisters
```

## User-level Context

A new user-level context is a list of values for the machine's registers, all of which are initially set to 0.

```
newContext :: USERCONTEXT
newContext = UC $ listArray (minBound,maxBound) $ repeat 0
```

Functions are provided to get and set a single register.

```
getRegister :: REGISTER → USERMONAD WORD
getRegister r = gets $ (!r) . fromUC

setRegister :: REGISTER → WORD → USERMONAD ()
setRegister r v = modify $ UC . (//[[(r, v)]) . fromUC
```

## Miscellaneous

The `mask` function is a trivial function which, given a number of bits, returns a word with that number of low-order bits set.

```
mask :: BITS w ⇒ INT → w
mask bits = bit bits - 1
```

## 8.2 Hardware Functions

This module contains definitions of the types and functions that the kernel uses for interaction with the hardware, such as performing memory accesses and controlling virtual memory mappings.

We use the C preprocessor to select a target architecture.

```
{-# OPTIONS_GHC -cpp #-}  
  
module SEL4.MACHINE.HARDWARE where  
  
import qualified SEL4.MACHINE.HARDWARE.TARGET as ARCH  
  
import SEL4.MACHINE.REGISTERSET  
  
import DATA.BITS
```

### 8.2.1 Types

#### Hardware Monad

Each simulator must define a monad that encapsulates the state of the underlying hardware.

```
type MACHINEMONAD = ARCH.MACHINEMONAD
```

### 8.2.2 Hardware Operations

The simulator must define several operations on the underlying hardware. These operations include physical memory accesses, memory management operations, and operations to fetch the configuration of the hardware. There is also a debugging print operation, which is not required to do anything, but will typically print its argument to the console.

The required operations are:

- fetching the number of address bits covered by a virtual page;

```
pageBits :: INT  
pageBits = ARCH.pageBits
```

- fetching the address of the word immediately after the end of physical memory, which is assumed to be contiguous;

```
getMemoryTop :: MACHINEMONAD (PPTR ())
getMemoryTop = ARCH.getMemoryTop
```

- fetching a list of the base addresses and sizes (in address bits) of the regions of the physical address space occupied by memory-mapped I/O devices;

```
getDeviceRegions :: MACHINEMONAD [(PPTR (), INT)]
getDeviceRegions = ARCH.getDeviceRegions
```

- loading or storing a word in the physical address space;

```
loadWord :: PPTR WORD → MACHINEMONAD WORD
loadWord = ARCH.loadWord
```

```
storeWord :: PPTR WORD → WORD → MACHINEMONAD ()
storeWord = ARCH.storeWord
```

- loading an entry into the translation cache (used only on architectures with software-loaded TLBs);

```
insertMapping :: PPTR WORD → VPTR → INT → BOOL → MACHINEMONAD ()
insertMapping = ARCH.insertMapping
```

- flushing the translation, data and instruction caches (if necessary) when switching or modifying address spaces;

```
flushCaches :: MACHINEMONAD ()
flushCaches = ARCH.flushCaches
```

- checking for any pending interrupts which might preempt the kernel;

```
checkForInterrupt :: MACHINEMONAD BOOL
checkForInterrupt = ARCH.checkForInterrupt
```

- and printing a string to the debugging console.

```
debugPrint :: STRING → MACHINEMONAD ()
debugPrint = ARCH.debugPrint
```

There are also two operations performed on the machine's register set that are implemented as functions, because they may vary considerably between architectures. They are used to fetch and set the program counter register (or registers) after the kernel receives a fault.

There are also two functions that perform specific operations on the register set. These are used to fetch and set the program counter. We must use functions here, rather than aliases for register names, because the handling of these registers may differ significantly depending on the architecture and the type of fault. These functions:

- fetch the address of the instruction that is currently executing, from the user level thread's point of view (such as a system call instruction, or an instruction which accesses virtual memory and has faulted);

```
getRestartPC :: USERMONAD WORD
getRestartPC = ARCH.getRestartPC
```

- and set the address of the next instruction to be executed, which by default is the instruction after the current one.

```
setNextPC :: WORD → USERMONAD ()
setNextPC = ARCH.setNextPC
```

### 8.2.3 Constants

The constant `nullPointer` is a physical pointer guaranteed to be invalid.

```
nullPointer :: PPTR a
nullPointer = PPTR 0
```

## 9 Implementation-Specific Features

The definitions in this chapter are instances for specific platforms of the implementation-defined parts of the kernel API, along with default definitions for platforms with no special requirements.

### 9.1 Generic VSpace Implementation

This module contains definitions of the VSpace functions for platforms that do not define specific data structures for virtual memory page tables. On these platforms, the kernel uses CSpace structures to represent virtual memory address spaces.

```
module SEL4.KERNEL.VSPACE.CSPACE where

import SEL4.MACHINE
import SEL4.MODEL
import SEL4.OBJECT
import SEL4.KERNEL.CSPACE
import SEL4.API.TYPES
import SEL4.API.FAILURES

import DATA.BITS

initVSpace :: [PPTR ()] → CAPABILITY → KERNEL ([PPTR ()], CAPABILITY)
initVSpace _ root = return ([], root)

lookupVPtr :: PPTR TCB → VPTR → BOOL → KERNELF LOOKUPFAILURE (PPTR WORD)
lookupVPtr thread vptr isWrite = do
    threadRootSlot ← withoutFailure $ getThreadVSpaceRoot thread
    threadRoot ← withoutFailure $ getSlotCap threadRootSlot allRights

    let cptr = CPTR $ fromVPtr vptr
        bits = bitSize cptr
        (slot, endBits, rightsMask) ←
            resolveAddressBits threadRoot (const TRUE) cptr bits
    cap ← withoutFailure $ getSlotCap slot rightsMask
```

```

case (isWrite, cap) of
  (_, FRAMECAP { capVPSizeBits = sizeBits })
    | sizeBits ≠ endBits → throw $ DEPTHMISMATCH {
      depthMismatchBitsLeft = endBits,
      depthMismatchBitsFound = sizeBits }
    (TRUE, FRAMECAP { capVPCanWrite = TRUE }) → return ()
    (FALSE, FRAMECAP { capVPCanRead = TRUE }) → return ()
    _ → throw $ MISSINGCAPABILITY { missingCapBitsLeft = 0 }

let offset = fromVPtr vpPtr .&. mask (capVPSizeBits cap)
return $! capVPBasePtr cap + PPTR offset

handleVMFault :: PPTR TCB → VPTR → BOOL → KERNELF FAULT ()
handleVMFault thread vpPtr isWrite = do
  pptr ← vmFaultOnFailure vpPtr isWrite $ lookupVPtr thread vpPtr isWrite
  withoutFailure $ doMachineOp $
    insertMapping pptr vpPtr pageBits isWrite

isValidVTableRoot :: CAPABILITY → BOOL
isValidVTableRoot (CNODECAP {}) = TRUE
isValidVTableRoot _ = FALSE

```

## 9.2 ARM

This section defines the implementation-specific parts of the API for the ARM architecture. This presently only supports a simplified version of the ARM with no hardware-defined MMU data structures.

### 9.2.1 Register Set

```

module SEL4.MACHINE.REGISTERSET.ARM where

import qualified DATA.WORD
import DATA.ARRAY

data REGISTER =
  R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | SL | FP | IP | SP |
  LR | LR_SVC | FAULTINSTRUCTION | CPSR
  deriving (Eq, Enum, Bounded, Ord, Ix, Show)

```

```

type WORD = DATA.WORD.WORD32

msgInfoRegister = R2
msgRegisters = [R3 .. R7]
syscallRegisters = (R0, R1)
badgeRegister = R1
frameRegisters = FAULTINSTRUCTION : SP : CPSR : [R0 .. R7]
gpRegisters = [R8 .. IP] ++ [LR]

```

## 9.2.2 Hardware Interface

```

{-# OPTIONS_GHC -fffi -fglasgow-exts #-}

module SEL4.MACHINE.HARDWARE.ARM where

import SEL4.MACHINE.REGISTERSET
import qualified SEL4.MACHINE.REGISTERSET.ARM as ARM

import qualified FOREIGN.STORABLE
import FOREIGN.PTR
import CONTROL.MONAD.READER
import CONTROL.MONAD.STATE
import DATA.BITS

type MACHINEMONAD = READERT MACHINEDATA IO

initMachine :: PPTR () → [(PPTR (), INT)] → PTR CALLBACKDATA → MACHINEDATA
initMachine = MACHINEDATA

data CALLBACKDATA

data MACHINEDATA = MACHINEDATA {
    sdMemoryTop :: PPTR (),
    sdDeviceRegions :: [(PPTR (), INT)],
    sdCallbackPtr :: PTR CALLBACKDATA }

pageBits :: INT
pageBits = 12

getMemoryTop :: MACHINEMONAD (PPTR ())
getMemoryTop = asks sdMemoryTop

```

```

getDeviceRegions :: MACHINEMONAD [(PPTR (), INT)]
getDeviceRegions = asks sdDeviceRegions

loadWord :: PPTR WORD → MACHINEMONAD WORD
loadWord ptr = do
    cbptr ← asks sdCallbackPtr
    lift $ loadWordCallback cbptr ptr

storeWord :: PPTR WORD → WORD → MACHINEMONAD ()
storeWord ptr val = do
    cbptr ← asks sdCallbackPtr
    lift $ storeWordCallback cbptr ptr val

insertMapping :: PPTR WORD → VPTR → INT → BOOL → MACHINEMONAD ()
insertMapping pptr vptr 12 isWrite = do
    cbptr ← asks sdCallbackPtr
    lift $ tlbInsertCallback cbptr
        (pptr `shiftR` pageBits) (vptr `shiftR` pageBits) isWrite
insertMapping _ _ _ _ = error "invalid mapping size"

flushCaches :: MACHINEMONAD ()
flushCaches = do
    cbptr ← asks sdCallbackPtr
    lift $ tlbFlushCallback cbptr

checkForInterrupt :: MACHINEMONAD BOOL
checkForInterrupt = do
    cbptr ← asks sdCallbackPtr
    lift $ interruptCallback cbptr

debugPrint :: STRING → MACHINEMONAD ()
debugPrint str = lift $ putStrLn str

foreign import ccall "arm_load_word" loadWordCallback ::
    PTR CALLBACKDATA → PPTR WORD → IO WORD

foreign import ccall "arm_store_word" storeWordCallback ::
    PTR CALLBACKDATA → PPTR WORD → WORD → IO ()

foreign import ccall "arm_tlb_insert" tlbInsertCallback ::
    PTR CALLBACKDATA → PPTR WORD → VPTR → BOOL → IO ()

foreign import ccall "arm_tlb_flush" tlbFlushCallback ::

```

```

PTR CALLBACKDATA → IO ()

foreign import ccall "arm_interrupt" interruptCallback ::
  PTR CALLBACKDATA → IO BOOL

getRestartPC = getRegister (REGISTER ARM.FAULTINSTRUCTION)
setNextPC = setRegister (REGISTER ARM.LR_SVC)

```

### 9.2.3 API

This module contains an instance of the machine-specific kernel API for the ARM architecture.

There are presently no architecture-specific types defined, so we just re-export the definitions of the universal object types.

```

module SEL4.API.TYPES.ARM (module SEL4.API.TYPES.UNIVERSAL) where

import SEL4.API.TYPES.UNIVERSAL

```

### 9.2.4 Kernel Objects

This module contains operations on machine-specific object types for the ARM. Since there are presently no such object types, the functions here are all undefined, and should never be called.

This module makes use of the GHC extension allowing data types with no constructors, so GHC language extensions are enabled.

```

{-# OPTIONS_GHC -fglasgow-exts #-}

module SEL4.OBJECT.OBJECTTYPE.ARM where

import SEL4.MACHINE
import SEL4.MODEL
import SEL4.API.TYPES
import SEL4.API.FAILURES
import SEL4.OBJECT.STRUCTURES

data INVOCATION

```

```

deriveCap :: ARCHCAPABILITY → KERNELF SYSCALLERROR ARCHCAPABILITY
deriveCap = ⊥

deleteCap :: ARCHCAPABILITY → PPTR CTE → KERNEL ()
deleteCap = ⊥

sameRegionAs :: ARCHCAPABILITY → ARCHCAPABILITY → BOOL
sameRegionAs = ⊥

updateCapData :: WORD → ARCHCAPABILITY → CAPABILITY
updateCapData = ⊥

maskCapRights :: CAPRIGHTS → ARCHCAPABILITY → CAPABILITY
maskCapRights = ⊥

createNewCaps :: OBJECTTYPE → PPTR () → INT → INT →
                KERNEL [ARCHCAPABILITY]
createNewCaps = ⊥

retypeRegion :: PPTR () → INT → OBJECTTYPE → KERNEL INT
retypeRegion = ⊥

detypeObject :: ARCHCAPABILITY → KERNELP ()
detypeObject = ⊥

decodeInvocation :: WORD → [WORD] → CPTR → PPTR CTE → CAPABILITY →
                  KERNELF SYSCALLERROR INVOCATION
decodeInvocation = ⊥

invoke :: INVOCATION → KERNELP (MAYBE [WORD])
invoke = ⊥

```

## 9.2.5 Virtual Address Space

At present, the ARM model uses the capability space for virtual memory. This will change in the near future to use the ARM's hardware-defined page tables.

```
module SEL4.KERNEL.VSPACE.ARM (
    module SEL4.KERNEL.VSPACE.CSPACE
) where

import SEL4.KERNEL.VSPACE.CSPACE
```

## 9.3 Simple Haskell-based Simulator

This section defines a language and interpreter that may be used for writing simple test programs for the seL4 API. The language is intended to look similar to the assembly language of a simple RISC processor. This chapter also defines implementation-specific API features for the simulated machine.

### 9.3.1 Register Set

```
module SEL4.MACHINE.REGISTERSET.HASKELLCPU where

import qualified DATA.WORD
import DATA.ARRAY

data REGISTER =
    PC | SP |
    ---_syscall argument registers
    AR0 | AR1 | AR2 | AR3 | AR4 | AR5 | AR6 | AR7 |
    ---_general purpose integer registers
    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 (Eq, ENUM, BOUNDED, ORD, IX, SHOW)

type WORD = DATA.WORD.WORD32

msgInfoRegister = AR2
msgRegisters = [AR3 .. AR7]
```

```

syscallRegisters = (AR0, AR1)
badgeRegister = AR1
frameRegisters = PC : SP : [AR0 .. AR7]
gpRegisters = [R0 .. R31]

```

### 9.3.2 Hardware Interface

```

module SEL4.MACHINE.HARDWARE.HASKELLCPU where

import SEL4.MACHINE.REGISTERSET
import qualified SEL4.MACHINE.REGISTERSET.HASKELLCPU as HASKELLCPU

import DATA.BITS
import qualified DATA.MAP as MAP
import CONTROL.MONAD.STATE
import CONTROL.MONAD.ERROR

type MACHINEMONAD = STATE MACHINESTATE

data MACHINESTATE = MS {
    msMemory :: MAP.MAP PPTR WORD,
    msTLB :: MAP.MAP VPN (PFN, BOOL),
    msConsoleOutput :: [STRING] }

pageBits :: INT
pageBits = 12

intSize :: WORD
intSize = fromIntegral $ bitSize (⊥::WORD) `div` 8

getMemoryTop :: MACHINEMONAD PPTR
getMemoryTop = return maxBound

getDeviceRegions :: MACHINEMONAD [(PPTR, INT)]
getDeviceRegions = return []

loadWord :: PPTR → MACHINEMONAD WORD
loadWord address = do
    when (address .&. PPTR (intSize-1) ≠ 0) $
        error 'unalignedLaccess'
    gets $ MAP.findWithDefault 0 address . msMemory

```

```

storeWord :: PPTR → WORD → MACHINEMONAD ()
storeWord address value = do
  when (address .&. PPTR (intSize-1) ≠ 0) $
    error 'unaligned␣access'
  modify (λms → ms {
    msMemory = MAP.insert address value $ msMemory ms })

debugPrint :: STRING → MACHINEMONAD ()
debugPrint str = modify (λms → ms {
  msConsoleOutput = msConsoleOutput ms ++ [str] })

tlbInsertEntry :: PFN → VPN → BOOL → MACHINEMONAD ()
tlbInsertEntry pptr vptr writable = modify (λms → ms {
  msTLB = MAP.insert vptr (pptr, writable) $ msTLB ms })

tlbFlushAll :: MACHINEMONAD ()
tlbFlushAll = modify (λms → ms { msTLB = MAP.empty })

getFaultingPC = getRegister (REGISTER HASKELLCPU.PC)
setNextPC = setRegister (REGISTER HASKELLCPU.PC)

```

### 9.3.3 API

This module contains an instance of the machine-specific kernel API for the Haskell-based CPU simulator. This simulator is for the machine-independent API, and defines no additional object types; so this module re-exports the definitions of the universal object types.

```

module SEL4.API.TYPES.HASKELLCPU (module SEL4.API.TYPES.UNIVERSAL) where

import SEL4.API.TYPES.UNIVERSAL

```

### 9.3.4 Kernel Objects

This module contains operations on machine-specific object types for the Haskell-based CPU simulator. Since there are no such object types, the functions here are all undefined, and should never be called.

```
module SEL4.OBJECT.OBJECTTYPE.HASKELLCPU where

import SEL4.MACHINE
import SEL4.MODEL.STATEDATA
import SEL4.API.TYPES
import SEL4.OBJECT.TYPES

retypeRegion :: PPtr → INT → OBJECTTYPE → KERNEL INT
retypeRegion = ⊥

detypeObject :: PPtr → INT → OBJECTTYPE → KERNEL (MAYBE THREADSUICIDE)
detypeObject = ⊥

invokeObject :: WORD → [WORD] → CPtr → CTEPtr → CAPABILITY →
              KERNELF FAULT (MAYBE (WORD, [WORD]))
invokeObject = ⊥
```

### 9.3.5 Virtual Address Space

The generic Haskell-based CPU uses the capability space for virtual memory.

```
module SEL4.KERNEL.VSPACE.HASKELLCPU (
    module SEL4.KERNEL.VSPACE.CSPACE
) where

import SEL4.KERNEL.VSPACE.CSPACE
```

## Bibliography

- [1] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5):295–357, July 2002.
- [2] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell version 98. <http://www.haskell.org/tutorial/>, June 2000.
- [3] L4Ka Team. L4Ka::Pistachio kernel. <http://l4ka.org/projects/pistachio/>.
- [4] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, 1994.
- [5] Jeff Newbern. All about monads. <http://www.nomaware.com/monads/html/>.
- [6] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.
- [7] 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.
- [8] 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.