Abstract Formal Specification of the seL4/ARMv6 API

June Andronick Timothy Bourke Philip Derrin
Kevin Elphinstone David Greenaway Gerwin Klein
Rafal Kolanski Daniel Matichuk Thomas Sewell
Simon Winwood

Version 1.3

Abstract

This document is the text version of the abstract, formal Isabelle/HOL specification of the seL4 microkernel. It is intended to give a precise, operational definition of the seL4 microkernel on the ARMv6 architecture. The document contains a short overview, followed by text generated from the formal Isabelle/HOL definitions.

This document is not a tutorial or user manual and is not intended to be read as such. Please see the bundled user manual for a higher-level introduction to the kernel.

Contents

1 Introduction

The seL4 microkernel is an operating system kernel designed to be a secure, safe, and reliable foundation for systems in a wide variety of application domains. As a microkernel, seL4 provides a minimal number of services to applications. This small number of services directly translates to a small implementation of approximately 8700 lines of C code, which has allowed the kernel to be formally proven in the Isabelle/HOL theorem prover to adhere to a formal specification.

This document gives the text version of the formal Isabelle/HOL specification used in this proof. The document starts by giving a brief overview of the seL4 microkernel design, followed by text generated from the Isabelle/HOL definitions.

This document is not a user manual to seL4, nor is it intended to be read as such. Instead, it is a precise reference to the behaviour of the seL4 kernel.

Further information on the models and verification techniques can be found in previous publications [1–3, 5–13, 16–22].

1.1 The seL4 Microkernel

The seL4 microkernel is a small operating system kernel of the L4 family. SeL4 provides a minimal number of services to applications, such as abstractions for virtual address spaces, threads, inter-process communication (IPC).

SeL4 uses a capability-based access-control model. All memory, devices, and microkernel-provided services require an associated *capability* (access right) to utilise them [4]. The set of capabilities an application possesses determines what resources that application can directly access. SeL4 enforces this access control by using the hardware's memory management unit (MMU) to ensure that userspace applications only have access to memory they possess capabilities to.

Figure 1.1 shows a representative seL4-based system. It depicts the microkernel executing on top of the hardware as the only software running in privileged mode of the processor. The first application to execute is the supervisor OS. The supervisor OS (also termed a *booter* for simple scenarios) is responsible for initialising, configuring and delegating authority to the specific system layered on top.

In Figure 1.1, the example system set up by the supervisor consists of an instance of Linux on the left, and several instances of trusted or sensitive applications on the right. The group of applications on the left and the group on the right are unable to directly communicate or interfere with each other without explicit involvement of the supervisor (and the microkernel) — a barrier is thus created between the untrusted left and the trusted right, as indicated in the figure. The supervisor has a kernel-provided mechanism to determine the relationship between applications and the presence or absence of any such barriers.

1.1.1 Kernel Services

A limited number of service primitives are provided by the microkernel; more complex services may be implemented as applications on top of these primitives. In this way, the functionality of the system can be extended without increasing the code and complexity in privileged mode, while still supporting a potentially wide number of services for varied application domains.

The basic services the microkernel provides are as follows:

Threads are an abstraction of CPU execution that support running software;

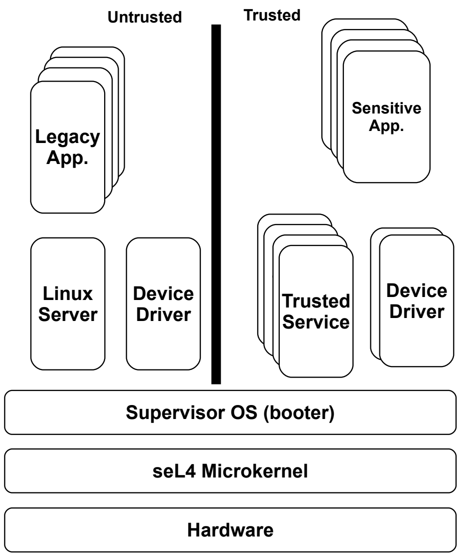


Figure 1.1: Sample seL4 based system

Address Spaces are virtual memory spaces that each contain an application. Applications are limited to accessing memory in their address space;

Interprocess Communication (IPC) via *endpoints* allows threads to communicate using message passing;

Device Primitives allow device drivers to be implemented as unprivileged applications. The kernel exports hardware device interrupts via IPC messages; and

Capability Spaces store capabilities (i.e., access rights) to kernel services along with their book-keeping information.

All kernel services are accessed using kernel-provided system calls that *invoke* a capability; the semantics of the system call depends upon the type of the capability invoked. For example, invoking the `Call()` system call on a thread control block (TCB) with certain arguments will suspend the target thread, while invoking `Call()` on an endpoint will result in a message being sent. In general, the message sent to a capability will have an entry indicating the desired operation, along with any arguments.

The kernel provides to clients the following system calls:

`Send()` delivers the system call arguments to the target object and allows the application to continue. If the target object is unable to receive and/or process the arguments immediately, the sending application will be blocked until the arguments can be delivered.

`NBSend()` performs non-blocking send in a similar fashion to `Send()` except that if the object is unable to receive the arguments immediately, the message is silently dropped.

`Call()` is a `Send()` that blocks the application until the object provides a response, or the receiving application replies. In the case of delivery to an application (via an Endpoint), an additional capability is added to the arguments and delivered to the receiver to give it the right to respond to the sender.

`Wait()` is used by an application to block until the target object is ready.

`Reply()` is used to respond to a `Call()`, using the capability generated by the `Call()` operation.

`ReplyWait()` is a combination of `Reply()` and `Wait()`. It exists for efficiency reasons: the common case of replying to a request and waiting for the next can be performed in a single kernel system call instead of two.

1.1.2 Capability-based Access Control

The seL4 microkernel provides a capability-based access control model. Access control governs all kernel services; in order to perform any system call, an application must invoke a capability in its possession that has sufficient access rights for the requested service. With this, the system can be configured to isolate software components from each other, and also to enable authorised controlled communication between components by selectively granting specific communication capabilities. This enables software component isolation with a high degree of assurance, as only those operations explicitly authorised by capability possession are permitted.

A capability is an unforgeable token that references a specific kernel object (such as a thread control block) and carries access rights that control what operations may be performed when it is invoked. Conceptually, a capability resides in an application's *capability space*; an address in this space refers to a *slot* which may or may not contain a capability. An application may refer to a capability — to request a kernel service, for example — using the address of the slot holding that capability. The seL4 capability model is an instance of a *segregated* (or *partitioned*) capability model, where capabilities are managed by the kernel.

Capability spaces are implemented as a directed graph of kernel-managed *capability nodes* (CNodes). A CNode is a table of slots, where each slot may contain further CNode capabilities. An address in a capability space is then the concatenation of the indices of the CNode capabilities forming the path to the destination slot; we discuss CNode objects further in section 1.1.3.

Capabilities can be copied and moved within capability spaces, and also sent via IPC. This allows creation of applications with specific access rights, the delegation of authority to another application, and passing to an application authority to a newly created (or selected) kernel service. Furthermore, capabilities can be *minted* to create a derived capability with a subset of the rights of the original capability (never with more rights). A newly minted capability can be used for partial delegation of authority.

Capabilities can also be revoked in their entirety to withdraw authority. Revocation includes any capabilities that may have been derived from the original capabilities. The propagation of capabilities through the system is controlled by a *take-grant*-based model [6].

1.1.3 Kernel Objects

In this section we give a brief overview of the kernel implemented objects that can be invoked by applications. The interface to these objects forms the interface to the kernel itself. The creation and use of the high-level kernel services is achieved by the creation, manipulation, and combination of these kernel objects.

CNodes

As mentioned in the previous section, capabilities in seL4 are stored in kernel objects called CNodes. A CNode has a fixed number of slots, always a power of two, determined when the CNode is created. Slots can be empty or contain a capability.

CNodes have the following operations:

`Mint()` creates a new capability in a specified CNode slot from an existing capability. The newly created capability may have fewer rights than the original.

`Copy()` is similar to `Mint()`, but the newly created capability has the same rights as the original.

`Move()` moves a capability between two specified capability slots.

`Mutate()` is an atomic combination of `Move()` and `Mint()`. It is a performance optimisation.

`Rotate()` moves two capabilities between three specified capability slots. It is essentially two `Move()` operations: one from the second specified slot to the first, and one from the third to the second. The first and third specified slots may be the same, in which case the capability in it is swapped

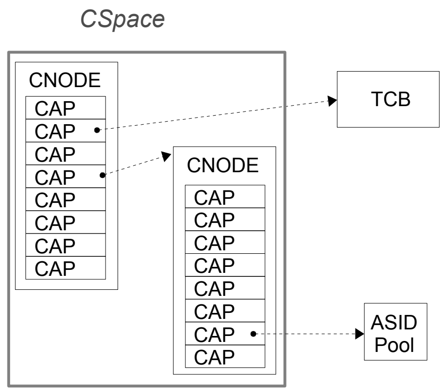


Figure 1.2: CNodes forming a CSpace

with the capability in the second slot. The operation is atomic; either both or neither capabilities are moved.

`Delete()` removes a capability from the specified slot.

`Revoke()` is equivalent to calling `Delete()` on each derived child of the specified capability. It has no effect on the capability itself.

`SaveCaller()` moves a kernel-generated reply capability of the current thread from the special TCB slot it was created in, into the designated CSpace slot.

`Recycle()` is equivalent to `Revoke()`, except that it also resets most aspects of the object to its initial state.

IPC Endpoints

The seL4 microkernel supports both *synchronous* (EP) and *asynchronous* (AsyncEP) IPC endpoints, used to facilitate interprocess communication between threads. Capabilities to endpoints can be restricted to be send-only or receive-only. They can also specify whether capabilities can be passed through the endpoint.

Synchronous endpoints allow both data and capabilities to be transferred between threads, depending on the rights on the endpoint capability. Sending a message will block the sender until the message has been received; similarly, a waiting thread will be blocked until a message is available (but see `NBSend()` above).

When only notification of an event is required together with a very limited message, asynchronous endpoints can be used. Asynchronous endpoints have a single invocation:

`Notify()` simply sets the given set of bits in the endpoint. Multiple `Notify()` system calls without an intervening `Wait()` result in the bits being "or-ed" with any bits already set. As such, `Notify()` is always non-blocking, and has no indication of whether a receiver has received the notification.

Additionally, the `Wait()` system call may be used with an asynchronous endpoint, allowing the calling thread to retrieve all set bits from the asynchronous endpoint. If no `Notify()` operations have taken place since the last `Wait()` call, the calling thread will block until the next `Notify()` takes place.

TCB

The *thread control block* (TCB) object represents a thread of execution in seL4. Threads are the unit of execution that is scheduled, blocked, unblocked, etc., depending on the applications interaction with other threads.

As illustrated in Figure 1.3, a thread needs both a CSpace and a VSpace in which to execute to form an application (plus some additional information not represented here). The CSpace provides the capabilities (authority) required to manipulated kernel objects, in order to send messages to another application for example. The VSpace provides the virtual memory environment required to contain the code and data of the application. A CSpace is associated with a thread by installing a capability to the root CNode of a CSpace into the TCB. Likewise, a VSpace is associated with a thread by installing a capability to a Page Directory (described shortly) into the TCB. Note that multiple threads can share the same CSpace and VSpace.

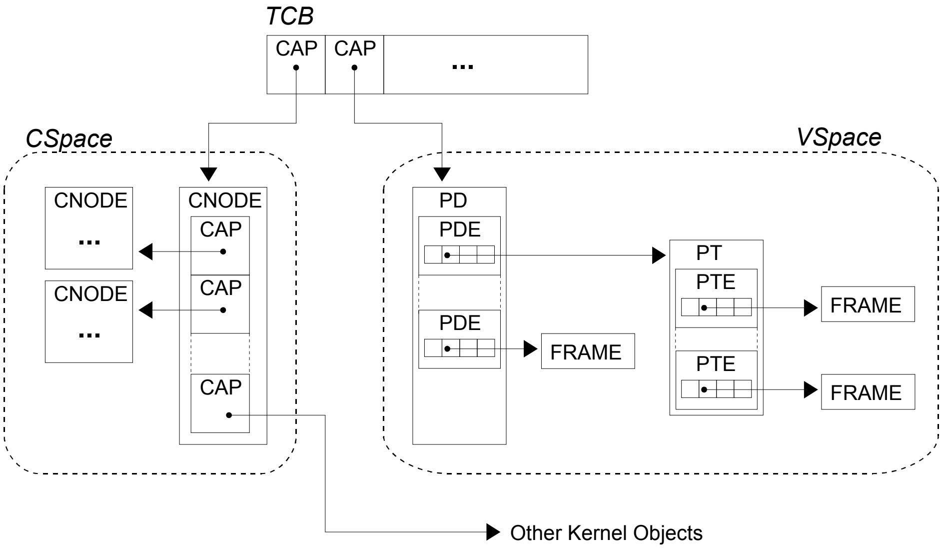


Figure 1.3: Internal representation of an application in seL4

The TCB object has the following methods:

`CopyRegisters()` is used for copying the state of a thread. The method is given an additional capability argument, which must refer to a TCB that will be used as the source of the transfer; the invoked thread is the destination. The caller may select which of several subsets of the register context will be transferred between the threads. The operation may also suspend the source thread, and resume the destination thread.

Two subsets of the context that might be copied (if indicated by the caller) include: firstly, the parts of the register state that are used or preserved by system calls, including the instruction and stack pointers, and the argument and message registers; and secondly, the remaining integer registers. Other subsets are architecture-defined, and typically include coprocessor registers such as the floating point registers. Note that many integer registers are modified or destroyed by system calls, so it is not generally useful to use `CopyRegisters()` to copy integer registers to or from the current thread.

`ReadRegisters()` is a variant of `CopyRegisters()` for which the destination is the calling thread. It uses the message registers to transfer the two subsets of the integer registers; the message format has the more commonly transferred instruction pointer, stack pointer and argument registers at the start, and will be truncated at the caller's request if the other registers are not required.

`WriteRegisters()` is a variant of `CopyRegisters()` for which the source is the calling thread. It uses the message registers to transfer the integer registers, in the same order used by `ReadRegisters()`. It may be truncated if the later registers are not required; an explicit length argument is given to allow error detection when the message is inadvertently truncated by a missing IPC buffer.

`SetPriority()` configures the thread's scheduling parameters. In the current version of seL4, this is simply a priority for the round-robin scheduler.

`SetIPCBuffer()` configures the thread's local storage, particularly the IPC buffer used for sending parts of the message payload that don't fit in hardware registers.

`SetSpace()` configures the thread's virtual memory and capability address spaces. It sets the roots of the trees (or other architecture-specific page table structures) that represent the two address spaces, and also nominates the Endpoint that the kernel uses to notify the thread's pager[1] of faults and exceptions.

`Configure()` is a batched version of the three configuration system calls: `SetPriority()`, `SetIPCBuffer()`, and `SetSpace()`. `Configure()` is simply a performance optimisation.

`Suspend()` makes a thread inactive. The thread will not be scheduled again until a `Resume()` operation is performed on it. A `CopyRegisters()` or `ReadRegisters()` operation may optionally include a `Suspend()` operation on the source thread.

`Resume()` resumes 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. `Resume()`-ing a thread that is already ready has no effect. `Resume()`-ing a thread that is in the waiting phase of a `Call()` operation may cause the sending phase to be performed again, even if it has previously succeeded.

A `CopyRegisters()` or `WriteRegisters()` operation may optionally include a `Resume()` operation on the destination thread.

Virtual Memory

A virtual address space in seL4 is called a VSpace. In a similar way to CSpaces, a VSpace is composed of objects provided by the microkernel. Unlike CSpaces, these objects for managing virtual memory largely directly correspond to those of the hardware, that is, a page directory pointing to page tables, which in turn map physical frames. The kernel also includes ASID Pool and ASID Control objects for tracking the status of address spaces.

Figure 1.4 illustrates a VSpace with the requisite components required to implement a virtual address space.

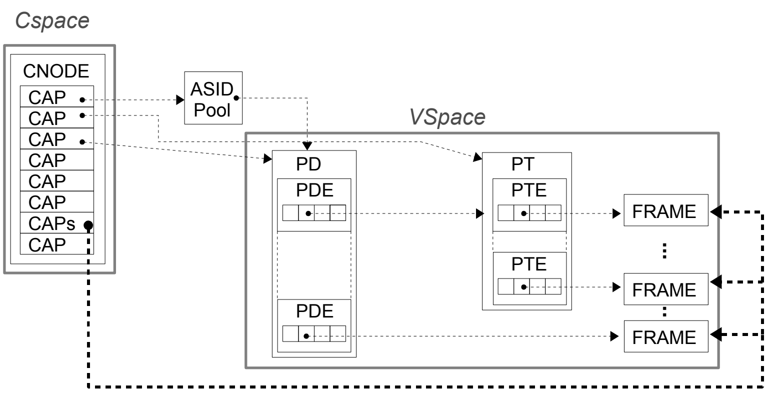


Figure 1.4: Virtual Memory in seL4.

These VSpace-related objects are sufficient to implement the hardware data structures required to create, manipulate, and destroy virtual memory address spaces. It should be noted that, as usual, the manipulator of a virtual memory space needs the appropriate capabilities to the required objects.

Page Directory The Page Directory (PD) is the top-level page table of the ARM two-level page table structure. It has a hardware defined format, but conceptually contains 1024 page directory entries (PDE), which are one of a pointer to a page table, a 4 megabyte Page, or an invalid entry . The Page

[1] A *pager* is a term for a thread that manages the VSpace of another application. For example, Linux would be called the pager of its applications.

Directory has no methods itself, but it is used as an argument to several other virtual memory related object calls.

Page Table The Page Table object forms the second level of the ARM page table. It contains 1024 slots, each of which contains a page table entry (PTE). A page table entry contains either an invalid entry, or a pointer to a 4 kilobyte Page.

Page Table objects possess only a single method:

`Map()` takes a Page Directory capability as an argument, and installs a reference to the invoked Page Table to a specified slot in the Page Directory.

Page A Page object is a region of physical memory that is used to implement virtual memory pages in a virtual address space. The Page object has the following methods:

`Map()` takes a Page Directory or a Page Table capability as an argument and installs a PDE or PTE referring to the Page in the specified location, respectively.

`Remap()` changes the permissions of an existing mapping.

`Unmap()` removes an existing mapping.

ASID Control For internal kernel book-keeping purposes, there is a fixed maximum number of applications the system can support. In order to manage this limited resource, the microkernel provides an ASID Control capability. The ASID Control capability is used to generate a capability that authorises the use of a subset of available address space identifiers. This newly created capability is called an ASID Pool. ASID Control only has a single method:

`MakePool()` together with a capability to Untyped Memory (described shortly) as argument creates an ASID Pool.

ASID Pool An ASID Pool confers the right to create a subset of the available maximum applications. For a VSpace to be usable by an application, it must be assigned to an ASID. This is done using a capability to an ASID Pool. The ASID Pool object has a single method:

`Assign()` assigns an ASID to the VSpace associated with the Page Directory passed in as an argument.

Interrupt Objects

Device driver applications need the ability to receive and acknowledge interrupts from hardware devices.

A capability to IRQControl has the ability to create a new capability to manage a specific interrupt source associated with a specific device. The new capability is then delegated to a device driver to access an interrupt source. IRQControl has one method:

`Get()` creates an IRQHandler capability for the specified interrupt source.

An IRQHandler object is used by driver application to handle interrupts for the device it manages. It has three methods:

`SetEndpoint()` specifies the AsyncEP that a `Notify()` should be sent to when an interrupt occurs. The driver application usually `Wait()`-s on this endpoint for interrupts to process.

`Ack()` informs the kernel that the userspace driver has finished processing the interrupt and the microkernel can send further pending or new interrupts to the application.

`Clear()` de-registers the AsyncEP from the IRQHandler object.

Untyped Memory

The Untyped Memory object is the foundation of memory allocation in the seL4 kernel. Untyped memory capabilities have a single method:

`Retype()` creates a number of new kernel objects. If this method succeeds, it returns capabilities to the newly-created objects.

In particular, untyped memory objects can be divided into a group of smaller untyped memory objects. We discuss memory management in general in the following section.

1.1.4 Kernel Memory Allocation

The seL4 microkernel has no internal memory allocator: all kernel objects must be explicitly created from application controlled memory regions via Untyped Memory capabilities. Applications must have explicit authority to memory (via Untyped Memory capabilities) in order to create other services, and services consume no extra memory once created (other than the amount of untyped memory from which they were originally created). The mechanisms can be used to precisely control the specific amount of physical memory available to applications, including being able to enforce isolation of physical memory access between applications or a device. Thus, there are no arbitrary resource limits in the kernel apart from those dictated by the hardware[2], and so many denial-of-service attacks via resource exhaustion are obviated.

At boot time, seL4 pre-allocates all the memory required for the kernel itself, including the code, data, and stack sections (seL4 is a single kernel-stack operating system). The remainder of the memory is given to the first task in the form of capabilities to Untyped Memory, and some additional capabilities to kernel objects that were required to bootstrap the supervisor task. These objects can then be split into smaller untyped memory regions or other kernel objects using the `Retype()` method; the created objects are termed *children* of the original untyped memory object.

See Figure 1.5 for an example.

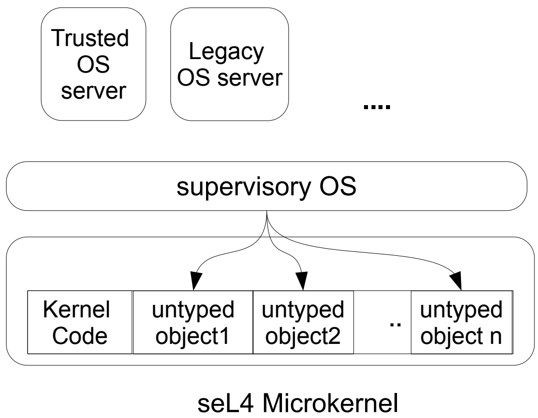


Figure 1.5: Memory layout at boot time

The user-level application that creates an object using `Retype()` receives full authority over the resulting object. It can then delegate all or part of the authority it possesses over this object to one or more of its clients. This is done by selectively granting each client a capability to the kernel object, thereby allowing the client to obtain kernel services by invoking the object.

For obvious security reasons, kernel data must be protected from user access. The seL4 kernel prevents such access by using two mechanisms. First, the above allocation policy guarantees that typed objects never overlap. Second, the kernel ensures that each physical frame mapped by the MMU at a user-accessible address corresponds to a Page object (described above); Page objects contain no kernel data, so direct user access to kernel data is not possible. All other kernel objects are only indirectly manipulated via their corresponding capabilities.

[2] The treatment of virtual ASIDs imposes a fixed number of address spaces, but this limitation is to be removed in future versions of seL4.

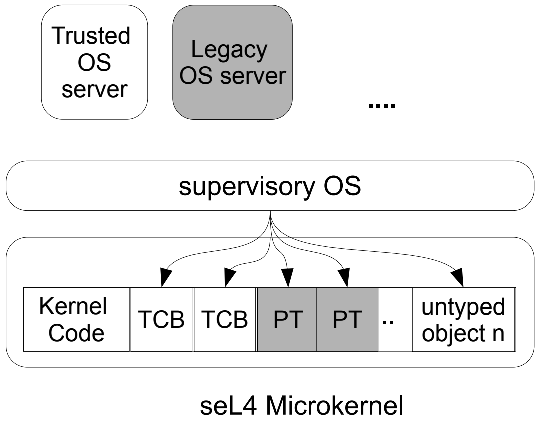


Figure 1.6: Memory layout after supervisor creates kernel services.

Re-using Memory

The model described thus far is sufficient for applications to allocate kernel objects, distribute authority among client applications, and obtain various kernel services provided by these objects. This alone is sufficient for a simple static system configuration.

The seL4 kernel also allows memory re-use. Reusing a region of memory is sound only when there are no dangling references (e.g. capabilities) left to the objects implemented by that memory. The kernel tracks *capability derivations*, that is, the children generated by various `CNode` methods (`Retype()`, `Mint()`, `Copy()`, and `Mutate()`). Whenever a user requests that the kernel create new objects in an untyped memory region, the kernel uses this information to check that there are no children in the region, and thus no live capability references.

The tree structure so generated is termed the *capability derivation tree* (CDT)[3]. For example, when a user creates new kernel objects by retyping untyped memory, the newly created capabilities would be inserted into the CDT as children of the untyped memory capability.

Finally, recall that the `Revoke()` operation destroys all capabilities derived from the argument capability. Revoking the last capability to a kernel object is easily detectable, and triggers the *destroy* operation on the now unreferenced object. Destroy simply deactivates the object if it was active, and cleans up any in-kernel dependencies between it and other objects.

By calling `Revoke()` on the original capability to an untyped memory object, the user removes all of the untyped memory object's children — that is, all capabilities pointing to objects in the untyped memory region. Thus, after this operation there are no valid references to any object within the untyped region, and the region may be safely retyped and reused.

1.2 Summary

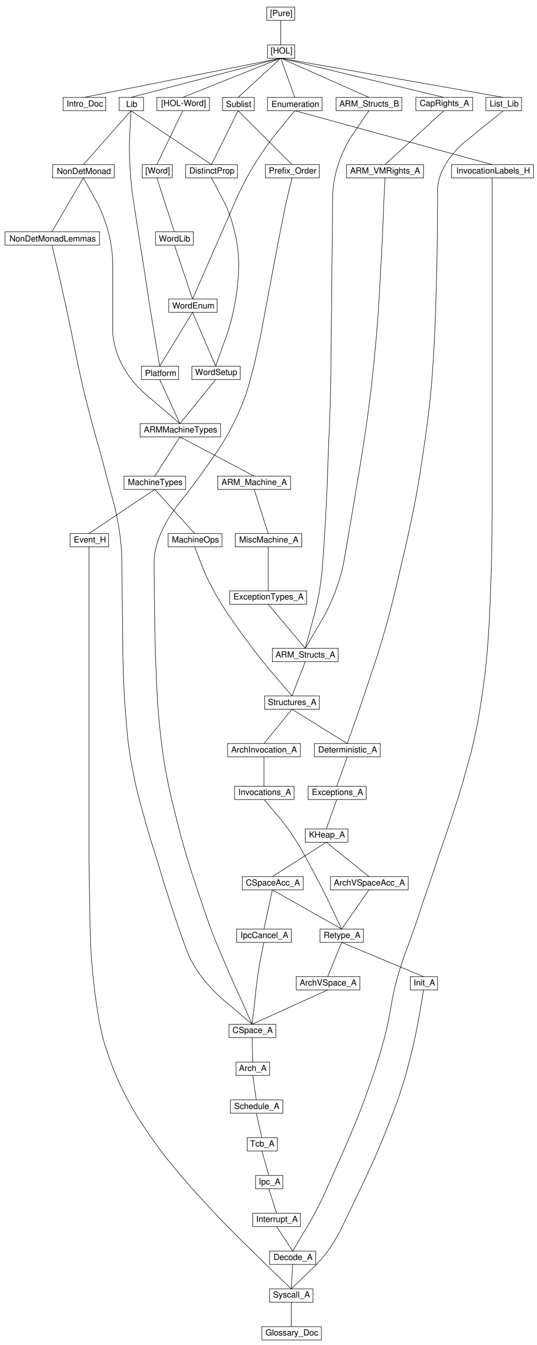
This chapter has given an overview of the seL4 microkernel. The following chapters are generated from the formal Isabelle/HOL definitions that comprise the formal specification of the seL4 kernel on the ARM11 architecture. The specification does not cover any other architectures or platforms.

The order of definitions in this document is as processed by Isabelle/HOL: bottom up. All concepts are defined before first used. This means the first chapters mainly introduce basic data types and structures while the top-level kernel entry point is defined in the last chapter (chapter 39). The following section shows the dependency graph between the theory modules in this specification. We assume a familiarity with Isabelle syntax; see Nipkow et al. [15] for an introduction. In addition to the

[3]Although we model the CDT as a separate data structure, it is implemented as part of the CNode object and so requires no additional kernel meta-data.

standard Isabelle/HOL notation, we sometimes write `f $ x` for `(f x)` and use monadic do-notation extensively. The latter is defined in chapter 2.

1.3 Theory Dependencies



2 Nondeterministic State Monad with Failure

```
theory NonDetMonad
imports "../Lib"
begin
```

State monads are used extensively in the seL4 specification. They are defined below.

2.1 The Monad

The basic type of the nondeterministic state monad with failure is very similar to the normal state monad. Instead of a pair consisting of result and new state, we return a set of these pairs coupled with a failure flag. Each element in the set is a potential result of the computation. The flag is `True` if there is an execution path in the computation that may have failed. Conversely, if the flag is `False`, none of the computations resulting in the returned set can have failed.

```
type_synonym ('s,'a) nondet_monad = "'s ⇒ ('a × 's) set × bool"
```

The definition of fundamental monad functions `return` and `bind`. The monad function `return x` does not change the state, does not fail, and returns `x`.

```
definition
  return :: "'a ⇒ ('s,'a) nondet_monad" where
  "return a ≡ λs. ({(a,s)},False)"
```

The monad function `bind f g`, also written `f >>= g`, is the execution of `f` followed by the execution of `g`. The function `g` takes the result value *and* the result state of `f` as parameter. The definition says that the result of the combined operation is the union of the set of sets that is created by `g` applied to the result sets of `f`. The combined operation may have failed, if `f` may have failed or `g` may have failed on any of the results of `f`.

```
definition
  bind :: "('s, 'a) nondet_monad ⇒ ('a ⇒ ('s, 'b) nondet_monad) ⇒
           ('s, 'b) nondet_monad" (infixl ">>=" 60)
  where
  "bind f g ≡ λs. (⋃(fst ` split g ` fst (f s)),
                   True ∈ snd ` split g ` fst (f s) ∨ snd (f s))"
```

Sometimes it is convenient to write `bind` in reverse order.

```
abbreviation(input)
  bind_rev :: "('c ⇒ ('a, 'b) nondet_monad) ⇒ ('a, 'c) nondet_monad ⇒
               ('a, 'b) nondet_monad" (infixl "=<<" 60) where
  "g =<< f ≡ f >>= g"
```

The basic accessor functions of the state monad. `get` returns the current state as result, does not fail, and does not change the state. `put s` returns nothing (`unit`), changes the current state to `s` and does not fail.

```
definition
  get :: "('s,'s) nondet_monad" where
  "get ≡ λs. ({(s,s)}, False)"
```

```
definition
```

```
  put :: "'s ⇒ ('s, unit) nondet_monad" where
  "put s ≡ λ_. ({((),s)}, False)"
```

2.1.1 Nondeterminism

Basic nondeterministic functions. `select A` chooses an element of the set `A`, does not change the state, and does not fail (even if the set is empty). `f OR g` executes `f` or executes `g`. It retuns the union of results of `f` and `g`, and may have failed if either may have failed.

```
definition
  select :: "'a set ⇒ ('s,'a) nondet_monad" where
  "select A ≡ λs. (A <*> {s}, False)"
```

```
definition
  alternative :: "('s,'a) nondet_monad ⇒ ('s,'a) nondet_monad ⇒
                 ('s,'a) nondet_monad"
  (infixl "OR" 20)
where
  "f OR g ≡ λs. (fst (f s) ∪ fst (g s), snd (f s) ∨ snd (g s))"
```

Alternative notation for `OR`

```
notation (xsymbols)  alternative (infixl "⊓" 20)
```

A variant of `select` that takes a pair. The first component is a set as in normal `select`, the second component indicates whether the execution failed. This is useful to lift monads between different state spaces.

```
definition
  select_f :: "'a set × bool  ⇒ ('s,'a) nondet_monad" where
  "select_f S ≡ λs. (fst S × {s}, snd S)"
```

`select_state` takes a relationship between states, and outputs nondeterministically a state related to the input state.

```
definition
  state_select :: "('s × 's) set ⇒ ('s, unit) nondet_monad"
where
  "state_select r ≡ λs. ((λx. ((), x)) ` {s'. (s, s') ∈ r}, ¬ (∃s'. (s, s') ∈ r))"
```

2.1.2 Failure

The monad function that always fails. Returns an empty set of results and sets the failure flag.

```
definition
  fail :: "('s, 'a) nondet_monad" where
 "fail ≡ λs. ({}, True)"
```

Assertions: fail if the property `P` is not true

```
definition
  assert :: "bool ⇒ ('a, unit) nondet_monad" where
 "assert P ≡ if P then return () else fail"
```

Fail if the value is `None`, return result `v` for `Some v`

```
definition
  assert_opt :: "'a option ⇒ ('b, 'a) nondet_monad" where
 "assert_opt v ≡ case v of None ⇒ fail | Some v ⇒ return v"
```

An assertion that also can introspect the current state.

```
definition
  state_assert :: "('s ⇒ bool) ⇒ ('s, unit) nondet_monad"
where
  "state_assert P ≡ get >>= (λs. assert (P s))"
```

2.1.3 Generic functions on top of the state monad

Apply a function to the current state and return the result without changing the state.

```
definition
  gets :: "('s ⇒ 'a) ⇒ ('s, 'a) nondet_monad" where
 "gets f ≡ get >>= (λs. return (f s))"
```

Modify the current state using the function passed in.

```
definition
  modify :: "('s ⇒ 's) ⇒ ('s, unit) nondet_monad" where
 "modify f ≡ get >>= (λs. put (f s))"

lemma simpler_gets_def: "gets f = (λs. ({(f s, s)}, False))"

lemma simpler_modify_def:
  "modify f = (λs. ({((), f s)}, False))"
```

Execute the given monad when the condition is true, return `()` otherwise.

```
definition
  when :: "bool ⇒ ('s, unit) nondet_monad ⇒
          ('s, unit) nondet_monad" where
  "when P m ≡ if P then m else return ()"
```

Execute the given monad unless the condition is true, return `()` otherwise.

```
definition
  unless :: "bool ⇒ ('s, unit) nondet_monad ⇒
            ('s, unit) nondet_monad" where
  "unless P m ≡ when (¬P) m"
```

Perform a test on the current state, performing the left monad if the result is true or the right monad if the result is false.

```
definition
  condition :: "('s ⇒ bool)
      ⇒ ('s, 'r) nondet_monad
      ⇒ ('s, 'r) nondet_monad
      ⇒ ('s, 'r) nondet_monad"
where
  "(condition P L R) ≡ λs. if (P s) then (L s) else (R s)"

notation (output)
  condition  ("(condition (_)//  (_)//  (_))" [1000,1000,1000] 1000)
```

Apply an option valued function to the current state, fail if it returns `None`, return `v` if it returns `Some v`.

```
definition
  gets_the :: "('s ⇒ 'a option) ⇒ ('s, 'a) nondet_monad" where
 "gets_the f ≡ gets f >>= assert_opt"
```

2.1.4 The Monad Laws

A more expanded definition of `bind`

```
lemma bind_def':
  "(f >>= g) ≡
      λs. ({(r'', s''). ∃(r', s') ∈ fst (f s). (r'', s'') ∈ fst (g r' s') },
                  snd (f s) ∨ (∃(r', s') ∈ fst (f s). snd (g r' s')))"
```

Each monad satisfies at least the following three laws.

`return` is absorbed at the left of a `op >>=`, applying the return value directly:

```
lemma return_bind [simp]: "(return x >>= f) = f x"
```

`return` is absorbed on the right of a `op >>=`

```
lemma bind_return [simp]: "(m >>= return) = m"
```

`op >>=` is associative

```
lemma bind_assoc:
  fixes m :: "('a,'b) nondet_monad"
  fixes f :: "'b ⇒ ('a,'c) nondet_monad"
  fixes g :: "'c ⇒ ('a,'d) nondet_monad"
  shows "(m >>= f) >>= g  =  m >>= (λx. f x >>= g)"
```

2.2 Adding Exceptions

The type `('s, 'a) nondet_monad` gives us nondeterminism and failure. We now extend this monad with exceptional return values that abort normal execution, but can be handled explicitly. We use the sum type to indicate exceptions.

In `('s, 'e + 'a) nondet_monad`, `'s` is the state, `'e` is an exception, and `'a` is a normal return value.

This new type itself forms a monad again. Since type classes in Isabelle are not powerful enough to express the class of monads, we provide new names for the `return` and `op >>=` functions in this monad. We call them `returnOk` (for normal return values) and `bindE` (for composition). We also define `throwError` to return an exceptional value.

```
definition
  returnOk :: "'a ⇒ ('s, 'e + 'a) nondet_monad" where
  "returnOk ≡ return o Inr"
```

```
definition
  throwError :: "'e ⇒ ('s, 'e + 'a) nondet_monad" where
  "throwError ≡ return o Inl"
```

Lifting a function over the exception type: if the input is an exception, return that exception; otherwise continue execution.

```
definition
  lift :: "('a ⇒ ('s, 'e + 'b) nondet_monad) ⇒
           'e +'a ⇒ ('s, 'e + 'b) nondet_monad"
where
  "lift f v ≡ case v of Inl e ⇒ throwError e
                      | Inr v' ⇒ f v'"
```

The definition of `op >>=` in the exception monad (new name `bindE`): the same as normal `op >>=`, but the right-hand side is skipped if the left-hand side produced an exception.

```
definition
  bindE :: "('s, 'e + 'a) nondet_monad ⇒
            ('a ⇒ ('s, 'e + 'b) nondet_monad) ⇒
            ('s, 'e + 'b) nondet_monad"  (infixl ">>=E" 60)
where
  "bindE f g ≡ bind f (lift g)"
```

Lifting a normal nondeterministic monad into the exception monad is achieved by always returning its result as normal result and never throwing an exception.

```
definition
  liftE :: "('s,'a) nondet_monad ⇒ ('s, 'e+'a) nondet_monad"
where
  "liftE f ≡ f >>= (λr. return (Inr r))"
```

Since the underlying type and `return` function changed, we need new definitions for when and unless:

```
definition
  whenE :: "bool ⇒ ('s, 'e + unit) nondet_monad ⇒
            ('s, 'e + unit) nondet_monad"
  where
  "whenE P f ≡ if P then f else returnOk ()"
```

```
definition
  unlessE :: "bool ⇒ ('s, 'e + unit) nondet_monad ⇒
            ('s, 'e + unit) nondet_monad"
  where
  "unlessE P f ≡ if P then returnOk () else f"
```

Throwing an exception when the parameter is `None`, otherwise returning `v` for `Some v`.

```
definition
  throw_opt :: "'e ⇒ 'a option ⇒ ('s, 'e + 'a) nondet_monad" where
  "throw_opt ex x ≡
  case x of None ⇒ throwError ex | Some v ⇒ returnOk v"
```

Failure in the exception monad is redefined in the same way as `whenE` and `unlessE`, with `returnOk` instead of `return`.

```
definition
  assertE :: "bool ⇒ ('a, 'e + unit) nondet_monad" where
 "assertE P ≡ if P then returnOk () else fail"
```

2.2.1 Monad Laws for the Exception Monad

More direct definition of `liftE`:

```
lemma liftE_def2:
  "liftE f = (λs. ((λ(v,s'). (Inr v, s')) ` fst (f s), snd (f s)))"
```

Left `returnOk` absorbtion over `op >>=E`:

```
lemma returnOk_bindE [simp]: "(returnOk x >>=E f) = f x"
```

```
lemma lift_return [simp]:
  "lift (return ∘ Inr) = return"
```

Right `returnOk` absorbtion over `op >>=E`:

```
lemma bindE_returnOk [simp]: "(m >>=E returnOk) = m"
```

Associativity of `op >>=E`:

```
lemma bindE_assoc:
  "(m >>=E f) >>=E g = m >>=E (λx. f x >>=E g)"
```

`returnOk` could also be defined via `liftE`:

```
lemma returnOk_liftE:
  "returnOk x = liftE (return x)"
```

Execution after throwing an exception is skipped:

```
lemma throwError_bindE [simp]:
  "(throwError E >>=E f) = throwError E"
```

2.3 Syntax

This section defines traditional Haskell-like do-syntax for the state monad in Isabelle.

2.3.1 Syntax for the Nondeterministic State Monad

We use `K_bind` to syntactically indicate the case where the return argument of the left side of a `op >>=` is ignored

```
definition
  K_bind_def [iff]: "K_bind ≡ λx y. x"

nonterminal
  dobinds and dobind and nobind

syntax
  "_dobind"    :: "[pttrn, 'a] => dobind"            ("(_ <-/ _)" 10)
  ""           :: "dobind => dobinds"                ("_")
  "_nobind"    :: "'a => dobind"                     ("_")
  "_dobinds"   :: "[dobind, dobinds] => dobinds"     ("(_);//(_)")

  "_do"        :: "[dobinds, 'a] => 'a"              ("(do ((_);//(_))//od)" 100)
syntax (xsymbols)
  "_dobind"    :: "[pttrn, 'a] => dobind"            ("(_ ←/ _)" 10)

translations
  "_do (_dobinds b bs) e"  == "_do b (_do bs e)"
  "_do (_nobind b) e"      == "b >>= (CONST K_bind e)"
  "do x <- a; e od"        == "a >>= (λx. e)"
```

Syntax examples:

```
lemma "do x ← return 1;
          return (2::nat);
          return x
       od =
       return 1 >>=
```

```
       (λx. return (2::nat) >>=
            K_bind (return x))"

lemma "do x ← return 1;
          return 2;
          return x
       od = return 1"
```

2.3.2 Syntax for the Exception Monad

Since the exception monad is a different type, we need to syntactically distinguish it in the syntax. We use doE/odE for this, but can re-use most of the productions from do/od above.

```
syntax
  "_doE" :: "[dobinds, 'a] => 'a"  ("(doE ((_);//(_))//odE)" 100)

translations
  "_doE (_dobinds b bs) e"  == "_doE b (_doE bs e)"
  "_doE (_nobind b) e"      == "b >>=E (CONST K_bind e)"
  "doE x <- a; e odE"       == "a >>=E (λx. e)"
```

Syntax examples:

```
lemma "doE x ← returnOk 1;
           returnOk (2::nat);
           returnOk x
       odE =
       returnOk 1 >>=E
       (λx. returnOk (2::nat) >>=E
            K_bind (returnOk x))"

lemma "doE x ← returnOk 1;
           returnOk 2;
           returnOk x
       odE = returnOk 1"
```

2.4 Library of Monadic Functions and Combinators

Lifting a normal function into the monad type:

```
definition
  liftM :: "('a ⇒ 'b) ⇒ ('s,'a) nondet_monad ⇒ ('s, 'b) nondet_monad"
where
  "liftM f m ≡ do x ← m; return (f x) od"
```

The same for the exception monad:

```
definition
  liftME :: "('a ⇒ 'b) ⇒ ('s,'e+'a) nondet_monad ⇒ ('s,'e+'b) nondet_monad"
where
  "liftME f m ≡ doE x ← m; returnOk (f x) odE"
```

Run a sequence of monads from left to right, ignoring return values.

```
definition
  sequence_x :: "('s, 'a) nondet_monad list ⇒ ('s, unit) nondet_monad"
where
  "sequence_x xs ≡ foldr (λx y. x >>= (λ_. y)) xs (return ())"
```

Map a monadic function over a list by applying it to each element of the list from left to right, ignoring return values.

definition
```
  mapM_x :: "('a ⇒ ('s,'b) nondet_monad) ⇒ 'a list ⇒ ('s, unit) nondet_monad"
```
where
```
  "mapM_x f xs ≡ sequence_x (map f xs)"
```

Map a monadic function with two parameters over two lists, going through both lists simultanously, left to right, ignoring return values.

definition
```
  zipWithM_x :: "('a ⇒ 'b ⇒ ('s,'c) nondet_monad) ⇒
                 'a list ⇒ 'b list ⇒ ('s, unit) nondet_monad"
```
where
```
  "zipWithM_x f xs ys ≡ sequence_x (zipWith f xs ys)"
```

The same three functions as above, but returning a list of return values instead of `unit`

definition
```
  sequence :: "('s, 'a) nondet_monad list ⇒ ('s, 'a list) nondet_monad"
```
where
```
  "sequence xs ≡ let mcons = (λp q. p >>= (λx. q >>= (λy. return (x#y))))
                 in foldr mcons xs (return [])"
```

definition
```
  mapM :: "('a ⇒ ('s,'b) nondet_monad) ⇒ 'a list ⇒ ('s, 'b list) nondet_monad"
```
where
```
  "mapM f xs ≡ sequence (map f xs)"
```

definition
```
  zipWithM :: "('a ⇒ 'b ⇒ ('s,'c) nondet_monad) ⇒
                 'a list ⇒ 'b list ⇒ ('s, 'c list) nondet_monad"
```
where
```
  "zipWithM f xs ys ≡ sequence (zipWith f xs ys)"
```

definition
```
  foldM :: "('b ⇒ 'a ⇒ ('s, 'a) nondet_monad) ⇒ 'b list ⇒ 'a ⇒ ('s, 'a) nondet_monad"
```
where
```
  "foldM m xs a ≡ foldr (λp q. q >>= m p) xs (return a) "
```

The sequence and map functions above for the exception monad, with and without lists of return value

definition
```
  sequenceE_x :: "('s, 'e+'a) nondet_monad list ⇒ ('s, 'e+unit) nondet_monad"
```
where
```
  "sequenceE_x xs ≡ foldr (λx y. doE _ <- x; y odE) xs (returnOk ())"
```

definition
```
  mapME_x :: "('a ⇒ ('s,'e+'b) nondet_monad) ⇒ 'a list ⇒
              ('s,'e+unit) nondet_monad"
```
where
```
  "mapME_x f xs ≡ sequenceE_x (map f xs)"
```

definition
```
  sequenceE :: "('s, 'e+'a) nondet_monad list ⇒ ('s, 'e+'a list) nondet_monad"
```
where
```
  "sequenceE xs ≡ let mcons = (λp q. p >>=E (λx. q >>=E (λy. returnOk (x#y))))
                 in foldr mcons xs (returnOk [])"
```

```
definition
  mapME :: "('a ⇒ ('s,'e+'b) nondet_monad) ⇒ 'a list ⇒
             ('s,'e+'b list) nondet_monad"
where
  "mapME f xs ≡ sequenceE (map f xs)"
```

Filtering a list using a monadic function as predicate:

```
primrec
  filterM :: "('a ⇒ ('s, bool) nondet_monad) ⇒ 'a list ⇒ ('s, 'a list) nondet_monad"
where
  "filterM P []       = return []"
| "filterM P (x # xs) = do
     b  <- P x;
     ys <- filterM P xs;
     return (if b then (x # ys) else ys)
   od"
```

2.5 Catching and Handling Exceptions

Turning an exception monad into a normal state monad by catching and handling any potential exceptions:

```
definition
  catch :: "('s, 'e + 'a) nondet_monad ⇒
            ('e ⇒ ('s, 'a) nondet_monad) ⇒
            ('s, 'a) nondet_monad" (infix "<catch>" 10)
where
  "f <catch> handler ≡
     do x ← f;
        case x of
          Inr b ⇒ return b
        | Inl e ⇒ handler e
     od"
```

Handling exceptions, but staying in the exception monad. The handler may throw a type of exceptions different from the left side.

```
definition
  handleE' :: "('s, 'e1 + 'a) nondet_monad ⇒
               ('e1 ⇒ ('s, 'e2 + 'a) nondet_monad) ⇒
               ('s, 'e2 + 'a) nondet_monad" (infix "<handle2>" 10)
where
  "f <handle2> handler ≡
   do
      v ← f;
      case v of
        Inl e ⇒ handler e
      | Inr v' ⇒ return (Inr v')
   od"
```

A type restriction of the above that is used more commonly in practice: the exception handle (potentially) throws exception of the same type as the left-hand side.

```
definition
  handleE :: "('s, 'x + 'a) nondet_monad ⇒
             ('x ⇒ ('s, 'x + 'a) nondet_monad) ⇒
             ('s, 'x + 'a) nondet_monad" (infix "<handle>" 10)
```

```
where
  "handleE ≡ handleE'"
```

Handling exceptions, and additionally providing a continuation if the left-hand side throws no exception:

```
definition
  handle_elseE :: "('s, 'e + 'a) nondet_monad ⇒
                   ('e ⇒ ('s, 'ee + 'b) nondet_monad) ⇒
                   ('a ⇒ ('s, 'ee + 'b) nondet_monad) ⇒
                   ('s, 'ee + 'b) nondet_monad"
  ("_ <handle> _ <else> _" 10)
where
  "f <handle> handler <else> continue ≡
   do v ← f;
   case v of Inl e  ⇒ handler e
          | Inr v' ⇒ continue v'
   od"
```

2.5.1 Loops

Loops are handled using the following inductive predicate; non-termination is represented using the failure flag of the monad.

```
inductive_set
  whileLoop_results :: "('r ⇒ 's ⇒ bool) ⇒ ('r ⇒ ('s, 'r) nondet_monad) ⇒ ((('r × 's) option)
× (('r × 's) option)) set"
  for C B
where
    "(None, None) ∈ whileLoop_results C B"
  | "⟦ ¬ C r s ⟧ ⟹ (Some (r, s), Some (r, s)) ∈ whileLoop_results C B"
  | "⟦ C r s; snd (B r s) ⟧ ⟹ (Some (r, s), None) ∈ whileLoop_results C B"
  | "⟦ C r s; (r', s') ∈ fst (B r s); (Some (r', s'), z) ∈ whileLoop_results C B  ⟧
      ⟹ (Some (r, s), z) ∈ whileLoop_results C B"

inductive_cases whileLoop_results_cases_valid: "(Some x, Some y) ∈ whileLoop_results C B"
inductive_cases whileLoop_results_cases_fail: "(Some x, None) ∈ whileLoop_results C B"
inductive_simps whileLoop_results_simps: "(Some x, y) ∈ whileLoop_results C B"
inductive_simps whileLoop_results_simps_valid: "(Some x, Some y) ∈ whileLoop_results C B"
inductive_simps whileLoop_results_simps_start_fail [simp]: "(None, x) ∈ whileLoop_results C B"

inductive
  whileLoop_terminates :: "('r ⇒ 's ⇒ bool) ⇒ ('r ⇒ ('s, 'r) nondet_monad) ⇒ 'r ⇒ 's ⇒ bool"
  for C B
where
    "¬ C r s ⟹ whileLoop_terminates C B r s"
  | "⟦ C r s; ∀(r', s') ∈ fst (B r s). whileLoop_terminates C B r' s' ⟧
       ⟹ whileLoop_terminates C B r s"

inductive_cases whileLoop_terminates_cases: "whileLoop_terminates C B r s"
inductive_simps whileLoop_terminates_simps: "whileLoop_terminates C B r s"

definition
  "whileLoop C B ≡ (λr s.
     ({(r',s'). (Some (r, s), Some (r', s')) ∈ whileLoop_results C B},
        (Some (r, s), None) ∈ whileLoop_results C B ∨ (¬ whileLoop_terminates C B r s)))"

notation (output)
  whileLoop  ("(whileLoop (_)//  (_))" [1000, 1000] 1000)
```

```
definition
  whileLoopE :: "('r ⇒ 's ⇒ bool) ⇒ ('r ⇒ ('s, 'e + 'r) nondet_monad)
      ⇒ 'r ⇒ 's ⇒ (('e + 'r) × 's) set × bool"
where
  "whileLoopE C body ≡
      λr. whileLoop (λr s. (case r of Inr v ⇒ C v s | _ ⇒ False)) (lift body) (Inr r)"

notation (output)
  whileLoopE  ("(whileLoopE (_)//  (_))" [1000, 1000] 1000)
```

2.6 Hoare Logic

2.6.1 Validity

This section defines a Hoare logic for partial correctness for the nondeterministic state monad as well as the exception monad. The logic talks only about the behaviour part of the monad and ignores the failure flag.

The logic is defined semantically. Rules work directly on the validity predicate.

In the nondeterministic state monad, validity is a triple of precondition, monad, and postcondition. The precondition is a function from state to bool (a state predicate), the postcondition is a function from return value to state to bool. A triple is valid if for all states that satisfy the precondition, all result values and result states that are returned by the monad satisfy the postcondition. Note that if the computation returns the empty set, the triple is trivially valid. This means `assert P` does not require us to prove that `P` holds, but rather allows us to assume `P`! Proving non-failure is done via separate predicate and calculus (see below).

```
definition
  valid :: "('s ⇒ bool) ⇒ ('s,'a) nondet_monad ⇒ ('a ⇒ 's ⇒ bool) ⇒ bool"
  ("⦃_⦄/ _ /⦃_⦄")
where
  "⦃P⦄ f ⦃Q⦄ ≡ ∀s. P s ⟶ (∀(r,s') ∈ fst (f s). Q r s')"
```

Validity for the exception monad is similar and build on the standard validity above. Instead of one postcondition, we have two: one for normal and one for exceptional results.

```
definition
  validE :: "('s ⇒ bool) ⇒ ('s, 'a + 'b) nondet_monad ⇒
             ('b ⇒ 's ⇒ bool) ⇒
             ('a ⇒ 's ⇒ bool) ⇒ bool"
("⦃_⦄/ _ /(⦃_⦄,/ ⦃_⦄)")
where
  "⦃P⦄ f ⦃Q⦄,⦃E⦄ ≡ ⦃P⦄ f ⦃ λv s. case v of Inr r ⇒ Q r s | Inl e ⇒ E e s ⦄"
```

The following two instantiations are convenient to separate reasoning for exceptional and normal case.

```
definition
  validE_R :: "('s ⇒ bool) ⇒ ('s, 'e + 'a) nondet_monad ⇒
               ('a ⇒ 's ⇒ bool) ⇒ bool"
   ("⦃_⦄/ _ /⦃_⦄, -")
where
 "⦃P⦄ f ⦃Q⦄,- ≡ validE P f Q (λx y. True)"

definition
  validE_E :: "('s ⇒ bool) ⇒  ('s, 'e + 'a) nondet_monad ⇒
               ('e ⇒ 's ⇒ bool) ⇒ bool"
   ("⦃_⦄/ _ /-, ⦃_⦄")
where
```

```
"⦃P⦄ f -,⦃Q⦄ ≡ validE P f (λx y. True) Q"
```

Abbreviations for trivial preconditions:

```
abbreviation(input)
  top :: "'a ⇒ bool" ("⊤")
where
  "⊤ ≡ λ_. True"

abbreviation(input)
  bottom :: "'a ⇒ bool" ("⊥")
where
  "⊥ ≡ λ_. False"
```

Abbreviations for trivial postconditions (taking two arguments):

```
abbreviation(input)
  toptop :: "'a ⇒ 'b ⇒ bool" ("⊤⊤")
where
 "⊤⊤ ≡ λ_ _. True"

abbreviation(input)
  botbot :: "'a ⇒ 'b ⇒ bool" ("⊥⊥")
where
 "⊥⊥ ≡ λ_ _. False"
```

Lifting ∧ and ∨ over two arguments. Lifting ∧ and ∨ over one argument is already defined (written `and` and `or`).

```
definition
  bipred_conj :: "('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool)"
  (infixl "And" 96)
where
  "bipred_conj P Q ≡ λx y. P x y ∧ Q x y"

definition
  bipred_disj :: "('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool)"
  (infixl "Or" 91)
where
  "bipred_disj P Q ≡ λx y. P x y ∨ Q x y"
```

2.6.2 Determinism

A monad of type `nondet_monad` is deterministic iff it returns exactly one state and result and does not fail

```
definition
  det :: "('a,'s) nondet_monad ⇒ bool"
where
  "det f ≡ ∀s. ∃r. f s = ({r},False)"
```

A deterministic `nondet_monad` can be turned into a normal state monad:

```
definition
  the_run_state :: "('s,'a) nondet_monad ⇒ 's ⇒ 'a × 's"
where
  "the_run_state M ≡ λs. THE s'. fst (M s) = {s'}"
```

2.6.3 Non-Failure

With the failure flag, we can formulate non-failure separately from validity. A monad `m` does not fail under precondition `P`, if for no start state in that precondition it sets the failure flag.

```
definition
  no_fail :: "('s ⇒ bool) ⇒ ('s,'a) nondet_monad ⇒ bool"
where
  "no_fail P m ≡ ∀s. P s ⟶ ¬ (snd (m s))"
```

It is often desired to prove non-failure and a Hoare triple simulataneously, as the reasoning is often similar. The following definitions allow such reasoning to take place.

```
definition
  validNF ::"('s ⇒ bool) ⇒ ('s,'a) nondet_monad ⇒ ('a ⇒ 's ⇒ bool) ⇒ bool"
      ("⦃_⦄/ _ /⦃_⦄!")
where
  "validNF P f Q ≡ valid P f Q ∧ no_fail P f"

definition
  validE_NF :: "('s ⇒ bool) ⇒ ('s, 'a + 'b) nondet_monad ⇒
             ('b ⇒ 's ⇒ bool) ⇒
             ('a ⇒ 's ⇒ bool) ⇒ bool"
  ("⦃_⦄/ _ /(⦃_⦄,/ ⦃_⦄!)")
where
  "validE_NF P f Q E ≡ validE P f Q E ∧ no_fail P f"

lemma validE_NF_alt_def:
  "⦃ P ⦄ B ⦃ Q ⦄,⦃ E ⦄! = ⦃ P ⦄ B ⦃ λv s. case v of Inl e ⇒ E e s | Inr r ⇒ Q r s ⦄!"
```

Usually, well-formed monads constructed from the primitives above will have the following property: if they return an empty set of results, they will have the failure flag set.

```
definition
  empty_fail :: "('s,'a) nondet_monad ⇒ bool"
where
  "empty_fail m ≡ ∀s. fst (m s) = {} ⟶ snd (m s)"
```

Useful in forcing otherwise unknown executions to have the `empty_fail` property.

```
definition
  mk_ef :: "'a set × bool ⇒ 'a set × bool"
where
  "mk_ef S ≡ (fst S, fst S = {} ∨ snd S)"
```

2.7 Basic exception reasoning

The following predicates `no_throw` and `no_return` allow reasoning that functions in the exception monad either do no throw an exception or never return normally.

```
definition "no_throw P A ≡ ⦃ P ⦄ A ⦃ λ_ _. True ⦄,⦃ λ_ _. False ⦄"

definition "no_return P A ≡ ⦃ P ⦄ A ⦃λ_ _. False⦄,⦃λ_ _. True ⦄"

end
```

3 Enumerations

```
theory Enumeration
imports "~~/src/HOL/Main"
begin

abbreviation
  "enum ≡ enum_class.enum"
abbreviation
  "enum_all ≡ enum_class.enum_all"
abbreviation
  "enum_ex ≡ enum_class.enum_ex"

primrec
  the_index :: "'a list ⇒ 'a ⇒ nat"
where
  "the_index (x # xs) y = (if x = y then 0 else Suc (the_index xs y))"

lemma the_index_bounded:
  "x ∈ set xs ⟹ the_index xs x < length xs"

lemma nth_the_index:
  "x ∈ set xs ⟹ xs ! the_index xs x = x"

lemma distinct_the_index_is_index[simp]:
  "⟦ distinct xs ; n < length xs ⟧ ⟹ the_index xs (xs ! n) = n"

lemma the_index_last_distinct:
  "distinct xs ∧ xs ≠ [] ⟹ the_index xs (last xs) = length xs - 1"

context enum begin


  lemmas enum_surj[simp] = enum_UNIV
  declare enum_distinct[simp]

lemma enum_nonempty[simp]: "(enum :: 'a list) ≠ []"

definition
  maxBound :: 'a where
  "maxBound ≡ last enum"

definition
  minBound :: 'a where
  "minBound ≡ hd enum"
definition
  toEnum :: "nat ⇒ 'a" where
  "toEnum n ≡ if n < length (enum :: 'a list) then enum ! n else the None"

definition
  fromEnum :: "'a ⇒ nat" where
  "fromEnum x ≡ the_index enum x"
```

```
lemma maxBound_is_length:
  "fromEnum maxBound = length (enum :: 'a list) - 1"

lemma maxBound_less_length:
  "(x ≤ fromEnum maxBound) = (x < length (enum :: 'a list))"

lemma maxBound_is_bound [simp]:
 "fromEnum x ≤ fromEnum maxBound"

lemma to_from_enum [simp]:
  fixes x :: 'a
  shows "toEnum (fromEnum x) = x"

lemma from_to_enum [simp]:
  "x ≤ fromEnum maxBound ⟹ fromEnum (toEnum x) = x"

lemma map_enum:
  fixes x :: 'a
  shows "map f enum ! fromEnum x = f x"

definition
  assocs :: "('a ⇒ 'b) ⇒ ('a × 'b) list" where
 "assocs f ≡ map (λx. (x, f x)) enum"

end


lemmas enum_bool = enum_bool_def

lemma fromEnumTrue [simp]: "fromEnum True = 1"

lemma fromEnumFalse [simp]: "fromEnum False = 0"


class enum_alt =
  fixes enum_alt :: "nat ⇒ 'a option"

class enumeration_alt = enum_alt +
  assumes enum_alt_one_bound:
    "enum_alt x = (None :: 'a option) ⟹ enum_alt (Suc x) = (None :: 'a option)"
  assumes enum_alt_surj: "range enum_alt ∪ {None} = UNIV"
  assumes enum_alt_inj:
    "(enum_alt x :: 'a option) = enum_alt y ⟹ (x = y) ∨ (enum_alt x = (None :: 'a option))"
begin

lemma enum_alt_inj_2:
  "⟦ enum_alt x = (enum_alt y :: 'a option);
     enum_alt x ≠ (None :: 'a option) ⟧
    ⟹ x = y"

lemma enum_alt_surj_2:
  "∃x. enum_alt x = Some y"

end

definition
  alt_from_ord :: "'a list ⇒ nat ⇒ 'a option" where
```

```
 "alt_from_ord L ≡ λn. if (n < length L) then Some (L ! n) else None"

lemma handy_enum_lemma1: "((if P then Some A else None) = None) = (¬ P)"

lemma handy_enum_lemma2: "Some x ∉ empty ‘ S"

lemma handy_enum_lemma3: "((if P then Some A else None) = Some B) = (P ∧ (A = B))"

class enumeration_both = enum_alt + enum +
  assumes enum_alt_rel: "enum_alt = alt_from_ord enum"

instance enumeration_both < enumeration_alt

instantiation bool :: enumeration_both
begin

definition
  enum_alt_bool: "enum_alt ≡ alt_from_ord [False, True]"

instance
end

definition
  toEnumAlt :: "nat ⇒ ('a :: enum_alt)" where
 "toEnumAlt n ≡ the (enum_alt n)"

definition
  fromEnumAlt :: "('a :: enum_alt) ⇒ nat" where
 "fromEnumAlt x ≡ THE n. enum_alt n = Some x"

definition
  upto_enum :: "('a :: enumeration_alt) ⇒ 'a ⇒ 'a list" ("(1[_.e._])") where
 "upto_enum n m ≡ map toEnumAlt [fromEnumAlt n ..< Suc (fromEnumAlt m)]"

lemma fromEnum_alt_red[simp]:
  "fromEnumAlt = (fromEnum :: ('a :: enumeration_both) ⇒ nat)"

lemma toEnum_alt_red[simp]:
  "toEnumAlt = (toEnum :: nat ⇒ ('a :: enumeration_both))"

lemma upto_enum_red:
  "[(n :: ('a :: enumeration_both)) .e. m] = map toEnum [fromEnum n ..< Suc (fromEnum m)]"

instantiation nat :: enumeration_alt
begin

definition
  enum_alt_nat: "enum_alt ≡ Some"

instance

end

lemma toEnumAlt_nat[simp]: "toEnumAlt = id"

lemma fromEnumAlt_nat[simp]: "fromEnumAlt = id"

lemma upto_enum_nat[simp]: "[n .e. m] = [n ..< Suc m]"
```

```
definition
  zipE1 :: "('a :: enum_alt) ⇒ 'b list ⇒ ('a × 'b) list" where
 "zipE1 x L ≡ zip (map toEnumAlt [(fromEnumAlt x) ..< (fromEnumAlt x) + length L]) L"

definition
  zipE2 :: "('a :: enum_alt) ⇒ 'a ⇒ 'b list ⇒ ('a × 'b) list" where
 "zipE2 x xn L ≡ zip (map (λn. toEnumAlt ((fromEnumAlt x) + ((fromEnumAlt xn) - (fromEnumAlt x))
* n)) [0 ..< length L]) L"

definition
  zipE3 :: "'a list ⇒ ('b :: enum_alt) ⇒ ('a × 'b) list" where
 "zipE3 L x ≡ zip L (map toEnumAlt [(fromEnumAlt x) ..< (fromEnumAlt x) + length L])"

definition
  zipE4 :: "'a list ⇒ ('b :: enum_alt) ⇒ 'b ⇒ ('a × 'b) list" where
 "zipE4 L x xn ≡ zip L (map (λn. toEnumAlt ((fromEnumAlt x) + ((fromEnumAlt xn) - (fromEnumAlt
x)) * n)) [0 ..< length L])"

lemma handy_lemma: "a = Some b ⟹ the a = b"

lemma to_from_enum_alt[simp]:
 "toEnumAlt (fromEnumAlt x) = (x :: ('a :: enumeration_alt))"

end
```

```
theory WordLib
imports NICTACompat SignedWords
begin
```

```
lemma shiftl_power:
  "(shiftl1 ^^ x) (y::'a::len word) = 2 ^ x * y"

lemma to_bl_use_of_bl:
   "(to_bl w = bl) = (w = of_bl bl ∧ length bl = length (to_bl w))"

lemmas of_bl_reasoning = to_bl_use_of_bl of_bl_append

lemma uint_of_bl_is_bl_to_bin:
  "length l≤len_of TYPE('a) ⟹
   uint ((of_bl::bool list⇒ ('a :: len) word) l) = bl_to_bin l"

lemma bin_to_bl_or:
  "bin_to_bl n (a OR b) = map2 (op ∨) (bin_to_bl n a) (bin_to_bl n b)"

lemma word_ops_nth [simp]:
  shows
  word_or_nth:  "(x || y) !! n = (x !! n ∨ y !! n)" and
  word_and_nth: "(x && y) !! n = (x !! n ∧ y !! n)" and
  word_xor_nth: "(x xor y) !! n = (x !! n ≠ y !! n)"

lemma word_nth_1 [iff]:
```

```
  "(1::'a::len word) !! n = (n = 0)"


lemma "1 < (1024::32 word) ∧ 1 ≤ (1024::32 word)"

lemma and_not_mask: "w AND NOT mask n = (w >> n) << n"

lemma and_mask: "w AND mask n = (w << (size w - n)) >> (size w - n)"

lemma AND_twice:
  "(w && m) && m = w && m"

lemma nth_w2p_same: "(2^n :: 'a :: len word) !! n = (n < len_of TYPE('a::len))"

lemma p2_gt_0: "(0 < (2 ^ n :: 'a :: len word)) = (n < len_of TYPE('a))"

lemmas uint_2p_alt = uint_2p [unfolded p2_gt_0]

lemma shiftr_div_2n_w: "n < size w ⟹ w >> n = w div (2^n :: 'a :: len word)"

lemmas less_def = less_eq [symmetric]
lemmas le_def = not_less [symmetric, where ?'a = nat]

lemmas p2_eq_0 = trans [OF eq_commute
  iffD2 [OF Not_eq_iff p2_gt_0, folded le_def, unfolded word_gt_0 not_not]]

lemma neg_mask_is_div':
  "n < size w ⟹ w AND NOT mask n = ((w div (2 ^ n)) * (2 ^ n))"

lemma neg_mask_is_div: "w AND NOT mask n = ((w div (2 ^ n)) * (2 ^ n))"

lemma and_mask_arith':
  "0 < n ⟹ w AND mask n = ((w * (2 ^ (size w - n))) div (2 ^ (size w - n)))"

lemma mask_0 [simp]: "mask 0 = 0"

lemmas p2len = iffD2 [OF p2_eq_0 order_refl]

lemma and_mask_arith:
  "w AND mask n = ((w * (2 ^ (size w - n))) div (2 ^ (size w - n)))"

lemma mask_2pm1: "mask n = 2 ^ n - 1"

lemma is_aligned_AND_less_0:
  "u && mask n = 0 ⟹ v < 2^n ⟹ u && v = 0"

lemma len_0_eq: "len_of (TYPE('a :: len0)) = 0 ==> (x :: 'a :: len0 word) = y"

lemma le_shiftr1: "u <= v ==> shiftr1 u <= shiftr1 v"

lemma le_shiftr: "u ≤ v ⟹ u >> (n :: nat) ≤ (v :: 'a :: len0 word) >> n"

lemma shiftr_mask_le: "n <= m ==> mask n >> m = 0"

lemmas shiftr_mask = order_refl [THEN shiftr_mask_le, simp]

lemma word_leI: "(!!n::nat.  n < size (u::'a::len0 word)
   ==> u !! n ==> (v::'a::len0 word) !! n) ==> u <= v"
```

lemma le_mask_iff: "(w ≤ mask n) = (w >> n = 0)"

lemma and_mask_eq_iff_shiftr_0: "(w AND mask n = w) = (w >> n = 0)"

lemmas and_mask_eq_iff_le_mask = trans
  [OF and_mask_eq_iff_shiftr_0 le_mask_iff [THEN sym]]

lemma one_bit_shiftl: "set_bit 0 n True = (1 :: 'a :: len word) << n"

lemmas one_bit_pow = trans [OF one_bit_shiftl shiftl_1]

lemma bin_sc_minus: "0 < n ==> bin_sc (Suc (n - 1)) b i = bin_sc n b i"

lemmas bin_sc_minus_simps =
  bin_sc_simps (2,3,4) [THEN [2] trans, OF bin_sc_minus [THEN sym]]

lemma NOT_eq: "NOT (x :: 'a :: len word) = - x - 1"

lemma NOT_mask: "NOT (mask n :: 'a :: len word) = - (2 ^ n)"

lemma power_2_pos_iff:
  "(2 ^ n > (0 :: 'a :: len word)) = (n < len_of TYPE ('a))"

lemma le_m1_iff_lt: "(x > (0 :: 'a :: len word)) = ((y ≤ x - 1) = (y < x))"

lemmas gt0_iff_gem1 =
  iffD1 [OF iffD1 [OF eq_left_commute le_m1_iff_lt] order_refl]

lemmas power_2_ge_iff = trans [OF gt0_iff_gem1 [THEN sym] power_2_pos_iff]

lemma le_mask_iff_lt_2n:
  "n < len_of TYPE ('a) = (((w :: 'a :: len word) ≤ mask n) = (w < 2 ^ n))"

lemmas mask_lt_2pn =
  le_mask_iff_lt_2n [THEN iffD1, THEN iffD1, OF _ order_refl]

lemma mask_eq_iff_w2p_alt:
  "n < size (w::'a::len word) ==> (w AND mask n = w) = (w < 2 ^ n)"

lemma and_mask_less_size_alt: "n < size x ==> x AND mask n < 2 ^ n"

lemma bang_eq:
  fixes x :: "'a::len0 word"
  shows "(x = y) = (∀n. x !! n = y !! n)"

declare of_nat_power [simp]
declare of_nat_mult [simp]

lemma word_unat_power:
  "(2 :: ('a :: len) word) ^ n = of_nat (2 ^ n)"

lemma of_nat_mono_maybe:
  fixes Y :: "nat"
  assumes xlt: "X < 2 ^ len_of TYPE ('a :: len)"

```
  shows   "(Y < X) ⟹ of_nat Y < (of_nat X :: 'a :: len word)"

lemma shiftl_over_and_dist:
  fixes a::"'a::len word"
  shows "(a AND b) << c = (a << c) AND (b << c)"

lemma shiftr_over_and_dist:
  fixes a::"'a::len word"
  shows "a AND b >> c = (a >> c) AND (b >> c)"

lemma sshiftr_over_and_dist:
  fixes a::"'a::len word"
  shows "a AND b >>> c = (a >>> c) AND (b >>> c)"

lemma shiftl_over_or_dist:
  fixes a::"'a::len word"
  shows "a OR b << c = (a << c) OR (b << c)"

lemma shiftr_over_or_dist:
  fixes a::"'a::len word"
  shows "a OR b >> c = (a >> c) OR (b >> c)"

lemma sshiftr_over_or_dist:
  fixes a::"'a::len word"
  shows "a OR b >>> c = (a >>> c) OR (b >>> c)"

lemmas shift_over_ao_dists =
  shiftl_over_or_dist shiftr_over_or_dist
  sshiftr_over_or_dist shiftl_over_and_dist
  shiftr_over_and_dist sshiftr_over_and_dist

lemma shiftl_shiftl:
  fixes a::"'a::len word"
  shows "a << b << c = a << (b + c)"

lemma shiftr_shiftr:
  fixes a::"'a::len word"
  shows "a >> b >> c = a >> (b + c)"

lemma shiftl_shiftr1:
  fixes a::"'a::len word"
  shows "c ≤ b ⟹
    a << b >> c =
    a AND (mask (size a - b)) << (b - c)"

lemma shiftl_shiftr2:
  fixes a::"'a::len word"
  shows "b < c ⟹
    a << b >> c =
    (a >> (c - b)) AND (mask (size a - c))"

lemma shiftr_shiftl1:
  fixes a::"'a::len word"
  shows "c ≤ b ⟹
         a >> b << c = (a >> (b - c)) AND
                       (NOT mask c)"

lemma shiftr_shiftl2:
```

```
  fixes a::"'a::len word"
  shows "b < c ⟹
         a >> b << c = (a << (c - b)) AND
                       (NOT mask c)"

lemmas multi_shift_simps =
  shiftl_shiftl shiftr_shiftr
  shiftl_shiftr1 shiftl_shiftr2
  shiftr_shiftl1 shiftr_shiftl2

lemma word_and_max_word:
  fixes a::"'a::len word"
  shows "x = max_word ⟹ a AND x = a"

lemma word_and_1:
  fixes x::"'a::len word"
  shows "(x AND 1) = (if x!!0 then 1 else 0)"

lemma word_and_1_bl:
  fixes x::"'a::len word"
  shows "(x AND 1) = of_bl [x !! 0]"

lemma word_1_and_bl:
  fixes x::"'a::len word"
  shows "(1 AND x) = of_bl [x !! 0]"




notation (input)
  test_bit ("testBit")

definition
  w2byte :: "'a :: len word ⇒ 8 word" where
  "w2byte ≡ ucast"

lemma scast_scast_id':
  fixes x :: "('a::len) word"
  assumes is_up: "is_up (scast :: 'a word ⇒ ('b::len) word)"
  shows "scast (scast x :: 'b word) = (x :: 'a word)"

lemma scast_scast_id [simp]:
  "scast (scast x :: ('a::len) signed word) = (x :: 'a word)"
  "scast (scast y :: ('a::len) word) = (y :: 'a signed word)"

lemma scast_ucast_id [simp]:
    "scast (ucast (x :: 'a::len word) :: 'a signed word) = x"

lemma ucast_scast_id [simp]:
    "ucast (scast (x :: 'a::len signed word) :: 'a word) = x"

lemma scast_of_nat [simp]:
    "scast (of_nat x :: 'a::len signed word) = (of_nat x :: 'a word)"

lemma ucast_of_nat:
  "is_down (ucast :: ('a :: len) word ⇒ ('b :: len) word)
    ⟹ ucast (of_nat n :: 'a word) = (of_nat n :: 'b word)"
```

```
lemma word32_sint_1[simp]:
 "sint (1::word32) = 1"

lemma sint_1 [simp]:
  "sint (1::'a::len word) = (if len_of TYPE('a) = 1 then -1 else 1)"

lemma scast_1':
  "(scast (1::'a::len word) :: 'b::len word) =
   (word_of_int (sbintrunc (len_of TYPE('a::len) - Suc 0) (1::int)))"

lemma scast_1 [simp]:
  "(scast (1::'a::len word) :: 'b::len word) =
      (if len_of TYPE('a) = 1 then -1 else 1)"

lemma scast_eq_scast_id [simp]:
    "((scast (a :: 'a::len signed word) :: 'a word) = scast b) = (a = b)"

lemma ucast_eq_ucast_id [simp]:
    "((ucast (a :: 'a::len word) :: 'a signed word) = ucast b) = (a = b)"

lemma scast_ucast_norm [simp]:
  "(ucast (a :: 'a::len word) = (b :: 'a signed word)) = (a = scast b)"
  "((b :: 'a signed word) = ucast (a :: 'a::len word)) = (a = scast b)"

lemma of_bl_drop:
  "of_bl (drop n xs) = (of_bl xs && mask (length xs - n))"

lemma of_int_uint [simp]:
    "of_int (uint x) = x"

lemma shiftr_mask2:
  "n ≤ len_of TYPE('a) ⟹ (mask n >> m :: ('a :: len) word) = mask (n - m)"


corollary word_plus_and_or_coroll:
  "x && y = 0 ⟹ x + y = x || y"


corollary word_plus_and_or_coroll2:
  "(x && w) + (x && ~~ w) = x"


lemma less_le_mult_nat':
  "w * c < b * c ==> 0 ≤ c ==> Suc w * c ≤ b * (c::nat)"

lemmas less_le_mult_nat = less_le_mult_nat'[simplified distrib_right, simplified]


class signed_div =
  fixes sdiv :: "'a ⇒ 'a ⇒ 'a" (infixl "sdiv" 70)
  fixes smod :: "'a ⇒ 'a ⇒ 'a" (infixl "smod" 70)

instantiation int :: signed_div
begin
definition "(a :: int) sdiv b ≡ sgn (a * b) * (abs a div abs b)"
definition "(a :: int) smod b ≡ a - (a sdiv b) * b"
instance
end
```

```
instantiation word :: (len) signed_div
begin
definition "(a :: ('a::len) word) sdiv b = word_of_int (sint a sdiv sint b)"
definition "(a :: ('a::len) word) smod b = word_of_int (sint a smod sint b)"
instance
end


lemma
  "( 4 :: word32) sdiv  4 =  1"
  "(-4 :: word32) sdiv  4 = -1"
  "(-3 :: word32) sdiv  4 =  0"
  "( 3 :: word32) sdiv -4 =  0"
  "(-3 :: word32) sdiv -4 =  0"
  "(-5 :: word32) sdiv -4 =  1"
  "( 5 :: word32) sdiv -4 = -1"

lemma
  "( 4 :: word32) smod  4 =   0"
  "( 3 :: word32) smod  4 =   3"
  "(-3 :: word32) smod  4 =  -3"
  "( 3 :: word32) smod -4 =   3"
  "(-3 :: word32) smod -4 =  -3"
  "(-5 :: word32) smod -4 =  -1"
  "( 5 :: word32) smod -4 =   1"

end
```

4 Enumeration instances for Words

```
theory WordEnum
imports Enumeration WordLib
begin

instantiation word :: (len) enum
begin

definition
  "(enum_class.enum :: ('a :: len) word list) ≡ map of_nat [0 ..< 2 ^ len_of TYPE('a)]"

definition
  "enum_class.enum_all (P :: ('a :: len) word ⇒ bool) ⟷ Ball UNIV P"

definition
  "enum_class.enum_ex (P :: ('a :: len) word ⇒ bool) ⟷ Bex UNIV P"

instance

end

lemma fromEnum_unat[simp]: "fromEnum (x :: ('a :: len) word) = unat x"

lemma length_word_enum: "length (enum :: ('a :: len) word list) = 2 ^ len_of TYPE('a)"

lemma toEnum_of_nat[simp]: "n < 2 ^ len_of TYPE('a) ⟹ ((toEnum n) :: ('a :: len) word) = of_nat
n"

declare of_nat_diff [simp]
declare word_pow_0 [simp]

lemma "(maxBound :: ('a :: len) word) = -1"

lemma "(minBound :: ('a :: len) word) = 0"

instantiation word :: (len) enumeration_both
begin

definition
  enum_alt_word_def: "enum_alt ≡ alt_from_ord (enum :: ('a :: len) word list)"

instance

end


definition
  upto_enum_step :: "word32 ⇒ word32 ⇒ word32 ⇒ word32 list" ("[_ , _ .e. _]")
where
  "upto_enum_step a b c ≡
     if c < a then [] else map (λx. a + x * (b - a)) [0 .e. (c - a) div (b - a)]"
```

end

5 Machine Word Setup

```
theory WordSetup
imports WordEnum DistinctProp

begin
```

This theory defines the standard platform-specific word size and alignment.

```
definition
  word_bits :: nat where
  "word_bits ≡ len_of TYPE(32)"

definition
  word_size :: "'a :: numeral" where
  "word_size ≡ 4"

lemma word_bits_conv:
  "word_bits = 32"

lemma word_bits_word_size_conv:
  "word_bits = word_size * 8"

definition
  is_aligned :: "'a :: len word ⇒ nat ⇒ bool" where
  "is_aligned ptr n ≡ 2^n dvd unat ptr"

definition
  ptr_add :: "word32 ⇒ nat ⇒ word32" where
  "ptr_add ptr n ≡ ptr + of_nat n"

definition
  complement :: "('a :: len) word ⇒ 'a word"  where
 "complement x ≡ x xor -1"

definition
  alignUp :: "'a::len word ⇒ nat ⇒ 'a word" where
 "alignUp x n ≡ x + 2 ^ n - 1 && complement (2 ^ n - 1)"

end
```

6 Platform Definitions

```
theory Platform
imports
  "../../lib/Lib"
  "../../lib/WordEnum"
begin
```

This theory lists platform-specific types and basic constants, in particular the types of interrupts and physical addresses, constants for the kernel location, the offsets between physical and virtual kernel addresses, as well as the range of IRQs on the platform.

```
type_synonym irq = word8
type_synonym paddr = word32

abbreviation "toPAddr ≡ id"
abbreviation "fromPAddr ≡ id"

definition
  pageColourBits :: nat where
  "pageColourBits ≡ 2"

definition
  cacheLineBits :: nat where
  "cacheLineBits = 5"

definition
  cacheLine :: nat where
  "cacheLine = 2^cacheLineBits"

definition
  kernelBase_addr :: word32 where
  "kernelBase_addr ≡ 0xf0000000"

definition
  physBase :: word32 where
  "physBase ≡ 0x80000000"

definition
  physMappingOffset :: word32 where
  "physMappingOffset ≡ kernelBase_addr - physBase"

definition
  ptrFromPAddr :: "paddr ⇒ word32" where
  "ptrFromPAddr paddr ≡ paddr + physMappingOffset"

definition
  addrFromPPtr :: "word32 ⇒ paddr" where
  "addrFromPPtr pptr ≡ pptr - physMappingOffset"

definition
  minIRQ :: "irq" where
  "minIRQ ≡ 0"
```

definition

```
maxIRQ :: "irq" where
"maxIRQ ≡ 63"
```

end

7 ARM Machine Types

```
theory ARMMachineTypes
imports
  "../../lib/Enumeration"
  "../../lib/WordSetup"
  "../../lib/wp/NonDetMonad"
  "../machine/Platform"
begin
```

An implementation of the machine's types, defining register set and some observable machine state.

7.1 Types

```
datatype register =
    R0
  | R1
  | R2
  | R3
  | R4
  | R5
  | R6
  | R7
  | R8
  | R9
  | SL
  | FP
  | IP
  | SP
  | LR
  | LR_svc
  | FaultInstruction
  | CPSR

type_synonym machine_word = "word32"

consts
initContext :: "(register * machine_word) list"

consts
sanitiseRegister :: "register ⇒ machine_word ⇒ machine_word"
definition
"capRegister ≡ R0"

definition
"msgInfoRegister ≡ R1"

definition
"msgRegisters ≡ [R2 .e. R5]"

definition
"badgeRegister ≡ R0"
```

definition
```
"frameRegisters ≡ FaultInstruction # SP # CPSR # [R0, R1] @ [R8  .e.  IP]"
```

definition
```
"gpRegisters ≡ [R2, R3, R4, R5, R6, R7, LR]"
```

definition
```
"exceptionMessage ≡ [FaultInstruction, SP, CPSR]"
```

definition
```
"syscallMessage ≡ [R0  .e.  R7] @ [FaultInstruction, SP, LR, CPSR]"
```

defs `initContext_def:`
```
"initContext≡ [(CPSR,0x150)]"
```

defs `sanitiseRegister_def:`
```
"sanitiseRegister x0 v≡ (case x0 of
    CPSR ⇒    (v && 0xf8000000) || 0x150
  | _ ⇒    v
  )"
```

7.2 Machine State

Most of the machine state is left underspecified at this level. We know it exists, we will declare some interface functions, but at this level we do not have access to how this state is transformed or what effect it has on the machine.

typedecl `machine_state_rest`

The exclusive monitors state is observable in user mode. The type for this is the type used in the Cambridge HOL4 ARM model.

type_synonym `exclusive_monitors = "(word32 ⇒ bool) list × (word32 × nat ⇒ bool)"`

The full machine state is the state observable by the kernel plus the underspecified rest above. The observable parts are the interrupt controller (which IRQs are masked) and the memory of the machine. The latter is shadow state: kernel memory is kept in a separate, more abstract datatype; user memory is reflected down to the underlying memory of the machine.

record
```
  machine_state =
  irq_masks :: "irq ⇒ bool"
  irq_state :: nat
  underlying_memory :: "word32 ⇒ word8"
  exclusive_state :: exclusive_monitors
  machine_state_rest :: machine_state_rest
```

consts `irq_oracle :: "nat ⇒ word8"`

The machine monad is used for operations on the state defined above.

type_synonym `'a machine_monad = "(machine_state, 'a) nondet_monad"`

translations
```
  (type) "'c machine_monad" <= (type) "(machine_state, 'c) nondet_monad"
```

After kernel initialisation all IRQs are masked.

definition

```
  "init_irq_masks ≡ λ_. True"
```

The initial contents of the user-visible memory is 0.

```
definition
  init_underlying_memory :: "word32 ⇒ word8"
  where
  "init_underlying_memory ≡ λ_. 0"
```

The initial exclusive state is the same constant that clearExMonitor defaults it to.

```
consts default_exclusive_state :: exclusive_monitors
```

We leave open the underspecified rest of the machine state in the initial state.

```
definition
  init_machine_state :: machine_state where
 "init_machine_state ≡ (| irq_masks = init_irq_masks,
                         irq_state = 0,
                         underlying_memory = init_underlying_memory,
                         exclusive_state = default_exclusive_state,
                         machine_state_rest = undefined |)"



type_synonym hardware_asid = "word8"

definition
  HardwareASID :: "hardware_asid ⇒ hardware_asid"
where HardwareASID_def[simp]:
 "HardwareASID ≡ id"

definition
  fromHWASID :: "hardware_asid ⇒ hardware_asid"
where
  fromHWASID_def[simp]:
 "fromHWASID ≡ id"

definition  fromHWASID_update :: "(hardware_asid ⇒ hardware_asid) ⇒ hardware_asid ⇒ hardware_asid"
where
  fromHWASID_update_def[simp]:
 "fromHWASID_update f y ≡ f y"

abbreviation (input)
  HardwareASID_trans :: "(word8) ⇒ hardware_asid" ("HardwareASID'_ (| fromHWASID= _ |)")
where
  "HardwareASID_ (| fromHWASID= v0 |) == HardwareASID v0"

datatype vmpage_size =
    ARMSmallPage
  | ARMLargePage
  | ARMSection
  | ARMSuperSection

datatype vmfault_type =
    ARMDataAbort
  | ARMPrefetchAbort

definition
pageBits :: "nat"
```

where
"pageBits ≡ 12"

definition
pageBitsForSize :: "vmpage_size ⇒ nat"
where
```
"pageBitsForSize x0≡ (case x0 of
    ARMSmallPage ⇒    12
  | ARMLargePage ⇒    16
  | ARMSection ⇒    20
  | ARMSuperSection ⇒    24
  )"
```

end

8 Machine Types

theory MachineTypes
imports ARMMachineTypes
begin

We select ARM based machine types by importing them above.

end

9 Kernel Events

```
theory Event_H
imports "../machine/MachineTypes"
begin
```

These are the user-level and machine generated events the kernel reacts to.

```
datatype syscall =
    SysSend
  | SysNBSend
  | SysCall
  | SysWait
  | SysReply
  | SysReplyWait
  | SysYield

datatype event =
    SyscallEvent syscall
  | UnknownSyscall nat
  | UserLevelFault machine_word machine_word
  | Interrupt
  | VMFaultEvent vmfault_type


end
```

10 Common, Architecture-Specific Data Types

```
theory ARM_Structs_B
imports "~~/src/HOL/Main"
begin

datatype arm_vspace_region_use =
    ArmVSpaceUserRegion
  | ArmVSpaceInvalidRegion
  | ArmVSpaceKernelWindow
  | ArmVSpaceDeviceWindow


end
```

11 ARM Machine Instantiation

```
theory ARM_Machine_A
imports
  "../../lib/WordSetup"
  "../../lib/wp/NonDetMonad"
  "../machine/ARMMachineTypes"
begin
```

The specification is written with abstract type names for object references, user pointers, word-based data, cap references, and so on. This theory provides an instantiation of these names to concrete types for the ARM architecture. Other architectures may have slightly different instantations.

```
type_synonym obj_ref          = machine_word
type_synonym vspace_ref       = machine_word
type_synonym data_offset      = "12 word"

type_synonym data             = machine_word
type_synonym cap_ref          = "bool list"
type_synonym length_type      = machine_word
```

With the definitions above, most conversions between abstract type names boil down to just the identity function, some convert from `word` to `nat` and others between different word sizes using `ucast`.

```
definition
  oref_to_data   :: "obj_ref ⇒ data" where
  "oref_to_data ≡ id"

definition
  data_to_oref   :: "data ⇒ obj_ref" where
  "data_to_oref ≡ id"

definition
  vref_to_data   :: "vspace_ref ⇒ data" where
  "vref_to_data ≡ id"

definition
  data_to_vref   :: "data ⇒ vspace_ref" where
  "data_to_vref ≡ id"

definition
  nat_to_len     :: "nat ⇒ length_type" where
  "nat_to_len ≡ of_nat"

definition
  data_to_nat    :: "data ⇒ nat" where
  "data_to_nat ≡ unat"

definition
  data_to_16     :: "data ⇒ 16 word" where
  "data_to_16 ≡ ucast"

definition
  data_to_cptr :: "data ⇒ cap_ref" where
```

```
  "data_to_cptr ≡ to_bl"
```

definition
```
  data_offset_to_nat :: "data_offset ⇒ nat" where
  "data_offset_to_nat ≡ unat"
```

definition
```
  combine_aep_badges :: "data ⇒ data ⇒ data" where
  "combine_aep_badges ≡ bitOR"
```

definition
```
  combine_aep_msgs :: "data ⇒ data ⇒ data" where
  "combine_aep_msgs ≡ bitOR"
```

These definitions will be unfolded automatically in proofs.

lemmas `data_convs [simp] =`
```
  oref_to_data_def data_to_oref_def vref_to_data_def data_to_vref_def
  nat_to_len_def data_to_nat_def data_to_16_def data_to_cptr_def
  data_offset_to_nat_def
```

The following definitions provide architecture-dependent sizes such as the standard page size and capability size of the underlying machine.

definition
```
  slot_bits :: nat where
  "slot_bits ≡ 4"
```

end

12 Machine Accessor Functions

```
theory MiscMachine_A
imports ARM_Machine_A
begin
```

Miscellaneous definitions of constants used in modelling machine operations.

```
definition
  nat_to_cref :: "nat ⇒ nat ⇒ cap_ref" where
  "nat_to_cref ln n ≡ drop (word_bits - ln)
                           (to_bl (of_nat n :: machine_word))"

type_synonym user_context = "register ⇒ data"
type_synonym 'a user_monad = "(user_context, 'a) nondet_monad"

definition
 "msg_info_register ≡ msgInfoRegister"
definition
 "msg_registers ≡ msgRegisters"
definition
 "cap_register ≡ capRegister"
definition
 "badge_register ≡ badgeRegister"
definition
 "frame_registers ≡ frameRegisters"
definition
 "gp_registers ≡ gpRegisters"
definition
 "exception_message ≡ exceptionMessage"
definition
 "syscall_message ≡ syscallMessage"

definition
  new_context :: "user_context" where
  "new_context ≡ (λr. 0) (CPSR := 0x150)"

definition
  get_register :: "register ⇒ data user_monad" where
  "get_register r ≡ gets (λuc. uc r)"

definition
  set_registers :: "(register ⇒ data) ⇒ unit user_monad" where
  "set_registers ≡ put"

definition
  set_register :: "register ⇒ data ⇒ unit user_monad" where
  "set_register r v ≡ modify (λuc. uc (r := v))"

end
```

13 Error and Fault Messages

```
theory ExceptionTypes_A
imports MiscMachine_A
begin
```

There are two types of exceptions that can occur in the kernel: faults and errors. Faults are reported to the user's fault handler. Errors are reported to the user directly.

Capability lookup failures can be be either fault or error, depending on context.

```
datatype lookup_failure
     = InvalidRoot
     | MissingCapability nat
     | DepthMismatch nat nat
     | GuardMismatch nat "bool list"

datatype fault
         = CapFault word32 bool lookup_failure
         | VMFault data "data list"
         | UnknownSyscallException data
         | UserException data data

datatype syscall_error
         = InvalidArgument nat
         | InvalidCapability nat
         | IllegalOperation
         | RangeError data data
         | AlignmentError
         | FailedLookup bool lookup_failure
         | TruncatedMessage
         | DeleteFirst
         | RevokeFirst
         | NotEnoughMemory data
```

Preemption in the system is caused by the arrival of hardware interrupts which are tagged with their hardware IRQ.

```
datatype interrupt = Interrupted irq
```

Create a message from a system-call failure to be returned to the thread attempting the operation that failed.

```
primrec
  msg_from_lookup_failure :: "lookup_failure ⇒ data list"
where
  "msg_from_lookup_failure InvalidRoot           = [1]"
| "msg_from_lookup_failure (MissingCapability n) = [2, of_nat n]"
| "msg_from_lookup_failure (DepthMismatch n m)   = [3, of_nat n, of_nat m]"
| "msg_from_lookup_failure (GuardMismatch n g)   = [4, of_nat n, of_bl g, of_nat (size g)]"

primrec
  msg_from_syscall_error :: "syscall_error ⇒ (data × data list)"
where
  "msg_from_syscall_error (InvalidArgument n)    = (1, [of_nat n])"
```

```
| "msg_from_syscall_error (InvalidCapability n)  = (2, [of_nat n])"
| "msg_from_syscall_error IllegalOperation       = (3, [])"
| "msg_from_syscall_error (RangeError minv maxv) = (4, [minv, maxv])"
| "msg_from_syscall_error AlignmentError         = (5, [])"
| "msg_from_syscall_error (FailedLookup s lf)    = (6, [if s then 1 else 0]@(msg_from_lookup_failure
lf))"
| "msg_from_syscall_error TruncatedMessage       = (7, [])"
| "msg_from_syscall_error DeleteFirst            = (8, [])"
| "msg_from_syscall_error RevokeFirst            = (9, [])"
| "msg_from_syscall_error (NotEnoughMemory n)    = (10, [n])"
```

end

14 Access Rights

theory CapRights_A
imports "~~/src/HOL/Main"
begin

The possible access-control rights that exist in the system. Note that some rights are synonyms for others.

datatype rights = AllowRead | AllowWrite | AllowGrant

definition
  "AllowSend ≡ AllowWrite"
definition
  "AllowRecv ≡ AllowRead"
definition
  "CanModify ≡ AllowWrite"

Cap rights are just a set of access rights

type_synonym cap_rights = "rights set"

The set of all rights:

definition
  all_rights :: cap_rights
where
 "all_rights ≡ UNIV"

end

15 ARM-Specific Virtual-Memory Rights

theory ARM_VMRights_A
imports CapRights_A
begin

This theory provides architecture-specific definitions and datatypes for virtual-memory support.

15.1 Architecture-specific virtual memory

Page access rights.

```
type_synonym vm_rights = cap_rights

definition
  vm_kernel_only :: vm_rights where
  "vm_kernel_only ≡ {}"
definition
  vm_read_only :: vm_rights where
  "vm_read_only ≡ {AllowRead}"
definition
  vm_read_write :: vm_rights where
  "vm_read_write ≡ {AllowRead,AllowWrite}"
```

Note that only the above combinations of virtual-memory rights are permitted. We introduce the following definitions to reflect this fact: The predicate `valid_vm_rights` holds iff a given set of rights is valid (i.e., a permitted combination). The function `validate_vm_rights` takes an arbitrary set of rights and returns the largest permitted subset.

```
definition
  "valid_vm_rights ≡ {vm_read_write, vm_read_only, vm_kernel_only}"
definition
  "validate_vm_rights rs ≡
   (if AllowRead ∈ rs
     then (if AllowWrite ∈ rs then vm_read_write else vm_read_only)
   else vm_kernel_only)"


end
```

16 ARM-Specific Data Types

```
theory ARM_Structs_A
imports
  "../design/ARM_Structs_B"
  ExceptionTypes_A
  ARM_VMRights_A
begin
```

This theory provides architecture-specific definitions and datatypes including architecture-specific capabilities and objects.

16.1 Architecture-specific virtual memory

An ASID is simply a word.

```
type_synonym asid = "word32"

datatype vm_attribute = ParityEnabled | PageCacheable | Global | XNever
type_synonym vm_attributes = "vm_attribute set"
```

16.2 Architecture-specific capabilities

The ARM kernel supports capabilities for ASID pools and an ASID controller capability, along with capabilities for page directories, page tables, and page mappings.

```
datatype arch_cap =
   ASIDPoolCap obj_ref asid
 | ASIDControlCap
 | PageCap obj_ref cap_rights vmpage_size "(asid * vspace_ref) option"
 | PageTableCap obj_ref "(asid * vspace_ref) option"
 | PageDirectoryCap obj_ref "asid option"

definition
  is_page_cap :: "arch_cap ⇒ bool" where
  "is_page_cap c ≡ ∃x0 x1 x2 x3. c = PageCap x0 x1 x2 x3"

definition
  asid_high_bits :: nat where
  "asid_high_bits ≡ 8"
definition
  asid_low_bits :: nat where
  "asid_low_bits ≡ 10 :: nat"
definition
  asid_bits :: nat where
  "asid_bits ≡ 18 :: nat"
```

16.3 Architecture-specific objects

This section gives the types and auxiliary definitions for the architecture-specific objects: a page directory entry (`pde`) contains either an invalid entry, a page table reference, a section reference, or a super-section reference; a page table entry contains either an invalid entry, a large page, or a small page mapping; finally, an architecture-specific object is either an ASID pool, a page table, a page directory, or a data page used to model user memory.

```
datatype pde =
   InvalidPDE
 | PageTablePDE obj_ref vm_attributes machine_word
 | SectionPDE obj_ref vm_attributes machine_word cap_rights
 | SuperSectionPDE obj_ref vm_attributes cap_rights

datatype pte =
   InvalidPTE
 | LargePagePTE obj_ref vm_attributes cap_rights
 | SmallPagePTE obj_ref vm_attributes cap_rights

datatype arch_kernel_obj =
   ASIDPool "10 word ⇀ obj_ref"
 | PageTable "word8 ⇒ pte"
 | PageDirectory "12 word ⇒ pde"
 | DataPage vmpage_size

primrec
  arch_obj_size :: "arch_cap ⇒ nat"
where
  "arch_obj_size (ASIDPoolCap p as) = pageBits"
| "arch_obj_size ASIDControlCap = 0"
| "arch_obj_size (PageCap x rs sz as4) = pageBitsForSize sz"
| "arch_obj_size (PageDirectoryCap x as2) = 14"
| "arch_obj_size (PageTableCap x as3) = 10"

primrec
  arch_kobj_size :: "arch_kernel_obj ⇒ nat"
where
  "arch_kobj_size (ASIDPool p) = pageBits"
| "arch_kobj_size (PageTable pte) = 10"
| "arch_kobj_size (PageDirectory pde) = 14"
| "arch_kobj_size (DataPage sz) = pageBitsForSize sz"

primrec
  aobj_ref :: "arch_cap ⇀ obj_ref"
where
  "aobj_ref (ASIDPoolCap p as) = Some p"
| "aobj_ref ASIDControlCap = None"
| "aobj_ref (PageCap x rs sz as4) = Some x"
| "aobj_ref (PageDirectoryCap x as2) = Some x"
| "aobj_ref (PageTableCap x as3) = Some x"

primrec
  acap_rights :: "arch_cap ⇒ cap_rights"
where
 "acap_rights (PageCap x rs sz as) = rs"

definition
  acap_rights_update :: "cap_rights ⇒ arch_cap ⇒ arch_cap" where
 "acap_rights_update rs ac ≡ case ac of
    PageCap x rs' sz as ⇒ PageCap x (validate_vm_rights rs) sz as
  | _                   ⇒ ac"
```

16.4 Architecture-specific object types and default objects

```
datatype
  aobject_type =
    SmallPageObj
  | LargePageObj
  | SectionObj
  | SuperSectionObj
  | PageTableObj
  | PageDirectoryObj
  | ASIDPoolObj
```

```
definition
  arch_default_cap :: "aobject_type ⇒ obj_ref ⇒ nat ⇒ arch_cap" where
 "arch_default_cap tp r n ≡ case tp of
  SmallPageObj ⇒ PageCap r vm_read_write ARMSmallPage None
  | LargePageObj ⇒ PageCap r vm_read_write ARMLargePage None
  | SectionObj ⇒ PageCap r vm_read_write ARMSection None
  | SuperSectionObj ⇒ PageCap r vm_read_write ARMSuperSection None
  | PageTableObj ⇒ PageTableCap r None
  | PageDirectoryObj ⇒ PageDirectoryCap r None
  | ASIDPoolObj ⇒ ASIDPoolCap r 0"
```

```
definition
  default_arch_object :: "aobject_type ⇒ nat ⇒ arch_kernel_obj" where
 "default_arch_object tp n ≡ case tp of
    SmallPageObj ⇒ DataPage ARMSmallPage
  | LargePageObj ⇒ DataPage ARMLargePage
  | SectionObj ⇒ DataPage ARMSection
  | SuperSectionObj ⇒ DataPage ARMSuperSection
  | PageTableObj ⇒ PageTable (λx. InvalidPTE)
  | PageDirectoryObj ⇒ PageDirectory (λx. InvalidPDE)
  | ASIDPoolObj ⇒ ASIDPool (λ_. None)"
```

```
type_synonym hw_asid = word8
```

```
type_synonym arm_vspace_region_uses = "vspace_ref ⇒ arm_vspace_region_use"
```

16.5 Architecture-specific state

The architecture-specific state for the ARM model consists of a reference to the globals page (`arm_globals_frame`), the first level of the ASID table (`arm_asid_table`), a map from hardware ASIDs to seL4 ASIDs (`arm_hwasid_table`), the next hardware ASID to preempt (`arm_next_asid`), the inverse map from seL4 ASIDs to hardware ASIDs (first component of `arm_asid_map`), and the address of the page directory and page tables mapping the shared address space, along with a description of this space (`arm_global_pd`, `arm_global_pts`, and `arm_kernel_vspace` respectively).

Hardware ASIDs are only ever associated with seL4 ASIDs that have a currently active page directory. The second component of `arm_asid_map` values is the address of that page directory.

```
record arch_state =
  arm_globals_frame :: obj_ref
  arm_asid_table    :: "word8 ⇀ obj_ref"
  arm_hwasid_table  :: "hw_asid ⇀ asid"
  arm_next_asid     :: hw_asid
  arm_asid_map      :: "asid ⇀ (hw_asid × obj_ref)"
  arm_global_pd     :: obj_ref
  arm_global_pts    :: "obj_ref list"
```

```
  arm_kernel_vspace :: arm_vspace_region_uses
```

definition
```
  pd_bits :: "nat" where
  "pd_bits ≡ pageBits + 2"
```

definition
```
  pt_bits :: "nat" where
  "pt_bits ≡ pageBits - 2"
```

end

17 Machine Operations

```
theory MachineOps
imports
  "../../lib/WordSetup"
  "../../lib/wp/NonDetMonad"
  MachineTypes
begin
```

17.1 Wrapping and Lifting Machine Operations

Most of the machine operations below work on the underspecified part of the machine state `machine_state_rest` and cannot fail. We could express the latter by type (leaving out the failure flag), but if we later wanted to implement them, we'd have to set up a new hoare-logic framework for that type. So instead, we provide a wrapper for these operations that explicitly ignores the fail flag and sets it to False. Similarly, these operations never return an empty set of follow-on states, which would require the operation to fail. So we explicitly make this (non-existing) case a null operation.

All this is done only to avoid a large number of axioms (2 for each operation).

```
definition
  ignore_failure :: "('s,unit) nondet_monad ⇒ ('s,unit) nondet_monad"
  where
  "ignore_failure f ≡
  λs. if fst (f s) = {} then ({((),s)},False) else (fst (f s), False)"
```

The wrapper doesn't do anything for usual operations:

```
lemma failure_consistent:
  "⟦ empty_fail f; no_fail ⊤ f ⟧ ⟹ ignore_failure f = f"
```

And it has the desired properties

```
lemma ef_ignore_failure [simp]:
  "empty_fail (ignore_failure f)"

lemma no_fail_ignore_failure [simp, intro!]:
  "no_fail ⊤ (ignore_failure f)"


type_synonym 'a machine_rest_monad = "(machine_state_rest, 'a) nondet_monad"

definition
  machine_rest_lift :: "'a machine_rest_monad ⇒ 'a machine_monad"
where
  "machine_rest_lift f ≡ do
    mr ← gets machine_state_rest;
    (r, mr') ← select_f (f mr);
    modify (λs. s ⦇ machine_state_rest := mr' ⦈);
    return r
  od"

lemma ef_machine_rest_lift [simp, intro!]:
```

```
  "empty_fail f ⟹ empty_fail (machine_rest_lift f)"
```

lemma no_fail_machine_state_rest [intro!]:
```
  "no_fail P f ⟹ no_fail (P o machine_state_rest) (machine_rest_lift f)"
```

lemma no_fail_machine_state_rest_T [simp, intro!]:
```
  "no_fail ⊤ f ⟹ no_fail ⊤ (machine_rest_lift f)"
```

definition
```
  "machine_op_lift ≡ machine_rest_lift o ignore_failure"
```

17.2 The Operations

consts
```
  memory_regions :: "(paddr × paddr) list"
  device_regions :: "(paddr × paddr) list"
```

definition
```
  getMemoryRegions :: "(paddr * paddr) list machine_monad"
```
  where "getMemoryRegions ≡ return memory_regions"

consts
```
  getDeviceRegions_impl :: "unit machine_rest_monad"
  getDeviceRegions_val :: "machine_state ⇒ (paddr * paddr) list"
```

definition
```
  getDeviceRegions :: "(paddr * paddr) list machine_monad"
```
where
```
  "getDeviceRegions ≡ return device_regions"
```

consts
```
  getKernelDevices_impl :: "unit machine_rest_monad"
  getKernelDevices_val :: "machine_state ⇒ (paddr * machine_word) list"
```

definition
```
  getKernelDevices :: "(paddr * machine_word) list machine_monad"
```
where
```
  "getKernelDevices ≡ do
    machine_op_lift getKernelDevices_impl;
    gets getKernelDevices_val
  od"
```

definition
```
  loadWord :: "machine_word ⇒ machine_word machine_monad"
  where "loadWord p ≡ do m ← gets underlying_memory;
                       assert (p && mask 2 = 0);
                       return (word_rcat [m (p + 3), m (p + 2), m (p + 1), m p])
                    od"
```

definition
```
  storeWord :: "machine_word ⇒ machine_word ⇒ unit machine_monad"
  where "storeWord p w ≡ do
                          assert (p && mask 2 = 0);
                          modify (underlying_memory_update (λm.
                                    m(p := word_rsplit w ! 3,
                                      p + 1 := word_rsplit w ! 2,
```

```
                                            p + 2 := word_rsplit w ! 1,
                                            p + 3 := word_rsplit w ! 0)))
                               od"
```

lemma `loadWord_storeWord_is_return:`

```
  "p && mask 2 = 0 ⟹ (do w ← loadWord p; storeWord p w od) = return ()"
```

This instruction is required in the simulator, only.

definition

```
  storeWordVM :: "machine_word ⇒ machine_word ⇒ unit machine_monad"
```

  where `"storeWordVM w p ≡ return ()"`

consts

```
  configureTimer_impl :: "unit machine_rest_monad"
  configureTimer_val :: "machine_state ⇒ irq"
```

definition

```
  configureTimer :: "irq machine_monad"
```

where

```
  "configureTimer ≡ do
    machine_op_lift configureTimer_impl;
    gets configureTimer_val
  od"
```

consts

```
  initTimer_impl :: "unit machine_rest_monad"
```

definition

```
  initTimer :: "unit machine_monad"
```

where `"initTimer ≡ machine_op_lift initTimer_impl"`

consts

```
  resetTimer_impl :: "unit machine_rest_monad"
```

definition

```
  resetTimer :: "unit machine_monad"
```

where `"resetTimer ≡ machine_op_lift resetTimer_impl"`

consts

```
  setCurrentPD_impl :: "paddr ⇒ unit machine_rest_monad"
```

definition

```
  setCurrentPD :: "paddr ⇒ unit machine_monad"
```

where `"setCurrentPD pd ≡ machine_op_lift (setCurrentPD_impl pd)"`

consts

```
  setHardwareASID_impl :: "hardware_asid ⇒ unit machine_rest_monad"
```

definition

```
  setHardwareASID:: "hardware_asid ⇒ unit machine_monad"
```

where `"setHardwareASID a ≡ machine_op_lift (setHardwareASID_impl a)"`

consts

```
  isb_impl :: "unit machine_rest_monad"
```

definition

```
  isb :: "unit machine_monad"
```

where "isb ≡ machine_op_lift isb_impl"

consts
  dsb_impl :: "unit machine_rest_monad"
definition
  dsb :: "unit machine_monad"
where "dsb ≡ machine_op_lift dsb_impl"

consts
  dmb_impl :: "unit machine_rest_monad"
definition
  dmb :: "unit machine_monad"
where "dmb ≡ machine_op_lift dmb_impl"

consts
  invalidateTLB_impl :: "unit machine_rest_monad"
definition
  invalidateTLB :: "unit machine_monad"
where "invalidateTLB ≡ machine_op_lift invalidateTLB_impl"

consts
  invalidateTLB_ASID_impl :: "hardware_asid ⇒ unit machine_rest_monad"
definition
  invalidateTLB_ASID :: "hardware_asid ⇒ unit machine_monad"
where "invalidateTLB_ASID a ≡ machine_op_lift (invalidateTLB_ASID_impl a)"

consts
  invalidateTLB_VAASID_impl :: "machine_word ⇒ unit machine_rest_monad"
definition
  invalidateTLB_VAASID :: "machine_word ⇒ unit machine_monad"
where "invalidateTLB_VAASID w ≡ machine_op_lift (invalidateTLB_VAASID_impl w)"

consts
  cleanByVA_impl :: "machine_word ⇒ paddr ⇒ unit machine_rest_monad"
definition
  cleanByVA :: "machine_word ⇒ paddr ⇒ unit machine_monad"
where "cleanByVA w p ≡ machine_op_lift (cleanByVA_impl w p)"

consts
  cleanByVA_PoU_impl :: "machine_word ⇒ paddr ⇒ unit machine_rest_monad"
definition
  cleanByVA_PoU :: "machine_word ⇒ paddr ⇒ unit machine_monad"
where "cleanByVA_PoU w p ≡ machine_op_lift (cleanByVA_PoU_impl w p)"

consts
  invalidateByVA_impl :: "machine_word ⇒ paddr ⇒ unit machine_rest_monad"
definition
  invalidateByVA :: "machine_word ⇒ paddr ⇒ unit machine_monad"
where "invalidateByVA w p ≡ machine_op_lift (invalidateByVA_impl w p)"

consts
  invalidateByVA_I_impl :: "machine_word ⇒ paddr ⇒ unit machine_rest_monad"

definition
```
  invalidateByVA_I :: "machine_word ⇒ paddr ⇒ unit machine_monad"
```
where `"invalidateByVA_I w p ≡ machine_op_lift (invalidateByVA_I_impl w p)"`

consts
```
  invalidate_I_PoU_impl :: "unit machine_rest_monad"
```
definition
```
  invalidate_I_PoU :: "unit machine_monad"
```
where `"invalidate_I_PoU ≡ machine_op_lift invalidate_I_PoU_impl"`

consts
```
  cleanInvalByVA_impl :: "machine_word ⇒ paddr ⇒ unit machine_rest_monad"
```
definition
```
  cleanInvalByVA :: "machine_word ⇒ paddr ⇒ unit machine_monad"
```
where `"cleanInvalByVA w p ≡ machine_op_lift (cleanInvalByVA_impl w p)"`

consts
```
  branchFlush_impl :: "machine_word ⇒ paddr ⇒ unit machine_rest_monad"
```
definition
```
  branchFlush :: "machine_word ⇒ paddr ⇒ unit machine_monad"
```
where `"branchFlush w p ≡ machine_op_lift (branchFlush_impl w p)"`

consts
```
  clean_D_PoU_impl :: "unit machine_rest_monad"
```
definition
```
  clean_D_PoU :: "unit machine_monad"
```
where `"clean_D_PoU ≡ machine_op_lift clean_D_PoU_impl"`

consts
```
  cleanInvalidate_D_PoC_impl :: "unit machine_rest_monad"
```
definition
```
  cleanInvalidate_D_PoC :: "unit machine_monad"
```
where `"cleanInvalidate_D_PoC ≡ machine_op_lift cleanInvalidate_D_PoC_impl"`

consts
```
  cleanInvalidateL2Range_impl :: "paddr ⇒ paddr ⇒ unit machine_rest_monad"
```
definition
```
  cleanInvalidateL2Range :: "paddr ⇒ paddr ⇒ unit machine_monad"
```
where `"cleanInvalidateL2Range w p ≡ machine_op_lift (cleanInvalidateL2Range_impl w p)"`

consts
```
  invalidateL2Range_impl :: "paddr ⇒ paddr ⇒ unit machine_rest_monad"
```
definition
```
  invalidateL2Range :: "paddr ⇒ paddr ⇒ unit machine_monad"
```
where `"invalidateL2Range w p ≡ machine_op_lift (invalidateL2Range_impl w p)"`

consts
```
  cleanL2Range_impl :: "paddr ⇒ paddr ⇒ unit machine_rest_monad"
```
definition
```
  cleanL2Range :: "paddr ⇒ paddr ⇒ unit machine_monad"
```
where `"cleanL2Range w p ≡ machine_op_lift (cleanL2Range_impl w p)"`

consts
```
  initL2Cache_impl :: "unit machine_rest_monad"
```
definition
```
  initL2Cache :: "unit machine_monad"
```
where `"initL2Cache ≡ machine_op_lift initL2Cache_impl"`

```
definition
  clearExMonitor :: "unit machine_monad"
where "clearExMonitor ≡ modify (λs. s (| exclusive_state := default_exclusive_state |))"

consts
  flushBTAC_impl :: "unit machine_rest_monad"
definition
  flushBTAC :: "unit machine_monad"
where "flushBTAC ≡ machine_op_lift flushBTAC_impl"

consts
  writeContextID_impl :: "unit machine_rest_monad"
definition
  writeContextID :: "unit machine_monad"
where "writeContextID ≡ machine_op_lift writeContextID_impl"

lemmas cache_machine_op_defs = isb_def dsb_def dmb_def writeContextID_def flushBTAC_def
                               clearExMonitor_def cleanL2Range_def invalidateL2Range_def
                               cleanInvalidateL2Range_def cleanInvalidate_D_PoC_def
                               clean_D_PoU_def branchFlush_def cleanInvalByVA_def
                               invalidate_I_PoU_def invalidateByVA_I_def invalidateByVA_def
                               cleanByVA_PoU_def cleanByVA_def invalidateTLB_VAASID_def
                               invalidateTLB_ASID_def invalidateTLB_def
consts
  IFSR_val :: "machine_state ⇒ machine_word"
  DFSR_val :: "machine_state ⇒ machine_word"
  FAR_val :: "machine_state ⇒ machine_word"

definition
  getIFSR :: "machine_word machine_monad"
  where "getIFSR ≡ gets IFSR_val"

definition
  getDFSR :: "machine_word machine_monad"
  where "getDFSR ≡ gets DFSR_val"

definition
  getFAR :: "machine_word machine_monad"
  where "getFAR ≡ gets FAR_val"

definition
  debugPrint :: "unit list ⇒ unit machine_monad"
where
  debugPrint_def[simp]:
 "debugPrint ≡ λmessage. return ()"
```

— Interrupt controller operations

`getActiveIRQ` is now derministic. It 'updates' the irq state to the reflect the passage of time since last the irq was gotten, then it gets the active IRQ (if there is one).

```
definition
  getActiveIRQ :: "(irq option) machine_monad"
where
  "getActiveIRQ ≡ do
    is_masked ← gets $ irq_masks;
    modify (λs. s (| irq_state := irq_state s + 1 |));
    active_irq ← gets $ irq_oracle ∘ irq_state;
```

```
    if is_masked active_irq ∨ active_irq = 0xFF
    then return None
    else return ((Some active_irq) :: irq option)
  od"
```

definition
```
  maskInterrupt :: "bool ⇒ irq ⇒ unit machine_monad"
```
where
```
  "maskInterrupt m irq ≡
  modify (λs. s (| irq_masks := (irq_masks s) (irq := m) |))"
```

Does nothing on imx31

definition
```
  ackInterrupt :: "irq ⇒ unit machine_monad"
```
where
```
  "ackInterrupt ≡ λirq. return ()"
```

definition
```
  lineStart :: "machine_word ⇒ machine_word"
```
where
```
  "lineStart addr = (addr >> cacheLineBits) << cacheLineBits"
```

Performs the given operation on every cache line that intersects the supplied range.

definition
```
  cacheRangeOp :: "(machine_word ⇒ paddr ⇒ unit machine_monad)
               ⇒ machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "cacheRangeOp operation vstart vend pstart ≡
    let pend = pstart + (vend - vstart);
        vptrs = [lineStart vstart, lineStart vstart + of_nat cacheLine .e. lineStart vend];
        pptrs = [lineStart pstart, lineStart pstart + of_nat cacheLine .e. lineStart pend]
    in mapM_x (λ(v, p). operation v p) (zip vptrs pptrs)"
```

definition
```
  cleanCacheRange_PoC :: "machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "cleanCacheRange_PoC vstart vend pstart ≡ cacheRangeOp cleanByVA vstart vend pstart"
```

definition
```
  cleanInvalidateCacheRange_RAM :: "machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "cleanInvalidateCacheRange_RAM vstart vend pstart ≡ do
    cleanCacheRange_PoC vstart vend pstart;
    dsb;
    cleanInvalidateL2Range pstart (pstart + (vend - vstart));
    cacheRangeOp cleanInvalByVA vstart vend pstart;
    dsb
  od"
```

definition
```
  cleanCacheRange_RAM :: "machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "cleanCacheRange_RAM vstart vend pstart ≡ do
    cleanCacheRange_PoC vstart vend pstart;
    dsb;
    cleanL2Range pstart (pstart + (vend - vstart))
  od"
```

definition
```
  cleanCacheRange_PoU :: "machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "cleanCacheRange_PoU vstart vend pstart ≡ cacheRangeOp cleanByVA_PoU vstart vend pstart"
```

definition
```
  invalidateCacheRange_RAM :: "machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "invalidateCacheRange_RAM vstart vend pstart ≡ do
    when (vstart ≠ lineStart vstart) $
        cleanCacheRange_RAM vstart vstart pstart;
    when (vend + 1 ≠ lineStart (vend + 1)) $
        cleanCacheRange_RAM (lineStart vend) (lineStart vend)
           (pstart + ((lineStart vend) - vstart));
    invalidateL2Range pstart (pstart + (vend - vstart));
    cacheRangeOp invalidateByVA vstart vend pstart;
    dsb
  od"
```

definition
```
  invalidateCacheRange_I :: "machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "invalidateCacheRange_I vstart vend pstart ≡ cacheRangeOp invalidateByVA_I vstart vend pstart"
```

definition
```
  branchFlushRange :: "machine_word ⇒ machine_word ⇒ paddr ⇒ unit machine_monad"
```
where
```
  "branchFlushRange vstart vend pstart ≡ cacheRangeOp branchFlush vstart vend pstart"
```

definition
```
  cleanCaches_PoU :: "unit machine_monad"
```
where
```
  "cleanCaches_PoU ≡ do
    dsb;
    clean_D_PoU;
    dsb;
    invalidate_I_PoU;
    dsb
  od"
```

definition
```
  cleanInvalidateL1Caches :: "unit machine_monad"
```
where
```
  "cleanInvalidateL1Caches ≡ do
    dsb;
    cleanInvalidate_D_PoC;
    dsb;
    invalidate_I_PoU;
    dsb
  od"
```

17.3 Memory Clearance

Clear memory contents to recycle it as user memory

definition
```
  clearMemory :: "machine_word ⇒ nat ⇒ unit machine_monad"
```
  where

```
"clearMemory ptr bytelength ≡
  do mapM_x (λp. storeWord p 0) [ptr, ptr + word_size .e. ptr + (of_nat bytelength) - 1];
     cleanCacheRange_PoU ptr (ptr + of_nat bytelength - 1) (addrFromPPtr ptr)
  od"
```

definition

```
  clearMemoryVM :: "machine_word ⇒ nat ⇒ unit machine_monad"
  where
  "clearMemoryVM ptr bits ≡ return ()"
```

Initialize memory to be used as user memory. Note that zeroing out the memory is redundant in the specifications. In any case, we cannot abstract from the call to cleanCacheRange, which appears in the implementation.

abbreviation `(input) "initMemory == clearMemory"`

Free memory that had been initialized as user memory. While freeing memory is a no-op in the implementation, we zero out the underlying memory in the specifications to avoid garbage. If we know that there is no garbage, we can compute from the implementation state what the exact memory content in the specifications is.

definition

```
  freeMemory :: "machine_word ⇒ nat ⇒ unit machine_monad"
  where
 "freeMemory ptr bits ≡
  mapM_x (λp. storeWord p 0) [ptr, ptr + word_size  .e.  ptr + 2 ^ bits - 1]"
```

17.4 User Monad

type_synonym `user_context = "register ⇒ machine_word"`

type_synonym `'a user_monad = "(user_context, 'a) nondet_monad"`

translations

```
  (type) "'a user_monad" <= (type) "(register ⇒ machine_word, 'a) nondet_monad"
```

definition

```
  getRegister :: "register ⇒ machine_word user_monad"
where
  "getRegister r ≡ gets (λuc. uc r)"
```

definition

```
  setRegister :: "register ⇒ machine_word ⇒ unit user_monad"
where
  "setRegister r v ≡ modify (λuc. uc (r := v))"
```

definition

```
  "getRestartPC ≡ getRegister FaultInstruction"
```

definition

```
  "setNextPC ≡ setRegister LR_svc"
```

end

18 Basic Data Structures

```
theory Structures_A
imports
  ARM_Structs_A
  "../machine/MachineOps"
begin
```

User mode can request these objects to be created by retype:

```
datatype apiobject_type =
    Untyped
  | TCBObject
  | EndpointObject
  | AsyncEndpointObject
  | CapTableObject
  | ArchObject aobject_type
```

These allow more informative type signatures for IPC operations.

```
type_synonym badge = data
type_synonym msg_label = data
type_synonym message = data
```

This type models refences to capability slots. The first element of the tuple points to the object the capability is contained in. The second element is the index of the slot inside a slot-containing object. The default slot-containing object is a cnode, thus the name `cnode_index`.

```
type_synonym cnode_index = "bool list"
type_synonym cslot_ptr = "obj_ref × cnode_index"
```

Capabilities. Capabilities represent explicit authority to perform some action and are required for all system calls. Capabilities to Endpoint, AsyncEndpoint, Thread and CNode objects allow manipulation of standard kernel objects. Untyped capabilities allow the creation and removal of kernel objects from a memory region. Reply capabilities allow sending a one-off message to a thread waiting for a reply. IRQHandler and IRQControl caps allow a user to configure the way interrupts on one or all IRQs are handled. Capabilities to architecture-specific facilities are provided through the `arch_cap` type. Null capabilities are the contents of empty capability slots; they confer no authority and can be freely replaced. Zombie capabilities are stored when the deletion of CNode and Thread objects is partially completed; they confer no authority but cannot be replaced until the deletion is finished.

```
datatype cap
         = NullCap
         | UntypedCap obj_ref nat nat
           — pointer, size in bits (i.e. size = 2^bits) and freeIndex (i.e. freeRef = obj_ref + (freeIndex
* 2^4))
         | EndpointCap obj_ref badge cap_rights
         | AsyncEndpointCap obj_ref badge cap_rights
         | ReplyCap obj_ref bool
         | CNodeCap obj_ref nat "bool list"
           — CNode ptr, number of bits translated, guard
         | ThreadCap obj_ref
         | DomainCap
         | IRQControlCap
         | IRQHandlerCap irq
```

```
        | Zombie obj_ref "nat option" nat
          — cnode ptr * nat + tcb or cspace ptr
        | ArchObjectCap arch_cap
```

The CNode object is an array of capability slots. The domain of the function will always be the set of boolean lists of some specific length. Empty slots contain a Null capability.

```
type_synonym cnode_contents = "cnode_index ⇒ cap option"
```

Various access functions for the cap type are defined for convenience.

```
definition
  the_cnode_cap :: "cap ⇒ obj_ref × nat × bool list" where
  "the_cnode_cap cap ≡
  case cap of
    CNodeCap oref bits guard ⇒ (oref, bits, guard)"

definition
  the_arch_cap :: "cap ⇒ arch_cap" where
  "the_arch_cap cap ≡ case cap of ArchObjectCap a ⇒ a"

primrec
  cap_ep_badge :: "cap ⇒ badge"
where
  "cap_ep_badge (EndpointCap _ badge _) = badge"
| "cap_ep_badge (AsyncEndpointCap _ badge _) = badge"

primrec
  cap_ep_ptr :: "cap ⇒ badge"
where
  "cap_ep_ptr (EndpointCap obj_ref _ _) = obj_ref"
| "cap_ep_ptr (AsyncEndpointCap obj_ref _ _) = obj_ref"

definition
  bits_of :: "cap ⇒ nat" where
  "bits_of cap ≡ case cap of
    UntypedCap _ bits _ ⇒ bits
  | CNodeCap _ radix_bits _ ⇒ radix_bits"

definition
  free_index_of :: "cap ⇒ nat" where
  "free_index_of cap ≡ case cap of
    UntypedCap _ _ free_index ⇒ free_index"

definition
  is_reply_cap :: "cap ⇒ bool" where
  "is_reply_cap cap ≡ case cap of ReplyCap _ m ⇒ ¬ m | _ ⇒ False"
definition
  is_master_reply_cap :: "cap ⇒ bool" where
  "is_master_reply_cap cap ≡ case cap of ReplyCap _ m ⇒ m | _ ⇒ False"
definition
  is_zombie :: "cap ⇒ bool" where
  "is_zombie cap ≡ case cap of Zombie _ _ _ ⇒ True | _ ⇒ False"
definition
  is_arch_cap :: "cap ⇒ bool" where
  "is_arch_cap cap ≡ case cap of ArchObjectCap _ ⇒ True | _ ⇒ False"

fun is_cnode_cap :: "cap ⇒ bool"
where
```

```
  "is_cnode_cap (CNodeCap _ _ _) = True"
| "is_cnode_cap _                = False"

fun is_thread_cap :: "cap ⇒ bool"
where
  "is_thread_cap (ThreadCap _) = True"
| "is_thread_cap _             = False"

fun is_domain_cap :: "cap ⇒ bool"
where
  "is_domain_cap DomainCap = True"
| "is_domain_cap _ = False"

fun is_untyped_cap :: "cap ⇒ bool"
where
  "is_untyped_cap (UntypedCap _ _ _) = True"
| "is_untyped_cap _                  = False"

fun is_ep_cap :: "cap ⇒ bool"
where
  "is_ep_cap (EndpointCap _ _ _) = True"
| "is_ep_cap _                   = False"

fun is_aep_cap :: "cap ⇒ bool"
where
  "is_aep_cap (AsyncEndpointCap _ _ _) = True"
| "is_aep_cap _                        = False"

primrec
  cap_rights :: "cap ⇒ cap_rights"
where
  "cap_rights (EndpointCap _ _ cr) = cr"
| "cap_rights (AsyncEndpointCap _ _ cr) = cr"
| "cap_rights (ArchObjectCap acap) = acap_rights acap"
```

Various update functions for cap data common to various kinds of cap are defined here.

```
definition
  cap_rights_update :: "cap_rights ⇒ cap ⇒ cap" where
  "cap_rights_update cr' cap ≡
   case cap of
     EndpointCap oref badge cr ⇒ EndpointCap oref badge cr'
   | AsyncEndpointCap oref badge cr
     ⇒ AsyncEndpointCap oref badge (cr' - {AllowGrant})
   | ArchObjectCap acap ⇒ ArchObjectCap (acap_rights_update cr' acap)
   | _ ⇒ cap"
```

For implementation reasons not all bits of the badge word can be used.

```
definition
  badge_bits :: nat where
  "badge_bits ≡ 28"

declare badge_bits_def [simp]

definition
  badge_update :: "badge ⇒ cap ⇒ cap" where
  "badge_update data cap ≡
   case cap of
     EndpointCap oref badge cr ⇒ EndpointCap oref (data && mask badge_bits) cr
```

```
  | AsyncEndpointCap oref badge cr ⇒ AsyncEndpointCap oref (data && mask badge_bits) cr
  | _ ⇒ cap"
```

definition
```
  mask_cap :: "cap_rights ⇒ cap ⇒ cap" where
  "mask_cap rights cap ≡ cap_rights_update (cap_rights cap ∩ rights) cap"
```

18.1 Message Info

The message info is the first thing interpreted on a user system call and determines the structure of the message the user thread is sending either to another user or to a system service. It is also passed to user threads receiving a message to indicate the structure of the message they have received. The `mi_length` parameter is the number of data words in the body of the message. The `mi_extra_caps` parameter is the number of caps to be passed together with the message. The `mi_caps_unwrapped` parameter is a bitmask allowing threads receiving a message to determine how extra capabilities were transferred. The `mi_label` parameter is transferred directly from sender to receiver as part of the message.

datatype `message_info = MI length_type length_type data msg_label`

primrec
```
  mi_label :: "message_info ⇒ msg_label"
```
where
```
  "mi_label (MI ln exc unw label) = label"
```

primrec
```
  mi_length :: "message_info ⇒ length_type"
```
where
```
  "mi_length (MI ln exc unw label) = ln"
```

primrec
```
  mi_extra_caps :: "message_info ⇒ length_type"
```
where
```
  "mi_extra_caps (MI ln exc unw label) = exc"
```

primrec
```
  mi_caps_unwrapped :: "message_info ⇒ data"
```
where
```
 "mi_caps_unwrapped (MI ln exc unw label) = unw"
```

Message infos are encoded to or decoded from a data word.

primrec
```
  message_info_to_data :: "message_info ⇒ data"
```
where
```
  "message_info_to_data (MI ln exc unw mlabel) =
   (let
        extra = exc << 7;
        unwrapped = unw << 9;
        label = mlabel << 12
    in
       label || extra || unwrapped || ln)"
```

Hard-coded to avoid recursive imports?

definition
```
  data_to_message_info :: "data ⇒ message_info"
```

where
```
  "data_to_message_info w ≡
   MI (let v = w && ((1 << 7) - 1) in if v > 120 then 120 else v) ((w >> 7) && ((1 << 2) - 1))
      ((w >> 9) && ((1 << 3) - 1)) (w >> 12)"
```

18.2 Kernel Objects

Endpoints are synchronous points of communication for threads. At any time an endpoint may contain a queue of threads waiting to send, a queue of threads waiting to receive or be idle. Whenever threads would be waiting to send and receive simultaneously messages are transferred immediately.

```
datatype endpoint
           = IdleEP
           | SendEP "obj_ref list"
           | RecvEP "obj_ref list"
```

AsyncEndpoints are asynchronous points of communication. Unlike regular endpoints, threads may block waiting to receive but not to send. Whenever a thread sends to an async endpoint, its message is stored in the endpoint immediately.

```
datatype async_ep
           = IdleAEP
           | WaitingAEP "obj_ref list"
           | ActiveAEP badge message
```

```
definition
  default_ep :: endpoint where
  "default_ep ≡ IdleEP"
```

```
definition
  default_async_ep :: async_ep where
  "default_async_ep ≡ IdleAEP"
```

Thread Control Blocks are the in-kernel representation of a thread.

Threads which can execute are either in the Running state for normal execution, in the Restart state if their last operation has not completed yet or in the IdleThreadState for the unique system idle thread. Threads can also be blocked waiting for any of the different kinds of system messages. The Inactive state indicates that the TCB is not currently used by a running thread.

TCBs also contain some special-purpose capability slots. The CTable slot is a capability to a CNode through which the thread accesses capabilities with which to perform system calls. The VTable slot is a capability to a virtual address space (an architecture-specific capability type) in which the thread runs. If the thread has issued a Reply cap to another thread and is awaiting a reply, that cap will have a "master" Reply cap as its parent in the Reply slot. The Caller slot is used to initially store any Reply cap issued to this thread. The IPCFrame slot stores a capability to a memory frame (an architecture-specific capability type) through which messages will be sent and received.

If the thread has encountered a fault and is waiting to send it to its supervisor the fault is stored in `tcb_fault`. The user register file is stored in `tcb_context`, the pointer to the cap in the IPCFrame slot in `tcb_ipc_buffer` and the identity of the Endpoint cap through which faults are to be sent in `tcb_fault_handler`.

```
record sender_payload =
 sender_badge     :: badge
 sender_can_grant :: bool
 sender_is_call   :: bool
```

```
datatype thread_state
```

```
  = Running
  | Inactive
  | Restart
  | BlockedOnReceive obj_ref bool
  | BlockedOnSend obj_ref sender_payload
  | BlockedOnReply
  | BlockedOnAsyncEvent obj_ref
  | IdleThreadState
```

record tcb =

```
 tcb_ctable        :: cap
 tcb_vtable        :: cap
 tcb_reply         :: cap
 tcb_caller        :: cap
 tcb_ipcframe      :: cap
 tcb_state         :: thread_state
 tcb_fault_handler :: cap_ref
 tcb_ipc_buffer    :: vspace_ref
 tcb_context       :: user_context
 tcb_fault         :: "fault option"
```

Determines whether a thread in a given state may be scheduled.

primrec

```
  runnable :: "Structures_A.thread_state ⇒ bool"
```

where

```
  "runnable (Running)                = True"
| "runnable (Inactive)               = False"
| "runnable (Restart)                = True"
| "runnable (BlockedOnReceive x y)   = False"
| "runnable (BlockedOnSend x y)      = False"
| "runnable (BlockedOnAsyncEvent x)  = False"
| "runnable (IdleThreadState)        = False"
| "runnable (BlockedOnReply)         = False"
```

definition

```
  default_tcb :: tcb where
  "default_tcb ≡ (|
      tcb_ctable   = NullCap,
      tcb_vtable   = NullCap,
      tcb_reply    = NullCap,
      tcb_caller   = NullCap,
      tcb_ipcframe = NullCap,
      tcb_state    = Inactive,
      tcb_fault_handler = to_bl (0::word32),
      tcb_ipc_buffer = 0,
      tcb_context    = new_context,
      tcb_fault      = None |)"
```

All kernel objects are CNodes, TCBs, Endpoints, AsyncEndpoints or architecture specific.

datatype kernel_object

```
         = CNode nat cnode_contents — size in bits, and contents
         | TCB tcb
         | Endpoint endpoint
         | AsyncEndpoint async_ep
         | ArchObj arch_kernel_obj
```

Checks whether a cnode's contents are well-formed.

```
definition
  well_formed_cnode_n :: "nat ⇒ cnode_contents ⇒ bool" where
 "well_formed_cnode_n n ≡ λcs. dom cs = {x. length x = n}"

definition
  cte_level_bits :: nat where
  "cte_level_bits  ≡ 4"

primrec
  obj_bits :: "kernel_object ⇒ nat"
where
  "obj_bits (CNode sz cs) = (if well_formed_cnode_n sz cs
                             then cte_level_bits + sz
                             else cte_level_bits)"
| "obj_bits (TCB t) = 9"
| "obj_bits (Endpoint ep) = 4"
| "obj_bits (AsyncEndpoint aep) = 4"
| "obj_bits (ArchObj ao) = arch_kobj_size ao"

primrec
  obj_size :: "cap ⇒ word32"
where
  "obj_size NullCap = 0"
| "obj_size (UntypedCap r bits f) = 1 << bits"
| "obj_size (EndpointCap r b R) = 1 << obj_bits (Endpoint undefined)"
| "obj_size (AsyncEndpointCap r b R) = 1 << obj_bits (AsyncEndpoint undefined)"
| "obj_size (CNodeCap r bits g) = 1 << (cte_level_bits + bits)"
| "obj_size (ThreadCap r) = 1 << obj_bits (TCB undefined)"
| "obj_size (Zombie r zb n) = (case zb of None ⇒ 1 << obj_bits (TCB undefined)
                                         | Some n ⇒ 1 << (cte_level_bits + n))"
| "obj_size (ArchObjectCap a) = 1 << arch_obj_size a"
```

18.3 Kernel State

The kernel's heap is a partial function containing kernel objects.

```
type_synonym kheap = "obj_ref ⇒ kernel_object option"
```

Capabilities are created either by cloning an existing capability or by creating a subordinate capability from it. This results in a capability derivation tree or CDT. The kernel provides a Revoke operation which deletes all capabilities derived from one particular capability. To support this, the kernel stores the CDT explicitly. It is here stored as a tree, a partial mapping from capability slots to parent capability slots.

```
type_synonym cdt = "cslot_ptr ⇒ cslot_ptr option"

datatype irq_state =
   IRQInactive
 | IRQNotifyAEP
 | IRQTimer
```

The kernel state includes a heap, a capability derivation tree (CDT), a bitmap used to determine if a capability is the original capability to that object, a pointer to the current thread, a pointer to the system idle thread, the state of the underlying machine, per-irq pointers to cnodes (each containing one async endpoint through which interrupts are delivered), an array recording which interrupts are used for which purpose, and the state of the architecture-specific kernel module.

Note: for each irq, `interrupt_irq_node irq` points to a cnode which can contain the async endpoint cap through which interrupts are delivered. In C, this all lives in a single array. In the abstract spec though, to prove security, we can't have a single object accessible by everyone. Hence the need to separate irq handlers.

```
record abstract_state =
  kheap              :: kheap
  cdt                :: cdt
  is_original_cap    :: "cslot_ptr ⇒ bool"
  cur_thread         :: obj_ref
  idle_thread        :: obj_ref
  machine_state      :: machine_state
  interrupt_irq_node :: "irq ⇒ obj_ref"
  interrupt_states   :: "irq ⇒ irq_state"
  arch_state         :: arch_state
```

The following record extends the abstract kernel state with extra state of type `'a`. The specification operates over states of this extended type. By choosing an appropriate concrete type for `'a` we may obtain different *instantiations* of the kernel specifications at differing levels of abstraction. See chapter 19 for further information.

```
record 'a state = abstract_state + exst :: 'a
```

This wrapper lifts monadic operations on the underlying machine state to monadic operations on the kernel state.

```
definition
  do_machine_op :: "(machine_state, 'a) nondet_monad ⇒ ('z state, 'a) nondet_monad"
where
 "do_machine_op mop ≡ do
    ms ← gets machine_state;
    (r, ms') ← select_f (mop ms);
    modify (λstate. state ⦇ machine_state := ms' ⦈);
    return r
  od"
```

This function generates the cnode indices used when addressing the capability slots within a TCB.

```
definition
  tcb_cnode_index :: "nat ⇒ cnode_index" where
  "tcb_cnode_index n ≡ to_bl (of_nat n :: 3 word)"
```

Zombie capabilities store the bit size of the CNode cap they were created from or None if they were created from a TCB cap. This function decodes the bit-length of cnode indices into the relevant kernel objects.

```
definition
  zombie_cte_bits :: "nat option ⇒ nat" where
 "zombie_cte_bits N ≡ case N of Some n ⇒ n | None ⇒ 3"

lemma zombie_cte_bits_simps[simp]:
 "zombie_cte_bits (Some n) = n"
 "zombie_cte_bits None     = 3"
```

The first capability slot of the relevant kernel object.

```
primrec
  first_cslot_of :: "cap ⇒ cslot_ptr"
where
  "first_cslot_of (ThreadCap oref) = (oref, tcb_cnode_index 0)"
| "first_cslot_of (CNodeCap oref bits g) = (oref, replicate bits False)"
| "first_cslot_of (Zombie oref bits n) = (oref, replicate (zombie_cte_bits bits) False)"
```

The set of all objects referenced by a capability.

```
primrec
  obj_refs :: "cap ⇒ obj_ref set"
where
  "obj_refs NullCap = {}"
| "obj_refs (ReplyCap r m) = {}"
| "obj_refs IRQControlCap = {}"
| "obj_refs (IRQHandlerCap irq) = {}"
| "obj_refs (UntypedCap r s f) = {}"
| "obj_refs (CNodeCap r bits guard) = {r}"
| "obj_refs (EndpointCap r b cr) = {r}"
| "obj_refs (AsyncEndpointCap r b cr) = {r}"
| "obj_refs (ThreadCap r) = {r}"
| "obj_refs DomainCap = {}"
| "obj_refs (Zombie ptr b n) = {ptr}"
| "obj_refs (ArchObjectCap x) = Option.set (aobj_ref x)"
```

The partial definition below is sometimes easier to work with. It also provides cases for UntypedCap and ReplyCap which are not true object references in the sense of the other caps.

```
primrec
  obj_ref_of :: "cap ⇒ obj_ref"
where
  "obj_ref_of (UntypedCap r s f) = r"
| "obj_ref_of (ReplyCap r m) = r"
| "obj_ref_of (CNodeCap r bits guard) = r"
| "obj_ref_of (EndpointCap r b cr) = r"
| "obj_ref_of (AsyncEndpointCap r b cr) = r"
| "obj_ref_of (ThreadCap r) = r"
| "obj_ref_of (Zombie ptr b n) = ptr"
| "obj_ref_of (ArchObjectCap x) = the (aobj_ref x)"

primrec
  cap_bits_untyped :: "cap ⇒ nat"
where
  "cap_bits_untyped (UntypedCap r s f) = s"

definition
  "tcb_cnode_map tcb ≡
   [tcb_cnode_index 0 ↦ tcb_ctable tcb,
    tcb_cnode_index 1 ↦ tcb_vtable tcb,
    tcb_cnode_index 2 ↦ tcb_reply tcb,
    tcb_cnode_index 3 ↦ tcb_caller tcb,
    tcb_cnode_index 4 ↦ tcb_ipcframe tcb]"

definition
  "cap_of kobj ≡
   case kobj of CNode _ cs ⇒ cs | TCB tcb ⇒ tcb_cnode_map tcb | _ ⇒ empty"
```

The set of all caps contained in a kernel object.

```
definition
  caps_of :: "kernel_object ⇒ cap set" where
  "caps_of kobj ≡ ran (cap_of kobj)"

end
```

19 Abstract Specification Instantiations

```
theory Deterministic_A
imports
  Structures_A
  "../../lib/List_Lib"

begin
```

The kernel specification operates over states of type `'a state`, which includes all of the abstract kernel state plus an extra field, `exst` of type `'a`. By choosing an appropriate concrete type for `'a`, we obtain different *instantiations* of this specification, at differing levels of abstraction. The abstract specification is thus *extensible*. The basic technique, and its motivation, are described in [14].

Here, we define two such instantiations. The first yields a largely-deterministic specification by instantiating `'a` with a record that includes concrete scheduler state and information about sibling ordering in the capability derivation tree (CDT). We call the resulting specification the *deterministic abstract specification* and it is defined below in section 19.1.

The second instantiation uses the type `unit` for `'a`, yielding a specification that is far more nondeterministic. In particular, the scheduling behaviour and the order in which capabilities are deleted during a *revoke* system call both become completely nondeterministic. We call this second instantiation the *nondeterministic abstract specification* and it is defined below in section 19.2.

Translate a state of type `'a state` to one of type `'b state` via a function `t` from `'a` to `'b`.

```
definition trans_state :: "('a ⇒ 'b) ⇒ 'a state ⇒ 'b state" where
"trans_state t s = (|kheap = kheap s, cdt = cdt s, is_original_cap = is_original_cap s,
                     cur_thread = cur_thread s, idle_thread = idle_thread s,
                     machine_state = machine_state s,
                     interrupt_irq_node = interrupt_irq_node s,
                     interrupt_states = interrupt_states s, arch_state = arch_state s,
                     exst = t(exst s)|)"
```

Truncate an extended state of type `'a state` by effectively throwing away all the `'a` information.

```
abbreviation "truncate_state ≡ trans_state (λ_. ())"
```

19.1 Deterministic Abstract Specification

The deterministic abstract specification tracks the state of the scheduler and ordering information about sibling nodes in the CDT.

The current scheduler action, which is part of the scheduling state.

```
datatype scheduler_action =
    resume_cur_thread
  | switch_thread obj_ref
  | choose_new_thread

type_synonym priority = word8

type_synonym domain = word8
```

```
record etcb =
 tcb_priority :: "priority"
 tcb_time_slice :: "nat"
 tcb_domain :: "domain"

definition num_domains :: nat where
  "num_domains ≡ 16"

definition time_slice :: "nat" where
  "time_slice ≡ 5"

definition default_priority :: "priority" where
  "default_priority ≡ minBound"

definition default_domain :: "domain" where
  "default_domain ≡ minBound"

definition default_etcb :: "etcb" where
  "default_etcb ≡ (|tcb_priority = default_priority, tcb_time_slice = time_slice, tcb_domain =
default_domain|)"

type_synonym ready_queue = "obj_ref list"
```

For each entry in the CDT, we record an ordered list of its children. This encodes the order of sibling nodes in the CDT.

```
type_synonym cdt_list = "cslot_ptr ⇒ cslot_ptr list"

definition work_units_limit :: "32 word" where
  "work_units_limit = 0x64"
```

The extended state of the deterministic abstract specification.

```
record det_ext =
   work_units_completed_internal :: "32 word"
   scheduler_action_internal :: scheduler_action
   ekheap_internal :: "obj_ref ⇒ etcb option"
   domain_list_internal :: "(domain × 32 word) list"
   domain_index_internal :: nat
   cur_domain_internal :: domain
   domain_time_internal :: "32 word"
   ready_queues_internal :: "domain ⇒ priority ⇒ ready_queue"
   cdt_list_internal :: cdt_list
```

The state of the deterministic abstract specification extends the abstract state with the `det_ext` record.

```
type_synonym det_state = "det_ext state"
```

Accessor and update functions for the extended state of the deterministic abstract specification.

```
abbreviation
  "work_units_completed (s::det_state) ≡ work_units_completed_internal (exst s)"

abbreviation
  "work_units_completed_update f (s::det_state) ≡  trans_state (work_units_completed_internal_update
f) s"

abbreviation
  "scheduler_action (s::det_state) ≡ scheduler_action_internal (exst s)"

abbreviation
```

```
  "scheduler_action_update f (s::det_state) ≡  trans_state (scheduler_action_internal_update f)
s"
```

abbreviation
```
  "ekheap (s::det_state) ≡ ekheap_internal (exst s)"
```

abbreviation
```
  "ekheap_update f (s::det_state) ≡ trans_state (ekheap_internal_update f) s"
```

abbreviation
```
  "domain_list (s::det_state) ≡ domain_list_internal (exst s)"
```

abbreviation
```
  "domain_list_update f (s::det_state) ≡ trans_state (domain_list_internal_update f) s"
```

abbreviation
```
  "domain_index (s::det_state) ≡ domain_index_internal (exst s)"
```

abbreviation
```
  "domain_index_update f (s::det_state) ≡ trans_state (domain_index_internal_update f) s"
```

abbreviation
```
  "cur_domain (s::det_state) ≡ cur_domain_internal (exst s)"
```

abbreviation
```
  "cur_domain_update f (s::det_state) ≡ trans_state (cur_domain_internal_update f) s"
```

abbreviation
```
  "domain_time (s::det_state) ≡ domain_time_internal (exst s)"
```

abbreviation
```
  "domain_time_update f (s::det_state) ≡ trans_state (domain_time_internal_update f) s"
```

abbreviation
```
  "ready_queues (s::det_state) ≡ ready_queues_internal (exst s)"
```

abbreviation
```
  "ready_queues_update f (s::det_state) ≡ trans_state (ready_queues_internal_update f) s"
```

abbreviation
```
  "cdt_list (s::det_state) ≡ cdt_list_internal (exst s)"
```

abbreviation
```
  "cdt_list_update f (s::det_state) ≡ trans_state (cdt_list_internal_update f) s"
```

type_synonym `'a det_ext_monad = "(det_state,'a) nondet_monad"`

Basic monadic functions for operating on the extended state of the deterministic abstract specification.

definition
```
  get_etcb :: "obj_ref ⇒ det_state ⇒ etcb option"
```
where
```
  "get_etcb tcb_ref es ≡ ekheap es tcb_ref"
```

definition
```
  ethread_get :: "(etcb ⇒ 'a) ⇒ obj_ref ⇒ 'a det_ext_monad"
```
where
```
  "ethread_get f tptr ≡ do
     tcb ← gets_the $ get_etcb tptr;
```

```
    return $ f tcb
  od"
```

definition `set_eobject :: "obj_ref ⇒ etcb ⇒ unit det_ext_monad"`
where

```
 "set_eobject ptr obj ≡
  do es ← get;
    ekh ← return $ ekheap es(ptr ↦ obj);
    put (es⦇ekheap := ekh⦈)
  od"
```

definition

```
  ethread_set :: "(etcb ⇒ etcb) ⇒ obj_ref ⇒ unit det_ext_monad"
```

where

```
  "ethread_set f tptr ≡ do
     tcb ← gets_the $ get_etcb tptr;
     set_eobject tptr $ f tcb
   od"
```

definition

```
  set_scheduler_action :: "scheduler_action ⇒ unit det_ext_monad" where
  "set_scheduler_action action ≡
     modify (λes. es⦇scheduler_action := action⦈)"
```

definition

```
  thread_set_priority :: "obj_ref ⇒ priority ⇒ unit det_ext_monad" where
  "thread_set_priority tptr prio ≡ ethread_set (λtcb. tcb⦇tcb_priority := prio⦈) tptr"
```

definition

```
  thread_set_time_slice :: "obj_ref ⇒ nat ⇒ unit det_ext_monad" where
  "thread_set_time_slice tptr time ≡ ethread_set (λtcb. tcb⦇tcb_time_slice := time⦈) tptr"
```

definition

```
  thread_set_domain :: "obj_ref ⇒ domain ⇒ unit det_ext_monad" where
  "thread_set_domain tptr domain ≡ ethread_set (λtcb. tcb⦇tcb_domain := domain⦈) tptr"
```

definition

```
  get_tcb_queue :: "domain ⇒ priority ⇒ ready_queue det_ext_monad" where
  "get_tcb_queue d prio ≡ do
     queues ← gets ready_queues;
     return (queues d prio)
   od"
```

definition

```
  set_tcb_queue :: "domain ⇒ priority ⇒ ready_queue ⇒ unit det_ext_monad" where
  "set_tcb_queue d prio queue ≡
     modify (λes. es⦇ ready_queues :=
      (λd' p. if d' = d ∧ p = prio then queue else ready_queues es d' p)⦈)"
```

definition

```
  tcb_sched_action :: "(obj_ref ⇒ obj_ref list ⇒ obj_ref list) ⇒ obj_ref  ⇒ unit det_ext_monad"
```

where

```
  "tcb_sched_action action thread ≡ do
     d ← ethread_get tcb_domain thread;
     prio ← ethread_get tcb_priority thread;
     queue ← get_tcb_queue d prio;
```

```
    set_tcb_queue d prio (action thread queue)
  od"
```

definition

```
  tcb_sched_enqueue :: "obj_ref ⇒ obj_ref list ⇒ obj_ref list" where
  "tcb_sched_enqueue thread queue ≡ if (thread ∉ set queue) then thread # queue else queue"
```

definition

```
  tcb_sched_append :: "obj_ref ⇒ obj_ref list ⇒ obj_ref list" where
  "tcb_sched_append thread queue ≡ if (thread ∉ set queue) then queue @ [thread] else queue"
```

definition

```
  tcb_sched_dequeue :: "obj_ref ⇒ obj_ref list ⇒ obj_ref list" where
  "tcb_sched_dequeue thread queue ≡ filter (λx. x ≠ thread) queue"
```

definition `reschedule_required :: "unit det_ext_monad"` **where**

```
  "reschedule_required ≡ do
    action ← gets scheduler_action;
    case action of switch_thread t ⇒ tcb_sched_action (tcb_sched_enqueue) t | _ ⇒ return ();
    set_scheduler_action choose_new_thread
  od"
```

definition

```
  possible_switch_to :: "obj_ref ⇒ bool ⇒ unit det_ext_monad" where
  "possible_switch_to target on_same_prio ≡ do
    cur ← gets cur_thread;
    cur_dom ← gets cur_domain;
    cur_prio ← ethread_get tcb_priority cur;
    target_dom ← ethread_get tcb_domain target;
    target_prio ← ethread_get tcb_priority target;
    action ← gets scheduler_action;
    if (target_dom ≠ cur_dom) then tcb_sched_action tcb_sched_enqueue target
    else do
      if ((target_prio > cur_prio ∨ (target_prio = cur_prio ∧ on_same_prio))
             ∧ action = resume_cur_thread) then set_scheduler_action $ switch_thread target
        else tcb_sched_action tcb_sched_enqueue target;
      case action of switch_thread _ ⇒ reschedule_required | _ ⇒ return ()
    od
  od"
```

definition

```
  attempt_switch_to :: "obj_ref ⇒ unit det_ext_monad" where
  "attempt_switch_to target ≡ possible_switch_to target True"
```

definition

```
  switch_if_required_to :: "obj_ref ⇒ unit det_ext_monad" where
  "switch_if_required_to target ≡ possible_switch_to target False"
```

definition

```
  next_domain :: "unit det_ext_monad" where
  "next_domain ≡
    modify (λs.
      let domain_index' = (domain_index s + 1) mod length (domain_list s) in
      let next_dom = (domain_list s)!domain_index'
      in s⦇ domain_index := domain_index',
            cur_domain := fst next_dom,
            domain_time := snd next_dom,
```

```
          work_units_completed := 0⦈)"
```

definition
```
  dec_domain_time :: "unit det_ext_monad" where
  "dec_domain_time = modify (λs. s⦇domain_time := domain_time s - 1⦈)"
```

definition `set_cdt_list :: "cdt_list ⇒ (det_state, unit) nondet_monad"` **where**
```
  "set_cdt_list t ≡ do
    s ← get;
    put $ s⦇ cdt_list := t ⦈
  od"
```

definition
```
  update_cdt_list :: "(cdt_list ⇒ cdt_list) ⇒ (det_state, unit) nondet_monad"
where
  "update_cdt_list f ≡ do
     t ← gets cdt_list;
     set_cdt_list (f t)
  od"
```

The CDT in the implementation is stored in prefix traversal order. The following functions traverse its abstract representation here to yield corresponding information.

definition `next_child :: "cslot_ptr ⇒ cdt_list ⇒ cslot_ptr option"` **where**
```
  "next_child slot t ≡ case (t slot) of [] ⇒ None |
                                        x # xs ⇒ Some x"
```

definition `next_sib :: "cslot_ptr ⇒ cdt_list ⇒ cdt ⇒ cslot_ptr option"` **where**
```
  "next_sib slot t m ≡ case m slot of None ⇒ None |
                       Some p ⇒ after_in_list (t p) slot"
```

function `(domintros) next_not_child :: "cslot_ptr ⇒ cdt_list ⇒ cdt ⇒ cslot_ptr option"` **where**
```
  "next_not_child slot t m = (if next_sib slot t m = None
                              then (case m slot of
                                None ⇒ None |
                                Some p ⇒ next_not_child p t m)
                              else next_sib slot t m)"
```

definition `next_slot :: "cslot_ptr ⇒ cdt_list ⇒ cdt ⇒ cslot_ptr option"` **where**
```
  "next_slot slot t m ≡ if t slot ≠ []
                        then next_child slot t
                        else next_not_child slot t m"
```

Extended operations for the deterministic abstract specification.

definition `max_non_empty_queue :: "(priority ⇒ ready_queue) ⇒ ready_queue"` **where**
```
  "max_non_empty_queue queues ≡ queues (Max {prio. queues prio ≠ []})"
```

definition `decode_set_priority_error_choice`
```
  :: "priority ⇒ obj_ref ⇒ bool det_ext_monad" where
  "decode_set_priority_error_choice new_prio cur ≡
    do
      prio ← ethread_get tcb_priority cur;
      return (new_prio > prio)
    od"
```

```
definition default_ext :: "apiobject_type ⇒ domain ⇒ etcb option" where
  "default_ext type cdom ≡
      case type of TCBObject ⇒ Some (default_etcb(|tcb_domain := cdom|))
                         | _ ⇒ None"

definition retype_region_ext :: "obj_ref list ⇒ apiobject_type ⇒ unit det_ext_monad" where
  "retype_region_ext ptrs type ≡  do
                                     ekh ← gets ekheap;
                                     cdom ← gets cur_domain;
                                     ekh' ← return $ foldr (λp ekh. (ekh(p := default_ext type
cdom))) ptrs ekh;
                                     modify (λs. s(|ekheap := ekh'|))
                                  od"

definition recycle_cap_ext where
  "recycle_cap_ext ptr ≡ do cdom ← gets cur_domain; ethread_set (K (default_etcb(|tcb_domain :=
cdom|))) ptr od"

definition cap_swap_ext where
"cap_swap_ext ≡ (λ slot1 slot2 slot1_op slot2_op.
      do
       update_cdt_list (λlist. list(slot1 := list slot2, slot2 := list slot1));
       update_cdt_list
        (λlist. case if slot2_op = Some slot1 then Some slot2
                     else if slot2_op = Some slot2 then Some slot1 else slot2_op of
                None ⇒ case if slot1_op = Some slot1 then Some slot2
                            else if slot1_op = Some slot2 then Some slot1 else slot1_op of
                       None ⇒ list
                       | Some slot2_p ⇒ list(slot2_p := list_replace (list slot2_p) slot1 slot2)
                | Some slot1_p ⇒
                    case if slot1_op = Some slot1 then Some slot2
                         else if slot1_op = Some slot2 then Some slot1 else slot1_op of
                    None ⇒ list(slot1_p := list_replace (list slot1_p) slot2 slot1)
                    | Some slot2_p ⇒
                        if slot1_p = slot2_p
                        then list(slot1_p := list_swap (list slot1_p) slot1 slot2)
                        else list(slot1_p := list_replace (list slot1_p) slot2 slot1,
                                  slot2_p := list_replace (list slot2_p) slot1 slot2))
    od)"

definition cap_move_ext where
"cap_move_ext ≡ (λ src_slot dest_slot src_p dest_p.
 do

    update_cdt_list (λlist. case (dest_p) of
      None ⇒ list |
      Some p ⇒ list (p := list_remove (list p) dest_slot));

   if (src_slot = dest_slot) then return () else

    (do
    update_cdt_list (λlist. case (src_p) of
      None ⇒ list |
      Some p ⇒ list (p := list_replace (list p) src_slot dest_slot));

    update_cdt_list (λlist. list (src_slot := [], dest_slot := (list src_slot) @ (list dest_slot)))
    od)
```

```
  od)"
```

definition cap_insert_ext **where**

```
"cap_insert_ext ≡ (λ src_parent src_slot dest_slot src_p dest_p.
 do

 update_cdt_list (λlist. case (dest_p) of
      None ⇒ list |
      Some p ⇒ (list (p := list_remove (list p) dest_slot)));

    update_cdt_list (λlist. case (src_p) of
      None ⇒ list (
        src_slot := if src_parent then [dest_slot] @ (list src_slot) else list src_slot) |
      Some p ⇒ list (
        src_slot := if src_parent then [dest_slot] @ (list src_slot) else list src_slot,
        p := if (src_parent ∧ p ≠ src_slot) then (list p) else if (src_slot ≠ dest_slot) then
(list_insert_after (list p) src_slot dest_slot) else (dest_slot # (list p))))
 od)"
```

definition empty_slot_ext **where**

```
"empty_slot_ext ≡ (λ slot slot_p.

    update_cdt_list (λlist. case slot_p of None ⇒ list (slot := []) |
      Some p ⇒ if (p = slot) then list(p := list_remove (list p) slot) else list (p := list_replace_list
(list p) slot (list slot), slot := [])))"
```

definition create_cap_ext **where**

```
"create_cap_ext ≡ (λ untyped dest dest_p. do

    update_cdt_list (λlist. case dest_p of
      None ⇒ list |
      Some p ⇒ (list (p := list_remove (list p) dest)));

    update_cdt_list (λlist. list (untyped := [dest] @ (list untyped)))
  od)"
```

definition next_revoke_cap **where**

```
"next_revoke_cap ≡ (λslot ext. the (next_child slot (cdt_list ext)))"
```

definition free_asid_select **where**

```
"free_asid_select ≡ (λ asid_table. fst (hd ((filter (λ (x,y). x ≤ 2 ^ asid_high_bits - 1 ∧ y
= None) (assocs asid_table))))) :: (word8 ⇀ word32) ⇒ word8"
```

definition free_asid_pool_select **where**

```
"free_asid_pool_select ≡ (λ pool base. fst (hd ((filter (λ (x,y). x ≤ 2 ^ asid_low_bits - 1 ∧
ucast x + base ≠ 0 ∧ y = None) (assocs pool))))) :: (10 word ⇀ word32) ⇒ word32 ⇒ 10 word"
```

definition update_work_units **where**

```
  "update_work_units ≡
     modify (λs. s(|work_units_completed := work_units_completed s + 1|))"
```

definition reset_work_units **where**

```
  "reset_work_units ≡
     modify (λs. s(|work_units_completed := 0|))"
```

```
definition work_units_limit_reached where
  "work_units_limit_reached ≡ do
     work_units ← gets work_units_completed;
     return (work_units_limit ≤ work_units)
   od"
```

The lowest virtual address in the kernel window. The kernel reserves the virtual addresses from here up in every virtual address space.

```
definition
  kernel_base :: "vspace_ref" where
  "kernel_base ≡ 0xf0000000"

definition
  idle_thread_ptr :: word32 where
  "idle_thread_ptr = kernel_base + 0x1000"
```

A type class for all instantiations of the abstract specification. In practice, this is restricted to basically allow only two sensible implementations at present: the deterministic abstract specification and the nondeterministic one.

```
class state_ext =
 fixes unwrap_ext :: "'a state ⇒ det_ext state"
 fixes wrap_ext :: "(det_ext ⇒ det_ext) ⇒ ('a ⇒ 'a)"
 fixes wrap_ext_op :: "unit det_ext_monad ⇒ ('a state,unit) nondet_monad"
 fixes wrap_ext_bool :: "bool det_ext_monad ⇒ ('a state,bool) nondet_monad"
 fixes select_switch :: "'a ⇒ bool"
 fixes ext_init :: "'a"

definition detype_ext :: "obj_ref set ⇒ 'z::state_ext ⇒ 'z" where
 "detype_ext S ≡ wrap_ext (λs. s(|ekheap_internal := (λx. if x ∈ S then None else ekheap_internal
s x)|))"

instantiation  det_ext_ext :: (type) state_ext
begin

definition "unwrap_ext_det_ext_ext == (λx. x) :: det_ext state ⇒ det_ext state"

definition "wrap_ext_det_ext_ext == (λx. x) ::
  (det_ext ⇒ det_ext) ⇒ det_ext ⇒ det_ext"

definition "wrap_ext_op_det_ext_ext == (λx. x) ::
  (det_ext state ⇒ ((unit × det_ext state) set) × bool)
  ⇒ det_ext state  ⇒ ((unit × det_ext state) set) × bool"

definition "wrap_ext_bool_det_ext_ext == (λx. x) ::
  (det_ext state ⇒ ((bool × det_ext state) set) × bool)
  ⇒ det_ext state ⇒ ((bool × det_ext state) set) × bool"

definition "select_switch_det_ext_ext == (λ_. True)  :: det_ext⇒ bool"


definition "ext_init_det_ext_ext ≡
     (|work_units_completed_internal = 0,
      scheduler_action_internal = resume_cur_thread,
      ekheap_internal = Map.empty (idle_thread_ptr ↦ default_etcb),
      domain_list_internal = [],
      domain_index_internal = 0,
      cur_domain_internal = 0,
```

```
      domain_time_internal = 15,
      ready_queues_internal = const (const []),
      cdt_list_internal = const []⦈ :: det_ext"
```

instance

end

19.2 Nondeterministic Abstract Specification

The nondeterministic abstract specification instantiates the extended state with the unit type – i.e. it doesn't have any meaningful extended state.

instantiation `unit :: state_ext`
begin

definition `"unwrap_ext_unit == (λ_. undefined) :: unit state ⇒ det_ext state"`

definition `"wrap_ext_unit == (λf s. ()) :: (det_ext ⇒ det_ext) ⇒ unit ⇒ unit"`

definition `"wrap_ext_op_unit == (λm. return ()) ::`
```
  (det_ext state ⇒ ((unit × det_ext state) set) × bool) ⇒ unit state ⇒ ((unit × unit state)
set) × bool"
```

definition `"wrap_ext_bool_unit == (λm. select UNIV) ::`
```
  (det_ext state ⇒ ((bool × det_ext state ) set) × bool) ⇒ unit state ⇒ ((bool × unit state)
set) × bool"
```

definition `"select_switch_unit == (λs. False) :: unit ⇒ bool"`

definition `"ext_init_unit ≡ () :: unit"`

instance

end

Run an extended operation over the extended state without modifying it and use the return value to choose between two computations to run.

lemmas `ext_init_def = ext_init_det_ext_ext_def ext_init_unit_def`

definition `OR_choice :: "bool det_ext_monad ⇒ ('z::state_ext state,'a) nondet_monad ⇒ ('z state,'a) nondet_monad ⇒ ('z state,'a) nondet_monad"` **where**
```
"OR_choice c f g ≡
  do
    ex ← get;
    (rv,_) ← select_f (mk_ef ((wrap_ext_bool c) ex));
    if rv then f else g
  od"
```

definition `OR_choiceE :: "bool det_ext_monad ⇒ ('z::state_ext state,'e + 'a) nondet_monad ⇒ ('z state,'e + 'a) nondet_monad ⇒ ('z state,'e + 'a) nondet_monad"` **where**
```
"OR_choiceE c f g ≡
  doE
    ex ← liftE get;
    (rv,_) ← liftE $ select_f (mk_ef ((wrap_ext_bool c) ex));
```

```
    if rv then f else g
  odE"
```

Run an extended operation over the extended state to update the extended state, ignoring any return value that the extended operation might yield.

```
definition do_extended_op :: "unit det_ext_monad ⇒ ('z::state_ext state,unit) nondet_monad" where
 "do_extended_op eop ≡ do
                         ex ← get;
                         (_,es') ← select_f (mk_ef ((wrap_ext_op eop) ex));
                         modify (λ state. state⦇exst := (exst es')⦈)
                        od"
```

Use the extended state to choose a value from a bounding set `S` when `select_switch` is true. Otherwise just select from `S`.

```
definition select_ext :: "(det_ext state ⇒ 'd) ⇒ ('d set) ⇒ ('a::state_ext state,'d) nondet_monad"
where
  "select_ext a S ≡ do
                      s ← get;
                      x ← if (select_switch (exst s)) then (return (a (unwrap_ext s)))
                          else (select S);
                      assert (x ∈ S);
                      return x
                    od"


definition valid_list_2 :: "cdt_list ⇒ cdt ⇒ bool" where
  "valid_list_2 t m ≡ (∀p. set (t p) = {c. m c = Some p}) ∧ (∀p. distinct (t p))"

abbreviation valid_list :: "det_ext state ⇒ bool" where
  "valid_list s ≡ valid_list_2 (cdt_list s) (cdt s)"

end
```

20 Basic Kernel and Exception Monads

```
theory Exceptions_A
imports Deterministic_A
begin
```

This theory contains abbreviations for the monadic types used in the specification and a number of lifting functions between them.

The basic kernel monad without faults, interrupts, or errors.

```
type_synonym ('a,'z) s_monad = "('z state, 'a) nondet_monad"
```

The fault monad: may throw a `fault` exception which will usually be reported to the current thread's fault handler.

```
type_synonym ('a,'z) f_monad = "(fault + 'a,'z) s_monad"

term "a::(unit,'a) s_monad"
```

The error monad: may throw a `syscall_error` exception which will usually be reported to the current thread as system call result.

```
type_synonym ('a,'z) se_monad = "(syscall_error + 'a,'z) s_monad"
```

The lookup failure monad: may throw a `lookup_failure` exception. Depending on context it may either be reported directly to the current thread or to its fault handler.

```
type_synonym ('a,'z) lf_monad = "(lookup_failure + 'a,'z) s_monad"
```

The preemption monad. May throw an interrupt exception.

```
type_synonym ('a,'z) p_monad = "(interrupt + 'a,'z) s_monad"
```

Printing abbreviations for the above types.

```
translations
  (type) "'a s_monad" <= (type) "state ⇒ (('a × state) ⇒ bool) × bool"
  (type) "'a f_monad" <= (type) "(fault + 'a) s_monad"
  (type) "'a se_monad" <= (type) "(syscall_error + 'a) s_monad"
  (type) "'a lf_monad" <= (type) "(lookup_failure + 'a) s_monad"
  (type) "'a p_monad" <=(type) "(interrupt + 'a) s_monad"
```

Perform non-preemptible operations within preemptible blocks.

```
definition
  without_preemption :: "('a,'z::state_ext) s_monad ⇒ ('a,'z::state_ext) p_monad"
where without_preemption_def[simp]:
 "without_preemption ≡ liftE"
```

Allow preemption at this point.

```
definition
  preemption_point :: "(unit,'z::state_ext) p_monad" where
 "preemption_point ≡ doE liftE $ do_extended_op update_work_units;
                         OR_choiceE (work_units_limit_reached)
                           (doE liftE $ do_extended_op reset_work_units;
                                irq_opt ← liftE $ do_machine_op getActiveIRQ;
```

```
                                   option_case (returnOk ()) (throwError ∘ Interrupted) irq_opt
                              odE) (returnOk ())
                        odE"
```

Lift one kind of exception monad into another by converting the error into various other kinds of error or return value.

```
definition
  cap_fault_on_failure :: "word32 ⇒ bool ⇒ ('a,'z::state_ext) lf_monad ⇒ ('a,'z::state_ext)
f_monad" where
 "cap_fault_on_failure cptr rp m ≡ handleE' m (throwError ∘ CapFault cptr rp)"

definition
  lookup_error_on_failure ::  "bool ⇒ ('a,'z::state_ext) lf_monad ⇒ ('a,'z::state_ext) se_monad"
where
 "lookup_error_on_failure s m ≡ handleE' m (throwError ∘ FailedLookup s)"

definition
  null_cap_on_failure :: "(cap,'z::state_ext) lf_monad ⇒ (cap,'z::state_ext) s_monad" where
 "null_cap_on_failure ≡ liftM (sum_case (λx. NullCap) id)"

definition
  unify_failure :: "('f + 'a,'z::state_ext) s_monad ⇒ (unit + 'a,'z::state_ext) s_monad" where
 "unify_failure m ≡ handleE' m (λx. throwError ())"

definition
  empty_on_failure :: "('f + 'a list,'z::state_ext) s_monad ⇒ ('a list,'z::state_ext) s_monad"
where
 "empty_on_failure m ≡ m <catch> (λx. return [])"

definition
  const_on_failure :: "'a ⇒ ('f + 'a,'z::state_ext) s_monad ⇒ ('a,'z::state_ext) s_monad" where
 "const_on_failure c m ≡ m <catch> (λx. return c)"
```

end

21 Accessing the Kernel Heap

theory KHeap_A
imports Exceptions_A
begin

This theory gives auxiliary getter and setter methods for kernel objects.

21.1 General Object Access

```
definition
  get_object :: "obj_ref ⇒ (kernel_object,'z::state_ext) s_monad"
where
  "get_object ptr ≡ do
     kh ← gets kheap;
     assert (kh ptr ≠ None);
     return $ the $ kh ptr
   od"
```

```
definition
  set_object :: "obj_ref ⇒ kernel_object ⇒ (unit,'z::state_ext) s_monad"
where
  "set_object ptr obj ≡ do
     s ← get;
     kh ← return $ (kheap s)(ptr := Some obj);
     put (s ⦇ kheap := kh ⦈)
   od"
```

21.2 TCBs

```
definition
  get_tcb :: "obj_ref ⇒ 'z::state_ext state ⇒ tcb option"
where
  "get_tcb tcb_ref state ≡
   case kheap state tcb_ref of
      None      ⇒ None
    | Some kobj ⇒ (case kobj of
        TCB tcb ⇒ Some tcb
      | _       ⇒ None)"
```

```
definition
  thread_get :: "(tcb ⇒ 'a) ⇒ obj_ref ⇒ ('a,'z::state_ext) s_monad"
where
  "thread_get f tptr ≡ do
     tcb ← gets_the $ get_tcb tptr;
     return $ f tcb
   od"
```

```
definition
  thread_set :: "(tcb ⇒ tcb) ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad"
where
```

```
  "thread_set f tptr ≡ do
     tcb ← gets_the $ get_tcb tptr;
     set_object tptr $ TCB $ f tcb
   od"
```

definition
```
  get_thread_state :: "obj_ref ⇒ (thread_state,'z::state_ext) s_monad"
```
where
```
  "get_thread_state ref ≡ thread_get tcb_state ref"
```

definition `set_thread_state_ext :: "obj_ref ⇒ unit det_ext_monad"` **where**
```
  "set_thread_state_ext t ≡ do
     ts ← get_thread_state t;
     cur ← gets cur_thread;
     action ← gets scheduler_action;
     when (¬ (runnable ts) ∧ cur = t ∧ action = resume_cur_thread) (set_scheduler_action choose_new_thread)
   od"
```

definition
```
  set_thread_state :: "obj_ref ⇒ thread_state ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "set_thread_state ref ts ≡ do
     tcb ← gets_the $ get_tcb ref;
     set_object ref (TCB (tcb ⦇ tcb_state := ts ⦈));
     do_extended_op (set_thread_state_ext ref)
   od"
```

definition
```
  set_priority :: "obj_ref ⇒ priority ⇒ unit det_ext_monad" where
  "set_priority tptr prio ≡ do
     tcb_sched_action tcb_sched_dequeue tptr;
     thread_set_priority tptr prio;
     ts ← get_thread_state tptr;
     when (runnable ts) $ tcb_sched_action tcb_sched_enqueue tptr;
     cur ← gets cur_thread;
     when (tptr = cur) reschedule_required
   od"
```

21.3 Synchronous and Asyncronous Endpoints

definition
```
  get_endpoint :: "obj_ref ⇒ (endpoint,'z::state_ext) s_monad"
```
where
```
  "get_endpoint ptr ≡ do
     kobj ← get_object ptr;
     (case kobj of Endpoint e ⇒ return e
                 | _ ⇒ fail)
   od"
```

definition
```
  set_endpoint :: "obj_ref ⇒ endpoint ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "set_endpoint ptr ep ≡ do
     obj ← get_object ptr;
     assert (case obj of Endpoint ep ⇒ True | _ ⇒ False);
     set_object ptr (Endpoint ep)
   od"
```

definition
```
  get_async_ep :: "obj_ref ⇒ (async_ep,'z::state_ext) s_monad"
```
where
```
  "get_async_ep ptr ≡ do
     kobj ← get_object ptr;
     case kobj of AsyncEndpoint e ⇒ return e
                 | _ ⇒ fail
   od"
```

definition
```
  set_async_ep :: "obj_ref ⇒ async_ep ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "set_async_ep ptr aep ≡ do
     obj ← get_object ptr;
     assert (case obj of AsyncEndpoint aep ⇒ True | _ ⇒ False);
     set_object ptr (AsyncEndpoint aep)
   od"
```

21.4 IRQ State and Slot

definition
```
  get_irq_state :: "irq ⇒ (irq_state,'z::state_ext) s_monad" where
 "get_irq_state irq ≡ gets (λs. interrupt_states s irq)"
```

definition
```
  set_irq_state :: "irq_state ⇒ irq ⇒ (unit,'z::state_ext) s_monad" where
 "set_irq_state state irq ≡ do
    modify (λs. s (| interrupt_states := (interrupt_states s) (irq := state)|));
    do_machine_op $ maskInterrupt (state = IRQInactive) irq
  od"
```

definition
```
  get_irq_slot :: "irq ⇒ (cslot_ptr,'z::state_ext) s_monad" where
 "get_irq_slot irq ≡ gets (λst. (interrupt_irq_node st irq, []))"
```

21.5 User Context

Changes user context of specified thread by running specified user monad.

definition
```
  as_user :: "obj_ref ⇒ 'a user_monad ⇒ ('a,'z::state_ext) s_monad"
```
where
```
  "as_user tptr f ≡ do
    tcb ← gets_the $ get_tcb tptr;
    uc ← return $ tcb_context tcb;
    (a, uc') ← select_f $ f uc;
    new_tcb ← return $ tcb (| tcb_context := uc' |);
    set_object tptr (TCB new_tcb);
    return a
  od"
```

end

22 Accessing CSpace

theory CSpaceAcc_A
imports KHeap_A
begin

This theory contains basic definitions for manipulating capabilities and CDTs.

22.1 Capability access

Recall that a capability may reside in either a CNode, or inside a TCB; the following definitions allow the kernel model to retrieve and update capabilities in a uniform fashion.

```
definition
  get_cap :: "cslot_ptr ⇒ (cap,'z::state_ext) s_monad"
where
  "get_cap ≡ λ(oref, cref). do
     obj ← get_object oref;
     caps ← case obj of
             CNode sz cnode ⇒ do
                                 assert (well_formed_cnode_n sz cnode);
                                 return cnode
                               od
           | TCB tcb     ⇒ return (tcb_cnode_map tcb)
           | _ ⇒ fail;
     assert_opt (caps cref)
   od"

definition
  set_cap :: "cap ⇒ cslot_ptr ⇒ (unit,'z::state_ext) s_monad"
where
  "set_cap cap ≡ λ(oref, cref). do
     obj ← get_object oref;
     obj' ← case obj of
               CNode sz cn ⇒ if cref ∈ dom cn ∧ well_formed_cnode_n sz cn
                                 then return $ CNode sz $ cn (cref ↦ cap)
                                 else fail
             | TCB tcb ⇒
                   if cref = tcb_cnode_index 0 then
                       return $ TCB $ tcb (| tcb_ctable := cap |)
                   else if cref = tcb_cnode_index 1 then
                       return $ TCB $ tcb (| tcb_vtable := cap |)
                   else if cref = tcb_cnode_index 2 then
                       return $ TCB $ tcb (| tcb_reply := cap |)
                   else if cref = tcb_cnode_index 3 then
                       return $ TCB $ tcb (| tcb_caller := cap |)
                   else if cref = tcb_cnode_index 4 then
                       return $ TCB $ tcb (| tcb_ipcframe := cap |)
                   else
                       fail
             | _ ⇒ fail;
     set_object oref obj'
```

```
  od"
```

Ensure a capability slot is empty.

```
definition
  ensure_empty :: "cslot_ptr ⇒ (unit,'z::state_ext) se_monad"
where
  "ensure_empty slot ≡ doE
    cap ← liftE $ get_cap slot;
    whenE (cap ≠ NullCap) (throwError DeleteFirst)
  odE"
```

22.2 Accessing the capability derivation tree

Set the capability derivation tree.

```
definition
  set_cdt :: "cdt ⇒ (unit,'z::state_ext) s_monad"
  where
  "set_cdt t ≡ do
    s ← get;
    put $ s⦇ cdt := t ⦈
  od"
```

Update the capability derivation tree.

```
definition
  update_cdt :: "(cdt ⇒ cdt) ⇒ (unit,'z::state_ext) s_monad"
where
  "update_cdt f ≡ do
     t ← gets cdt;
     set_cdt (f t)
  od"
```

Set the original flag for a given cap slot.

```
definition
  set_original :: "cslot_ptr ⇒ bool ⇒ (unit,'z::state_ext) s_monad"
where
  "set_original slot v ≡ do
     r ← gets is_original_cap;
     modify (λs. s ⦇ is_original_cap := r (slot := v) ⦈)
  od"
```

Definitions and syntax for predicates on capability derivation.

```
definition
  is_cdt_parent :: "cdt ⇒ cslot_ptr ⇒ cslot_ptr ⇒ bool" where
  "is_cdt_parent t p c ≡ t c = Some p"

definition
  cdt_parent_rel :: "cdt ⇒ (cslot_ptr × cslot_ptr) set" where
  "cdt_parent_rel t ≡ {(p,c). is_cdt_parent t p c}"

abbreviation
  parent_of :: "cdt ⇒ cslot_ptr ⇒ cslot_ptr ⇒ bool"
  ("_ ⊢ _ cdt'_parent'_of _" [60,0,60] 61)
where
  "t ⊢ p cdt_parent_of c ≡ (p,c) ∈ cdt_parent_rel t"
```

abbreviation
```
parent_of_trancl :: "cdt ⇒ cslot_ptr ⇒ cslot_ptr ⇒ bool"
("_ ⊢ _ cdt'_parent'_of⁺ _" [60,0,60] 61)
```
where
```
"t ⊢ x cdt_parent_of⁺ y ≡ (x, y) ∈ (cdt_parent_rel t)⁺"
```

abbreviation
```
parent_of_rtrancl :: "cdt ⇒ cslot_ptr ⇒ cslot_ptr ⇒ bool"
("_ ⊢ _ cdt'_parent'_of* _" [60,0,60] 61)
```
where
```
"t ⊢ x cdt_parent_of* y ≡ (x, y) ∈ (cdt_parent_rel t)*"
```

notation
```
parent_of ("_ ⊨ _ ⇝ _" [60,0,60] 60)
```
and
```
parent_of_trancl ("_ ⊨ _ → _" [60,0,60] 60)
```

The set of descendants of a particular slot in the CDT.

definition
```
descendants_of :: "cslot_ptr ⇒ cdt ⇒ cslot_ptr set" where
"descendants_of p t ≡ {q. (p,q) ∈ (cdt_parent_rel t)⁺}"
```

end

23 Accessing the ARM VSpace

```
theory ArchVSpaceAcc_A
imports KHeap_A
begin
```

This part of the specification is fairly concrete as the machine architecture is visible to the user in seL4 and therefore needs to be described. The abstraction compared to the implementation is in the data types for kernel objects. The interface which is rich in machine details remains the same.

23.1 Encodings

The lowest virtual address in the kernel window. The kernel reserves the virtual addresses from here up in every virtual address space.

Convert a set of rights into binary form.

```
definition
  word_from_vm_rights :: "vm_rights ⇒ word32" where
  "word_from_vm_rights R ≡
  if vm_read_write ⊆ R then 3
  else if vm_read_only ⊆ R then 2
  else 1"
```

Encode a page directory entry into the equivalent entry that the page table walker implemented in ARM hardware would parse.

```
definition
  word_from_pde :: "pde ⇒ machine_word" where
  "word_from_pde pde ≡ case pde of
    InvalidPDE ⇒ 0
  | PageTablePDE table attrib domain ⇒ 1 ||
    table && 0xfffffc00 ||
    (if ParityEnabled ∈ attrib then 1 << 9 else 0) ||
    ((domain && 0xf) << 5)
  | SectionPDE frame attrib domain rights ⇒  2 ||
    frame && 0xfff00000 ||
    (if ParityEnabled ∈ attrib then (1 << 9) else 0) ||
    (if PageCacheable ∈ attrib then (1 << 2) || (1 << 3) else 0) ||
    ((domain && 0xf) << 5) ||
    (if Global ∈ attrib then 0 else (1 << 17)) ||
    (word_from_vm_rights rights << 10)
  | SuperSectionPDE frame attrib rights ⇒  2 ||
    (1 << 18) ||
    (frame && 0xff000000) ||
    (if ParityEnabled ∈ attrib then 1 << 9 else 0) ||
    (if PageCacheable ∈ attrib then (1 << 2) || (1 << 3) else 0) ||
    (if Global ∈ attrib then 0 else (1 << 17)) ||
    (word_from_vm_rights rights << 10)"
```

Encode a page table entry into the equivalent entry that the page table walker implemented in ARM hardware would parse.

definition
```
  word_from_pte :: "pte ⇒ machine_word" where
  "word_from_pte pte ≡ case pte of
    InvalidPTE ⇒ 0
  | LargePagePTE frame attrib rights ⇒    1 ||
    (frame && 0xffff0000) ||
    (if PageCacheable ∈ attrib then (1 << 2) || (1 << 3) else 0) ||
    (word_from_vm_rights rights * 85 << 4)
  | (SmallPagePTE frame attrib rights) ⇒    2 ||
    (frame && 0xfffff000) ||
    (if PageCacheable ∈ attrib then (1 << 2) || (1 << 3) else 0) ||
    (word_from_vm_rights rights * 85 << 4)"
```

The high bits of a virtual ASID.

definition
```
  asid_high_bits_of :: "asid ⇒ word8" where
  "asid_high_bits_of asid ≡ ucast (asid >> asid_low_bits)"
```

23.2 Kernel Heap Accessors

Manipulate ASID pools, page directories and page tables in the kernel heap.

definition
```
  get_asid_pool :: "obj_ref ⇒ (10 word ⇀ obj_ref,'z::state_ext) s_monad" where
  "get_asid_pool ptr ≡ do
     kobj ← get_object ptr;
     (case kobj of ArchObj (ASIDPool pool) ⇒ return pool
                 | _ ⇒ fail)
   od"
```

definition
```
  set_asid_pool :: "obj_ref ⇒ (10 word ⇀ obj_ref) ⇒ (unit,'z::state_ext) s_monad" where
 "set_asid_pool ptr pool ≡ do
    v ← get_object ptr;
    assert (case v of ArchObj (arch_kernel_obj.ASIDPool p) ⇒ True | _ ⇒ False);
    set_object ptr (ArchObj (arch_kernel_obj.ASIDPool pool))
  od"
```

definition
```
  get_pd :: "obj_ref ⇒ (12 word ⇒ pde,'z::state_ext) s_monad" where
  "get_pd ptr ≡ do
     kobj ← get_object ptr;
     (case kobj of ArchObj (PageDirectory pd) ⇒ return pd
                 | _ ⇒ fail)
   od"
```

definition
```
  set_pd :: "obj_ref ⇒ (12 word ⇒ pde) ⇒ (unit,'z::state_ext) s_monad" where
  "set_pd ptr pd ≡ do
     kobj ← get_object ptr;
     assert (case kobj of ArchObj (PageDirectory pd) ⇒ True | _ ⇒ False);
     set_object ptr (ArchObj (PageDirectory pd))
   od"
```

The following function takes a pointer to a PDE in kernel memory and returns the actual PDE.

definition
```
  get_pde :: "obj_ref ⇒ (pde,'z::state_ext) s_monad" where
```

```
  "get_pde ptr ≡ do
     base ← return (ptr && ~~mask pd_bits);
     offset ← return ((ptr && mask pd_bits) >> 2);
     pd ← get_pd base;
     return $ pd (ucast offset)
   od"
```

definition

```
  store_pde :: "obj_ref ⇒ pde ⇒ (unit,'z::state_ext) s_monad" where
  "store_pde p pde ≡ do
    base ← return (p && ~~mask pd_bits);
    offset ← return ((p && mask pd_bits) >> 2);
    pd ← get_pd base;
    pd' ← return $ pd (ucast offset := pde);
    set_pd base pd'
  od"
```

definition

```
  get_pt :: "obj_ref ⇒ (word8 ⇒ pte,'z::state_ext) s_monad" where
  "get_pt ptr ≡ do
     kobj ← get_object ptr;
     (case kobj of ArchObj (PageTable pt) ⇒ return pt
                 | _ ⇒ fail)
   od"
```

definition

```
  set_pt :: "obj_ref ⇒ (word8 ⇒ pte) ⇒ (unit,'z::state_ext) s_monad" where
  "set_pt ptr pt ≡ do
     kobj ← get_object ptr;
     assert (case kobj of ArchObj (PageTable _) ⇒ True | _ ⇒ False);
     set_object ptr (ArchObj (PageTable pt))
   od"
```

The following function takes a pointer to a PTE in kernel memory and returns the actual PTE.

definition

```
  get_pte :: "obj_ref ⇒ (pte,'z::state_ext) s_monad" where
  "get_pte ptr ≡ do
     base ← return (ptr && ~~mask pt_bits);
     offset ← return ((ptr && mask pt_bits) >> 2);
     pt ← get_pt base;
     return $ pt (ucast offset)
   od"
```

definition

```
  store_pte :: "obj_ref ⇒ pte ⇒ (unit,'z::state_ext) s_monad" where
  "store_pte p pte ≡ do
    base ← return (p && ~~mask pt_bits);
    offset ← return ((p && mask pt_bits) >> 2);
    pt ← get_pt base;
    pt' ← return $ pt (ucast offset := pte);
    set_pt base pt'
  od"
```

23.3 Basic Operations

The kernel window is mapped into every virtual address space from the `kernel_base` pointer upwards. This function copies the mappings which create the kernel window into a new page directory object.

definition

```
copy_global_mappings :: "obj_ref ⇒ (unit,'z::state_ext) s_monad" where
"copy_global_mappings new_pd ≡ do
    global_pd ← gets (arm_global_pd ∘ arch_state);
    pde_bits ← return 2;
    pd_size ← return (1 << (pd_bits - pde_bits));
    mapM_x (λindex. do
        offset ← return (index << pde_bits);
        pde ← get_pde (global_pd + offset);
        store_pde (new_pd + offset) pde
    od) [kernel_base >> 20  .e.  pd_size - 1]
od"
```

Walk the page directories and tables in software.

The following function takes a page-directory reference as well as a virtual address and then computes a pointer to the PDE in kernel memory

definition

```
lookup_pd_slot :: "word32 ⇒ vspace_ref ⇒ word32" where
"lookup_pd_slot pd vptr ≡
    let pd_index = vptr >> 20
    in pd + (pd_index << 2)"
```

The following function takes a page-directory reference as well as a virtual address and then computes a pointer to the PTE in kernel memory. Note that the function fails if the virtual address is mapped on a section or super section.

definition

```
lookup_pt_slot :: "word32 ⇒ vspace_ref ⇒ (word32,'z::state_ext) lf_monad" where
"lookup_pt_slot pd vptr ≡ doE
    pd_slot ← returnOk (lookup_pd_slot pd vptr);
    pde ← liftE $ get_pde pd_slot;
    (case pde of
          PageTablePDE ptab _ _ ⇒   (doE
            pt ← returnOk (ptrFromPAddr ptab);
            pt_index ← returnOk ((vptr >> 12) && 0xff);
            pt_slot ← returnOk (pt + (pt_index << 2));
            returnOk pt_slot
          odE)
        | _ ⇒ throwError $ MissingCapability 20)
odE"
```

A non-failing version of `lookup_pt_slot` when the pd is already known

definition

```
  lookup_pt_slot_no_fail :: "word32 ⇒ vspace_ref ⇒ word32"
where
  "lookup_pt_slot_no_fail pt vptr ≡
     let pt_index = ((vptr >> 12) && 0xff)
     in pt + (pt_index << 2)"
```

end

24 ARM Object Invocations

```
theory ArchInvocation_A
imports Structures_A
begin
```

These datatypes encode the arguments to the various possible ARM-specific system calls. Accessors are defined for various fields for convenience elsewhere.

```
datatype flush_type = Clean | Invalidate | CleanInvalidate | Unify

datatype page_directory_invocation =
    PageDirectoryFlush flush_type vspace_ref vspace_ref word32 obj_ref asid
  | PageDirectoryNothing

primrec
  pd_flush_type :: "page_directory_invocation ⇒ flush_type"
where
  "pd_flush_type (PageDirectoryFlush typ start end pstart pd asid) = typ"

primrec
  pd_flush_start :: "page_directory_invocation ⇒ vspace_ref"
where
  "pd_flush_start (PageDirectoryFlush typ start end pstart pd asid) = start"

primrec
  pd_flush_end :: "page_directory_invocation ⇒ vspace_ref"
where
  "pd_flush_end (PageDirectoryFlush typ start end pstart pd asid) = end"

primrec
  pd_flush_pstart :: "page_directory_invocation ⇒ word32"
where
  "pd_flush_pstart (PageDirectoryFlush typ start end pstart pd asid) = pstart"

primrec
  pd_flush_pd :: "page_directory_invocation ⇒ obj_ref"
where
  "pd_flush_pd (PageDirectoryFlush typ start end pstart pd asid) = pd"

primrec
  pd_flush_asid :: "page_directory_invocation ⇒ asid"
where
  "pd_flush_asid (PageDirectoryFlush typ start end pstart pd asid) = asid"

datatype page_table_invocation =
    PageTableMap cap cslot_ptr pde obj_ref
  | PageTableUnmap cap cslot_ptr

datatype asid_control_invocation =
    MakePool obj_ref cslot_ptr cslot_ptr asid

datatype asid_pool_invocation =
    Assign asid obj_ref cslot_ptr
```

```
datatype page_invocation
     = PageMap
         asid
         cap
         cslot_ptr
         "pte × (obj_ref list) + pde × (obj_ref list)"
     | PageRemap
         asid
         "pte × (obj_ref list) + pde × (obj_ref list)"
     | PageUnmap
         arch_cap
         cslot_ptr
     | PageFlush
         flush_type
         vspace_ref
         vspace_ref
         word32
         obj_ref
         asid
     | PageGetAddr
         obj_ref

primrec
  page_map_cap :: "page_invocation ⇒ cap"
where
  "page_map_cap (PageMap a c p x) = c"

primrec
  page_map_asid :: "page_invocation ⇒ asid"
where
  "page_map_asid (PageMap a c p x) = a"
primrec
  page_map_ct_slot :: "page_invocation ⇒ cslot_ptr"
where
  "page_map_ct_slot (PageMap a c p x) = p"
primrec
  page_map_entries :: "page_invocation ⇒ pte × (obj_ref list) + pde × (obj_ref list)"
where
  "page_map_entries (PageMap a c p x) = x"

primrec
  page_remap_entries :: "page_invocation ⇒ pte × (obj_ref list) + pde × (obj_ref list)"
where
  "page_remap_entries (PageRemap a x) = x"

primrec
  page_remap_asid :: "page_invocation ⇒ asid"
where
  "page_remap_asid (PageRemap a x) = a"

primrec
  page_unmap_cap :: "page_invocation ⇒ arch_cap"
where
  "page_unmap_cap (PageUnmap c p) = c"

primrec
  page_unmap_cap_slot :: "page_invocation ⇒ cslot_ptr"
```

```
where
  "page_unmap_cap_slot (PageUnmap c p) = p"

primrec
  page_flush_pd :: "page_invocation ⇒ obj_ref"
where
  "page_flush_pd (PageFlush typ start end pstart pd asid) = pd"

primrec
  page_flush_asid :: "page_invocation ⇒ asid"
where
  "page_flush_asid (PageFlush typ start end pstart pd asid) = asid"

primrec
  page_flush_type :: "page_invocation ⇒ flush_type"
where
  "page_flush_type (PageFlush typ start end pstart pd asid) = typ"

primrec
  page_flush_start :: "page_invocation ⇒ vspace_ref"
where
  "page_flush_start (PageFlush typ start end pstart pd asid) = start"

primrec
  page_flush_end :: "page_invocation ⇒ vspace_ref"
where
  "page_flush_end (PageFlush typ start end pstart pd asid) = end"

primrec
  page_flush_pstart :: "page_invocation ⇒ word32"
where
  "page_flush_pstart (PageFlush typ start end pstart pd asid) = pstart"

primrec
  page_get_paddr :: "page_invocation ⇒ obj_ref"
where
  "page_get_paddr (PageGetAddr ptr) = ptr"

datatype arch_invocation
     = InvokePageTable page_table_invocation
     | InvokePageDirectory page_directory_invocation
     | InvokePage page_invocation
     | InvokeASIDControl asid_control_invocation
     | InvokeASIDPool asid_pool_invocation


typedecl arch_interrupt_control

end
```

25 Kernel Object Invocations

```
theory Invocations_A
imports ArchInvocation_A
begin
```

These datatypes encode the arguments to the available system calls.

```
datatype cnode_invocation =
    InsertCall cap cslot_ptr cslot_ptr
  | MoveCall cap cslot_ptr cslot_ptr
  | RevokeCall cslot_ptr
  | DeleteCall cslot_ptr
  | RotateCall cap cap cslot_ptr cslot_ptr cslot_ptr
  | SaveCall cslot_ptr
  | RecycleCall cslot_ptr

datatype untyped_invocation =
    Retype cslot_ptr obj_ref obj_ref apiobject_type nat "cslot_ptr list"

datatype arm_copy_register_sets =
    ARMNoExtraRegisters

datatype tcb_invocation =
    WriteRegisters word32 bool "word32 list" arm_copy_register_sets
  | ReadRegisters word32 bool word32 arm_copy_register_sets
  | CopyRegisters word32 word32 bool bool bool bool arm_copy_register_sets
  | ThreadControl word32 cslot_ptr "cap_ref option" "word8 option"
                  "(cap * cslot_ptr) option" "(cap * cslot_ptr) option"
                  "(vspace_ref * (cap * cslot_ptr) option) option"
  | Suspend "word32"
  | Resume "word32"

datatype irq_control_invocation =
    IRQControl irq cslot_ptr cslot_ptr
  | InterruptControl arch_interrupt_control

datatype irq_handler_invocation =
    ACKIrq irq
  | SetIRQHandler irq cap cslot_ptr
  | ClearIRQHandler irq

datatype invocation =
    InvokeUntyped untyped_invocation
  | InvokeEndpoint obj_ref word32 bool
  | InvokeAsyncEndpoint obj_ref word32 word32
  | InvokeReply obj_ref cslot_ptr
  | InvokeTCB tcb_invocation
  | InvokeDomain obj_ref word8
  | InvokeCNode cnode_invocation
  | InvokeIRQControl irq_control_invocation
  | InvokeIRQHandler irq_handler_invocation
  | InvokeArchObject arch_invocation
```

end

26 Retyping and Untyped Invocations

```
theory Retype_A
imports
  CSpaceAcc_A
  ArchVSpaceAcc_A
  Invocations_A
begin
```

26.1 Creating Caps

The original capability created when an object of a given type is created with a particular address and size.

```
primrec
  default_cap :: "apiobject_type  ⇒ obj_ref ⇒ nat ⇒ cap"
where
  "default_cap CapTableObject oref s = CNodeCap oref s []"
| "default_cap Untyped oref s = UntypedCap oref s 0"
| "default_cap TCBObject oref s = ThreadCap oref"
| "default_cap EndpointObject oref s = EndpointCap oref 0 UNIV"
| "default_cap AsyncEndpointObject oref s =
     AsyncEndpointCap oref 0 {AllowRead, AllowWrite}"
| "default_cap (ArchObject aobj) oref s = ArchObjectCap (arch_default_cap aobj oref s)"
```

Create and install a new capability to a newly created object.

```
definition
  create_cap ::
  "apiobject_type ⇒ nat ⇒ cslot_ptr ⇒ cslot_ptr × obj_ref ⇒ (unit,'z::state_ext) s_monad"
where
  "create_cap type bits untyped ≡ λ(dest,oref). do
    dest_p ← gets (λs. cdt s dest);
    cdt ← gets cdt;
    set_cdt (cdt (dest ↦ untyped));
    do_extended_op (create_cap_ext untyped dest dest_p);
    set_original dest True;
    set_cap (default_cap type oref bits) dest
   od"
```

26.2 Creating Objects

Properties of an empty CNode object.

```
definition
  empty_cnode :: "nat ⇒ cnode_contents" where
  "empty_cnode bits ≡ λx. if length x = bits then Some NullCap else None"
```

The initial state objects of various types are in when created.

```
definition
  default_object :: "apiobject_type ⇒ nat ⇒ kernel_object" where
  "default_object api n ≡ case api of
```

```
          Untyped ⇒ undefined
        | CapTableObject ⇒ CNode n (empty_cnode n)
        | TCBObject ⇒ TCB default_tcb
        | EndpointObject ⇒ Endpoint default_ep
        | AsyncEndpointObject ⇒ AsyncEndpoint default_async_ep
        | ArchObject aobj ⇒ ArchObj (default_arch_object aobj n)"
```

The size in bits of the objects that will be created when a given type and size is requested.

definition

```
  obj_bits_api :: "apiobject_type ⇒ nat ⇒ nat" where
  "obj_bits_api type obj_size_bits ≡ case type of
          Untyped ⇒ obj_size_bits
        | CapTableObject ⇒ obj_size_bits + slot_bits
        | TCBObject ⇒ obj_bits (TCB default_tcb)
        | EndpointObject ⇒ obj_bits (Endpoint undefined)
        | AsyncEndpointObject ⇒ obj_bits (AsyncEndpoint undefined)
        | ArchObject aobj ⇒ obj_bits $ ArchObj $ default_arch_object aobj obj_size_bits"
```

26.3 Main Retype Implementation

Create `numObjects` objects, starting from `obj_ref`, return of list pointers to them. For some types, each returned pointer points to a group of objects.

definition

```
  retype_region :: "obj_ref ⇒ nat ⇒ nat ⇒ apiobject_type ⇒ (obj_ref list,'z::state_ext) s_monad"
where
  "retype_region ptr numObjects o_bits type ≡ do
    obj_size ← return $ 2 ^ obj_bits_api type o_bits;
    ptrs ← return $ map (λp. ptr_add ptr (p * obj_size)) [0..< numObjects];
    when (type ≠ Untyped) (do
      kh ← gets kheap;
      kh' ← return $ foldr (λp kh. kh(p ↦ default_object type o_bits)) ptrs kh;
      do_extended_op (retype_region_ext ptrs type);
      modify $ kheap_update (K kh')
    od);
    return $ ptrs
  od"
```

26.4 Invoking Untyped Capabilities

Remove objects from a region of the heap.

definition

```
  detype :: "(obj_ref set) ⇒ 'z::state_ext state ⇒ 'z::state_ext state" where
 "detype S s ≡ s (| kheap := (λx. if x ∈ S then None else kheap s x), exst := detype_ext S (exst
s)|)"
```

Delete objects within a specified region.

definition

```
  delete_objects :: "word32 ⇒ nat ⇒ (unit,'z::state_ext) s_monad" where
 "delete_objects ptr bits = do
     do_machine_op (freeMemory ptr bits);
     modify (detype {ptr..ptr + 2 ^ bits - 1})
  od"
```

This is a placeholder function. We may wish to extend the specification with explicitly tagging kernel data regions in memory.

definition
```
  reserve_region :: "obj_ref ⇒ nat ⇒ bool ⇒ (unit,'z::state_ext) s_monad" where
  "reserve_region ptr byteLength is_kernel ≡ return ()"
```

Create 4096-byte frame objects that can be mapped into memory. These must be cleared to prevent past contents being revealed.

definition
```
  create_word_objects :: "word32 ⇒ nat ⇒ nat ⇒ (unit,'z::state_ext) s_monad" where
  "create_word_objects ptr numObjects sz ≡
  do
    byteLength ← return $ numObjects * 2 ^ sz;
    reserve_region ptr byteLength True;
    rst ←  return (map (λ n. (ptr + (n << sz))) [0  .e.  (of_nat numObjects) - 1]);
    do_machine_op $ mapM_x (λx. clearMemory x (2 ^ sz)) rst
 od"
```

Initialise architecture-specific objects.

definition
```
  init_arch_objects :: "apiobject_type ⇒ obj_ref ⇒ nat ⇒ nat ⇒ obj_ref list ⇒ (unit,'z::state_ext)
s_monad"
where
  "init_arch_objects new_type ptr num_objects obj_sz refs ≡ case new_type of
    ArchObject SmallPageObj ⇒ create_word_objects ptr num_objects 12
  | ArchObject LargePageObj ⇒ create_word_objects ptr num_objects 16
  | ArchObject SectionObj ⇒ create_word_objects ptr num_objects 20
  | ArchObject SuperSectionObj ⇒ create_word_objects ptr num_objects 24
  | ArchObject PageTableObj ⇒
      do_machine_op $ mapM_x (λx. cleanCacheRange_PoU x (x + ((1::word32) << pt_bits) - 1)
                                                  (addrFromPPtr x)) refs
  | ArchObject PageDirectoryObj ⇒ do
      mapM_x copy_global_mappings refs;
      do_machine_op $ mapM_x (λx. cleanCacheRange_PoU x (x + ((1::word32) << pd_bits) - 1)
                                                  (addrFromPPtr x)) refs
    od
  | _ ⇒ return ()"
```

Untyped capabilities confer authority to the Retype method. This clears existing objects from a region, creates new objects of the requested type, initialises them and installs new capabilities to them.

fun
```
  invoke_untyped :: "untyped_invocation ⇒ (unit,'z::state_ext) s_monad"
where
"invoke_untyped (Retype src_slot base free_region_base new_type obj_sz slots) =
do
  cap ← get_cap src_slot;

  (* If we are creating the first object, detype the entire region. *)
  when (base = free_region_base)
    $ delete_objects base (bits_of cap);

  (* Update the untyped cap to track the amount of space used. *)
  total_object_size ← return $ (of_nat (length slots) << (obj_bits_api new_type obj_sz));
  free_ref ← return $ free_region_base + total_object_size;
  set_cap (UntypedCap base (bits_of cap) (unat (free_ref - base))) src_slot;

  (* Create new objects. *)
  orefs ← retype_region free_region_base (length slots) obj_sz new_type;
```

```
  init_arch_objects new_type free_region_base (length slots) obj_sz orefs;
  sequence_x (map (create_cap new_type obj_sz src_slot) (zip slots orefs))
od"
```

end

27 ARM VSpace Functions

```
theory ArchVSpace_A
imports Retype_A
begin
```

Save the set of entries that would be inserted into a page table or page directory to map various different sizes of frame at a given virtual address.

```
fun create_mapping_entries ::
  "paddr ⇒ vspace_ref ⇒ vmpage_size ⇒ vm_rights ⇒ vm_attributes ⇒ word32 ⇒
  ((pte * word32 list) + (pde * word32 list),'z::state_ext) se_monad"
where
  "create_mapping_entries base vptr ARMSmallPage vm_rights attrib pd =
  doE
    p ← lookup_error_on_failure False $ lookup_pt_slot pd vptr;
    returnOk $ Inl (SmallPagePTE base (attrib - {Global, ParityEnabled})
                               vm_rights, [p])
  odE"

| "create_mapping_entries base vptr ARMLargePage vm_rights attrib pd =
  doE
    p ← lookup_error_on_failure False $ lookup_pt_slot pd vptr;
    returnOk $ Inl (LargePagePTE base (attrib - {Global, ParityEnabled})
                               vm_rights, [p, p + 4  .e.  p + 60])
  odE"

| "create_mapping_entries base vptr ARMSection vm_rights attrib pd =
  doE
    p ← returnOk (lookup_pd_slot pd vptr);
    returnOk $ Inr (SectionPDE base (attrib - {Global}) 0 vm_rights, [p])
  odE"

| "create_mapping_entries base vptr ARMSuperSection vm_rights attrib pd =
  doE
    p ← returnOk (lookup_pd_slot pd vptr);
    returnOk $ Inr (SuperSectionPDE base (attrib - {Global}) vm_rights, [p, p + 4  .e.  p + 60])
  odE"

definition get_master_pde :: "word32 ⇒ (ARM_Structs_A.pde,'z::state_ext)s_monad"
  where "get_master_pde ptr ≡ do
    pde ← (get_pde (ptr && ~~ mask 6));
    (case pde of ARM_Structs_A.pde.SuperSectionPDE _ _ _ ⇒ return pde
    | _ ⇒ get_pde ptr)
  od"

definition get_master_pte :: "word32 ⇒ (ARM_Structs_A.pte, 'z::state_ext)s_monad"
  where "get_master_pte ptr ≡ do
    pte ← (get_pte (ptr && ~~ mask 6));
    (case pte of ARM_Structs_A.pte.LargePagePTE _ _ _ ⇒ return pte
    | _ ⇒ get_pte ptr)
  od"
```

Placing an entry which maps a frame within the set of entries that map a larger frame is unsafe. This function checks that given entries replace either invalid entries or entries of the same granularity.

```
fun ensure_safe_mapping ::
  "(pte * word32 list) + (pde * word32 list) ⇒ (unit,'z::state_ext) se_monad"
where
"ensure_safe_mapping (Inl (InvalidPTE, _)) = returnOk ()"
|
"ensure_safe_mapping (Inl (SmallPagePTE _ _ _, pt_slots)) =
    mapME_x (λslot. (doE
        pte ← liftE $ get_master_pte slot;
        (case pte of
              InvalidPTE ⇒ returnOk ()
            | SmallPagePTE _ _ _ ⇒ returnOk ()
            | _ ⇒ throwError DeleteFirst)
    odE)) pt_slots"
|
"ensure_safe_mapping (Inl (LargePagePTE _ _ _, pt_slots)) =
    mapME_x (λ slot. (doE
        pte ← liftE $ get_master_pte slot;
        (case pte of
              InvalidPTE ⇒ returnOk ()
            | LargePagePTE _ _ _ ⇒ returnOk ()
            | _ ⇒ throwError DeleteFirst
            )
    odE)) pt_slots"
|
"ensure_safe_mapping (Inr (InvalidPDE, _)) = returnOk ()"
|
"ensure_safe_mapping (Inr (PageTablePDE _ _ _, _)) = fail"
|
"ensure_safe_mapping (Inr (SectionPDE _ _ _ _, pd_slots)) =
    mapME_x (λ slot. (doE
        pde ← liftE $ get_master_pde slot;
        (case pde of
              InvalidPDE ⇒ returnOk ()
            | SectionPDE _ _ _ _ ⇒ returnOk ()
            | _ ⇒ throwError DeleteFirst
            )
    odE)) pd_slots"
|
"ensure_safe_mapping (Inr (SuperSectionPDE _ _ _, pd_slots)) =
    mapME_x (λ slot. (doE
        pde ← liftE $ get_master_pde slot;
        (case pde of
              InvalidPDE ⇒ returnOk ()
            | SuperSectionPDE _ _ _ ⇒ returnOk ()
            | _ ⇒ throwError DeleteFirst
            )
    odE)) pd_slots"
```

Look up a thread's IPC buffer and check that the thread has the right authority to read or (in the receiver case) write to it.

```
definition
lookup_ipc_buffer :: "bool ⇒ word32 ⇒ (word32 option,'z::state_ext) s_monad" where
"lookup_ipc_buffer is_receiver thread ≡ do
    buffer_ptr ← thread_get tcb_ipc_buffer thread;
    buffer_frame_slot ← return (thread, tcb_cnode_index 4);
    buffer_cap ← get_cap buffer_frame_slot;
```

```
    (case buffer_cap of
      ArchObjectCap (PageCap p R vms _) ⇒
        if vm_read_write ⊆ R ∨ vm_read_only ⊆ R ∧ ¬is_receiver
        then return $ Some $ p + (buffer_ptr && mask (pageBitsForSize vms))
        else return None
    | _ ⇒ return None)
od"
```

Locate the page directory associated with a given virtual ASID.

definition
```
find_pd_for_asid :: "asid ⇒ (word32,'z::state_ext) lf_monad" where
"find_pd_for_asid asid ≡ doE
    assertE (asid > 0);
    asid_table ← liftE $ gets (arm_asid_table ∘ arch_state);
    pool_ptr ← returnOk (asid_table (asid_high_bits_of asid));
    pool ← (case pool_ptr of
               Some ptr ⇒ liftE $ get_asid_pool ptr
             | None ⇒ throwError InvalidRoot);
    pd ← returnOk (pool (ucast asid));
    (case pd of
          Some ptr ⇒ returnOk ptr
        | None ⇒ throwError InvalidRoot)
odE"
```

Locate the page directory and check that this process succeeds and returns a pointer to a real page directory.

definition
```
find_pd_for_asid_assert :: "asid ⇒ (word32,'z::state_ext) s_monad" where
"find_pd_for_asid_assert asid ≡ do
   pd ← find_pd_for_asid asid <catch> K fail;
   get_pde pd;
   return pd
 od"
```

Format a VM fault message to be passed to a thread's supervisor after it encounters a page fault.

fun
```
handle_vm_fault :: "word32 ⇒ vmfault_type ⇒ (unit,'z::state_ext) f_monad"
```
where
```
"handle_vm_fault thread ARMDataAbort = doE
    addr ← liftE $ do_machine_op getFAR;
    fault ← liftE $ do_machine_op getDFSR;
    throwError $ VMFault addr [0, fault && mask 12]
odE"
|
"handle_vm_fault thread ARMPrefetchAbort = doE
    pc ← liftE $ as_user thread $ getRestartPC;
    fault ← liftE $ do_machine_op getIFSR;
    throwError $ VMFault pc [1, fault && mask 12]
odE"
```

Load the optional hardware ASID currently associated with this virtual ASID.

definition
```
load_hw_asid :: "asid ⇒ (hardware_asid option,'z::state_ext) s_monad" where
"load_hw_asid asid ≡ do
    asid_map ← gets (arm_asid_map ∘ arch_state);
    return $ option_map fst $ asid_map asid
od"
```

Associate a hardware ASID with a virtual ASID.

```
definition
store_hw_asid :: "asid ⇒ hardware_asid ⇒ (unit,'z::state_ext) s_monad" where
"store_hw_asid asid hw_asid ≡ do
    pd ← find_pd_for_asid_assert asid;
    asid_map ← gets (arm_asid_map ∘ arch_state);
    asid_map' ← return (asid_map (asid ↦ (hw_asid, pd)));
    modify (λs. s ⦇ arch_state := (arch_state s) ⦇ arm_asid_map := asid_map' ⦈⦈);
    hw_asid_map ← gets (arm_hwasid_table ∘ arch_state);
    hw_asid_map' ← return (hw_asid_map (hw_asid ↦ asid));
    modify (λs. s ⦇ arch_state := (arch_state s) ⦇ arm_hwasid_table := hw_asid_map' ⦈⦈)
od"
```

Clear all TLB mappings associated with this virtual ASID.

```
definition
invalidate_tlb_by_asid :: "asid ⇒ (unit,'z::state_ext) s_monad" where
"invalidate_tlb_by_asid asid ≡ do
    maybe_hw_asid ← load_hw_asid asid;
    (case maybe_hw_asid of
          None ⇒ return ()
        | Some hw_asid ⇒ do_machine_op $ invalidateTLB_ASID hw_asid)
od"
```

Flush all cache and TLB entries associated with this virtual ASID.

```
definition
flush_space :: "asid ⇒ (unit,'z::state_ext) s_monad" where
"flush_space asid ≡ do
    maybe_hw_asid ← load_hw_asid asid;
    do_machine_op cleanCaches_PoU;
    (case maybe_hw_asid of
          None ⇒ return ()
        | Some hw_asid ⇒ do_machine_op $ invalidateTLB_ASID hw_asid)
od"
```

Remove any mapping from this virtual ASID to a hardware ASID.

```
definition
invalidate_asid :: "asid ⇒ (unit,'z::state_ext) s_monad" where
"invalidate_asid asid ≡ do
    asid_map ← gets (arm_asid_map ∘ arch_state);
    asid_map' ← return (asid_map (asid:= None));
    modify (λs. s ⦇ arch_state := (arch_state s) ⦇ arm_asid_map := asid_map' ⦈⦈)
od"
```

Remove any mapping from this hardware ASID to a virtual ASID.

```
definition
invalidate_hw_asid_entry :: "hardware_asid ⇒ (unit,'z::state_ext) s_monad" where
"invalidate_hw_asid_entry hw_asid ≡ do
  hw_asid_map ← gets (arm_hwasid_table ∘ arch_state);
  hw_asid_map' ← return (hw_asid_map (hw_asid:= None));
  modify (λs. s ⦇ arch_state := (arch_state s) ⦇ arm_hwasid_table := hw_asid_map' ⦈⦈)
od"
```

Remove virtual to physical mappings in either direction involving this virtual ASID.

```
definition
invalidate_asid_entry :: "asid ⇒ (unit,'z::state_ext) s_monad" where
"invalidate_asid_entry asid ≡ do
```

```
  maybe_hw_asid ← load_hw_asid asid;
  when (maybe_hw_asid ≠ None) $ invalidate_hw_asid_entry (the maybe_hw_asid);
  invalidate_asid asid
od"
```

Locate a hardware ASID that is not in use, if necessary by reclaiming one from another virtual ASID in a round-robin manner.

definition

```
find_free_hw_asid :: "(hardware_asid,'z::state_ext) s_monad" where
"find_free_hw_asid ≡ do
    hw_asid_table ← gets (arm_hwasid_table ∘ arch_state);
    next_asid ← gets (arm_next_asid ∘ arch_state);
    maybe_asid ← return (find (λa. hw_asid_table a = None)
                    (take (length [minBound :: hardware_asid .e. maxBound])
                        ([next_asid .e. maxBound] @ [minBound .e. next_asid])));
    (case maybe_asid of
       Some hw_asid ⇒ return hw_asid
     | None ⇒  do
            invalidate_asid $ the $ hw_asid_table next_asid;
            do_machine_op $ invalidateTLB_ASID next_asid;
            invalidate_hw_asid_entry next_asid;
            new_next_asid ← return (next_asid + 1);
            modify (λs. s ⦇ arch_state := (arch_state s) ⦇ arm_next_asid := new_next_asid ⦈⦈);
            return next_asid
       od)
od"
```

Get the hardware ASID associated with a virtual ASID, assigning one if none is already assigned.

definition

```
get_hw_asid :: "asid ⇒ (hardware_asid,'z::state_ext) s_monad" where
"get_hw_asid asid ≡ do
  maybe_hw_asid ← load_hw_asid asid;
  (case maybe_hw_asid of
    Some hw_asid ⇒ return hw_asid
  | None ⇒  do
      new_hw_asid ← find_free_hw_asid;
      store_hw_asid asid new_hw_asid;
      return new_hw_asid
  od)
od"
```

Set the current virtual ASID by setting the hardware ASID to one associated with it.

definition

```
set_current_asid :: "asid ⇒ (unit,'z::state_ext) s_monad" where
"set_current_asid asid ≡ do
    hw_asid ← get_hw_asid asid;
    do_machine_op $ setHardwareASID hw_asid
od"
```

Switch into the address space of a given thread or the global address space if none is correctly configured.

definition

```
  set_vm_root :: "word32 ⇒ (unit,'z::state_ext) s_monad" where
"set_vm_root tcb ≡ do
    thread_root_slot ← return (tcb, tcb_cnode_index 1);
    thread_root ← get_cap thread_root_slot;
    (case thread_root of
```

```
       ArchObjectCap (PageDirectoryCap pd (Some asid)) ⇒ doE
           pd' ← find_pd_for_asid asid;
           whenE (pd ≠ pd') $ throwError InvalidRoot;
           liftE $ do
               do_machine_op $ setCurrentPD $ addrFromPPtr pd;
               set_current_asid asid
           od
       odE
     | _ ⇒ throwError InvalidRoot) <catch>
    (λ_. do
       global_pd ← gets (arm_global_pd ∘ arch_state);
       do_machine_op $ setCurrentPD $ addrFromPPtr global_pd
    od)
od"
```

Before deleting an ASID pool object we must deactivate all page directories that are installed in it.

definition
```
delete_asid_pool :: "asid ⇒ word32 ⇒ (unit,'z::state_ext) s_monad" where
"delete_asid_pool base ptr ≡ do
  assert (base && mask asid_low_bits = 0);
  asid_table ← gets (arm_asid_table ∘ arch_state);
  when (asid_table (asid_high_bits_of base) = Some ptr) $ do
    pool ← get_asid_pool ptr;
    mapM (λoffset. (when (pool (ucast offset) ≠ None) $ do
                      flush_space $ base + offset;
                      invalidate_asid_entry $ base + offset
                od)) [0  .e.  (1 << asid_low_bits) - 1];
    asid_table' ← return (asid_table (asid_high_bits_of base:= None));
    modify (λs. s ⦇ arch_state := (arch_state s) ⦇ arm_asid_table := asid_table' ⦈⦈);
    tcb ← gets cur_thread;
    set_vm_root tcb
  od
od"
```

When deleting a page directory from an ASID pool we must deactivate it.

definition
```
delete_asid :: "asid ⇒ word32 ⇒ (unit,'z::state_ext) s_monad" where
"delete_asid asid pd ≡ do
  asid_table ← gets (arm_asid_table ∘ arch_state);
  (case asid_table (asid_high_bits_of asid) of
    None ⇒ return ()
  | Some pool_ptr ⇒  do
     pool ← get_asid_pool pool_ptr;
     when (pool (ucast asid) = Some pd) $ do
                flush_space asid;
                invalidate_asid_entry asid;
                pool' ← return (pool (ucast asid := None));
                set_asid_pool pool_ptr pool';
                tcb ← gets cur_thread;
                set_vm_root tcb
            od
    od)
od"
```

Switch to a particular address space in order to perform a flush operation.

definition
```
set_vm_root_for_flush :: "word32 ⇒ asid ⇒ (bool,'z::state_ext) s_monad" where
```

```
"set_vm_root_for_flush pd asid ≡ do
    tcb ← gets cur_thread;
    thread_root_slot ← return (tcb, tcb_cnode_index 1);
    thread_root ← get_cap thread_root_slot;
    not_is_pd ← (case thread_root of
                    ArchObjectCap (PageDirectoryCap cur_pd (Some _)) ⇒ return (cur_pd ≠ pd)
                  | _ ⇒ return True);
    (if not_is_pd then do
        do_machine_op $ setCurrentPD $ addrFromPPtr pd;
        set_current_asid asid;
        return True
    od
    else return False)
od"
```

definition
```
do_flush :: "flush_type ⇒ vspace_ref ⇒ vspace_ref ⇒ paddr ⇒ unit machine_monad" where
"do_flush flush_type vstart vend pstart ≡
    case flush_type of
       Clean ⇒ cleanCacheRange_RAM vstart vend pstart
     | Invalidate ⇒ invalidateCacheRange_RAM vstart vend pstart
     | CleanInvalidate ⇒ cleanInvalidateCacheRange_RAM vstart vend pstart
     | Unify ⇒ do
         cleanCacheRange_PoU vstart vend pstart;
         dsb;
         invalidateCacheRange_I vstart vend pstart;
         branchFlushRange vstart vend pstart;
         isb
     od"
```

Flush mappings associated with a page table.

definition
```
flush_table :: "word32 ⇒ asid ⇒ vspace_ref ⇒ word32 ⇒ (unit,'z::state_ext) s_monad" where
"flush_table pd asid vptr pt ≡ do
    assert (vptr && mask (pageBitsForSize ARMSection) = 0);
    root_switched ← set_vm_root_for_flush pd asid;
    maybe_hw_asid ← load_hw_asid asid;
    when (maybe_hw_asid ≠ None) $ do
      hw_asid ← return (the maybe_hw_asid);
      do_machine_op $ invalidateTLB_ASID hw_asid;
      when root_switched $ do
        tcb ← gets cur_thread;
        set_vm_root tcb
      od
    od
od"
```

Flush mappings associated with a given page.

definition
```
flush_page :: "vmpage_size ⇒ word32 ⇒ asid ⇒ vspace_ref ⇒ (unit,'z::state_ext) s_monad" where
"flush_page page_size pd asid vptr≡ do
    assert (vptr && mask pageBits = 0);
    root_switched ← set_vm_root_for_flush pd asid;
    maybe_hw_asid ← load_hw_asid asid;
    when (maybe_hw_asid ≠ None) $ do
      hw_asid ← return (the maybe_hw_asid);
      do_machine_op $ invalidateTLB_VAASID (vptr || ucast hw_asid);
      when root_switched $ do
```

```
          tcb ← gets cur_thread;
          set_vm_root tcb
      od
   od
od"
```

Return the optional page directory a page table is mapped in.

definition
```
page_table_mapped :: "asid ⇒ vspace_ref ⇒ obj_ref ⇒ (obj_ref option,'z::state_ext) s_monad"
```
where
```
"page_table_mapped asid vaddr pt ≡ doE
    pd ← find_pd_for_asid asid;
    pd_slot ← returnOk $ lookup_pd_slot pd vaddr;
    pde ← liftE $ get_pde pd_slot;
    case pde of
      PageTablePDE addr _ _ ⇒ returnOk $
             if addrFromPPtr pt = addr then Some pd else None
    | _ ⇒ returnOk None
odE <catch> (K $ return None)"
```

Unmap a page table from its page directory.

definition
```
unmap_page_table :: "asid ⇒ vspace_ref ⇒ word32 ⇒ (unit,'z::state_ext) s_monad" where
"unmap_page_table asid vaddr pt ≡ do
    pdOpt ← page_table_mapped asid vaddr pt;
    case pdOpt of
      None ⇒ return ()
    | Some pd ⇒ do
        pd_slot ← return $ lookup_pd_slot pd vaddr;
        store_pde pd_slot InvalidPDE;
        do_machine_op $ cleanByVA_PoU pd_slot (addrFromPPtr pd_slot);
        flush_table pd asid vaddr pt
    od
od"
```

Check that a given frame is mapped by a given mapping entry.

definition
```
check_mapping_pptr :: "obj_ref ⇒ vmpage_size ⇒ (obj_ref + obj_ref) ⇒ (bool,'z::state_ext) s_monad"
```
where
```
"check_mapping_pptr pptr pgsz tablePtr ≡ case tablePtr of
   Inl ptePtr ⇒ do
     pte ← get_pte ptePtr;
     return $ case pte of
       SmallPagePTE x _ _ ⇒ x = addrFromPPtr pptr ∧ pgsz = ARMSmallPage
     | LargePagePTE x _ _ ⇒ x = addrFromPPtr pptr ∧ pgsz = ARMLargePage
     | _ ⇒ False
   od
 | Inr pdePtr ⇒ do
     pde ← get_pde pdePtr;
     return $ case pde of
       SectionPDE x _ _ _ ⇒ x = addrFromPPtr pptr ∧ pgsz = ARMSection
     | SuperSectionPDE x _ _ ⇒ x = addrFromPPtr pptr ∧ pgsz = ARMSuperSection
     | _ ⇒ False
   od"
```

Raise an exception if a property does not hold.

definition

```
throw_on_false :: "'e ⇒ (bool,'z::state_ext) s_monad ⇒ ('e + unit,'z::state_ext) s_monad" where
"throw_on_false ex f ≡ doE v ← liftE f; unlessE v $ throwError ex odE"
```

definition

```
  "last_byte_pte x ≡ let pte_bits = 2 in x + ((1 << pte_bits) - 1)"
```

definition

```
  "last_byte_pde x ≡ let pde_bits = 2 in x + ((1 << pde_bits) - 1)"
```

Unmap a mapped page if the given mapping details are still current.

definition

```
unmap_page :: "vmpage_size ⇒ asid ⇒ vspace_ref ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad" where
"unmap_page pgsz asid vptr pptr ≡ doE
    pd ← find_pd_for_asid asid;
    (case pgsz of
          ARMSmallPage ⇒ doE
            p ← lookup_pt_slot pd vptr;
            throw_on_false undefined $
                check_mapping_pptr pptr pgsz (Inl p);
            liftE $ do
                store_pte p InvalidPTE;
                do_machine_op $ cleanByVA_PoU p (addrFromPPtr p)
            od
          odE
        | ARMLargePage ⇒ doE
            p ← lookup_pt_slot pd vptr;
            throw_on_false undefined $
                check_mapping_pptr pptr pgsz (Inl p);
            liftE $ do
                assert $ p && mask 6 = 0;
                slots ← return (map (λx. x + p) [0, 4  .e.  60]);
                mapM (swp store_pte InvalidPTE) slots;
                do_machine_op $ cleanCacheRange_PoU (hd slots) (last_byte_pte (last slots))
                                                    (addrFromPPtr (hd slots))
            od
          odE
        | ARMSection ⇒ doE
            p ← returnOk (lookup_pd_slot pd vptr);
            throw_on_false undefined $
                check_mapping_pptr pptr pgsz (Inr p);
            liftE $ do
                store_pde p InvalidPDE;
                do_machine_op $ cleanByVA_PoU p (addrFromPPtr p)
            od
          odE
        | ARMSuperSection ⇒ doE
            p ← returnOk (lookup_pd_slot pd vptr);
            throw_on_false undefined $
                check_mapping_pptr pptr pgsz (Inr p);
            liftE $ do
                assert $ p && mask 6 = 0;
                slots ← return (map (λx. x + p) [0, 4  .e.  60]);
                mapM (swp store_pde InvalidPDE) slots;
                do_machine_op $ cleanCacheRange_PoU (hd slots) (last_byte_pde (last slots))
                                                    (addrFromPPtr (hd slots))
            od
          odE);
    liftE $ flush_page pgsz pd asid vptr
```

```
odE <catch> (K $ return ())"
```

PageDirectory and PageTable capabilities cannot be copied until they have a virtual ASID and location assigned. This is because page directories cannot have multiple current virtual ASIDs and page tables cannot be shared between address spaces or virtual locations.

```
definition
  arch_derive_cap :: "arch_cap ⇒ (arch_cap,'z::state_ext) se_monad"
where
  "arch_derive_cap c ≡ case c of
     PageTableCap _ (Some x) ⇒ returnOk c
   | PageTableCap _ None ⇒ throwError IllegalOperation
   | PageDirectoryCap _ (Some x) ⇒ returnOk c
   | PageDirectoryCap _ None ⇒ throwError IllegalOperation
   | PageCap r R pgs x ⇒ returnOk (PageCap r R pgs None)
   | ASIDControlCap ⇒ returnOk c
   | ASIDPoolCap _ _ ⇒ returnOk c"
```

No user-modifiable data is stored in ARM-specific capabilities.

```
definition
  arch_update_cap_data :: "data ⇒ arch_cap ⇒ arch_cap"
where
  "arch_update_cap_data data c ≡ c"
```

Actions that must be taken on finalisation of ARM-specific capabilities.

```
definition
  arch_finalise_cap :: "arch_cap ⇒ bool ⇒ (cap,'z::state_ext) s_monad"
where
  "arch_finalise_cap c x ≡ case (c, x) of
    (ASIDPoolCap ptr b, True) ⇒  do
    delete_asid_pool b ptr;
    return NullCap
    od
  | (PageDirectoryCap ptr (Some a), True) ⇒ do
    delete_asid a ptr;
    return NullCap
  od
  | (PageTableCap ptr (Some (a, v)), True) ⇒ do
    unmap_page_table a v ptr;
    return NullCap
  od
  | (PageCap ptr _ s (Some (a, v)), _) ⇒ do
     unmap_page s a v ptr;
     return NullCap
  od
  | _ ⇒ return NullCap"
```

Remove record of mappings to a page cap, page table cap or page directory cap

```
fun
  arch_reset_mem_mapping :: "arch_cap ⇒ arch_cap"
where
  "arch_reset_mem_mapping (PageCap p rts sz mp) = PageCap p rts sz None"
| "arch_reset_mem_mapping (PageTableCap ptr mp) = PageTableCap ptr None"
| "arch_reset_mem_mapping (PageDirectoryCap ptr ma) = PageDirectoryCap ptr None"
| "arch_reset_mem_mapping cap = cap"
```

Actions that must be taken to recycle ARM-specific capabilities.

```
definition
```

```
  arch_recycle_cap :: "bool ⇒ arch_cap ⇒ (arch_cap,'z::state_ext) s_monad"
where
  "arch_recycle_cap is_final cap ≡ case cap of
    PageCap p _ sz _ ⇒ do
      do_machine_op $ clearMemory p (2 ^ (pageBitsForSize sz));
      arch_finalise_cap cap is_final;
      return $ arch_reset_mem_mapping cap
    od
  | PageTableCap ptr mp ⇒ do
      pte_bits ← return 2;
      slots ← return [ptr, ptr + (1 << pte_bits) .e. ptr + (1 << pt_bits) - 1];
      mapM_x (swp store_pte InvalidPTE) slots;
      do_machine_op $ cleanCacheRange_PoU ptr (ptr + (1 << pt_bits) - 1)
                                    (addrFromPPtr ptr);
      case mp of None ⇒ return ()
       | Some (a, v) ⇒ do
          pdOpt ← page_table_mapped a v ptr;
          when (pdOpt ≠ None) $ invalidate_tlb_by_asid a
          od;
      arch_finalise_cap cap is_final;
      return (if is_final then arch_reset_mem_mapping cap else cap)
    od
  | PageDirectoryCap ptr ma ⇒ do
      pde_bits ← return 2;
      indices ← return [0 .e. (kernel_base >> pageBitsForSize ARMSection) - 1];
      offsets ← return (map (swp (op <<) pde_bits) indices);
      slots ← return (map (λx. x + ptr) offsets);
      mapM_x (swp store_pde InvalidPDE) slots;
      do_machine_op $ cleanCacheRange_PoU ptr (ptr + (1 << pd_bits) - 1)
                                    (addrFromPPtr ptr);
      case ma of None ⇒ return ()
              | Some a ⇒ doE
                          pd' ← find_pd_for_asid a;
                          liftE $ when (pd' = ptr) $ invalidate_tlb_by_asid a
                         odE <catch> K (return ());
      arch_finalise_cap cap is_final;
      return (if is_final then arch_reset_mem_mapping cap else cap)
    od
  | ASIDControlCap ⇒ return ASIDControlCap
  | ASIDPoolCap ptr base ⇒ do
      asid_table ← gets (arm_asid_table ∘ arch_state);
      when (asid_table (asid_high_bits_of base) = Some ptr) $ do
          delete_asid_pool base ptr;
          set_asid_pool ptr empty;
          asid_table ← gets (arm_asid_table ∘ arch_state);
          asid_table' ← return (asid_table (asid_high_bits_of base ↦ ptr));
          modify (λs. s ⦇ arch_state := (arch_state s) ⦇ arm_asid_table := asid_table' ⦈⦈)
      od;
      return cap
    od"
```

A thread's virtual address space capability must be to a page directory to be valid on the ARM architecture.

definition

```
  is_valid_vtable_root :: "cap ⇒ bool" where
  "is_valid_vtable_root c ≡ ∃r a. c = ArchObjectCap (PageDirectoryCap r (Some a))"
```

A thread's IPC buffer capability must be to a page that is capable of containing the IPC buffer without the end of the buffer spilling into another page.

```
definition
  cap_transfer_data_size :: nat where
  "cap_transfer_data_size ≡ 3"

definition
  msg_max_length :: nat where
 "msg_max_length ≡ 120"

definition
  msg_max_extra_caps :: nat where
 "msg_max_extra_caps ≡ 3"

definition
  msg_align_bits :: nat
  where
  "msg_align_bits ≡ 2 + (LEAST n. (cap_transfer_data_size + msg_max_length + msg_max_extra_caps
+ 2) ≤ 2 ^ n)"

lemma msg_align_bits:
  "msg_align_bits = 9"

definition
check_valid_ipc_buffer :: "vspace_ref ⇒ cap ⇒ (unit,'z::state_ext) se_monad" where
"check_valid_ipc_buffer vptr c ≡ case c of
  (ArchObjectCap (PageCap _ _ magnitude _)) ⇒ doE
    whenE (¬ is_aligned vptr msg_align_bits) $ throwError AlignmentError;
    returnOk ()
  odE
| _ ⇒ throwError IllegalOperation"
```

On the abstract level, capability and VM rights share the same type. Nevertheless, a simple set intersection might lead to an invalid value like {AllowWrite}. Hence, validate_vm_rights.

```
definition
  mask_vm_rights :: "vm_rights ⇒ cap_rights ⇒ vm_rights" where
  "mask_vm_rights V R ≡ validate_vm_rights (V ∩ R)"
```

Decode a user argument word describing the kind of VM attributes a mapping is to have.

```
definition
attribs_from_word :: "word32 ⇒ vm_attributes" where
"attribs_from_word w ≡
  let V = (if w !!0 then {PageCacheable} else {})
  in if w!!1 then insert ParityEnabled V else V"
```

Update the mapping data saved in a page or page table capability.

```
definition
  update_map_data :: "arch_cap ⇒ (word32 × word32) option ⇒ arch_cap" where
  "update_map_data cap m ≡ case cap of PageCap p R sz _ ⇒ PageCap p R sz m
                                   | PageTableCap p _ ⇒ PageTableCap p m"
```

Get information about the frame of a given virtual address

```
definition
  resolve_vaddr :: "word32 ⇒ vspace_ref ⇒ ((vmpage_size × obj_ref) option, 'z::state_ext) s_monad"
where
  "resolve_vaddr pd vaddr ≡ do
     pd_slot ← return $ lookup_pd_slot pd vaddr;
```

```
    pde ← get_master_pde pd_slot;
    case pde of
        SectionPDE f _ _ _ ⇒ return $ Some (ARMSection, f)
      | SuperSectionPDE f _ _ ⇒ return $ Some (ARMSuperSection, f)
      | PageTablePDE t _ _ ⇒ (do
          pt ← return $ ptrFromPAddr t;
          pte_slot ← return $ lookup_pt_slot_no_fail pt vaddr;
          pte ← get_master_pte pte_slot;
          case pte of
              LargePagePTE f _ _ ⇒ return $ Some (ARMLargePage, f)
            | SmallPagePTE f _ _ ⇒ return $ Some (ARMSmallPage, f)
            | _ ⇒ return None
        od)
      | _ ⇒ return None
  od"
```

end

28 IPC Cancelling

```
theory IpcCancel_A
imports CSpaceAcc_A
begin
```

Getting and setting endpoint queues.

```
definition
  get_ep_queue :: "endpoint ⇒ (obj_ref list,'z::state_ext) s_monad"
where
 "get_ep_queue ep ≡ case ep of SendEP q ⇒ return q
                              | RecvEP q ⇒ return q
                              | _ ⇒ fail"
```

```
primrec
  update_ep_queue :: "endpoint ⇒ obj_ref list ⇒ endpoint"
where
  "update_ep_queue (RecvEP q) q' = RecvEP q'"
| "update_ep_queue (SendEP q) q' = SendEP q'"
```

Cancel all message operations on threads currently queued within this synchronous message endpoint. Threads so queued are placed in the Restart state. Once scheduled they will reattempt the operation that previously caused them to be queued here.

```
definition
  ep_cancel_all :: "obj_ref ⇒ (unit,'z::state_ext) s_monad"
where
  "ep_cancel_all epptr ≡ do
     ep ← get_endpoint epptr;
     case ep of IdleEP ⇒ return ()
               | _ ⇒ do
                        queue ← get_ep_queue ep;
                        set_endpoint epptr IdleEP;
                        mapM_x (λt. do set_thread_state t Restart;
                                       do_extended_op (tcb_sched_action (tcb_sched_enqueue) t)
od) $ queue;
                        do_extended_op (reschedule_required)
                     od
   od"
```

The badge stored by thread waiting on a message send operation.

```
primrec
  blocking_ipc_badge :: "thread_state ⇒ badge"
where
  "blocking_ipc_badge (BlockedOnSend t payload) = sender_badge payload"
```

Cancel all message send operations on threads queued in this endpoint and using a particular badge.

```
definition
  ep_cancel_badged_sends :: "obj_ref ⇒ badge ⇒ (unit,'z::state_ext) s_monad"
where
  "ep_cancel_badged_sends epptr badge ≡ do
    ep ← get_endpoint epptr;
```

```
    case ep of
          IdleEP ⇒ return ()
        | RecvEP _ ⇒  return ()
        | SendEP queue ⇒  do
            set_endpoint epptr IdleEP;
            queue' ← (swp filterM queue) (λ t. do
                st ← get_thread_state t;
                if blocking_ipc_badge st = badge then do
                  set_thread_state t Restart;
                  do_extended_op (tcb_sched_action (tcb_sched_enqueue) t);
                  return False od
                else return True
            od);
            ep' ← return (case queue' of
                           [] ⇒ IdleEP
                         | _ ⇒ SendEP queue');
            set_endpoint epptr ep';
            do_extended_op (reschedule_required)
        od
  od"
```

Cancel all message operations on threads queued in an asynchronous endpoint.

definition

```
  aep_cancel_all :: "obj_ref ⇒ (unit,'z::state_ext) s_monad"
```

where

```
  "aep_cancel_all aepptr ≡ do
     aep ← get_async_ep aepptr;
     case aep of WaitingAEP queue ⇒ do
                      _ ← set_async_ep aepptr IdleAEP;
                      mapM_x (λt. do set_thread_state t Restart;
                                     do_extended_op (tcb_sched_action tcb_sched_enqueue t) od)
queue;
                      do_extended_op (reschedule_required)
                     od
               | _ ⇒ return ()
   od"
```

The endpoint pointer stored by a thread waiting for a message to be transferred in either direction.

definition

```
  get_blocking_ipc_endpoint :: "thread_state ⇒ (obj_ref,'z::state_ext) s_monad"
```

where

```
 "get_blocking_ipc_endpoint state ≡
       case state of BlockedOnReceive epptr d ⇒ return epptr
                   | BlockedOnSend epptr x ⇒ return epptr
                   | _ ⇒ fail"
```

Cancel whatever IPC operation a thread is engaged in.

definition

```
  blocked_ipc_cancel :: "thread_state ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad"
```

where

```
  "blocked_ipc_cancel state tptr ≡ do
     epptr ← get_blocking_ipc_endpoint state;
     ep ← get_endpoint epptr;
     queue ← get_ep_queue ep;
     queue' ← return $ remove1 tptr queue;
     ep' ← return (case queue' of [] ⇒ IdleEP
                                |  _ ⇒ update_ep_queue ep queue');
```

```
    set_endpoint epptr ep';
    set_thread_state tptr Inactive
  od"
```

Finalise a capability if the capability is known to be of the kind which can be finalised immediately. This is a simplified version of the `finalise_cap` operation.

```
fun
  fast_finalise :: "cap ⇒ bool ⇒ (unit,'z::state_ext) s_monad"
where
  "fast_finalise NullCap                  final = return ()"
| "fast_finalise (ReplyCap r m)           final = return ()"
| "fast_finalise (EndpointCap r b R)      final =
      (when final $ ep_cancel_all r)"
| "fast_finalise (AsyncEndpointCap r b R) final =
      (when final $ aep_cancel_all r)"
| "fast_finalise (CNodeCap r bits g)      final = fail"
| "fast_finalise (ThreadCap r)            final = fail"
| "fast_finalise DomainCap                final = fail"
| "fast_finalise (Zombie r b n)           final = fail"
| "fast_finalise IRQControlCap            final = fail"
| "fast_finalise (IRQHandlerCap irq)      final = fail"
| "fast_finalise (UntypedCap r n f)       final = fail"
| "fast_finalise (ArchObjectCap a)        final = fail"
```

The optional IRQ stored in a capability, presented either as an optional value or a set.

```
definition
  cap_irq_opt :: "cap ⇒ irq option" where
 "cap_irq_opt cap ≡ case cap of IRQHandlerCap irq ⇒ Some irq | _ ⇒ None"
```

```
definition
  cap_irqs :: "cap ⇒ irq set" where
 "cap_irqs cap ≡ Option.set (cap_irq_opt cap)"
```

Detect whether a capability is the final capability to a given object remaining in the system. Finalisation actions need to be taken when the final capability to the object is deleted.

```
definition
  is_final_cap' :: "cap ⇒ 'z::state_ext state ⇒ bool" where
 "is_final_cap' cap s ≡
    ∃cref. {cref. ∃cap'. fst (get_cap cref s) = {(cap', s)}
                    ∧ (obj_refs cap ∩ obj_refs cap' ≠ {}
                            ∨ cap_irqs cap ∩ cap_irqs cap' ≠ {})}
        = {cref}"
```

```
definition
  is_final_cap :: "cap ⇒ (bool,'z::state_ext) s_monad" where
  "is_final_cap cap ≡ gets (is_final_cap' cap)"
```

Actions to be taken after an IRQ handler capability is deleted.

```
definition
  deleted_irq_handler :: "irq ⇒ (unit,'z::state_ext) s_monad"
where
 "deleted_irq_handler irq ≡ set_irq_state IRQInactive irq"
```

Empty a capability slot assuming that the capability in it has been finalised already.

```
definition
  empty_slot :: "cslot_ptr ⇒ irq option ⇒ (unit,'z::state_ext) s_monad"
```

```
where
 "empty_slot slot free_irq ≡ do
      cap ← get_cap slot;
      if cap = NullCap then
        return ()
      else do
        slot_p ← gets (λs. cdt s slot);
        cdt ← gets cdt;
        parent ← return $ cdt slot;
        set_cdt ((λp. if cdt p = Some slot
                     then parent
                     else cdt p) (slot := None));
        do_extended_op (empty_slot_ext slot slot_p);
        set_original slot False;
        set_cap NullCap slot;

        case free_irq of Some irq ⇒ deleted_irq_handler irq
           | None ⇒ return ()
      od
  od"
```

Delete a capability with the assumption that the fast finalisation process will be sufficient.

```
definition
  cap_delete_one :: "cslot_ptr ⇒ (unit,'z::state_ext) s_monad" where
 "cap_delete_one slot ≡ do
    cap ← get_cap slot;
    unless (cap = NullCap) $ do
      final ← is_final_cap cap;
      fast_finalise cap final;
      empty_slot slot None
    od
  od"
```

Cancel the message receive operation of a thread waiting for a Reply capability it has issued to be invoked.

```
definition
  reply_ipc_cancel :: "obj_ref ⇒ (unit,'z::state_ext) s_monad"
where
 "reply_ipc_cancel tptr ≡ do
    thread_set (λtcb. tcb (| tcb_fault := None |)) tptr;
    cap ← get_cap (tptr, tcb_cnode_index 2);
    descs ← gets (descendants_of (tptr, tcb_cnode_index 2) o cdt);
    when (descs ≠ {}) $ do
      assert (∃cslot_ptr. descs = {cslot_ptr});
      cslot_ptr ← select descs;
      cap_delete_one cslot_ptr
    od
  od"
```

Cancel the message receive operation of a thread queued in an asynchronous endpoint.

```
definition
  async_ipc_cancel :: "obj_ref ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad"
where
  "async_ipc_cancel threadptr aepptr ≡ do
     aep ← get_async_ep aepptr;
     queue ← (case aep of WaitingAEP queue ⇒ return queue
                        | _ ⇒ fail);
```

```
    queue’ ← return $ remove1 threadptr queue;
    set_async_ep aepptr (case queue’ of [] ⇒ IdleAEP
                                      | _  ⇒ WaitingAEP queue’);
    set_thread_state threadptr Inactive
  od"
```

Cancel any message operations a given thread is waiting on.

definition
```
  ipc_cancel :: "obj_ref ⇒ (unit,’z::state_ext) s_monad"
```
where
```
  "ipc_cancel tptr ≡ do
     state ← get_thread_state tptr;
     case state
       of
          BlockedOnSend x y ⇒ blocked_ipc_cancel state tptr
        | BlockedOnReceive x d ⇒ blocked_ipc_cancel state tptr
        | BlockedOnAsyncEvent event ⇒ async_ipc_cancel tptr event
        | BlockedOnReply ⇒ reply_ipc_cancel tptr
        | _ ⇒ return ()
   od"
```

Suspend a thread, cancelling any pending operations and preventing it from further execution by setting it to the Inactive state.

definition
```
  suspend :: "obj_ref ⇒ (unit,’z::state_ext) s_monad"
```
where
```
  "suspend thread ≡ do
     ipc_cancel thread;
     set_thread_state thread Inactive;
     do_extended_op (tcb_sched_action (tcb_sched_dequeue) thread)
   od"
```

end

29 Prefix order on lists as order class instance

```
theory Prefix_Order
imports Sublist
begin

instantiation list :: (type) order
begin

definition "(xs::'a list) ≤ ys ≡ prefixeq xs ys"
definition "(xs::'a list) < ys ≡ xs ≤ ys ∧ ¬ (ys ≤ xs)"

instance

end

lemmas prefixI [intro?] = prefixeqI [folded less_eq_list_def]
lemmas prefixE [elim?] = prefixeqE [folded less_eq_list_def]
lemmas strict_prefixI' [intro?] = prefixI' [folded less_list_def]
lemmas strict_prefixE' [elim?] = prefixE' [folded less_list_def]
lemmas strict_prefixI [intro?] = prefixI [folded less_list_def]
lemmas strict_prefixE [elim?] = prefixE [folded less_list_def]
theorems Nil_prefix [iff] = Nil_prefixeq [folded less_eq_list_def]
theorems prefix_Nil [simp] = prefixeq_Nil [folded less_eq_list_def]
lemmas prefix_snoc [simp] = prefixeq_snoc [folded less_eq_list_def]
lemmas Cons_prefix_Cons [simp] = Cons_prefixeq_Cons [folded less_eq_list_def]
lemmas same_prefix_prefix [simp] = same_prefixeq_prefixeq [folded less_eq_list_def]
lemmas same_prefix_nil [iff] = same_prefixeq_nil [folded less_eq_list_def]
lemmas prefix_prefix [simp] = prefixeq_prefixeq [folded less_eq_list_def]
theorems prefix_Cons = prefixeq_Cons [folded less_eq_list_def]
theorems prefix_length_le = prefixeq_length_le [folded less_eq_list_def]
lemmas strict_prefix_simps [simp, code] = prefix_simps [folded less_list_def]
lemmas not_prefix_induct [consumes 1, case_names Nil Neq Eq] =
  not_prefixeq_induct [folded less_eq_list_def]

end
```

30 CSpace

```
theory CSpace_A
imports
  ArchVSpace_A
  IpcCancel_A
  "../../lib/wp/NonDetMonadLemmas"
  "~~/src/HOL/Library/Prefix_Order"
begin
```

This theory develops an abstract model of *capability spaces*, or CSpace, in seL4. The CSpace of a thread can be thought of as the set of all capabilities it has access to. More precisely, it is a directed graph of CNodes starting in the CSpace slot of a TCB. Capabilities are accessed from the user side by specifying a path in this graph. The kernel internally uses references to CNodes with an index into the CNode to identify capabilities.

The following sections show basic manipulation of capabilities, resolving user-specified, path-based capability references into internal kernel references, transfer, revokation, deletion, and finally toplevel capability invocations.

30.1 Basic capability manipulation

Interpret a set of rights from a user data word.

```
definition
  data_to_rights :: "data ⇒ cap_rights" where
  "data_to_rights data ≡ let
    w = data_to_16 data
   in {x. case x of AllowWrite ⇒ w !! 0
                  | AllowRead ⇒ w !! 1
                  | AllowGrant ⇒ w !! 2}"
```

Check that a capability stored in a slot is not a parent of any other capability.

```
definition
  ensure_no_children :: "cslot_ptr ⇒ (unit,'z::state_ext) se_monad" where
  "ensure_no_children cslot_ptr ≡ doE
    cdt ← liftE $ gets cdt;
    whenE (∃c. cdt c = Some cslot_ptr) (throwError RevokeFirst)
  odE"

definition
  max_free_index :: "nat ⇒ nat" where
  "max_free_index magnitude_bits ≡ 2 ^ magnitude_bits"

definition
  free_index_update :: "(nat ⇒ nat) ⇒ cap ⇒ cap"
where
  "free_index_update g cap ≡
   case cap of UntypedCap ref sz f ⇒ UntypedCap ref sz (g f) | _ ⇒ cap"

primrec
  untyped_sz_bits :: "cap ⇒ nat"
```

```
where
  "untyped_sz_bits (UntypedCap ref sz f) = sz"

abbreviation
  max_free_index_update :: "cap ⇒ cap"
where
  "max_free_index_update cap ≡ cap (| free_index:= max_free_index (untyped_sz_bits cap) |)"

definition
  set_untyped_cap_as_full :: "cap ⇒ cap ⇒ word32 × bool list⇒ (unit,'z::state_ext) s_monad"
where
  "set_untyped_cap_as_full src_cap new_cap src_slot ≡
   if (is_untyped_cap src_cap ∧ is_untyped_cap new_cap
      ∧ obj_ref_of src_cap = obj_ref_of new_cap ∧ cap_bits_untyped src_cap = cap_bits_untyped
new_cap)
      then set_cap (max_free_index_update src_cap) src_slot else return ()"
```

Derive a cap into a form in which it can be copied. For internal reasons not all capability types can be copied at all times and not all capability types can be copied unchanged.

```
definition
derive_cap :: "cslot_ptr ⇒ cap ⇒ (cap,'z::state_ext) se_monad" where
"derive_cap slot cap ≡
 case cap of
    ArchObjectCap c ⇒ liftME ArchObjectCap $ arch_derive_cap c
    | UntypedCap ptr sz f ⇒ doE ensure_no_children slot; returnOk cap odE
    | Zombie ptr n sz ⇒ returnOk NullCap
    | ReplyCap ptr m ⇒ returnOk NullCap
    | IRQControlCap ⇒ returnOk NullCap
    | _ ⇒ returnOk cap"
```

Transform a capability on request from a user thread. The user-supplied argument word is interpreted differently for different cap types. If the preserve flag is set this transformation is being done in-place which means some changes are disallowed because they would invalidate existing CDT relationships.

```
definition
  update_cap_data :: "bool ⇒ data ⇒ cap ⇒ cap" where
"update_cap_data preserve w cap ≡
  if is_ep_cap cap then
    if cap_ep_badge cap = 0 ∧ ¬ preserve then
      badge_update w cap
    else NullCap
  else if is_aep_cap cap then
    if cap_ep_badge cap = 0 ∧ ¬ preserve then
      badge_update w cap
    else NullCap
  else if is_cnode_cap cap then
    let
        (oref, bits, guard) = the_cnode_cap cap;
        rights_bits = 3;
        guard_bits = 18;
        guard_size_bits = 5;
        guard_size' = unat ((w >> rights_bits) && mask guard_size_bits);
        guard'' = (w >> (rights_bits + guard_size_bits)) && mask guard_bits;
        guard' = drop (size guard'' - guard_size') (to_bl guard'')
    in
        if guard_size' + bits > word_bits
        then NullCap
        else CNodeCap oref bits guard'
```

```
  else if is_arch_cap cap then
    ArchObjectCap $ arch_update_cap_data w (the_arch_cap cap)
  else
    cap"
```

30.2 Resolving capability references

Recursively looks up a capability address to a CNode slot by walking over multiple CNodes until all the bits in the address are used or there are no further CNodes.

```
function resolve_address_bits' :: "'z itself ⇒ cap × cap_ref ⇒ (cslot_ptr × cap_ref,'z::state_ext)
lf_monad"
where
  "resolve_address_bits' z (cap, cref) =
  (case cap of
     CNodeCap oref radix_bits guard  ⇒
     if radix_bits + size guard = 0 then
       fail (* nothing is translated: table broken *)
     else doE
       whenE (¬ guard ≤ cref)
             (* guard does not match *)
             (throwError $ GuardMismatch (size cref) guard);

       whenE (size cref < radix_bits + size guard)
             (* not enough bits to resolve: table malformed *)
             (throwError $ DepthMismatch (size cref) (radix_bits+size guard));

       offset ← returnOk $ take radix_bits (drop (size guard) cref);
       rest ← returnOk $ drop (radix_bits + size guard) cref;
       if rest = [] then
         returnOk ((oref,offset), [])
       else doE
         next_cap ← liftE $ get_cap (oref, offset);
         if is_cnode_cap next_cap then
           resolve_address_bits' z (next_cap, rest)
         else
           returnOk ((oref,offset), rest)
       odE
     odE
   | _ ⇒ throwError InvalidRoot)"

lemma rab_termination:
  "∀cref guard radix_bits.
    ¬ length cref ≤ radix_bits + length guard ∧
    (0 < radix_bits ∨ guard ≠ []) ⟶
      length cref - (radix_bits + length guard) < length cref"

termination

definition resolve_address_bits where
"resolve_address_bits ≡ resolve_address_bits' TYPE('z::state_ext)"
```

Specialisations of the capability lookup process to various standard cases.

```
definition
  lookup_slot_for_thread :: "obj_ref ⇒ cap_ref ⇒ (cslot_ptr × cap_ref,'z::state_ext) lf_monad"
where
  "lookup_slot_for_thread thread cref ≡ doE
```

```
     tcb ← liftE $ gets_the $ get_tcb thread;
     resolve_address_bits (tcb_ctable tcb, cref)
  odE"
```

definition
```
  lookup_cap_and_slot :: "obj_ref ⇒ cap_ref ⇒ (cap × cslot_ptr,'z::state_ext) lf_monad" where
  "lookup_cap_and_slot thread cptr ≡ doE
      (slot, cr) ← lookup_slot_for_thread thread cptr;
      cap ← liftE $ get_cap slot;
      returnOk (cap, slot)
  odE"
```

definition
```
  lookup_cap :: "obj_ref ⇒ cap_ref ⇒ (cap,'z::state_ext) lf_monad" where
  "lookup_cap thread ref ≡ doE
     (ref', _) ← lookup_slot_for_thread thread ref;
     liftE $ get_cap ref'
   odE"
```

definition
```
  lookup_slot_for_cnode_op ::
  "bool ⇒ cap ⇒ cap_ref ⇒ nat ⇒ (cslot_ptr,'z::state_ext) se_monad"
where
 "lookup_slot_for_cnode_op is_source root ptr depth ≡
  if is_cnode_cap root then
  doE
    whenE (depth < 1 ∨ depth > word_bits)
      $ throwError (RangeError 1 (of_nat word_bits));
    lookup_error_on_failure is_source $ doE
      ptrbits_for_depth ← returnOk $ drop (length ptr - depth) ptr;
      (slot, rem) ← resolve_address_bits (root, ptrbits_for_depth);
      case rem of
        [] ⇒ returnOk slot
      | _  ⇒ throwError $ DepthMismatch (length rem) 0
    odE
  odE
  else
    throwError (FailedLookup is_source InvalidRoot)"
```

definition
```
  lookup_source_slot :: "cap ⇒ cap_ref ⇒ nat ⇒ (cslot_ptr,'z::state_ext) se_monad"
where
 "lookup_source_slot ≡ lookup_slot_for_cnode_op True"
```

definition
```
  lookup_target_slot :: "cap ⇒ cap_ref ⇒ nat ⇒ (cslot_ptr,'z::state_ext) se_monad"
where
 "lookup_target_slot ≡ lookup_slot_for_cnode_op False"
```

definition
```
  lookup_pivot_slot :: "cap ⇒ cap_ref ⇒ nat ⇒ (cslot_ptr,'z::state_ext) se_monad"
where
 "lookup_pivot_slot  ≡ lookup_slot_for_cnode_op True"
```

30.3 Transferring capabilities

These functions are used in interpreting from user arguments the manner in which a capability transfer should take place.

```
record captransfer =
  ct_receive_root :: cap_ref
  ct_receive_index :: cap_ref
  ct_receive_depth :: data


definition
  captransfer_size :: "nat" — in words
where
  "captransfer_size ≡ 3"

definition
  captransfer_from_words :: "word32 ⇒ (captransfer,'z::state_ext) s_monad"
where
  "captransfer_from_words ptr ≡ do
     w0 ← do_machine_op $ loadWord ptr;
     w1 ← do_machine_op $ loadWord (ptr + word_size);
     w2 ← do_machine_op $ loadWord (ptr + 2 * word_size);
     return (| ct_receive_root = data_to_cptr w0,
               ct_receive_index = data_to_cptr w1,
               ct_receive_depth = w2 |)
   od"


definition
  load_cap_transfer :: "obj_ref ⇒ (captransfer,'z::state_ext) s_monad" where
 "load_cap_transfer buffer ≡ do
     offset ← return $ msg_max_length + msg_max_extra_caps + 2;
     captransfer_from_words (buffer + of_nat offset * word_size)
  od"

fun
  get_receive_slots :: "obj_ref ⇒ obj_ref option ⇒
                        (cslot_ptr list,'z::state_ext) s_monad"
where
  "get_receive_slots thread (Some buffer) = do
     ct ← load_cap_transfer buffer;

     empty_on_failure $ doE
       cnode ← unify_failure $
                 lookup_cap thread (ct_receive_root ct);
       slot ← unify_failure $ lookup_target_slot cnode
                 (ct_receive_index ct) (unat (ct_receive_depth ct));

       cap ← liftE $ get_cap slot;

       whenE (cap ≠ NullCap) (throwError ());

       returnOk [slot]
     odE
   od"
| "get_receive_slots x None = return []"
```

30.4 Revoking and deleting capabilities

Deletion of the final capability to any object is a long running operation if the capability is of these types.

definition
```
  long_running_delete :: "cap ⇒ bool" where
 "long_running_delete cap ≡ case cap of
    CNodeCap ptr bits gd ⇒ True
  | Zombie ptr bits n ⇒ True
  | ThreadCap ptr ⇒ True
  | _ ⇒ False"
```

definition
```
  slot_cap_long_running_delete :: "cslot_ptr ⇒ (bool,'z::state_ext) s_monad"
where
  "slot_cap_long_running_delete slot ≡ do
     cap ← get_cap slot;
     case cap of
         NullCap ⇒ return False
       | _ ⇒ do
           final ← is_final_cap cap;
           return (final ∧ long_running_delete cap)
         od
   od"
```

Swap the contents of two capability slots. The capability parameters are the new states of the capabilities, as the user may request that the capabilities are transformed as they are swapped.

definition
```
  cap_swap :: "cap ⇒ cslot_ptr ⇒ cap ⇒ cslot_ptr ⇒ (unit,'z::state_ext) s_monad"
where
  "cap_swap cap1 slot1 cap2 slot2 ≡
  do
    set_cap cap2 slot1;
    set_cap cap1 slot2;
    slot1_p ← gets (λs. cdt s slot1);
    slot2_p ← gets (λs. cdt s slot2);
    cdt ← gets cdt;
    (* update children: *)
    cdt' ← return (λn. if cdt n = Some slot1
                      then Some slot2
                      else if cdt n = Some slot2
                      then Some slot1
                      else cdt n);
    (* update parents: *)
    set_cdt (cdt' (slot1 := cdt' slot2, slot2 := cdt' slot1));
    do_extended_op (cap_swap_ext slot1 slot2 slot1_p slot2_p);
    is_original ← gets is_original_cap;
    set_original slot1 (is_original slot2);
    set_original slot2 (is_original slot1)
  od"
```

Move a capability from one slot to another. Once again the new capability is a parameter as it may be transformed while it is moved.

definition
```
  cap_move :: "cap ⇒ cslot_ptr ⇒ cslot_ptr ⇒ (unit,'z::state_ext) s_monad"
where
```

```
"cap_move new_cap src_slot dest_slot ≡ do
  set_cap new_cap dest_slot;
  set_cap NullCap src_slot;
  src_p ← gets (λs. cdt s src_slot);
  dest_p ← gets (λs. cdt s dest_slot);
  cdt ← gets cdt;
  parent ← return $ cdt src_slot;
  cdt' ← return $ cdt(dest_slot := parent, src_slot := None);
  set_cdt (λr. if cdt' r = Some src_slot then Some dest_slot else cdt' r);
  do_extended_op (cap_move_ext src_slot dest_slot src_p dest_p);
  is_original ← gets is_original_cap;
  set_original dest_slot (is_original src_slot);
  set_original src_slot False
od"
```

This version of capability swap does not change the capabilities that are swapped, passing the existing capabilities to the more general function.

definition
```
  cap_swap_for_delete :: "cslot_ptr ⇒ cslot_ptr ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "cap_swap_for_delete slot1 slot2 ≡
  when (slot1 ≠ slot2) $ do
    cap1 ← get_cap slot1;
    cap2 ← get_cap slot2;
    cap_swap cap1 slot1 cap2 slot2
  od"
```

The type of possible recursive deletes.

datatype
```
  rec_del_call
  = CTEDeleteCall cslot_ptr bool
  | FinaliseSlotCall cslot_ptr bool
  | ReduceZombieCall cap cslot_ptr bool
```

Locate the nth capability beyond some base capability slot.

definition
```
  locate_slot :: "cslot_ptr ⇒ nat ⇒ cslot_ptr" where
 "locate_slot ≡ λ(a, b) n. (a, drop (32 - length b)
                          (to_bl (of_bl b + of_nat n :: word32)))"
```

Actions to be taken after deleting an IRQ Handler capability.

definition
```
  deleting_irq_handler :: "irq ⇒ (unit,'z::state_ext) s_monad"
```
where
```
 "deleting_irq_handler irq ≡ do
    slot ← get_irq_slot irq;
    cap_delete_one slot
  od"
```

Actions that must be taken when a capability is deleted. Returns a Zombie capability if deletion requires a long-running operation and also a possible IRQ to be cleared.

fun
```
  finalise_cap :: "cap ⇒ bool ⇒ (cap × irq option,'z::state_ext) s_monad"
```
where
```
  "finalise_cap NullCap                  final = return (NullCap, None)"
| "finalise_cap (UntypedCap r bits f)    final = return (NullCap, None)"
```

```
| "finalise_cap (ReplyCap r m)           final = return (NullCap, None)"
| "finalise_cap (EndpointCap r b R)      final =
      (liftM (K (NullCap, None)) $ when final $ ep_cancel_all r)"
| "finalise_cap (AsyncEndpointCap r b R) final =
      (liftM (K (NullCap, None)) $ when final $ aep_cancel_all r)"
| "finalise_cap (CNodeCap r bits g)  final =
      return (if final then Zombie r (Some bits) (2 ^ bits) else NullCap, None)"
| "finalise_cap (ThreadCap r)            final =
      do
         when final $ suspend r;
         return (if final then (Zombie r None 5) else NullCap, None)
      od"
| "finalise_cap DomainCap                final = return (NullCap, None)"
| "finalise_cap (Zombie r b n)           final =
      do assert final; return (Zombie r b n, None) od"
| "finalise_cap IRQControlCap            final = return (NullCap, None)"
| "finalise_cap (IRQHandlerCap irq)      final = (
       if final then do
         deleting_irq_handler irq;
         return (NullCap, Some irq)
       od
       else return (NullCap, None))"
| "finalise_cap (ArchObjectCap a)        final =
      (liftM (λx. (x, None)) $ arch_finalise_cap a final)"
```

definition
```
  can_fast_finalise :: "cap ⇒ bool" where
 "can_fast_finalise cap ≡ case cap of
    ReplyCap r m ⇒ True
  | EndpointCap r b R ⇒ True
  | AsyncEndpointCap r b R ⇒ True
  | NullCap ⇒ True
  | _ ⇒ False"
```

This operation is used to delete a capability when it is known that a long-running operation is impossible. It is equivalent to calling the regular finalisation operation. It cannot be defined in that way as doing so would create a circular definition.

lemma fast_finalise_def2:
```
  "fast_finalise cap final = do
     assert (can_fast_finalise cap);
     result ← finalise_cap cap final;
     assert (result = (NullCap, None))
   od"
```

The finalisation process on a Zombie or Null capability is finished for all Null capabilities and for Zombies that cover no slots or only the slot they are currently stored in.

fun
```
  cap_removeable :: "cap ⇒ cslot_ptr ⇒ bool"
```
where
```
  "cap_removeable NullCap slot = True"
| "cap_removeable (Zombie slot' bits n) slot =
    ((n = 0) ∨ (n = 1 ∧ (slot', replicate (zombie_cte_bits bits) False) = slot))"
```

Checks for Zombie capabilities that refer to the CNode or TCB they are stored in.

definition
```
  cap_cyclic_zombie :: "cap ⇒ cslot_ptr ⇒ bool" where
```

```
"cap_cyclic_zombie cap slot ≡ case cap of
        Zombie slot' bits n ⇒ (slot', replicate (zombie_cte_bits bits) False) = slot
      | _ ⇒ False"
```

The complete recursive delete operation.

```
function (sequential)
  rec_del :: "rec_del_call ⇒ (bool * irq option,'z::state_ext) p_monad"
where
  "rec_del (CTEDeleteCall slot exposed) s =
 (doE
    (success, irq_freed) ← rec_del (FinaliseSlotCall slot exposed);
    without_preemption $ when (exposed ∨ success) $ empty_slot slot irq_freed;
    returnOk undefined
  odE) s"
|
  "rec_del (FinaliseSlotCall slot exposed) s =
 (doE
    cap ← without_preemption $ get_cap slot;
    if (cap = NullCap)
    then returnOk (True, None)
    else (doE
      is_final ← without_preemption $ is_final_cap cap;
      (remainder, irqopt) ← without_preemption $ finalise_cap cap is_final;
      if (cap_removeable remainder slot)
      then returnOk (True, irqopt)
      else if (cap_cyclic_zombie remainder slot ∧ ¬ exposed)
      then doE
        without_preemption $ set_cap remainder slot;
        returnOk (False, None)
      odE
      else doE
        without_preemption $ set_cap remainder slot;
        rec_del (ReduceZombieCall remainder slot exposed);
        preemption_point;
        rec_del (FinaliseSlotCall slot exposed)
      odE
    odE)
  odE) s"

| "rec_del (ReduceZombieCall (Zombie ptr bits (Suc n)) slot False) s =
 (doE
    cn ← returnOk $ first_cslot_of (Zombie ptr bits (Suc n));
    assertE (cn ≠ slot);
    without_preemption $ cap_swap_for_delete cn slot;
    returnOk undefined
  odE) s"
|
 "rec_del (ReduceZombieCall (Zombie ptr bits (Suc n)) slot True) s =
 (doE
    end_slot ← returnOk (ptr, nat_to_cref (zombie_cte_bits bits) n);
    rec_del (CTEDeleteCall end_slot False);
    new_cap ← without_preemption $ get_cap slot;
    if (new_cap = Zombie ptr bits (Suc n))
    then without_preemption $ set_cap (Zombie ptr bits n) slot
    else assertE (new_cap = NullCap ∨
                  is_zombie new_cap ∧ first_cslot_of new_cap = slot
                   ∧ first_cslot_of (Zombie ptr bits (Suc n)) ≠ slot);
    returnOk undefined
```

```
  odE) s"
|
 "rec_del (ReduceZombieCall cap slot exposed) s =
  fail s"
```

Delete a capability by calling the recursive delete operation.

```
definition
  cap_delete :: "cslot_ptr ⇒ (unit,'z::state_ext) p_monad" where
 "cap_delete slot ≡ doE rec_del (CTEDeleteCall slot True); returnOk () odE"
```

Prepare the capability in a slot for deletion but do not delete it.

```
definition
  finalise_slot :: "cslot_ptr ⇒ bool ⇒ (bool * irq option,'z::state_ext) p_monad"
where
  "finalise_slot p e ≡ rec_del (FinaliseSlotCall p e)"
```

Helper functions for the type of recursive delete calls.

```
primrec
  exposed_rdcall :: "rec_del_call ⇒ bool"
where
  "exposed_rdcall (CTEDeleteCall slot exposed) = exposed"
| "exposed_rdcall (FinaliseSlotCall slot exposed) = exposed"
| "exposed_rdcall (ReduceZombieCall cap slot exposed) = exposed"

primrec
  isCTEDeleteCall :: "rec_del_call ⇒ bool"
where
  "isCTEDeleteCall (CTEDeleteCall slot exposed) = True"
| "isCTEDeleteCall (FinaliseSlotCall slot exposed) = False"
| "isCTEDeleteCall (ReduceZombieCall cap slot exposed) = False"

primrec
  slot_rdcall :: "rec_del_call ⇒ cslot_ptr"
where
  "slot_rdcall (CTEDeleteCall slot exposed) = slot"
| "slot_rdcall (FinaliseSlotCall slot exposed) = slot"
| "slot_rdcall (ReduceZombieCall cap slot exposed) = slot"
```

Revoke the derived capabilities of a given capability, deleting them all.

```
function cap_revoke :: "cslot_ptr ⇒ (unit,'z::state_ext) p_monad"
where
"cap_revoke slot s = (doE
    cap ← without_preemption $ get_cap slot;
    cdt ← without_preemption $ gets cdt;
    descendants ← returnOk $ descendants_of slot cdt;
    whenE (cap ≠ NullCap ∧ descendants ≠ {}) (doE
      child ← without_preemption $ select_ext (next_revoke_cap slot) descendants;
      cap ← without_preemption $ get_cap child;
      assertE (cap ≠ NullCap);
      cap_delete child;
      preemption_point;
      cap_revoke slot
    odE)
odE) s"
```

30.5 Inserting and moving capabilities

definition

```
  get_badge :: "cap ⇒ badge option" where
 "get_badge cap ≡ case cap of
    AsyncEndpointCap oref badge cr ⇒ Some badge
  | EndpointCap oref badge cr      ⇒ Some badge
  | _                              ⇒ None"
```

For some purposes capabilities to physical objects are treated differently to others.

definition

```
  arch_is_physical :: "arch_cap ⇒ bool" where
  "arch_is_physical cap ≡ case cap of ASIDControlCap ⇒ False | _ ⇒ True"
```

definition

```
  is_physical :: "cap ⇒ bool" where
  "is_physical cap ≡ case cap of
    NullCap ⇒ False
  | DomainCap ⇒ False
  | IRQControlCap ⇒ False
  | IRQHandlerCap _ ⇒ False
  | ReplyCap _ _ ⇒ False
  | ArchObjectCap c ⇒ arch_is_physical c
  | _ ⇒ True"
```

Check whether the second capability is to the same object or an object contained in the region of the first one.

fun

```
  arch_same_region_as :: "arch_cap ⇒ arch_cap ⇒ bool"
```

where

```
  "arch_same_region_as (PageCap r R s x) (PageCap r' R' s' x') =
   (let
     topA = r + (1 << pageBitsForSize s) - 1;
     topB = r' + (1 << pageBitsForSize s') - 1
   in r ≤ r' ∧ topA ≥ topB ∧ r' ≤ topB)"
| "arch_same_region_as (PageTableCap r x) (PageTableCap r' x') = (r' = r)"
| "arch_same_region_as (PageDirectoryCap r x) (PageDirectoryCap r' x') = (r' = r)"
| "arch_same_region_as ASIDControlCap ASIDControlCap = True"
| "arch_same_region_as (ASIDPoolCap r a) (ASIDPoolCap r' a') = (r' = r)"
| "arch_same_region_as _ _ = False"
```

fun

```
  same_region_as :: "cap ⇒ cap ⇒ bool"
```

where

```
  "same_region_as NullCap c' = False"
| "same_region_as (UntypedCap r bits free) c' =
    (is_physical c' ∧
     r ≤ obj_ref_of c' ∧
     obj_ref_of c' ≤ obj_ref_of c' + obj_size c' - 1 ∧
     obj_ref_of c' + obj_size c' - 1 ≤ r + (1 << bits) - 1)"
| "same_region_as (EndpointCap r b R) c' =
    (is_ep_cap c' ∧ obj_ref_of c' = r)"
| "same_region_as (AsyncEndpointCap r b R) c' =
    (is_aep_cap c' ∧ obj_ref_of c' = r)"
| "same_region_as (CNodeCap r bits g) c' =
    (is_cnode_cap c' ∧ obj_ref_of c' = r ∧ bits_of c' = bits)"
| "same_region_as (ReplyCap n m) c' = (∃m'. c' = ReplyCap n m')"
```

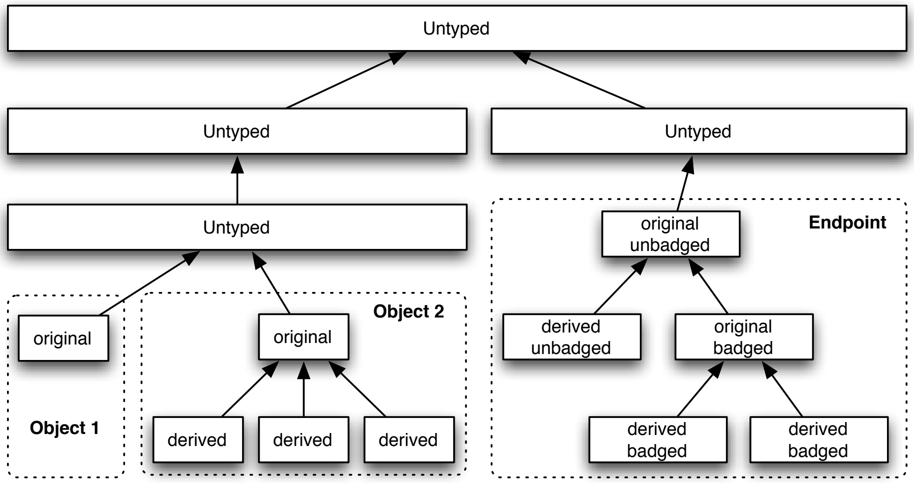


Figure 30.1: Example capability derivation tree.

```
| "same_region_as (ThreadCap r) c' =
    (is_thread_cap c' ∧ obj_ref_of c' = r)"
| "same_region_as (Zombie r b n) c' = False"
| "same_region_as (IRQControlCap) c' =
    (c' = IRQControlCap ∨ (∃n. c' = IRQHandlerCap n))"
| "same_region_as DomainCap c' = (c' = DomainCap)"
| "same_region_as (IRQHandlerCap n) c' =
    (c' = IRQHandlerCap n)"
| "same_region_as (ArchObjectCap a) c' =
    (case c' of ArchObjectCap a' ⇒ arch_same_region_as a a' | _ ⇒ False)"
```

Check whether two capabilities are to the same object.

```
definition
  same_object_as :: "cap ⇒ cap ⇒ bool" where
 "same_object_as cp cp' ≡
   (case (cp, cp') of
      (UntypedCap r bits free, _) ⇒ False
    | (IRQControlCap, IRQHandlerCap n) ⇒ False
    | (ArchObjectCap (PageCap ref _ pgsz _),
       ArchObjectCap (PageCap ref' _ pgsz' _))
          ⇒ (ref, pgsz) = (ref', pgsz')
              ∧ ref ≤ ref + 2 ^ pageBitsForSize pgsz - 1
    | _ ⇒ same_region_as cp cp')"
```

The function `should_be_parent_of` checks whether an existing capability should be a parent of another to-be-inserted capability. The test is the following: For capability `c` to be a parent of capability `c'`, `c` needs to be the original capability to the object and needs to cover the same memory region as `c'` (i.e. cover the same object). In the case of endpoint capabilities, if `c` is a badged endpoint cap (`badge ≠ 0`), then it should be a parent of `c'` if `c'` has the same badge and is itself not an original badged endpoint cap.

Figure 30.1 shows an example capability derivation tree that illustrates a standard scenario: the top level is a large untyped capability, the second level splits this capability into two regions covered by their own untyped caps, both are children of the first level. The third level on the left is a copy of the level 2 untyped capability. Untyped capabilities when copied always create children, never siblings. In this scenario, the untyped capability was typed into two separate objects, creating two capabilities on level 4, both are the original capability to the respective object, both are children of the untyped capability they were created from.

Ordinary original capabilities can have one level of derived capabilities (created, for instance, by the copy or mint operations). Further copies of these derived capabilities will create sibling, in this case remaining on level 5. There is an exception to this scheme for endpoint capabilities — they support an additional layer of depth with the concept of badged and unbadged endpoints. The original endpoint capability will be unbadged. Using the mint operation, a copy of the capability with a specific badge can be created. This new, badged capability to the same object is treated as an original capability (the "original badged endpoint capability") and supports one level of derived children like other capabilities.

definition

```
should_be_parent_of :: "cap ⇒ bool ⇒ cap ⇒ bool ⇒ bool" where
"should_be_parent_of c original c' original' ≡
 original ∧
 same_region_as c c' ∧
 (case c of
    EndpointCap ref badge R ⇒ badge ≠ 0 ⟶ cap_ep_badge c' = badge ∧ ¬original'
  | AsyncEndpointCap ref badge R ⇒ badge ≠ 0 ⟶ cap_ep_badge c' = badge ∧ ¬original'
  | _ ⇒ True)"
```

Insert a new capability as either a sibling or child of an existing capability. The function `should_be_parent_of` determines which it will be.

The term for `dest_original` determines if the new capability should be counted as the original capability to the object. This test is usually false, apart from the exceptions listed (newly badged endpoint capabilities, irq handlers, and untyped caps).

definition

```
cap_insert :: "cap ⇒ cslot_ptr ⇒ cslot_ptr ⇒ (unit,'z::state_ext) s_monad" where
"cap_insert new_cap src_slot dest_slot ≡ do
  src_cap ← get_cap src_slot;

  dest_original ← return (if is_ep_cap new_cap then
                             cap_ep_badge new_cap ≠ cap_ep_badge src_cap
                          else if is_aep_cap new_cap then
                             cap_ep_badge new_cap ≠ cap_ep_badge src_cap
                          else if ∃irq. new_cap = IRQHandlerCap irq then
                             src_cap = IRQControlCap
                          else is_untyped_cap new_cap);

  old_cap ← get_cap dest_slot;
  assert (old_cap = NullCap);
  set_untyped_cap_as_full src_cap new_cap src_slot;
  set_cap new_cap dest_slot;

  is_original ← gets is_original_cap;
  src_parent ← return $
     should_be_parent_of src_cap (is_original src_slot) new_cap dest_original;
  src_p ← gets (λs. cdt s src_slot);
  dest_p ← gets (λs. cdt s dest_slot);
  update_cdt (λcdt. cdt (dest_slot := if src_parent
                                      then Some src_slot
                                      else cdt src_slot));
  do_extended_op (cap_insert_ext src_parent src_slot dest_slot src_p dest_p);
  set_original dest_slot dest_original
od"
```

30.6 Recycling capabilities

Overwrite the capabilities stored in a TCB while preserving the register set and other fields.

```
definition
  tcb_registers_caps_merge :: "tcb ⇒ tcb ⇒ tcb"
where
 "tcb_registers_caps_merge regtcb captcb ≡
  regtcb ⦇ tcb_ctable := tcb_ctable captcb,
           tcb_vtable := tcb_vtable captcb,
           tcb_reply := tcb_reply captcb,
           tcb_caller := tcb_caller captcb,
           tcb_ipcframe := tcb_ipcframe captcb ⦈"
```

Restore a finalised capability to its original form and also restore some aspects of the associated object to their original state.

```
definition
  recycle_cap :: "bool ⇒ cap ⇒ (cap,'z::state_ext) s_monad" where
  "recycle_cap is_final cap ≡
  case cap of
    NullCap ⇒ fail
  | DomainCap ⇒ return cap
  | Zombie ptr tp n ⇒
    (case tp of
          None ⇒ do
            st ← get_thread_state ptr;
            assert (st = Inactive);
            thread_set (tcb_registers_caps_merge default_tcb) ptr;
            do_extended_op (recycle_cap_ext ptr);
            return $ ThreadCap ptr
          od
        | Some sz ⇒ return $ CNodeCap ptr sz [])
  | EndpointCap ep b _ ⇒
    do
      when (b ≠ 0) $ ep_cancel_badged_sends ep b;
      return cap
    od
  | ArchObjectCap c ⇒ liftM ArchObjectCap $ arch_recycle_cap is_final c
  | _ ⇒ return cap"
```

Recycle the capability stored in a slot, including finalising it as though it were to be deleted and then restoring it to its original state.

```
definition
  cap_recycle :: "cslot_ptr ⇒ (unit,'z::state_ext) p_monad" where
  "cap_recycle slot ≡ doE
    cap_revoke slot;
    finalise_slot slot True;
    without_preemption $ do
        cap ← get_cap slot;
        unless (cap = NullCap) $ do
            is_final' ← is_final_cap cap;
            cap' ← recycle_cap is_final' cap;
            set_cap cap' slot
        od
    od
  odE"
```

Only caps with sufficient rights can be recycled.

definition
```
has_recycle_rights :: "cap ⇒ bool" where
"has_recycle_rights cap ≡ case cap of
   NullCap ⇒ False
 | DomainCap ⇒ False
 | EndpointCap _ _ R ⇒ R = all_rights
 | AsyncEndpointCap _ _ R ⇒ {AllowRead,AllowWrite} ⊆ R
 | ArchObjectCap (PageCap _ R _ _) ⇒ {AllowRead,AllowWrite} ⊆ R
 | _ ⇒ True"
```

30.7 Invoking CNode capabilities

The CNode capability confers authority to various methods which act on CNodes and the capabilities within them. Copies of capabilities may be inserted in empty CNode slots by Insert. Capabilities may be moved to empty slots with Move or swapped with others in a three way rotate by Rotate. A Reply capability stored in a thread's last-caller slot may be saved into a regular CNode slot with Save. The Revoke, Delete and Recycle methods may also be invoked on the capabilities stored in the CNode.

definition
```
invoke_cnode :: "cnode_invocation ⇒ (unit,'z::state_ext) p_monad" where
"invoke_cnode i ≡ case i of
  RevokeCall dest_slot ⇒ cap_revoke dest_slot
| DeleteCall dest_slot ⇒ cap_delete dest_slot
| InsertCall cap src_slot dest_slot ⇒
     without_preemption $ cap_insert cap src_slot dest_slot
| MoveCall cap src_slot dest_slot ⇒
     without_preemption $ cap_move cap src_slot dest_slot
| RotateCall cap1 cap2 slot1 slot2 slot3 ⇒
     without_preemption $
     if slot1 = slot3 then
       cap_swap cap1 slot1 cap2 slot2
     else
       do cap_move cap2 slot2 slot3; cap_move cap1 slot1 slot2 od
| SaveCall slot ⇒ without_preemption $ do
  thread ← gets cur_thread;
  src_slot ← return (thread, tcb_cnode_index 3);
  cap ← get_cap src_slot;
  (case cap of
        NullCap ⇒ return ()
      | ReplyCap _ False ⇒ cap_move cap src_slot slot
      | _ ⇒ fail) od
| RecycleCall slot ⇒ cap_recycle slot"
```

end

31 Toplevel ARM Definitions

theory Arch_A
imports CSpace_A
begin

definition "page_bits ≡ pageBits"

The ARM architecture does not provide any additional operations on its interrupt controller.

```
definition
  arch_invoke_irq_control :: "arch_interrupt_control ⇒ (unit,'z::state_ext) p_monad" where
  "arch_invoke_irq_control aic ≡ fail"
```

Switch to a thread's virtual address space context and write its IPC buffer pointer into the globals frame. Clear the load-exclusive monitor.

```
definition
  arch_switch_to_thread :: "obj_ref ⇒ (unit,'z::state_ext) s_monad" where
  "arch_switch_to_thread t ≡ do
     set_vm_root t;
     globals ← gets (arm_globals_frame ∘ arch_state);
     buffer_ptr ← thread_get tcb_ipc_buffer t;
     do_machine_op $ storeWord globals buffer_ptr;
     do_machine_op $ clearExMonitor
   od"
```

The idle thread does not need to be handled specially on ARM.

```
definition
   arch_switch_to_idle_thread :: "(unit,'z::state_ext) s_monad" where
   "arch_switch_to_idle_thread ≡ do
      globals ← gets (arm_globals_frame ∘ arch_state);
      do_machine_op $ storeWord globals 0
    od"
```

```
definition
  arch_activate_idle_thread :: "obj_ref ⇒ (unit,'z::state_ext) s_monad" where
  "arch_activate_idle_thread t ≡ return ()"
```

The ASIDControl capability confers the authority to create a new ASID pool object. This operation creates the new ASID pool, provides a capability to it and connects it to the global virtual ASID table.

```
definition
perform_asid_control_invocation :: "asid_control_invocation ⇒ (unit,'z::state_ext) s_monad" where
"perform_asid_control_invocation iv ≡ case iv of
  MakePool frame slot parent base ⇒ do
    delete_objects frame page_bits;
    pcap ← get_cap parent;
    set_cap (max_free_index_update pcap) parent;
    retype_region frame 1 0 (ArchObject ASIDPoolObj);
    cap_insert (ArchObjectCap $ ASIDPoolCap frame base) parent slot;
    assert (base && mask asid_low_bits = 0);
    asid_table ← gets (arm_asid_table ∘ arch_state);
```

```
    asid_table' ← return (asid_table (asid_high_bits_of base ↦ frame));
    modify (λs. s (|arch_state := (arch_state s) (|arm_asid_table := asid_table'|)|))
od"
```

The ASIDPool capability confers the authority to assign a virtual ASID to a page directory.

definition
```
perform_asid_pool_invocation :: "asid_pool_invocation ⇒ (unit,'z::state_ext) s_monad" where
"perform_asid_pool_invocation iv ≡ case iv of Assign asid pool_ptr ct_slot ⇒
do
    pd_cap ← get_cap ct_slot;
    case pd_cap of
      ArchObjectCap (PageDirectoryCap pd_base _) ⇒ do
        pool ← get_asid_pool pool_ptr;
        pool' ← return (pool (ucast asid ↦ pd_base));
        set_cap (ArchObjectCap $ PageDirectoryCap pd_base (Some asid)) ct_slot;
        set_asid_pool pool_ptr pool'
      od
    | _ ⇒ fail
od"
```

The PageDirectory capability confers the authority to flush cache entries associated with that PD

definition
```
  perform_page_directory_invocation :: "page_directory_invocation ⇒ (unit,'z::state_ext) s_monad"
```

where
```
  "perform_page_directory_invocation iv ≡ case iv of
       PageDirectoryFlush typ start end pstart pd asid ⇒
         when (start < end) $ do
           root_switched ← set_vm_root_for_flush pd asid;
           do_machine_op $ do_flush typ start end pstart;
           when root_switched $ do
             tcb ← gets cur_thread;
             set_vm_root tcb
           od
        od
     | PageDirectoryNothing ⇒ return ()"
```

definition
```
  pte_check_if_mapped :: "32 word ⇒ (bool, 'z::state_ext) s_monad"
```
where
```
  "pte_check_if_mapped slot ≡ do
     pt ← get_master_pte slot;
     return (pt ≠ InvalidPTE)
  od"
```

definition
```
  pde_check_if_mapped :: "32 word ⇒ (bool, 'z::state_ext) s_monad"
```
where
```
  "pde_check_if_mapped slot ≡ do
     pd ← get_master_pde slot;
     return (pd ≠ InvalidPDE)
  od"
```

A pointer is inside a user frame if its top bits point to a `DataPage`.

definition
```
  in_user_frame :: "word32 ⇒ 'z::state_ext state ⇒ bool" where
  "in_user_frame p s ≡
```

```
  ∃sz. kheap s (p && ~~ mask (pageBitsForSize sz)) =
       Some (ArchObj (DataPage sz))"
```

Store or load a word at an offset from an IPC buffer.

definition
```
  store_word_offs :: "obj_ref ⇒ nat ⇒ machine_word ⇒ (unit,'z::state_ext) s_monad" where
 "store_word_offs ptr offs v ≡
    do s ← get;
       assert (in_user_frame (ptr + of_nat (offs * word_size)) s);
       do_machine_op $ storeWord (ptr + of_nat (offs * word_size)) v
    od"
```

Set the message registers of a thread.

definition
```
  set_mrs :: "obj_ref ⇒ obj_ref option ⇒ message list ⇒ (length_type,'z::state_ext) s_monad"
where
  "set_mrs thread buf msgs ≡
   do
     tcb ← gets_the $ get_tcb thread;
     context ← return (tcb_context tcb);
     new_regs ← return (λreg. if reg ∈ set (take (length msgs) msg_registers)
                              then msgs ! (the_index msg_registers reg)
                              else context reg);
     set_object thread (TCB (tcb ⦇ tcb_context := new_regs ⦈));
     remaining_msgs ← return (drop (length msg_registers) msgs);
     case buf of
     None      ⇒ return $ nat_to_len (min (length msg_registers) (length msgs))
   | Some pptr ⇒ do
       zipWithM_x (λx. store_word_offs pptr x)
          [length msg_registers + 1 ..< Suc msg_max_length] remaining_msgs;
       return $ nat_to_len $ min (length msgs) msg_max_length
     od
   od"
```

definition
```
  set_message_info :: "obj_ref ⇒ message_info ⇒ (unit,'z::state_ext) s_monad"
where
  "set_message_info thread info ≡
     as_user thread $ set_register msg_info_register $
                      message_info_to_data info"
```

The Page capability confers the authority to map, unmap and flush the memory page. The remap system call is a convenience operation that ensures the page is mapped in the same location as this cap was previously used to map it in.

definition
```
perform_page_invocation :: "page_invocation ⇒ (unit,'z::state_ext) s_monad" where
"perform_page_invocation iv ≡ case iv of
  PageMap asid cap ct_slot entries ⇒ do
    set_cap cap ct_slot;
    case entries of
          Inl (pte, slots) ⇒ do
            flush ← pte_check_if_mapped (hd slots);
            store_pte (hd slots) pte;
            mapM (swp store_pte InvalidPTE) (tl slots);
            do_machine_op $ cleanCacheRange_PoU (hd slots) (last_byte_pte (last slots))
                                                (addrFromPPtr (hd slots));
            if flush then (invalidate_tlb_by_asid asid) else return ()
```

```
          od
        | Inr (pde, slots) ⇒ do
            flush ← pde_check_if_mapped (hd slots);
            store_pde (hd slots) pde;
            mapM (swp store_pde InvalidPDE) (tl slots);
            do_machine_op $ cleanCacheRange_PoU (hd slots) (last_byte_pde (last slots))
                                                  (addrFromPPtr (hd slots));
            if flush then (invalidate_tlb_by_asid asid) else return ()
        od
    od
| PageRemap asid (Inl (pte, slots)) ⇒ do
    flush ← pte_check_if_mapped (hd slots);
    store_pte (hd slots) pte;
    mapM_x (swp store_pte InvalidPTE) (tl slots);
    do_machine_op $ cleanCacheRange_PoU (hd slots) (last_byte_pte (last slots))
                                          (addrFromPPtr (hd slots));
    if flush then (invalidate_tlb_by_asid asid) else return ()
  od
| PageRemap asid (Inr (pde, slots)) ⇒ do
    flush ← pde_check_if_mapped (hd slots);
    store_pde (hd slots) pde;
    mapM_x (swp store_pde InvalidPDE) (tl slots);
    do_machine_op $ cleanCacheRange_PoU (hd slots) (last_byte_pde (last slots))
                                          (addrFromPPtr (hd slots));
    if flush then (invalidate_tlb_by_asid asid) else return ()
  od
| PageUnmap cap ct_slot ⇒
    case cap of
      PageCap p R vp_size vp_mapped_addr ⇒ do
        case vp_mapped_addr of
            Some (asid, vaddr) ⇒ unmap_page vp_size asid vaddr p
          | None ⇒ return ();
        cap ← liftM the_arch_cap $ get_cap ct_slot;
        set_cap (ArchObjectCap $ update_map_data cap None) ct_slot
      od
    | _ ⇒ fail
| PageFlush typ start end pstart pd asid ⇒
    when (start < end) $ do
      root_switched ← set_vm_root_for_flush pd asid;
      do_machine_op $ do_flush typ start end pstart;
      when root_switched $ do
        tcb ← gets cur_thread;
        set_vm_root tcb
      od
   od
| PageGetAddr ptr ⇒ do
    ct ← gets cur_thread;
    n_msg ← set_mrs ct None [addrFromPPtr ptr];
    set_message_info ct $ MI n_msg 0 0 0
  od"
```

PageTable capabilities confer the authority to map and unmap page tables.

definition

```
perform_page_table_invocation :: "page_table_invocation ⇒ (unit,'z::state_ext) s_monad" where
"perform_page_table_invocation iv ≡
case iv of PageTableMap cap ct_slot pde pd_slot ⇒ do
    set_cap cap ct_slot;
    store_pde pd_slot pde;
```

```
    do_machine_op $ cleanByVA_PoU pd_slot (addrFromPPtr pd_slot)
  od
  | PageTableUnmap (ArchObjectCap (PageTableCap p mapped_address)) ct_slot ⇒ do
    case mapped_address of Some (asid, vaddr) ⇒ do
      unmap_page_table asid vaddr p;
      pte_bits ← return 2;
      slots ← return [p, p + (1 << pte_bits) .e. p + (1 << pt_bits) - 1];
      mapM_x (swp store_pte InvalidPTE) slots;
      do_machine_op $ cleanCacheRange_PoU p (p + (1 << pt_bits) - 1)
                                          (addrFromPPtr p)
    od | None ⇒ return ();
    cap ← liftM the_arch_cap $ get_cap ct_slot;
    set_cap (ArchObjectCap $ update_map_data cap None) ct_slot
  od
  | _ ⇒ fail"
```

Top level system call despatcher for all ARM-specific system calls.

definition

```
  arch_perform_invocation :: "arch_invocation ⇒ (data list,'z::state_ext) p_monad" where
  "arch_perform_invocation i ≡ liftE $ do
    case i of
          InvokePageTable oper ⇒ perform_page_table_invocation oper
        | InvokePageDirectory oper ⇒ perform_page_directory_invocation oper
        | InvokePage oper ⇒ perform_page_invocation oper
        | InvokeASIDControl oper ⇒ perform_asid_control_invocation oper
        | InvokeASIDPool oper ⇒ perform_asid_pool_invocation oper;
    return $ []
od"
```

end

32 Scheduler

```
theory Schedule_A
imports Arch_A
begin

abbreviation
  "idle st ≡ st = Structures_A.IdleThreadState"
```

Gets the TCB at an address if the thread can be scheduled.

```
definition
  getActiveTCB :: "obj_ref ⇒ 'z::state_ext state ⇒ tcb option"
where
  "getActiveTCB tcb_ref state ≡
   case (get_tcb tcb_ref state)
     of None           ⇒ None
      | Some tcb       ⇒ if (runnable $ tcb_state tcb)
                          then Some tcb else None"
```

Gets all schedulable threads in the system.

```
definition
  allActiveTCBs :: "(obj_ref set,'z::state_ext) s_monad" where
  "allActiveTCBs ≡ do
    state ← get;
    return {x. getActiveTCB x state ≠ None}
   od"
```

Switches the current thread to the specified one.

```
definition
  switch_to_thread :: "obj_ref ⇒ (unit,'z::state_ext) s_monad" where
  "switch_to_thread t ≡ do
     state ← get;
     assert (get_tcb t state ≠ None);
     arch_switch_to_thread t;
     do_extended_op (tcb_sched_action (tcb_sched_dequeue) t);
     modify (λs. s ⦇ cur_thread := t ⦈)
   od"
```

Asserts that a thread is runnable before switching to it.

```
definition guarded_switch_to :: "obj_ref ⇒ (unit,'z::state_ext) s_monad" where
"guarded_switch_to thread ≡ do ts ← get_thread_state thread;
                    assert (runnable ts);
                    switch_to_thread thread
                 od"
```

Switches to the idle thread.

```
definition
  switch_to_idle_thread :: "(unit,'z::state_ext) s_monad" where
  "switch_to_idle_thread ≡ do
     thread ← gets idle_thread;
     arch_switch_to_idle_thread;
```

```
    modify (λs. s (| cur_thread := thread |))
  od"
```

class state_ext_sched = state_ext +
  fixes schedule :: "(unit,'a) s_monad"

definition choose_thread :: "det_ext state ⇒ (unit × det_ext state) set × bool" **where**

```
"choose_thread ≡
      do
        d ← gets cur_domain;
        queues ← gets (λs. ready_queues s d);
        if (∀prio. queues prio = []) then (switch_to_idle_thread)
        else (guarded_switch_to (hd (max_non_empty_queue queues)))
      od"
```

instantiation det_ext_ext :: (type) state_ext_sched
begin

definition "schedule_det_ext_ext ≡ do

```
     cur ← gets cur_thread;
     cur_ts ← get_thread_state cur;
     action ← gets scheduler_action;
     (case action of
       resume_cur_thread ⇒ do
                            id ← gets idle_thread;
                            assert (runnable cur_ts ∨ cur = id);
                            return ()
                           od |
       choose_new_thread ⇒ do
         when (runnable cur_ts) ((tcb_sched_action tcb_sched_enqueue cur));
         dom_time ← gets domain_time;
         when (dom_time = 0) next_domain;
         choose_thread;
         (set_scheduler_action resume_cur_thread) od |
       switch_thread t ⇒ do
         when (runnable cur_ts) ((tcb_sched_action tcb_sched_enqueue cur));
         guarded_switch_to t;
         (set_scheduler_action resume_cur_thread) od)
    od"
```

instance
end

instantiation unit :: state_ext_sched
begin

The scheduler is heavily underspecified. It is allowed to pick any active thread or the idle thread. If the thread the scheduler picked is the current thread, it may omit the call to switch_to_thread.

definition schedule_unit :: "(unit,unit) s_monad" **where**

```
"schedule_unit ≡ do
   cur ← gets cur_thread;
   threads ← allActiveTCBs;
   thread ← select threads;
     if thread = cur then
     return () OR switch_to_thread thread
   else
```

```
    switch_to_thread thread
od OR
switch_to_idle_thread"
```

instance
end

lemmas `schedule_def = schedule_det_ext_ext_def schedule_unit_def`

end

33 Threads and TCBs

theory Tcb_A
imports Schedule_A
begin

33.1 Activating Threads

Threads that are active always have a master Reply capability to themselves stored in their reply slot. This is so that a derived Reply capability can be generated immediately if they wish to issue one. This function sets up a new master Reply capability if one does not exist.

definition
```
  "setup_reply_master thread ≡ do
     old_cap <- get_cap (thread, tcb_cnode_index 2);
     when (old_cap = NullCap) $ do
         set_original (thread, tcb_cnode_index 2) True;
         set_cap (ReplyCap thread True) (thread, tcb_cnode_index 2)
     od
  od"
```

Reactivate a thread if it is not already running.

definition
```
  restart :: "obj_ref ⇒ (unit,'z::state_ext) s_monad" where
 "restart thread ≡ do
    state ← get_thread_state thread;
    when (¬ runnable state ∧ ¬ idle state) $ do
      ipc_cancel thread;
      setup_reply_master thread;
      set_thread_state thread Restart;
      do_extended_op (tcb_sched_action (tcb_sched_enqueue) thread);
      do_extended_op (switch_if_required_to thread)
    od
  od"
```

This action is performed at the end of a system call immediately before control is restored to a used thread. If it needs to be restarted then its program counter is set to the operation it was performing rather than the next operation. The idle thread is handled specially.

definition
```
  activate_thread :: "(unit,'z::state_ext) s_monad" where
  "activate_thread ≡ do
     thread ← gets cur_thread;
     state ← get_thread_state thread;
     (case state
       of Running ⇒ return ()
       | Restart ⇒ (do
           pc ← as_user thread getRestartPC;
           as_user thread $ setNextPC pc;
           set_thread_state thread Running
         od)
       | IdleThreadState ⇒ arch_activate_idle_thread thread
```

```
      | _ ⇒ fail)
  od"
```

33.2 Thread Message Formats

The number of message registers in a maximum length message.

definition
```
  number_of_mrs :: nat where
 "number_of_mrs ≡ 32"
```

The size of a user IPC buffer.

definition
```
  ipc_buffer_size :: vspace_ref where
 "ipc_buffer_size ≡ of_nat ((number_of_mrs + captransfer_size) * word_size)"
```

definition
```
  load_word_offs :: "obj_ref ⇒ nat ⇒ (machine_word,'z::state_ext) s_monad" where
 "load_word_offs ptr offs ≡
    do_machine_op $ loadWord (ptr + of_nat (offs * word_size))"
```
definition
```
  load_word_offs_word :: "obj_ref ⇒ data ⇒ (machine_word,'z::state_ext) s_monad" where
 "load_word_offs_word ptr offs ≡
    do_machine_op $ loadWord (ptr + (offs * word_size))"
```

Get all of the message registers, both from the sending thread's current register file and its IPC buffer.

definition
```
  get_mrs :: "obj_ref ⇒ obj_ref option ⇒ message_info ⇒
             (message list,'z::state_ext) s_monad" where
  "get_mrs thread buf info ≡ do
     context ← thread_get tcb_context thread;
     cpu_mrs ← return (map context msg_registers);
     buf_mrs ← case buf
       of None      ⇒ return []
        | Some pptr ⇒ mapM (λx. load_word_offs pptr x)
               [length msg_registers + 1 ..< Suc msg_max_length];
     return (take (unat (mi_length info)) $ cpu_mrs @ buf_mrs)
   od"
```

Copy message registers from one thread to another.

definition
```
  copy_mrs :: "obj_ref ⇒ obj_ref option ⇒ obj_ref ⇒
              obj_ref option ⇒ length_type ⇒ (length_type,'z::state_ext) s_monad" where
  "copy_mrs sender sbuf receiver rbuf n ≡
   do
     hardware_mrs ← return $ take (unat n) msg_registers;
     mapM (λr. do
         v ← as_user sender $ get_register r;
         as_user receiver $ set_register r v
       od) hardware_mrs;
     buf_mrs ← case (sbuf, rbuf) of
       (Some sb_ptr, Some rb_ptr) ⇒ mapM (λx. do
                                      v ← load_word_offs sb_ptr x;
                                      store_word_offs rb_ptr x v
                                    od)
               [length msg_registers + 1 ..< Suc (unat n)]
     | _ ⇒ return [];
```

```
    return $ min n $ nat_to_len $ length hardware_mrs + length buf_mrs
  od"
```

The ctable and vtable slots of the TCB.

definition

```
get_tcb_ctable_ptr :: "obj_ref ⇒ cslot_ptr" where
"get_tcb_ctable_ptr tcb_ref ≡ (tcb_ref, tcb_cnode_index 0)"
```

definition

```
get_tcb_vtable_ptr :: "obj_ref ⇒ cslot_ptr" where
"get_tcb_vtable_ptr tcb_ref ≡ (tcb_ref, tcb_cnode_index 1)"
```

Copy a set of registers from a thread to memory and vice versa.

definition

```
copyRegsToArea :: "register list ⇒ obj_ref ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad" where
"copyRegsToArea regs thread ptr ≡ do
   context ← thread_get tcb_context thread;
   zipWithM_x (store_word_offs ptr)
     [0 ..< length regs]
     (map context regs)
od"
```

definition

```
copyAreaToRegs :: "register list ⇒ obj_ref ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad" where
"copyAreaToRegs regs ptr thread ≡ do
   old_regs ← thread_get tcb_context thread;
   vals ← mapM (load_word_offs ptr) [0 ..< length regs];
   vals2 ← return $ zip vals regs;
   vals3 ← return $ map (λ(v, r). (sanitiseRegister r v, r)) vals2;
   new_regs ← return $ foldl (λrs (v, r). rs ( r := v )) old_regs vals3;
   thread_set (λtcb. tcb (| tcb_context := new_regs |)) thread
 od"
```

Optionally update the tcb at an address.

definition

```
  option_update_thread :: "obj_ref ⇒ ('a ⇒ tcb ⇒ tcb) ⇒ 'a option ⇒ (unit,'z::state_ext) s_monad"
where
 "option_update_thread thread fn ≡ option_case (return ()) (λv. thread_set (fn v) thread)"
```

Check that a related capability is at an address. This is done before calling `cap_insert` to avoid a corner case where the would-be parent of the cap to be inserted has been moved or deleted.

definition

```
  check_cap_at :: "cap ⇒ cslot_ptr ⇒ (unit,'z::state_ext) s_monad ⇒ (unit,'z::state_ext) s_monad"
where
 "check_cap_at cap slot m ≡ do
    cap' ← get_cap slot;
    when (same_object_as cap cap') m
  od"
```

TCB capabilities confer authority to perform seven actions. A thread can request to yield its timeslice to another, to suspend or resume another, to reconfigure another thread, or to copy register sets into, out of or between other threads.

fun

```
  invoke_tcb :: "tcb_invocation ⇒ (data list,'z::state_ext) p_monad"
where
  "invoke_tcb (Suspend thread) = liftE (do suspend thread; return [] od)"
```

```
| "invoke_tcb (Resume thread) = liftE (do restart thread; return [] od)"

| "invoke_tcb (ThreadControl target slot faultep priority croot vroot buffer)
   = doE
    liftE $ option_update_thread target (tcb_fault_handler_update o K) faultep;
    liftE $ case priority of None ⇒ return()
     | Some prio ⇒ do_extended_op (set_priority target prio);
    (case croot of None ⇒ returnOk ()
     | Some (new_cap, src_slot) ⇒ doE
      cap_delete (target, tcb_cnode_index 0);
      liftE $ check_cap_at new_cap src_slot
            $ check_cap_at (ThreadCap target) slot
            $ cap_insert new_cap src_slot (target, tcb_cnode_index 0)
    odE);
    (case vroot of None ⇒ returnOk ()
     | Some (new_cap, src_slot) ⇒ doE
      cap_delete (target, tcb_cnode_index 1);
      liftE $ check_cap_at new_cap src_slot
            $ check_cap_at (ThreadCap target) slot
            $ cap_insert new_cap src_slot (target, tcb_cnode_index 1)
    odE);
    (case buffer of None ⇒ returnOk ()
     | Some (ptr, frame) ⇒ doE
      cap_delete (target, tcb_cnode_index 4);
      liftE $ thread_set (λt. t (| tcb_ipc_buffer := ptr |)) target;
      liftE $ case frame of None ⇒ return ()
       | Some (new_cap, src_slot) ⇒
            check_cap_at new_cap src_slot
          $ check_cap_at (ThreadCap target) slot
          $ cap_insert new_cap src_slot (target, tcb_cnode_index 4)
    odE);
    returnOk []
  odE"

| "invoke_tcb (CopyRegisters dest src suspend_source resume_target transfer_frame transfer_integer
transfer_arch) =
  (liftE $ do
    when suspend_source $ suspend src;
    when resume_target $ restart dest;
    when transfer_frame $ do
        mapM_x (λr. do
                v ← as_user src $ getRegister r;
                as_user dest $ setRegister r v
        od) frame_registers;
        pc ← as_user dest getRestartPC;
        as_user dest $ setNextPC pc
    od;
    when transfer_integer $
        mapM_x (λr. do
                v ← as_user src $ getRegister r;
                as_user dest $ setRegister r v
        od) gpRegisters;
    return []
  od)"

| "invoke_tcb (ReadRegisters src suspend_source n arch) =
  (liftE $ do
    when suspend_source $ suspend src;
```

```
    self ← gets cur_thread;
    regs ← return (take (unat n) $ frame_registers @ gp_registers);
    as_user src $ mapM getRegister regs
  od)"

| "invoke_tcb (WriteRegisters dest resume_target values arch) =
  (liftE $ do
    self ← gets cur_thread;
    as_user dest $ do
        zipWithM (λr v. setRegister r (sanitiseRegister r v))
            (frameRegisters @ gpRegisters) values;
        pc ← getRestartPC;
        setNextPC pc
    od;
    when resume_target $ restart dest;
    return []
  od)"
```

definition
```
  set_domain :: "obj_ref ⇒ domain ⇒ unit det_ext_monad" where
  "set_domain tptr new_dom ≡ do
     cur ← gets cur_thread;
     tcb_sched_action tcb_sched_dequeue tptr;
     thread_set_domain tptr new_dom;
     ts ← get_thread_state tptr;
     when (runnable ts) (tcb_sched_action tcb_sched_enqueue tptr);
     when (tptr = cur) reschedule_required
   od"
```

definition `invoke_domain:: "obj_ref ⇒ domain ⇒ (data list,'z::state_ext) p_monad"`
where
```
  "invoke_domain thread domain ≡
     liftE (do do_extended_op (set_domain thread domain); return [] od)"
```

end

34 IPC

theory Ipc_A
imports Tcb_A
begin

34.1 Getting and setting the message info register.

```
definition
  get_message_info :: "obj_ref ⇒ (message_info,'z::state_ext) s_monad"
where
  "get_message_info thread ≡ do
     x ← as_user thread $ get_register msg_info_register;
     return $ data_to_message_info x
   od"
```

34.2 IPC Capability Transfers

```
definition
  remove_rights :: "cap_rights ⇒ cap ⇒ cap"
where
 "remove_rights rights cap ≡ cap_rights_update (cap_rights cap - rights) cap"
```

In addition to the data payload a message may also contain capabilities. When a thread requests additional capabilities be transferred the identities of those capabilities are retreived from the thread's IPC buffer.

```
definition
  buffer_cptr_index :: nat
where
 "buffer_cptr_index ≡ (msg_max_length + 2)"

primrec
  get_extra_cptrs :: "obj_ref option ⇒ message_info ⇒ (cap_ref list,'z::state_ext) s_monad"
where
  "get_extra_cptrs (Some buf) mi =
    (liftM (map data_to_cptr) $ mapM (load_word_offs buf)
        [buffer_cptr_index ..< buffer_cptr_index + (unat (mi_extra_caps mi))])"
| "get_extra_cptrs None mi = return []"

definition
  get_extra_cptr :: "word32 ⇒ nat ⇒ (cap_ref,'z::state_ext) s_monad"
where
  "get_extra_cptr buffer n ≡ liftM data_to_cptr
      (load_word_offs buffer (n + buffer_cptr_index))"
```

This function both looks up the addresses of the additional capabilities and retreives them from the sender's CSpace.

```
definition
  lookup_extra_caps :: "obj_ref ⇒ data option ⇒ message_info ⇒ ((cap × cslot_ptr) list,'z::state_ext)
f_monad" where
  "lookup_extra_caps thread buffer mi ≡ doE
```

```
       cptrs ← liftE $ get_extra_cptrs buffer mi;
       mapME (λcptr. cap_fault_on_failure (of_bl cptr) False $ lookup_cap_and_slot thread cptr)
cptrs
  odE"
```

Capability transfers. Capabilities passed along with a message are split into two groups. Capabilities to the same endpoint as the message is passed through are not copied. Their badges are unwrapped and stored in the receiver's message buffer instead. Other capabilities are copied into the given slots.

Capability unwrapping allows a client to efficiently demonstrate to a server that it possesses authority to two or more services that server provides.

definition
```
  set_extra_badge :: "word32 ⇒ word32 ⇒ nat ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "set_extra_badge buffer badge n ≡
      store_word_offs buffer (buffer_cptr_index + n) badge"
```

type_synonym `transfer_caps_data = "(cap × cslot_ptr) list × cslot_ptr list"`

fun
```
  transfer_caps_loop :: "obj_ref option ⇒ bool ⇒ obj_ref ⇒ nat
                         ⇒ (cap × cslot_ptr) list ⇒ cslot_ptr list
                         ⇒ message_info ⇒ (message_info,'z::state_ext) s_monad"
```
where
```
  "transfer_caps_loop ep diminish rcv_buffer n [] slots
      mi = return (MI (mi_length mi) (of_nat n) (mi_caps_unwrapped mi)
                       (mi_label mi))"
| "transfer_caps_loop ep diminish rcv_buffer n ((cap, slot) # morecaps)
         slots mi =
  const_on_failure (MI (mi_length mi) (of_nat n) (mi_caps_unwrapped mi)
                      (mi_label mi)) (doE
    transfer_rest ← returnOk $ transfer_caps_loop ep diminish
         rcv_buffer (n + 1) morecaps;
    if (is_ep_cap cap ∧ ep = Some (obj_ref_of cap))
    then doE
       liftE $ set_extra_badge rcv_buffer (cap_ep_badge cap) n;
       liftE $ transfer_rest slots (MI (mi_length mi) (mi_extra_caps mi)
         (mi_caps_unwrapped mi || (1 << n)) (mi_label mi))
    odE
    else if slots ≠ []
    then doE
      cap' ← derive_cap slot (if diminish then remove_rights {AllowWrite} cap else cap);
      whenE (cap' = NullCap) $ throwError undefined;
      liftE $ cap_insert cap' slot (hd slots);
      liftE $ transfer_rest (tl slots) mi
    odE
    else returnOk (MI (mi_length mi) (of_nat n) (mi_caps_unwrapped mi)
                      (mi_label mi))
  odE)"
```

definition
```
  transfer_caps :: "message_info ⇒ (cap × cslot_ptr) list ⇒
                   obj_ref option ⇒ obj_ref ⇒ obj_ref option ⇒ bool ⇒
                   (message_info,'z::state_ext) s_monad"
```
where
```
  "transfer_caps info caps endpoint receiver recv_buffer diminish ≡ do
     dest_slots ← get_receive_slots receiver recv_buffer;
     mi' ← return $ MI (mi_length info) 0 0 (mi_label info);
```

```
    case recv_buffer of
      None ⇒ return mi'
    | Some receive_buffer ⇒
        transfer_caps_loop endpoint diminish receive_buffer 0 caps dest_slots mi'
  od"
```

34.3 Fault Handling

Threads fault when they attempt to access services that are not backed by any resources. Such a thread is then blocked and a fault messages is sent to its supervisor. When a reply to that message is sent the thread is reactivated.

Format a message for a given fault type.

fun
```
  make_fault_msg :: "fault ⇒ obj_ref ⇒ (data × data list,'z::state_ext) s_monad"
```
where
```
  "make_fault_msg (CapFault cptr rp lf) thread = (do
     pc ← as_user thread getRestartPC;
     return (1, pc # cptr # (if rp then 1 else 0) # msg_from_lookup_failure lf)
   od)"
| "make_fault_msg (VMFault vptr arch) thread = (do
     pc ← as_user thread getRestartPC;
     return (2, pc # vptr # arch)
   od)"
| "make_fault_msg (UnknownSyscallException n) thread = (do
     msg ← as_user thread $ mapM getRegister syscallMessage;
     return (3, msg @ [n])
   od)"
| "make_fault_msg (UserException exception code) thread = (do
     msg ← as_user thread $ mapM getRegister exceptionMessage;
     return (4, msg @ [exception, code])
   od)"
```

React to a fault reply. The reply message is interpreted in a manner that depends on the type of the original fault. For some fault types a thread reconfiguration is performed. This is done entirely to save the fault message recipient an additional system call. This function returns a boolean indicating whether the thread should now be restarted.

fun
```
  handle_fault_reply :: "fault ⇒ obj_ref ⇒
                          data ⇒ data list ⇒ (bool,'z::state_ext) s_monad"
```
where
```
  "handle_fault_reply (CapFault cptr rp lf) thread x y = return True"
| "handle_fault_reply (VMFault vptr arch) thread x y = return True"
| "handle_fault_reply (UnknownSyscallException n) thread label msg = do
     as_user thread $ zipWithM_x
         (λr v. set_register r $ sanitiseRegister r v)
         syscallMessage msg;
     return (label = 0)
   od"
| "handle_fault_reply (UserException exception code) thread label msg = do
     as_user thread $ zipWithM_x
         (λr v. set_register r $ sanitiseRegister r v)
         exceptionMessage msg;
     return (label = 0)
   od"
```

Transfer a fault message from a faulting thread to its supervisor.

definition
```
  do_fault_transfer :: "data ⇒ obj_ref ⇒ obj_ref
                              ⇒ obj_ref option ⇒ (unit,'z::state_ext) s_monad"
```
where
```
 "do_fault_transfer badge sender receiver buf ≡ do
    fault ← thread_get tcb_fault sender;
    f ← (case fault of
         Some f ⇒ return f
       | None ⇒ fail);
    (label, msg) ← make_fault_msg f sender;
    sent ← set_mrs receiver buf msg;
    set_message_info receiver $ MI sent 0 0 label;
    as_user receiver $ set_register badge_register badge
  od"
```

34.4 Synchronous Message Transfers

Transfer a non-fault message.

definition
```
  do_normal_transfer :: "obj_ref ⇒ obj_ref option ⇒ obj_ref option
                                   ⇒ data ⇒ bool ⇒ obj_ref
                                   ⇒ obj_ref option
                                   ⇒ bool ⇒ (unit,'z::state_ext) s_monad"
```
where
```
 "do_normal_transfer sender sbuf endpoint badge grant
                    receiver rbuf diminish ≡
  do
    mi ← get_message_info sender;
    caps ← if grant then lookup_extra_caps sender sbuf mi <catch> K (return [])
      else return [];
    mrs_transferred ← copy_mrs sender sbuf receiver rbuf (mi_length mi);
    mi' ← transfer_caps mi caps endpoint receiver rbuf diminish;
    set_message_info receiver $ MI mrs_transferred (mi_extra_caps mi')
                                   (mi_caps_unwrapped mi') (mi_label mi);
    as_user receiver $ set_register badge_register badge
  od"
```

Transfer a message either involving a fault or not.

definition
```
  do_ipc_transfer :: "obj_ref ⇒ obj_ref option ⇒
                      badge ⇒ bool ⇒ obj_ref ⇒ bool ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "do_ipc_transfer sender ep badge grant
     receiver diminish ≡ do

     recv_buffer ← lookup_ipc_buffer True receiver;
     fault ← thread_get tcb_fault sender;

     case fault
        of None ⇒ do
            send_buffer ← lookup_ipc_buffer False sender;
            do_normal_transfer sender send_buffer ep badge grant
                           receiver recv_buffer diminish
            od
         | Some f ⇒ do_fault_transfer badge sender receiver recv_buffer
   od"
```

Transfer a reply message and delete the one-use Reply capability.

definition
```
  do_reply_transfer :: "obj_ref ⇒ obj_ref ⇒ cslot_ptr ⇒ (unit,'z::state_ext) s_monad"
```
where
```
 "do_reply_transfer sender receiver slot ≡ do
    state ← get_thread_state receiver;
    assert (state = BlockedOnReply);
    fault ← thread_get tcb_fault receiver;
    case fault of
      None ⇒ do
         do_ipc_transfer sender None 0 True receiver False;
         cap_delete_one slot;
         set_thread_state receiver Running;
         do_extended_op (attempt_switch_to receiver)
      od
    | Some f ⇒ do
         cap_delete_one slot;
         mi ← get_message_info sender;
         buf ← lookup_ipc_buffer False sender;
         mrs ← get_mrs sender buf mi;
         restart ← handle_fault_reply f receiver (mi_label mi) mrs;
        thread_set (λtcb. tcb (| tcb_fault := None |)) receiver;
         set_thread_state receiver (if restart then Restart else Inactive);
         when restart $ do_extended_op (attempt_switch_to receiver)
       od
  od"
```

This function transfers a reply message to a thread when that message is generated by a kernel service.

definition
```
  reply_from_kernel :: "obj_ref ⇒ (data × data list) ⇒ (unit,'z::state_ext) s_monad"
```
where
```
 "reply_from_kernel thread x ≡ do
    (label, msg) ← return x;
    buf ← lookup_ipc_buffer True thread;
    as_user thread $ set_register badge_register 0;
    len ← set_mrs thread buf msg;
    set_message_info thread $ MI len 0 0 label
  od"
```

Install a one-use Reply capability.

definition
```
  setup_caller_cap :: "obj_ref ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad"
```
where
```
 "setup_caller_cap sender receiver ≡ do
    set_thread_state sender BlockedOnReply;
    cap_insert (ReplyCap sender False) (sender, tcb_cnode_index 2)
      (receiver, tcb_cnode_index 3)
  od"
```

Handle a message send operation performed on an endpoint by a thread. If a receiver is waiting then transfer the message. If no receiver is available and the thread is willing to block waiting to send then put it in the endpoint sending queue.

definition
```
  send_ipc :: "bool ⇒ bool ⇒ badge ⇒ bool
                ⇒ obj_ref ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad"
```
where

```
  "send_ipc block call badge can_grant thread epptr ≡ do
     ep ← get_endpoint epptr;
     case (ep, block) of
         (IdleEP, True) ⇒ do
               set_thread_state thread (BlockedOnSend epptr
                                    ⦇ sender_badge = badge,
                                      sender_can_grant = can_grant,
                                      sender_is_call = call ⦈);
               set_endpoint epptr $ SendEP [thread]
             od
       | (SendEP queue, True) ⇒ do
               set_thread_state thread (BlockedOnSend epptr
                                    ⦇ sender_badge = badge,
                                      sender_can_grant = can_grant,
                                      sender_is_call = call⦈);
               set_endpoint epptr $ SendEP (queue @ [thread])
             od
       | (IdleEP, False) ⇒ return ()
       | (SendEP queue, False) ⇒ return ()
       | (RecvEP (dest # queue), _) ⇒ do
             set_endpoint epptr $ (case queue of [] ⇒ IdleEP
                                                 | _ ⇒ RecvEP queue);
             recv_state ← get_thread_state dest;
             diminish ← (case recv_state
                           of BlockedOnReceive x d ⇒ return d
                            | _ ⇒ fail);
             do_ipc_transfer thread (Some epptr) badge
                             can_grant dest diminish;
             set_thread_state dest Running;
             do_extended_op (attempt_switch_to dest);
             fault ← thread_get tcb_fault thread;
             when (call ∨ fault ≠ None) $
               if can_grant ∧ ¬ diminish
               then setup_caller_cap thread dest
               else set_thread_state thread Inactive
           od
       | (RecvEP [], _) ⇒ fail
   od"
```

Handle a message receive operation performed on an endpoint by a thread. If a sender is waiting then transfer the message, otherwise put the thread in the endpoint receiving queue.

definition

```
  receive_ipc :: "obj_ref ⇒ cap ⇒ (unit,'z::state_ext) s_monad"
```

where

```
  "receive_ipc thread cap ≡ do
     (epptr,rights) ← (case cap
                       of EndpointCap ref badge rights ⇒ return (ref,rights)
                        | _ ⇒ fail);
     diminish ← return (AllowSend ∉ rights);
     ep ← get_endpoint epptr;
     case ep
       of IdleEP ⇒ do
            set_thread_state thread (BlockedOnReceive epptr diminish);
            set_endpoint epptr (RecvEP [thread])
          od
        | RecvEP queue ⇒ do
            set_thread_state thread (BlockedOnReceive epptr diminish);
            set_endpoint epptr (RecvEP (queue @ [thread]))
```

```
        od
      | SendEP q ⇒ do
          assert (q ≠ []);
          queue ← return $ tl q;
          sender ← return $ hd q;
          set_endpoint epptr $
            (case queue of [] ⇒ IdleEP | _ ⇒ SendEP queue);
          sender_state ← get_thread_state sender;
          data ← (case sender_state
                   of BlockedOnSend ref data ⇒ return data
                    | _ ⇒ fail);
          do_ipc_transfer sender (Some epptr)
                    (sender_badge data) (sender_can_grant data)
                    thread diminish;
          fault ← thread_get tcb_fault sender;
          if ((sender_is_call data) ∨ (fault ≠ None))
          then
            if sender_can_grant data ∧ ¬ diminish
            then setup_caller_cap sender thread
            else set_thread_state sender Inactive
          else do
            set_thread_state sender Running;
            do_extended_op (switch_if_required_to sender)
          od
        od
   od"
```

34.5 Asynchronous Message Transfers

Transfer an asynchronous message to a thread.

definition
```
  do_async_transfer :: "badge ⇒ message ⇒ obj_ref ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "do_async_transfer badge msg_word thread ≡ do
    receive_buffer ← lookup_ipc_buffer True thread;
    msg_transferred ← set_mrs thread receive_buffer [msg_word];
    as_user thread $ set_register badge_register badge;
    set_message_info thread $ MI msg_transferred 0 0 0
   od"
```

Helper function to handle an asynchronous send operation in the case where a receiver is waiting.

definition
```
  update_waiting_aep :: "obj_ref ⇒ obj_ref list ⇒ badge ⇒ message ⇒
                         (unit,'z::state_ext) s_monad"
```
where
```
  "update_waiting_aep aepptr queue badge val ≡ do
     assert (queue ≠ []);
     (dest,rest) ← return $ (hd queue, tl queue);

     set_async_ep aepptr $
         (case rest of [] ⇒ IdleAEP | _ ⇒ WaitingAEP rest);
     set_thread_state dest Running;
     do_async_transfer badge val dest;
     do_extended_op (switch_if_required_to dest)

   od"
```

Handle a message send operation performed on an asynchronous endpoint. If a receiver is waiting then transfer the message, otherwise combine the new message with whatever message is currently waiting.

definition
```
  send_async_ipc :: "obj_ref ⇒ badge ⇒ message ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "send_async_ipc aepptr badge val ≡ do
    aep ← get_async_ep aepptr;
    case aep
      of IdleAEP ⇒ set_async_ep aepptr $ ActiveAEP badge val
       | WaitingAEP queue ⇒ update_waiting_aep aepptr queue badge val
       | ActiveAEP badge' val' ⇒
           set_async_ep aepptr $
             ActiveAEP (combine_aep_badges badge badge')
                       (combine_aep_msgs val val')
   od"
```

Handle a receive operation performed on an asynchronous endpoint by a thread. If a message is waiting then perform the transfer, otherwise put the thread in the endpoint's receiving queue.

definition
```
  receive_async_ipc :: "obj_ref ⇒ cap ⇒ (unit,'z::state_ext) s_monad"
```
where
```
   "receive_async_ipc thread cap ≡ do
    aepptr ←
      case cap
        of AsyncEndpointCap aepptr badge rights ⇒ return aepptr
         | _ ⇒ fail;
    aep ← get_async_ep aepptr;
    case aep
      of IdleAEP ⇒ do
                  set_thread_state thread (BlockedOnAsyncEvent aepptr);
                  set_async_ep aepptr $ WaitingAEP [thread]
                od
       | WaitingAEP queue ⇒ do
                  set_thread_state thread (BlockedOnAsyncEvent aepptr);
                  set_async_ep aepptr $ WaitingAEP (queue @ [thread])
                od
       | ActiveAEP badge current_val ⇒ do
                    do_async_transfer badge current_val thread;
                    set_async_ep aepptr $ IdleAEP
                  od
    od"
```

34.6 Sending Fault Messages

When a thread encounters a fault, retreive its fault handler capability and send a fault message.

definition
```
  send_fault_ipc :: "obj_ref ⇒ fault ⇒ (unit,'z::state_ext) f_monad"
```
where
```
  "send_fault_ipc tptr fault ≡ doE
     handler_cptr ← liftE $ thread_get tcb_fault_handler tptr;
     handler_cap ← cap_fault_on_failure (of_bl handler_cptr) False $
         lookup_cap tptr handler_cptr;

     let f = CapFault (of_bl handler_cptr) False (MissingCapability 0)
```

```
    in
    (case handler_cap
      of EndpointCap ref badge rights ⇒
          if AllowSend ∈ rights ∧ AllowGrant ∈ rights
          then liftE $ (do
       thread_set (λtcb. tcb (| tcb_fault := Some fault |)) tptr;
       send_ipc True False (cap_ep_badge handler_cap)
                       True tptr (cap_ep_ptr handler_cap)
     od)
          else throwError f
       | _ ⇒ throwError f)
  odE"
```

If a fault message cannot be sent then leave the thread inactive.

definition
```
  handle_double_fault :: "obj_ref ⇒ fault ⇒ fault ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "handle_double_fault tptr ex1 ex2 ≡ set_thread_state tptr Inactive"
```

Handle a thread fault by sending a fault message if possible.

definition
```
  handle_fault :: "obj_ref ⇒ fault ⇒ (unit,'z::state_ext) s_monad"
```
where
```
  "handle_fault thread ex ≡ do
     _ ← gets_the $ get_tcb thread;
     send_fault_ipc thread ex
          <catch> handle_double_fault thread ex;
     return ()
   od"
```

end

35 Interrupts

```
theory Interrupt_A
imports Ipc_A
begin
```

Tests whether an IRQ identifier is in use.

```
definition
  is_irq_active :: "irq ⇒ (bool,'z::state_ext) s_monad" where
 "is_irq_active irq ≡ liftM (λst. st ≠ IRQInactive) $ get_irq_state irq"
```

The IRQControl capability can be used to create a new IRQHandler capability as well as to perform whatever architecture specific interrupt actions are available.

```
fun
  invoke_irq_control :: "irq_control_invocation ⇒ (unit,'z::state_ext) p_monad"
where
  "invoke_irq_control (IRQControl irq handler_slot control_slot) = liftE (do
     set_irq_state IRQNotifyAEP irq;
     cap_insert (IRQHandlerCap irq) control_slot handler_slot
   od)"
| "invoke_irq_control (InterruptControl invok) =
     arch_invoke_irq_control invok"
```

The IRQHandler capability may be used to configure how interrupts on an IRQ are delivered and to acknowledge a delivered interrupt. Interrupts are delivered when AsyncEndpoint capabilities are installed in the relevant per-IRQ slot. The IRQHandler operations load or clear those capabilities.

```
fun
  invoke_irq_handler :: "irq_handler_invocation ⇒ (unit,'z::state_ext) s_monad"
where
  "invoke_irq_handler (ACKIrq irq) = (do_machine_op $ maskInterrupt False irq)"
| "invoke_irq_handler (SetIRQHandler irq cap slot) = (do
     irq_slot ← get_irq_slot irq;
     cap_delete_one irq_slot;
     cap_insert cap slot irq_slot
   od)"
| "invoke_irq_handler (ClearIRQHandler irq) = (do
     irq_slot ← get_irq_slot irq;
     cap_delete_one irq_slot
   od)"
```

Handle an interrupt occurence. Timing and scheduling details are not included in this model, so no scheduling action needs to be taken on timer ticks. If the IRQ has a valid AsyncEndpoint cap loaded a message is delivered.

```
definition timer_tick :: "unit det_ext_monad" where
  "timer_tick ≡ do
     cur ← gets cur_thread;
     state ← get_thread_state cur;
     case state of Running ⇒ do
       ts ← ethread_get tcb_time_slice cur;
       let ts' = ts - 1 in
       if (ts' > 0) then thread_set_time_slice cur ts' else do
```

```
        thread_set_time_slice cur time_slice;
        tcb_sched_action tcb_sched_append cur;
        reschedule_required
      od
    od
    | _ ⇒ return ();
    when (num_domains > 1) (do
      dec_domain_time;
      dom_time ← gets domain_time;
      when (dom_time = 0) reschedule_required
    od)
  od"
```

definition

```
  handle_interrupt :: "irq ⇒ (unit,'z::state_ext) s_monad" where
 "handle_interrupt irq ≡ do
  st ← get_irq_state irq;
  case st of
    IRQNotifyAEP ⇒ do
      slot ← get_irq_slot irq;
      cap ← get_cap slot;
      when (is_aep_cap cap ∧ AllowSend ∈ cap_rights cap)
        $ send_async_ipc (obj_ref_of cap) (cap_ep_badge cap) (1 << ((unat irq) mod word_bits));
      do_machine_op $ maskInterrupt True irq
    od
  | IRQTimer ⇒ do
      do_extended_op timer_tick;
      do_machine_op resetTimer
    od
  | IRQInactive ⇒ fail (* not meant to be able to get IRQs from inactive lines *);
  do_machine_op $ ackInterrupt irq
  od"
```

end

36 Kernel Invocation Labels

```
theory InvocationLabels_H
imports "../../lib/Enumeration"
begin
```

An enumeration of all system call labels.

```
datatype invocation_label =
    InvalidInvocation
  | UntypedRetype
  | TCBReadRegisters
  | TCBWriteRegisters
  | TCBCopyRegisters
  | TCBConfigure
  | TCBSetPriority
  | TCBSetIPCBuffer
  | TCBSetSpace
  | TCBSuspend
  | TCBResume
  | CNodeRevoke
  | CNodeDelete
  | CNodeRecycle
  | CNodeCopy
  | CNodeMint
  | CNodeMove
  | CNodeMutate
  | CNodeRotate
  | CNodeSaveCaller
  | IRQIssueIRQHandler
  | IRQInterruptControl
  | IRQAckIRQ
  | IRQSetIRQHandler
  | IRQClearIRQHandler
  | DomainSetSet
  | ARMPDClean_Data
  | ARMPDInvalidate_Data
  | ARMPDCleanInvalidate_Data
  | ARMPDUnify_Instruction
  | ARMPageTableMap
  | ARMPageTableUnmap
  | ARMPageMap
  | ARMPageRemap
  | ARMPageUnmap
  | ARMPageClean_Data
  | ARMPageInvalidate_Data
  | ARMPageCleanInvalidate_Data
  | ARMPageUnify_Instruction
  | ARMPageGetAddress
  | ARMASIDControlMakePool
  | ARMASIDPoolAssign
```

```
definition
isPDFlush :: "invocation_label ⇒ bool"
```

where
```
"isPDFlush x≡ (case x of
        ARMPDClean_Data ⇒   True
      | ARMPDInvalidate_Data ⇒   True
      | ARMPDCleanInvalidate_Data ⇒   True
      | ARMPDUnify_Instruction ⇒   True
      | _ ⇒   False
      )"
```

definition
```
isPageFlush :: "invocation_label ⇒ bool"
```
where
```
"isPageFlush x≡ (case x of
        ARMPageClean_Data ⇒   True
      | ARMPageInvalidate_Data ⇒   True
      | ARMPageCleanInvalidate_Data ⇒   True
      | ARMPageUnify_Instruction ⇒   True
      | _ ⇒   False
      )"
```

end

37 Decoding System Calls

```
theory Decode_A
imports
  Interrupt_A
  "../../lib/WordLib"
  "../design/InvocationLabels_H"
begin
```

This theory includes definitions describing how user arguments are decoded into invocation structures; these structures are then used to perform the actual system call (see `perform_invocation`). In addition, these definitions check the validity of these arguments, throwing an error if given an invalid request.

As such, this theory describes the binary interface between the user and the kernel, along with the preconditions on each argument.

37.1 Helper definitions

This definition ensures that the given pointer is aligned to the given page size.

```
definition
  check_vp_alignment :: "vmpage_size ⇒ word32 ⇒ (unit,'z::state_ext) se_monad" where
  "check_vp_alignment sz vptr ≡
     unlessE (is_aligned vptr (pageBitsForSize sz)) $
       throwError AlignmentError"
```

This definition converts a user-supplied argument into an invocation label, used to determine the method to invoke.

```
definition
  invocation_type :: "data ⇒ invocation_label"
where
 "invocation_type x ≡ if ∃(v :: invocation_label). fromEnum v = data_to_nat x
                     then toEnum (data_to_nat x) else InvalidInvocation"

definition
  label_to_flush_type :: "invocation_label ⇒ flush_type"
where
  "label_to_flush_type label ≡ case label of
       ARMPDClean_Data ⇒ Clean
     | ARMPageClean_Data ⇒ Clean
     | ARMPDInvalidate_Data ⇒ Invalidate
     | ARMPageInvalidate_Data ⇒ Invalidate
     | ARMPDCleanInvalidate_Data ⇒ CleanInvalidate
     | ARMPageCleanInvalidate_Data ⇒ CleanInvalidate
     | ARMPDUnify_Instruction ⇒ Unify
     | ARMPageUnify_Instruction ⇒ Unify"
```

37.2 Architecture calls

This definition decodes architecture-specific invocations.

```
definition
```

```
  page_base :: "vspace_ref ⇒ vmpage_size ⇒ vspace_ref"
where
  "page_base vaddr vmsize ≡ vaddr && ~~ mask (pageBitsForSize vmsize)"
```

definition

```
  arch_decode_invocation ::
  "data ⇒ data list ⇒ cap_ref ⇒ cslot_ptr ⇒ arch_cap ⇒ (cap × cslot_ptr) list ⇒
   (arch_invocation,'z::state_ext) se_monad"
where
"arch_decode_invocation label args x_slot cte cap extra_caps ≡ case cap of

  PageDirectoryCap _ _ ⇒
    if isPDFlush (invocation_type label) then
    if length args > 1
    then let start = args ! 0;
             end = args ! 1
    in doE
            whenE (end ≤ start) $ throwError $ InvalidArgument 1;
            whenE (start ≥ kernel_base ∨ end > kernel_base) $ throwError IllegalOperation;
            (pd,asid) ← (case cap of
                    PageDirectoryCap pd (Some asid) ⇒ returnOk (pd,asid)
                  | _ ⇒ throwError $ InvalidCapability 0);
            pd' ← lookup_error_on_failure False $ find_pd_for_asid asid;
            whenE (pd' ≠ pd) $ throwError $ InvalidCapability 0;
            frame_info ← liftE $ resolve_vaddr pd start;
            case frame_info of
                None ⇒ returnOk $ InvokePageDirectory PageDirectoryNothing
              | Some (frame_size, frame_base) ⇒
                    let base_start = page_base start frame_size;
                        base_end = page_base (end - 1) frame_size;
                        offset = start && mask (pageBitsForSize frame_size);
                        pstart = frame_base + offset
                    in doE
                        whenE (base_start ≠ base_end) $ throwError $
                            RangeError start (base_start + mask (pageBitsForSize frame_size));
                        returnOk $ InvokePageDirectory $
                            PageDirectoryFlush (label_to_flush_type (invocation_type label))
                            start (end - 1) pstart pd asid
                    odE
    odE
    else throwError TruncatedMessage
    else throwError IllegalOperation

| PageTableCap p mapped_address ⇒
    if invocation_type label = ARMPageTableMap then
    if length args > 1 ∧ length extra_caps > 0
    then let vaddr = args ! 0;
             attr = args ! 1;
             pd_cap = fst (extra_caps ! 0)
    in doE
            whenE (mapped_address ≠ None) $ throwError $ InvalidCapability 0;
            (pd,asid) ← (case pd_cap of
                            ArchObjectCap (PageDirectoryCap pd (Some asid)) ⇒
                              returnOk (pd,asid)
                          | _ ⇒ throwError $ InvalidCapability 1);
            whenE (vaddr ≥ kernel_base) $ throwError $ InvalidArgument 0;
            pd' ← lookup_error_on_failure False $ find_pd_for_asid asid;
            whenE (pd' ≠ pd) $ throwError $ InvalidCapability 1;
```

```
          pd_index ← returnOk (shiftr vaddr 20);
          vaddr' ← returnOk (vaddr && ~~ mask 20);
          pd_slot ← returnOk (pd + (pd_index << 2));
          oldpde ← liftE $ get_master_pde pd_slot;
          unlessE (oldpde = InvalidPDE) $ throwError DeleteFirst;
          pde ← returnOk (PageTablePDE (addrFromPPtr p)
                              (attribs_from_word attr ∩ {ParityEnabled}) 0);
          returnOk $ InvokePageTable $
              PageTableMap
              (ArchObjectCap $ PageTableCap p (Some (asid, vaddr')))
              cte pde pd_slot
    odE
    else throwError TruncatedMessage
    else if invocation_type label = ARMPageTableUnmap
    then doE
          final ← liftE $ is_final_cap (ArchObjectCap cap);
          unlessE final $ throwError RevokeFirst;
          returnOk $ InvokePageTable $ PageTableUnmap (ArchObjectCap cap) cte
    odE
    else throwError IllegalOperation

| PageCap p R pgsz mapped_address ⇒
    if invocation_type label = ARMPageMap then
    if length args > 2 ∧ length extra_caps > 0
    then let vaddr = args ! 0;
             rights_mask = args ! 1;
             attr = args ! 2;
             pd_cap = fst (extra_caps ! 0)
        in doE
            whenE (mapped_address ≠ None) $ throwError $ InvalidCapability 0;
            (pd,asid) ← (case pd_cap of
                              ArchObjectCap (PageDirectoryCap pd (Some asid)) ⇒
                                returnOk (pd,asid)
                          | _ ⇒ throwError $ InvalidCapability 1);
            pd' ← lookup_error_on_failure False $ find_pd_for_asid asid;
            whenE (pd' ≠ pd) $ throwError $ InvalidCapability 1;
            vtop ← returnOk (vaddr + (1 << (pageBitsForSize pgsz)) - 1);
            whenE (vtop ≥ kernel_base) $ throwError $ InvalidArgument 0;
            vm_rights ← returnOk (mask_vm_rights R (data_to_rights rights_mask));
            check_vp_alignment pgsz vaddr;
            entries ← create_mapping_entries (addrFromPPtr p)
                                              vaddr pgsz vm_rights
                                              (attribs_from_word attr) pd;
            ensure_safe_mapping entries;
            returnOk $ InvokePage $ PageMap asid
                (ArchObjectCap $ PageCap p R pgsz (Some (asid, vaddr)))
                cte entries
        odE
    else  throwError TruncatedMessage
    else if invocation_type label = ARMPageRemap then
         if length args > 1 ∧ length extra_caps > 0
         then let rights_mask = args ! 0;
                  attr = args ! 1;
                  pd_cap = fst (extra_caps ! 0)
         in doE
            (pd,asid) ← (case pd_cap of
                              ArchObjectCap (PageDirectoryCap pd (Some asid)) ⇒
                                returnOk (pd,asid)
```

```
                     | _ ⇒ throwError $ InvalidCapability 1);
         (asid', vaddr) ← (case mapped_address of
               Some a ⇒ returnOk a
             | _ ⇒ throwError $ InvalidCapability 0);
         pd' ← lookup_error_on_failure False $ find_pd_for_asid asid';
         whenE (pd' ≠ pd ∨ asid' ≠ asid) $ throwError $ InvalidCapability 1;
         vm_rights ← returnOk (mask_vm_rights R $ data_to_rights rights_mask);
         check_vp_alignment pgsz vaddr;
         entries ← create_mapping_entries (addrFromPPtr p)
                                          vaddr pgsz vm_rights
                                          (attribs_from_word attr) pd;
         ensure_safe_mapping entries;
         returnOk $ InvokePage $ PageRemap asid' entries
     odE
  else  throwError TruncatedMessage
  else if invocation_type label = ARMPageUnmap
  then  returnOk $ InvokePage $ PageUnmap cap cte
  else if isPageFlush (invocation_type label) then
      if length args > 1
      then let start = args ! 0;
               end = args ! 1
      in doE
          (asid, vaddr) ← (case mapped_address of
              Some a ⇒ returnOk a
            | _ ⇒ throwError IllegalOperation);
          pd ← lookup_error_on_failure False $ find_pd_for_asid asid;
          whenE (end ≤ start) $ throwError $ InvalidArgument 1;
          page_size ← returnOk $ 1 << pageBitsForSize pgsz;
          whenE (start ≥ page_size ∨ end > page_size) $ throwError $ InvalidArgument 0;
          returnOk $ InvokePage $ PageFlush
              (label_to_flush_type (invocation_type label)) (start + vaddr)
              (end + vaddr - 1) (addrFromPPtr p + start) pd asid
  odE
  else throwError TruncatedMessage
  else if invocation_type label = ARMPageGetAddress
  then returnOk $ InvokePage $ PageGetAddr p
 else  throwError IllegalOperation

| ASIDControlCap ⇒
  if invocation_type label = ARMASIDControlMakePool then
  if length args > 1 ∧ length extra_caps > 1
  then let index = args ! 0;
           depth = args ! 1;
           (untyped, parent_slot) = extra_caps ! 0;
           root = fst (extra_caps ! 1)
       in doE
          asid_table ← liftE $ gets (arm_asid_table ∘ arch_state);
          free_set ← returnOk (- dom asid_table ∩ {x. x ≤ 2 ^ asid_high_bits - 1});
          whenE (free_set = {}) $ throwError DeleteFirst;
          free ← liftE $ select_ext (λ_. free_asid_select asid_table) free_set;
          base ← returnOk (ucast free << asid_low_bits);
          (p,n) ← (case untyped of UntypedCap p n f ⇒ returnOk (p,n)
                                 | _ ⇒ throwError $ InvalidCapability 1);
          frame ← (if n = pageBits
                   then doE
                     ensure_no_children parent_slot;
                     returnOk p
                   odE
```

```
                 else  throwError $ InvalidCapability 1);
          dest_slot ← lookup_target_slot root (to_bl index) (unat depth);
          ensure_empty dest_slot;
          returnOk $ InvokeASIDControl $ MakePool frame dest_slot parent_slot base
        odE
    else  throwError TruncatedMessage
    else  throwError IllegalOperation

| ASIDPoolCap p base ⇒
  if invocation_type label = ARMASIDPoolAssign then
  if length extra_caps > 0
  then
    let (pd_cap, pd_cap_slot) = extra_caps ! 0
     in case pd_cap of
         ArchObjectCap (PageDirectoryCap _ None) ⇒ doE
           asid_table ← liftE $ gets (arm_asid_table ∘ arch_state);
           pool_ptr ← returnOk (asid_table (asid_high_bits_of base));
           whenE (pool_ptr = None) $ throwError $ FailedLookup False InvalidRoot;
           whenE (p ≠ the pool_ptr) $ throwError $ InvalidCapability 0;
           pool ← liftE $ get_asid_pool p;
           free_set ← returnOk
                  (- dom pool ∩ {x. x ≤ 2 ^ asid_low_bits - 1 ∧ ucast x + base ≠ 0});
           whenE (free_set = {}) $ throwError DeleteFirst;
           offset ← liftE $ select_ext (λ_. free_asid_pool_select pool base) free_set;
           returnOk $ InvokeASIDPool $ Assign (ucast offset + base) p pd_cap_slot
         odE
       | _ ⇒  throwError $ InvalidCapability 1
  else  throwError TruncatedMessage
  else  throwError IllegalOperation"
```

ARM does not support additional interrupt control operations

definition

```
  arch_decode_interrupt_control ::
  "data list ⇒ cap list ⇒ (arch_interrupt_control,'z::state_ext) se_monad" where
  "arch_decode_interrupt_control d cs ≡ throwError IllegalOperation"
```

37.3 CNode

This definition decodes CNode invocations.

definition

```
  decode_cnode_invocation ::
  "data ⇒ data list ⇒ cap ⇒ cap list ⇒ (cnode_invocation,'z::state_ext) se_monad"
where
"decode_cnode_invocation label args cap excaps ≡ doE
  unlessE (invocation_type label ∈ set [CNodeRevoke .e. CNodeSaveCaller]) $
    throwError IllegalOperation;
  whenE (length args < 2) (throwError TruncatedMessage);
  index ← returnOk $ data_to_cptr $ args ! 0;
  bits ← returnOk $ data_to_nat $ args ! 1;
  args ← returnOk $ drop 2 args;
  dest_slot ← lookup_target_slot cap index bits;
  if length args ≥ 2 ∧ length excaps > 0
        ∧ invocation_type label ∈ set [CNodeCopy .e. CNodeMutate] then
  doE
    src_index ← returnOk $ data_to_cptr $ args ! 0;
    src_depth ← returnOk $ data_to_nat $ args ! 1;
```

```
    args ← returnOk $ drop 2 args;
    src_root_cap ← returnOk $ excaps ! 0;
    ensure_empty dest_slot;
    src_slot ←
         lookup_source_slot src_root_cap src_index src_depth;
    src_cap ← liftE $ get_cap src_slot;
    whenE (src_cap = NullCap) $
         throwError $ FailedLookup True $ MissingCapability src_depth;
    (rights, cap_data, is_move) ← case (invocation_type label, args) of
      (CNodeCopy, rightsWord # _) ⇒ doE
                   rights ← returnOk $ data_to_rights $ rightsWord;
                   returnOk $ (rights, None, False)
               odE
     | (CNodeMint, rightsWord # capData # _) ⇒ doE
                   rights ← returnOk $ data_to_rights $ rightsWord;
                   returnOk $ (rights, Some capData, False)
               odE
     | (CNodeMove, _) ⇒ returnOk (all_rights, None, True)
     | (CNodeMutate, capData # _) ⇒ returnOk (all_rights, Some capData, True)
     | _ ⇒ throwError TruncatedMessage;
    src_cap ← returnOk $ mask_cap rights src_cap;
    new_cap ← (if is_move then returnOk else derive_cap src_slot) (case cap_data of
                 Some w ⇒ update_cap_data is_move w src_cap
               | None ⇒ src_cap);
    whenE (new_cap = NullCap) $ throwError IllegalOperation;
    returnOk $ (if is_move then MoveCall else InsertCall) new_cap src_slot dest_slot
  odE
  else if invocation_type label = CNodeRevoke then returnOk $ RevokeCall dest_slot
  else if invocation_type label = CNodeDelete then returnOk $ DeleteCall dest_slot
  else if invocation_type label = CNodeSaveCaller then doE
    ensure_empty dest_slot;
    returnOk $ SaveCall dest_slot
  odE
  else if invocation_type label = CNodeRecycle then doE
    cap ← liftE $ get_cap dest_slot;
    unlessE (has_recycle_rights cap) $ throwError IllegalOperation;
    returnOk $ RecycleCall dest_slot
  odE
  else if invocation_type label = CNodeRotate ∧ length args > 5
          ∧ length excaps > 1 then
  doE
    pivot_new_data ← returnOk $ args ! 0;
    pivot_index ← returnOk $ data_to_cptr $ args ! 1;
    pivot_depth ← returnOk $ data_to_nat $ args ! 2;
    src_new_data ← returnOk $ args ! 3;
    src_index ← returnOk $ data_to_cptr $ args ! 4;
    src_depth ← returnOk $ data_to_nat $ args ! 5;
    pivot_root_cap <- returnOk $ excaps ! 0;
    src_root_cap <- returnOk $ excaps ! 1;

    src_slot <- lookup_source_slot src_root_cap src_index src_depth;
    pivot_slot <- lookup_pivot_slot pivot_root_cap pivot_index pivot_depth;

    whenE (pivot_slot = src_slot ∨ pivot_slot = dest_slot) $
      throwError IllegalOperation;

    unlessE (src_slot = dest_slot) $ ensure_empty dest_slot;
```

```
    src_cap <- liftE $ get_cap src_slot;
    whenE (src_cap = NullCap) $
      throwError $ FailedLookup True $ MissingCapability src_depth;

    pivot_cap <- liftE $ get_cap pivot_slot;
    whenE (pivot_cap = NullCap) $
      throwError $ FailedLookup False $ MissingCapability pivot_depth;

    new_src_cap ← returnOk $ update_cap_data True src_new_data src_cap;
    new_pivot_cap ← returnOk $ update_cap_data True pivot_new_data pivot_cap;

    whenE (new_src_cap = NullCap) $ throwError IllegalOperation;
    whenE (new_pivot_cap = NullCap) $ throwError IllegalOperation;

    returnOk $ RotateCall new_src_cap new_pivot_cap src_slot pivot_slot dest_slot
  odE
  else
    throwError TruncatedMessage
odE"
```

37.4 Threads

The definitions in this section decode invocations on TCBs.

This definition checks whether the first argument is between the second and third.

definition
```
  range_check :: "word32 ⇒ word32 ⇒ word32 ⇒ (unit,'z::state_ext) se_monad"
```
where
```
  "range_check v min_v max_v ≡
    unlessE (v ≥ min_v ∧ v ≤ max_v) $
        throwError $ RangeError min_v max_v"
```

definition
```
  decode_read_registers :: "data list ⇒ cap ⇒ (tcb_invocation,'z::state_ext) se_monad"
```
where
```
"decode_read_registers data cap ≡ case data of
  flags#n#_ ⇒ doE
    range_check n 1 $ of_nat (length frameRegisters + length gpRegisters);
    p ← case cap of ThreadCap p ⇒ returnOk p;
    self ← liftE $ gets cur_thread;
    whenE (p = self) $ throwError IllegalOperation;
    returnOk $ ReadRegisters p (flags !! 0) n ARMNoExtraRegisters
  odE
| _ ⇒ throwError TruncatedMessage"
```

definition
```
  decode_copy_registers :: "data list ⇒ cap ⇒ cap list ⇒ (tcb_invocation,'z::state_ext) se_monad"
```
where
```
"decode_copy_registers data cap extra_caps ≡ case data of
  flags#_ ⇒  doE
    suspend_source ← returnOk (flags !! 0);
    resume_target ← returnOk (flags !! 1);
    transfer_frame ← returnOk (flags !! 2);
    transfer_integer ← returnOk (flags !! 3);
    whenE (extra_caps = []) $ throwError TruncatedMessage;
    src_tcb ← (case extra_caps of
      ThreadCap p # _ ⇒ returnOk p
```

```
    | _ ⇒ throwError $ InvalidCapability 1);
    p ← case cap of ThreadCap p ⇒ returnOk p;
    returnOk $ CopyRegisters p src_tcb
                             suspend_source resume_target
                             transfer_frame transfer_integer
                             ARMNoExtraRegisters
  odE
| _ ⇒  throwError TruncatedMessage"
```

definition
```
  decode_write_registers :: "data list ⇒ cap ⇒ (tcb_invocation,'z::state_ext) se_monad"
```
where
```
"decode_write_registers data cap ≡ case data of
  flags#n#values ⇒ doE
    whenE (length values < unat n) $ throwError TruncatedMessage;
    p ← case cap of ThreadCap p ⇒ returnOk p;
    self ← liftE $ gets cur_thread;
    whenE (p = self) $ throwError IllegalOperation;
    returnOk $ WriteRegisters p (flags !! 0)
               (take (unat n) values) ARMNoExtraRegisters
  odE
| _ ⇒ throwError TruncatedMessage"
```

primrec
```
  tc_new_fault_ep :: "tcb_invocation ⇒ cap_ref option"
```
where
```
  "tc_new_fault_ep (ThreadControl target slot faultep prio croot vroot buffer) = faultep"
```

primrec
```
  tc_new_priority :: "tcb_invocation ⇒ word8 option"
```
where
```
  "tc_new_priority (ThreadControl target slot faultep prio croot vroot buffer) = prio"
```

primrec
```
  tc_new_croot :: "tcb_invocation ⇒ (cap × cslot_ptr) option"
```
where
```
  "tc_new_croot (ThreadControl target slot faultep prio croot vroot buffer) = croot"
```

primrec
```
  tc_new_vroot :: "tcb_invocation ⇒ (cap × cslot_ptr) option"
```
where
```
  "tc_new_vroot (ThreadControl target slot faultep prio croot vroot buffer) = vroot"
```

primrec
```
  tc_new_buffer :: "tcb_invocation ⇒ (vspace_ref × (cap × cslot_ptr) option) option"
```
where
```
  "tc_new_buffer (ThreadControl target slot faultep prio croot vroot buffer) = buffer"
```

definition
```
  decode_set_priority :: "data list ⇒ cap ⇒ cslot_ptr ⇒ (tcb_invocation,'z::state_ext) se_monad"
```
where
```
  "decode_set_priority args cap slot ≡
     if length args = 0 then throwError TruncatedMessage
     else doE
       cur ← liftE $ gets cur_thread;
       OR_choice (decode_set_priority_error_choice (ucast $ args ! 0) cur)
```

```
          (throwError IllegalOperation)
          (returnOk (ThreadControl (obj_ref_of cap) slot None
            (Some (ucast $ args ! 0)) None None None))
    odE"
```

definition

```
  decode_set_ipc_buffer ::
  "data list ⇒ cap ⇒ cslot_ptr ⇒ (cap × cslot_ptr) list ⇒ (tcb_invocation,'z::state_ext) se_monad"
```

where

```
"decode_set_ipc_buffer args cap slot excs ≡ doE
  whenE (length args = 0) $ throwError TruncatedMessage;
  whenE (length excs = 0) $ throwError TruncatedMessage;
  buffer ← returnOk $ data_to_vref $ args ! 0;
  (bcap, bslot) ← returnOk $ excs ! 0;
  newbuf ← if buffer = 0 then returnOk None
           else doE
      buffer_cap ← derive_cap bslot bcap;
      check_valid_ipc_buffer buffer buffer_cap;
      returnOk $ Some (buffer_cap, bslot)
    odE;
  returnOk $
    ThreadControl (obj_ref_of cap) slot None None None None (Some (buffer, newbuf))
odE"
```

definition

```
  decode_set_space
  :: "data list ⇒ cap ⇒ cslot_ptr ⇒ (cap × cslot_ptr) list ⇒ (tcb_invocation,'z::state_ext)
se_monad"
```

where

```
  "decode_set_space args cap slot excaps ≡ doE
   whenE (length args < 3 ∨ length excaps < 2) $ throwError TruncatedMessage;
   fault_ep ← returnOk $ args ! 0;
   croot_data  ← returnOk $ args ! 1;
   vroot_data  ← returnOk $ args ! 2;
   croot_arg  ← returnOk $ excaps ! 0;
   vroot_arg  ← returnOk $ excaps ! 1;
   can_chg_cr ← liftE $ liftM Not $ slot_cap_long_running_delete
                  $ get_tcb_ctable_ptr $ obj_ref_of cap;
   can_chg_vr ← liftE $ liftM Not $ slot_cap_long_running_delete
                  $ get_tcb_vtable_ptr $ obj_ref_of cap;
   unlessE (can_chg_cr ∧ can_chg_vr) $ throwError IllegalOperation;

   croot_cap  ← returnOk $ fst croot_arg;
   croot_slot ← returnOk $ snd croot_arg;
   croot_cap' ← derive_cap croot_slot $
                (if croot_data = 0 then id else update_cap_data False croot_data)
                croot_cap;
   unlessE (is_cnode_cap croot_cap') $ throwError IllegalOperation;
   croot ← returnOk (croot_cap', croot_slot);

   vroot_cap  ← returnOk $ fst vroot_arg;
   vroot_slot ← returnOk $ snd vroot_arg;
   vroot_cap' ← derive_cap vroot_slot $
                (if vroot_data = 0 then id else update_cap_data False vroot_data)
                vroot_cap;
   unlessE (is_valid_vtable_root vroot_cap') $ throwError IllegalOperation;
```

```
   vroot ← returnOk (vroot_cap', vroot_slot);

   returnOk $ ThreadControl (obj_ref_of cap) slot (Some (to_bl fault_ep)) None
                            (Some croot) (Some vroot) None
 odE"
```

definition

```
  decode_tcb_configure ::
  "data list ⇒ cap ⇒ cslot_ptr ⇒ (cap × cslot_ptr) list ⇒ (tcb_invocation,'z::state_ext) se_monad"
```

where

```
  "decode_tcb_configure args cap slot extra_caps ≡ doE
     whenE (length args < 5) $ throwError TruncatedMessage;
     whenE (length extra_caps < 3) $ throwError TruncatedMessage;
     fault_ep ← returnOk $ args ! 0;
     prio     ← returnOk $ args ! 1;
     croot_data ← returnOk $ args ! 2;
     vroot_data ← returnOk $ args ! 3;
     crootvroot ← returnOk $ take 2 extra_caps;
     buffer_cap ← returnOk $ extra_caps ! 2;
     buffer ← returnOk $ args ! 4;
     set_prio ← decode_set_priority [prio] cap slot;
     set_params ← decode_set_ipc_buffer [buffer] cap slot [buffer_cap];
     set_space ← decode_set_space [fault_ep, croot_data, vroot_data] cap slot crootvroot;
     returnOk $ ThreadControl (obj_ref_of cap) slot (tc_new_fault_ep set_space)
                              (tc_new_priority set_prio)
                              (tc_new_croot set_space) (tc_new_vroot set_space)
                              (tc_new_buffer set_params)
   odE"
```

definition

```
  decode_tcb_invocation ::
  "data ⇒ data list ⇒ cap ⇒ cslot_ptr ⇒ (cap × cslot_ptr) list ⇒
  (tcb_invocation,'z::state_ext) se_monad"
```

where

```
 "decode_tcb_invocation label args cap slot excs ≡
  case invocation_type label of
      TCBReadRegisters ⇒ decode_read_registers args cap
    | TCBWriteRegisters ⇒ decode_write_registers args cap
    | TCBCopyRegisters ⇒ decode_copy_registers args cap $ map fst excs
    | TCBSuspend ⇒ returnOk $ Suspend $ obj_ref_of cap
    | TCBResume ⇒ returnOk $ Resume $ obj_ref_of cap
    | TCBConfigure ⇒ decode_tcb_configure args cap slot excs
    | TCBSetPriority ⇒ decode_set_priority args cap slot
    | TCBSetIPCBuffer ⇒ decode_set_ipc_buffer args cap slot excs
    | TCBSetSpace ⇒ decode_set_space args cap slot excs
    | _ ⇒ throwError IllegalOperation"
```

definition

```
  decode_domain_invocation ::
  "data ⇒ data list ⇒ (cap × cslot_ptr) list ⇒
    ((obj_ref × domain), 'z::state_ext) se_monad"
```

where

```
  "decode_domain_invocation label args excs ≡ doE
     whenE (invocation_type label ≠ DomainSetSet) $ throwError IllegalOperation;
     domain ← (case args of
       x # xs ⇒ doE
```

```
        whenE (unat x ≥ num_domains) $ throwError $ InvalidArgument 0;
        returnOk (ucast x)
      odE
      | _ ⇒ throwError TruncatedMessage);
    whenE (length excs = 0) $ throwError TruncatedMessage;
    case (fst (hd excs)) of ThreadCap ptr ⇒ returnOk $ (ptr, domain)
      | _ ⇒ throwError $ InvalidArgument 1
  odE"
```

37.5 IRQ

The following two definitions decode system calls for the interrupt controller and interrupt handlers

definition
```
  decode_irq_control_invocation :: "data ⇒ data list ⇒ cslot_ptr ⇒ cap list
                                     ⇒ (irq_control_invocation,'z::state_ext) se_monad" where
 "decode_irq_control_invocation label args src_slot cps ≡
  (if invocation_type label = IRQIssueIRQHandler
    then if length args ≥ 3 ∧ length cps ≥ 1
      then let x = args ! 0; index = args ! 1; depth = args ! 2;
               cnode = cps ! 0; irqv = ucast x in doE
        whenE (x > ucast maxIRQ) $
          throwError (RangeError 0 (ucast maxIRQ));
        irq_active ← liftE $ is_irq_active irqv;
        whenE irq_active $ throwError RevokeFirst;

        dest_slot ← lookup_target_slot
               cnode (data_to_cptr index) (unat depth);
        ensure_empty dest_slot;

        returnOk $ IRQControl irqv dest_slot src_slot
      odE
    else throwError TruncatedMessage
  else if invocation_type label = IRQInterruptControl
    then liftME InterruptControl
         $ arch_decode_interrupt_control args cps
  else throwError IllegalOperation)"
```

definition
```
  decode_irq_handler_invocation :: "data ⇒ irq ⇒ (cap × cslot_ptr) list
                                     ⇒ (irq_handler_invocation,'z::state_ext) se_monad" where
 "decode_irq_handler_invocation label irq cps ≡
  if invocation_type label = IRQAckIRQ
    then returnOk $ ACKIrq irq
  else if invocation_type label = IRQSetIRQHandler
    then if cps ≠ []
      then let (cap, slot) = hd cps in
      if is_aep_cap cap ∧ AllowSend ∈ cap_rights cap
      then returnOk $ SetIRQHandler irq cap slot
      else throwError $ InvalidCapability 0
    else throwError TruncatedMessage
  else if invocation_type label = IRQClearIRQHandler
    then returnOk $ ClearIRQHandler irq
  else throwError IllegalOperation"
```

37.6 Untyped

The definitions in this section deal with decoding invocations of untyped memory capabilities.

```
definition
  arch_data_to_obj_type :: "nat ⇒ aobject_type option" where
 "arch_data_to_obj_type n ≡
  if n = 0 then Some SmallPageObj
  else if n = 1 then Some LargePageObj
  else if n = 2 then Some SectionObj
  else if n = 3 then Some SuperSectionObj
  else if n = 4 then Some PageTableObj
  else if n = 5 then Some PageDirectoryObj
  else None"

definition
  data_to_obj_type :: "data ⇒ (apiobject_type,'z::state_ext) se_monad" where
  "data_to_obj_type type ≡ doE
    n ← returnOk $ data_to_nat type;
    if n = 0 then
      returnOk $ Untyped
    else if n = 1 then
      returnOk $ TCBObject
    else if n = 2 then
      returnOk $ EndpointObject
    else if n = 3 then
      returnOk $ AsyncEndpointObject
    else if n = 4 then
      returnOk $ CapTableObject
    else (case arch_data_to_obj_type (n - 5)
       of Some tp ⇒ returnOk (ArchObject tp)
        | None ⇒ throwError (InvalidArgument 0))
  odE"

definition
  get_free_ref :: "obj_ref ⇒ nat ⇒ obj_ref" where
  "get_free_ref base free_index ≡ base +  (of_nat free_index)"

definition
  get_free_index :: "obj_ref ⇒ obj_ref ⇒ nat" where
  "get_free_index base free ≡ unat $ (free - base)"

definition
  decode_untyped_invocation ::
  "data ⇒ data list ⇒ cslot_ptr ⇒ cap ⇒ cap list ⇒ (untyped_invocation,'z::state_ext) se_monad"
where
"decode_untyped_invocation label args slot cap excaps ≡ doE
  unlessE (invocation_type label = UntypedRetype) $ throwError IllegalOperation;
  whenE (length args < 6) $ throwError TruncatedMessage;
  whenE (length excaps = 0) $ throwError TruncatedMessage;
  root_cap ← returnOk $ excaps ! 0;
  new_type ← data_to_obj_type (args!0);

  user_obj_size ← returnOk $ data_to_nat (args!1);
  unlessE (user_obj_size < word_bits - 1)
    $ throwError (RangeError 0 (of_nat word_bits - 2));
  whenE (new_type = CapTableObject ∧ user_obj_size = 0)
    $ throwError (InvalidArgument 1);
```

```
  whenE (new_type = Untyped ∧ user_obj_size < 4)
    $ throwError (InvalidArgument 1);
  node_index ← returnOk $ data_to_cptr (args!2);
  node_depth ← returnOk $ data_to_nat (args!3);

  node_cap ← if node_depth = 0
        then returnOk root_cap
        else doE
            node_slot ← lookup_target_slot
                root_cap node_index node_depth;
            liftE $ get_cap node_slot
        odE;

  if is_cnode_cap node_cap
        then  returnOk ()
        else  throwError $ FailedLookup False $ MissingCapability node_depth;

  node_offset ← returnOk $ data_to_nat (args ! 4);
  node_window ← returnOk $ data_to_nat (args ! 5);
  radix_bits ← returnOk $ bits_of node_cap;
  node_size ← returnOk (2 ^ radix_bits);

  whenE (node_offset < 0 ∨ node_offset > node_size - 1) $
    throwError $ RangeError 0 (of_nat (node_size - 1));

  whenE (node_window < 1 ∨ node_window > 256) $ throwError $ RangeError 1 256;

  whenE (node_window < 1 ∨ node_window > node_size - node_offset) $
    throwError $ RangeError 1 (of_nat (node_size - node_offset));

  oref ← returnOk $ obj_ref_of node_cap;
  offsets ← returnOk $ map (nat_to_cref radix_bits)
                         [node_offset ..< node_offset + node_window];
  slots ← returnOk $ map (λcref. (oref, cref)) offsets;

  mapME_x ensure_empty slots;

  free_index ← liftE $ const_on_failure (free_index_of cap) $ (doE
    ensure_no_children slot;
    returnOk 0
  odE);

  free_ref ← returnOk ( get_free_ref (obj_ref_of cap) free_index);
  object_size ← returnOk ( obj_bits_api new_type user_obj_size);
  aligned_free_ref ← returnOk ( alignUp free_ref object_size);
  untyped_free_bytes ← returnOk (obj_size cap - of_nat (free_index));

  max_count ← returnOk ( untyped_free_bytes >> object_size);
  whenE (unat max_count < node_window) $
        throwError $ NotEnoughMemory $ untyped_free_bytes;
  (ptr, block_size) ← case cap of
                        UntypedCap p n f ⇒ returnOk (p,n)
                      | _ ⇒ fail;
  returnOk $ Retype slot ptr aligned_free_ref new_type user_obj_size slots
odE"
```

37.7 Toplevel invocation decode.

This definition is the toplevel decoding definition; it dispatches to the above definitions, after checking, in some cases, whether the invocation is allowed.

definition

```
  decode_invocation ::
  "data ⇒ data list ⇒ cap_ref ⇒ cslot_ptr ⇒ cap ⇒ (cap × cslot_ptr) list ⇒ (invocation,'z::state_ext)
se_monad"
```

where

```
  "decode_invocation label args cap_index slot cap excaps ≡
  case cap of
    EndpointCap ptr badge rights ⇒
      if AllowSend ∈ rights then
        returnOk $ InvokeEndpoint ptr badge (AllowGrant ∈ rights)
      else throwError $ InvalidCapability 0
  | AsyncEndpointCap ptr badge rights ⇒
      if AllowSend ∈ rights then
        returnOk $ InvokeAsyncEndpoint ptr badge (if args = [] then 0 else hd args)
      else throwError $ InvalidCapability 0
  | ReplyCap thread False ⇒
      returnOk $ InvokeReply thread slot
  | IRQControlCap ⇒
      liftME InvokeIRQControl
        $ decode_irq_control_invocation label args slot (map fst excaps)
  | IRQHandlerCap irq ⇒
      liftME InvokeIRQHandler
        $ decode_irq_handler_invocation label irq excaps
  | ThreadCap ptr ⇒
      liftME InvokeTCB $ decode_tcb_invocation label args cap slot excaps
  | DomainCap ⇒
      liftME (split InvokeDomain) $ decode_domain_invocation label args excaps
  | CNodeCap ptr bits _ ⇒
      liftME InvokeCNode $ decode_cnode_invocation label args cap (map fst excaps)
  | UntypedCap ptr sz fi ⇒
      liftME InvokeUntyped $ decode_untyped_invocation label args slot cap (map fst excaps)
  | ArchObjectCap arch_cap ⇒
      liftME InvokeArchObject $
        arch_decode_invocation label args cap_index slot arch_cap excaps
  | _ ⇒
      throwError $ InvalidCapability 0"
```

end

38 An Initial Kernel State

```
theory Init_A
imports Retype_A
begin
```

This is not a specification of true kernel initialisation. This theory describes a dummy initial state only, to show that the invariants and refinement relation are consistent.

```
definition
  init_tcb_ptr :: word32 where
  "init_tcb_ptr = kernel_base + 0x2000"

definition
  init_irq_node_ptr :: word32 where
  "init_irq_node_ptr = kernel_base + 0x3000"

definition
  init_globals_frame :: word32 where
  "init_globals_frame = kernel_base + 0x5000"

definition
  init_global_pd :: word32 where
  "init_global_pd = kernel_base + 0x60000"

definition
  "init_arch_state ≡ (|
    arm_globals_frame = init_globals_frame,
    arm_asid_table = empty,
    arm_hwasid_table = empty,
    arm_next_asid = 0,
    arm_asid_map = empty,
    arm_global_pd = init_global_pd,
    arm_global_pts = [],
    arm_kernel_vspace = λref.
      if ref ∈ {kernel_base .. kernel_base + mask 20}
      then ArmVSpaceKernelWindow
      else ArmVSpaceInvalidRegion
  |)"

definition
  empty_context :: user_context where
  "empty_context ≡ λ_. 0"

definition
  [simp]:
  "global_pd ≡ (λ_. InvalidPDE)( ucast (kernel_base >> 20) := SectionPDE (addrFromPPtr kernel_base)
{} 0 {})"

definition
  "init_kheap ≡
  (λx. if ∃irq :: irq. init_irq_node_ptr + (ucast irq << cte_level_bits) = x
       then Some (CNode 0 (empty_cnode 0)) else None)
  (idle_thread_ptr ↦ TCB (|
```

```
    tcb_ctable = NullCap,
    tcb_vtable = NullCap,
    tcb_reply = NullCap,
    tcb_caller = NullCap,
    tcb_ipcframe = NullCap,
    tcb_state = IdleThreadState,
    tcb_fault_handler = replicate word_bits False,
    tcb_ipc_buffer = 0,
    tcb_context = empty_context,
    tcb_fault = None
  ⦈,
  init_globals_frame ↦ ArchObj (DataPage ARMSmallPage),
  init_global_pd ↦ ArchObj (PageDirectory global_pd)
  )"
```

definition

```
  "init_cdt ≡ empty"
```

definition

```
  "init_ioc ≡
   λ(a,b). (∃obj. init_kheap a = Some obj ∧
                 (∃cap. cap_of obj b = Some cap ∧ cap ≠ cap.NullCap))"
```

definition

```
  "init_A_st ≡ ⦇
    kheap = init_kheap,
    cdt = init_cdt,
    is_original_cap = init_ioc,
    cur_thread = idle_thread_ptr,
    idle_thread = idle_thread_ptr,
    machine_state = init_machine_state,
    interrupt_irq_node = λirq. init_irq_node_ptr + (ucast irq << cte_level_bits),
    interrupt_states = λ_. Structures_A.IRQInactive,
    arch_state = init_arch_state,
    exst = ext_init
  ⦈"
```

end

39 System Calls

theory Syscall_A
imports
  "../design/Event_H"
  Decode_A
  Init_A
begin

This theory defines the entry point to the kernel, `call_kernel`, which is called by the assembly stubs after switching into kernel mode and saving registers. There are five kinds of events that end up in a switch to kernel mode. These events are described by the enumerated type `event`, defined in chapter 9. One of the five events is an actual system call by the user, the other four are related to faults and interrupts. There are seven different kinds of user system calls, described by the enumerated type `syscall`, also defined in chapter 9.

The `call_kernel` function delegates the event-specific behaviour to `handle_event` which in turn further dispatches to system-call specific handler functions.

In particular, two of the system calls, namely `SysSend` and `SysCall`, correspond to a method invocation on capabilities. They are handled in the `handle_invocation` operation, which is made up of three phases: first checking if the caller has the capabilities to perform the operation, then decoding the arguments received from the user (using the `decode_invocation` operation), and finally actually performing the invocation (using the `perform_invocation`). These three phases are wrapped into a more generic `syscall` framework function described below.

39.1 Generic system call structure

The `syscall` operation generically describes the usual execution of system calls in three phases, where the first phase may result in a fault, the second phase may result in an error and the third phase may be interrupted. The first two phases are used for argument decoding and checking. The last phase commits and executes the system call.

The `syscall` operation has five arguments:

- the first operation `m_fault` to execute, that may result in a fault;
- the fault handler `h_fault` to execute if the first operation resulted in a fault;
- the second operation `m_error` to execute (if no fault occured in the first operation); this second operation may result in an error;
- the error handler `h_error` to execute if the second operation resulted in an error;
- the third and last operation `h_error` to execute (if no error occured in the second operation); this operation may be interrupted.

definition
```
  syscall :: "('a,'z::state_ext) f_monad
                 ⇒ (fault ⇒ ('c,'z::state_ext) s_monad)
                 ⇒ ('a ⇒ ('b,'z::state_ext) se_monad)
                 ⇒ (syscall_error ⇒ ('c,'z::state_ext) s_monad)
              ⇒ ('b ⇒ ('c,'z::state_ext) p_monad) ⇒ ('c,'z::state_ext) p_monad"
```
where
```
"syscall m_fault h_fault m_error h_error m_finalise ≡ doE
    r_fault ← without_preemption $ m_fault;
```

```
    case r_fault of
          Inl f ⇒   without_preemption $ h_fault f
        | Inr a ⇒   doE
            r_error ← without_preemption $ m_error a;
            case r_error of
                  Inl e ⇒   without_preemption $ h_error e
                | Inr b ⇒   m_finalise b
        odE
odE"
```

39.2 System call entry point

The kernel user can perform seven kinds of system calls, described by the enumerated type `syscall`, defined in section 39.1. These seven system calls can be categorised into two broad families: sending messages and receiving messages, the two main services provided by the kernel.

The usual case for sending messages (`Send` event) consists of the user sending a message to an object, without expecting any answer. The sender is blocked until the receiver is waiting to receive. In case the receiver is not trusted, an explicit non-blocking send operation can be used (`NBSend` event). If a reply is requested from the receiver, the Call operation can be used (`Call` event). The Call operation will automatically provide a `Reply` capability to the receiver.

All three sending operations are handled by the `handle_invocation` operation, which takes two boolean arguments, one to indicate if a reply is requested and the other to indicate if the send is blocking or not.

The other direction is the reception of messages. This is done by performing a Wait operation on an endpoint kernel object. The receiver is then blocked until a sender performs a Send operation on the endpoint object, resulting in a message transfer between the sender and the receiver. The receiver may also perform a Reply operation (`Reply` event) in response to a `Call`, which is always non-blocking. When the receiver is a user-level server, it generally runs a loop waiting for messages. On handling a received message, the server will send a reply and then return to waiting. To avoid excessive switching between user and kernel mode, the kernel provides a ReplyWait operation, which is simply a Reply followed by Wait.

Finally, the last event, `Yield`, enables the user to donate its remaining timeslice.

The invocation is made up of three phases. The first phase corresponds to a lookup of capabilities to check that the invocation is valid. This phase can result in a fault if a given CSpace address is invalid (see the function `resolve_address_bits`). The second phase is the decoding of the arguments given by the user. This is handled by the `decode_invocation` operation. This operation can result in an error if, for example, the number of arguments is less than required by the operation, or if some argument capability has the wrong type. Finally, the actual invocation is performed, using the `perform_invocation` function. Note that this last phase is preemptable.

```
fun
  perform_invocation :: "bool ⇒ bool ⇒ invocation ⇒ (data list,'z::state_ext) p_monad"
where
  "perform_invocation block call (InvokeUntyped i) =
    doE
      without_preemption $ invoke_untyped i;
      returnOk []
    odE"

| "perform_invocation block call (InvokeEndpoint ep badge canGrant) =
    (without_preemption $ do
       thread ← gets cur_thread;
       send_ipc block call badge canGrant thread ep;
       return []
```

```
    od)"

| "perform_invocation block call (InvokeAsyncEndpoint ep badge message) =
    doE
      without_preemption $ send_async_ipc ep badge message;
      returnOk []
    odE"

| "perform_invocation block call (InvokeTCB i) = invoke_tcb i"

| "perform_invocation block call (InvokeDomain tptr d) = invoke_domain tptr d"

| "perform_invocation block call (InvokeReply thread slot) =
    liftE (do
      sender ← gets cur_thread;
      do_reply_transfer sender thread slot;
      return []
    od)"

| "perform_invocation block call (InvokeCNode i) =
    doE
      invoke_cnode i;
      returnOk []
    odE"

| "perform_invocation block call (InvokeIRQControl i) =
   doE
     invoke_irq_control i;
     returnOk []
   odE"

| "perform_invocation block call (InvokeIRQHandler i) =
   doE
     liftE $ invoke_irq_handler i;
     returnOk []
   odE"

| "perform_invocation block call (InvokeArchObject i) =
    arch_perform_invocation i"
```

definition
```
  handle_invocation :: "bool ⇒ bool ⇒ (unit,'z::state_ext) p_monad"
```
where
```
  "handle_invocation calling blocking ≡ doE
    thread ← liftE $ gets cur_thread;
    info ← without_preemption $ get_message_info thread;
    ptr ← without_preemption $ liftM data_to_cptr $
          as_user thread $ get_register cap_register;
    syscall
      (doE
         (cap, slot) ← cap_fault_on_failure (of_bl ptr) False $
                            lookup_cap_and_slot thread ptr;
         buffer ← liftE $ lookup_ipc_buffer False thread;
         extracaps ← lookup_extra_caps thread buffer info;
         returnOk (slot, cap, extracaps, buffer)
       odE)
      (λfault. when blocking $ handle_fault thread fault)
```

```
    (λ(slot,cap,extracaps,buffer). doE
          args ← liftE $ get_mrs thread buffer info;
          decode_invocation (mi_label info) args ptr slot cap extracaps
     odE)
    (λerr. when calling $
          reply_from_kernel thread $ msg_from_syscall_error err)
    (λoper. doE
          without_preemption $ set_thread_state thread Restart;
          reply ← perform_invocation blocking calling oper;
          without_preemption $ do
              state ← get_thread_state thread;
              case state of
                    Restart ⇒ do
                        when calling $
                            reply_from_kernel thread (0, reply);
                        set_thread_state thread Running
                   od
                  | _ ⇒  return ()
          od
     odE)
odE"
```

definition

```
handle_yield :: "(unit,'z::state_ext) s_monad" where
"handle_yield ≡ do
   thread ← gets cur_thread;
   do_extended_op (tcb_sched_action (tcb_sched_dequeue) thread);
   do_extended_op (tcb_sched_action (tcb_sched_append) thread);
   do_extended_op (reschedule_required)
 od"
```

definition

```
handle_send :: "bool ⇒ (unit,'z::state_ext) p_monad" where
"handle_send bl ≡ handle_invocation False bl"
```

definition

```
handle_call :: "(unit,'z::state_ext) p_monad" where
"handle_call ≡ handle_invocation True True"
```

definition

```
delete_caller_cap :: "obj_ref ⇒ (unit,'z::state_ext) s_monad" where
"delete_caller_cap t ≡ cap_delete_one (t, tcb_cnode_index 3)"
```

definition

```
handle_wait :: "(unit,'z::state_ext) s_monad" where
"handle_wait ≡ do
   thread ← gets cur_thread;

   delete_caller_cap thread;

   ep_cptr ← liftM data_to_cptr $ as_user thread $
               get_register cap_register;

   (cap_fault_on_failure (of_bl ep_cptr) True $ doE
      ep_cap ← lookup_cap thread ep_cptr;

      let flt = (throwError $ MissingCapability 0)
```

```
        in
        case ep_cap
          of EndpointCap ref badge rights ⇒
             (if AllowRecv ∈ rights
              then liftE $ receive_ipc thread ep_cap
              else flt)
           | AsyncEndpointCap ref badge rights ⇒
             (if AllowRecv ∈ rights
              then liftE $ receive_async_ipc thread ep_cap
              else flt)
           | _ ⇒ flt
      odE)
      <catch> handle_fault thread
   od"
```

definition

```
  handle_reply :: "(unit,'z::state_ext) s_monad" where
 "handle_reply ≡ do
    thread ← gets cur_thread;
    caller_cap ← get_cap (thread, tcb_cnode_index 3);
    case caller_cap of
      ReplyCap caller False ⇒ do_reply_transfer thread caller (thread, tcb_cnode_index 3)
    | NullCap ⇒ return ()
    | _ ⇒ fail
  od"
```

39.3 Top-level event handling

fun

```
  handle_event :: "event ⇒ (unit,'z::state_ext) p_monad"
```

where

```
  "handle_event (SyscallEvent call) =
   (case call of
          SysSend ⇒ handle_send True
        | SysNBSend ⇒ handle_send False
        | SysCall ⇒ handle_call
        | SysWait ⇒ without_preemption handle_wait
        | SysYield ⇒ without_preemption handle_yield
        | SysReply ⇒ without_preemption handle_reply
        | SysReplyWait ⇒ without_preemption $ do
            handle_reply;
            handle_wait
          od)"

| "handle_event (UnknownSyscall n) = (without_preemption $ do
    thread ← gets cur_thread;
    handle_fault thread $ UnknownSyscallException $ of_nat n;
    return ()
  od)"

| "handle_event (UserLevelFault w1 w2) = (without_preemption $ do
    thread ← gets cur_thread;
    handle_fault thread $ UserException w1 (w2 && mask 29);
    return ()
  od)"

| "handle_event Interrupt = (without_preemption $ do
```

```
    active ← do_machine_op getActiveIRQ;
    case active of
       Some irq ⇒ handle_interrupt irq
     | None ⇒ return ()
  od)"

| "handle_event (VMFaultEvent fault_type) = (without_preemption $ do
    thread ← gets cur_thread;
    handle_vm_fault thread fault_type <catch> handle_fault thread;
    return ()
  od)"
```

39.4 Kernel entry point

This function is the main kernel entry point. The main event loop of the kernel handles events, handles a potential preemption interrupt, schedules and switches back to the active thread.

definition

```
  call_kernel :: "event ⇒ (unit,'z::state_ext_sched) s_monad" where
  "call_kernel ev ≡ do
       handle_event ev <handle>
           (λ_. without_preemption $ do
                 irq ← do_machine_op getActiveIRQ;
                 when (irq ≠ None) $ handle_interrupt (the irq)
               od);
       schedule;
       activate_thread
   od"
```

end

40 Glossary

aep, async_ep, AsyncEndpoint Asynchronous Communication Endpoint. A kernel object in seL4 for asynchronous message passing.

asid, asid pool Address Space Identifier. ASIDs are associated with page directories (PDs) and define the virtual address space of a thread. Multiple threads may be in the same address space. Since ARM hardware supports only 255 different ASIDs, seL4 on ARM supports the concept of virtual ASIDs that are mapped to hardware ASIDS managed in a two-level structure. The user manages only the second level of this structure: the asid pool. An asid pool can be seen as a set of virtual ASIDs that can be connected to and disconnected from page directories.

badge A badge is a piece of extra information stored in an endpoint capability. It can be used by applications to identify caps previously handed out to clients.

cap, capability The main access control concept in seL4. A capability conceptually is a reference to a kernel object together with a set of access rights. Most seL4 capabilities store additional bits of information. Some of this additional information may be exposed to the user, but the bulk of it is kernel-internal book-keeping information. Capabilities are stored in CNodes and TCBs.

cdt Capability Derivation Tree. A kernel-internal data structure that tracks the child/parent relationship between capabilities. Capabilities to new objects are children of the Untyped capability the object was created from. Capabilities can also be copied; in this case the user may specify if the operation should produce children or siblings of the source capability. The revoke operation will delete all children of the invoked capability.

cnode Capability Node. Kernel-controlled storage that holds capabilities. Capability nodes can be created in different sizes and be shared between CSpaces. CNodes can be pointed to by capabilities themselves.

cspace A directed graph of CNodes. The CSpace of a thread defines the set of capabilities it has access to. The root of the graph is the CNode capability in the CSpace slot of the thread. The edges of the graph are the CNode capabilities residing in the CNodes spanned by this root.

cptr Capability Pointer. A user-level reference to a capability, relative to a specified root CNode or the thread's CSpace root. In this specification, a user-level capability pointer is a sequence of bits that define a path in the CSpace graph that should end in a capability slot. The kernel resolves user-level capability pointers into capability slot pointers (cslot_ptr).

cslot_ptr Capability Slot Pointer. A kernel-internal reference to a capability. It identifies the kernel object the capability resides in as well as the location (slot) of the capability inside this object.

ep Endpoint. Without further qualifier refers to a synchronous communications (IPC) endpoint in seL4.

guard Guard of a CNode capability. From the user's perspetive the CSpace of a thread is organised as a guardedage table. The kernel will resolve user capability pointers into internal capability slot pointers. The guard of one link/edge in the CSpace graph defines a sequence of bits that will be stripped from the user-level capability pointer before resolving resumes at the next CNode.

ipc Inter Process Communication. In seL4: sending short messages between threads. The kernel supports both synchronous and asynchronous message passing. To communicate via IPC in seL4, the receiver listens at an endpoint object and the sender sends to the same endpoint object.

kheap Kernel Heap. This is not an actual C heap in the sense that it supports malloc and free. Rather, it is the kernel virtual memory view of physical memory in the machine.

pd Page Directory. The first level of an ARM virtual memory page table. A page directory can be seen as an array of page directory entries (PDEs).

pde Page Directory Entry. One entry in the page directory. It either is invalid, contains a translation to a frame, or a translation to a second level page table.

pt Page Table. The second level of an ARM virtual memory page table. It can be seen as an array of page table entries.

pte Page Table Entry. One entry of a second level ARM page table. It is either invalid or contains a translation to a frame.

replycap Reply Capability. Reply capabilities are created automatically in the receiver of a Call IPC. The reply capability points back to the sender of the call and can be used to send a reply efficiently without having to explicitly set up a return channel. Reply capabilities can be invoked only once. Internally, reply capabilities are created as copies of so-called Master Reply Capabilities that are always present in the master reply slot of the sender's TCB.

tcb Thread Control Block. The kernel object that stores management data for threads, such as the thread's CSpace, VSpace, thread state, or user registers.

thread The CPU execution abstraction of seL4.

vm Virtual Memory. The concept of translating virutal memory addresses to physical frames. SeL4 uses the MMU (Memory Management Unit) to provide controlled virtual memory mechanisms to the user, to protect kernel data and code from users, and to enforce separation between users (if set up correctly).

vspace In analogy to CSpace, the virtual memory space of a thread. In the ARM case, the VSpace of a thread is defined by its top-level page directory and that page directory's ASID.

zombie Zombie Capability. This capability is not accessible to the user. It stores continuation information for the preemtable capability delete operation.

Bibliography

[1] Andrew Boyton. A verified shared capability model. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 4th Workshop on Systems Software Verification*, volume 254 of *Electronic Notes in Computer Science*, pages 25–44, Aachen, Germany, October 2009. Elsevier.

[2] David Cock. Bitfields and tagged unions in C: Verification through automatic generation. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, August 2008.

[3] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182, Montreal, Canada, August 2008. Springer.

[4] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

[5] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, September 2006.

[6] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. Technical Report NRL-1474, NICTA, October 2007. Available from http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf.

[7] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In Jim Woodcock and Natarajan Shankar, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 99–114, Toronto, Canada, October 2008. Springer.

[8] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, San Diego, CA, USA, May 2007.

[9] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM Operating Systems Review*, 41(4):3–11, July 2007.

[10] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[11] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[12] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4 — formally verifying a high-performance microkernel. In *Proceedings of the 14th International Conference on Functional Programming*, pages 91–96, Edinburgh, UK, August 2009. ACM.

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[14] Daniel Matichuk and Toby Murray. Extensible specifications for automatic re-use of specifications and proofs. volume 7504 of *Lecture Notes in Computer Science*, pages 333–341, October 2012.

[15] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. 2002.

[16] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, August 2008.

[17] Harvey Tuch. Structured types and separation logic. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd Workshop on Systems Software Verification*, volume 217 of *Electronic Notes in Computer Science*, pages 41–59, Sydney, Australia, February 2008. Elsevier.

[18] Harvey Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning: Special Issue on Operating System Verification*, 42(2–4):125–187, April 2009.

[19] Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 474–488, Montego Bay, Jamaica, December 2005.

[20] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, Santa Fe, NM, USA, June 2005. USENIX.

[21] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, January 2007.

[22] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515, Munich, Germany, August 2009. Springer.