NICTA Trustworthy Systems
http://ssrg.nicta.com.au/projects/TS/

seL4 Reference Manual (RT) Version 1.0.0-rt-dev

Trustworthy Systems Team, NICTA
ssrg@nicta.com.au

1 June 2016

Acknowledgements

The primary authors of this document are Matthew Grosvenor and Adam Walker, with contributions from Adrian Danis, Andrew Boyton, David Greenaway, Etienne Le Sueur, Gernot Heiser, Gerwin Klein, Godfrey van der Linden, Kevin Elphinstone, Matthew Fernandez, Matthias Daum, Michael von Tessin, Peter Chubb, Simon Winwood, Thomas Sewell, Timothy Bourke and Toby Murray. All authors and contributors can be contacted at firstname.lastname@nicta.com.au.

Contents

List of Tables

List of Figures

Chapter 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, it provides a small number of services to applications, such as abstractions to create and manage virtual address spaces, threads, and inter-process communication (IPC). The small number of services provided by seL4 directly translates to a small implementation of approximately 8700 lines of C code. This has allowed the ARMv6 version of the kernel to be formally proven in the Isabelle/HOL theorem prover to adhere to its formal specification [Boy09, CKS08, DEK+06, EKE08, KEH+09, TKN07, WKS+09], which in turn enabled proofs of the kernel's enforcement of integrity [SWG+11] and confidentiality [MMB+13]. The kernel's small size was also instrumental in performing a complete and sound analysis of worst-case execution time [BSC+11, BSH12].

This manual describes the seL4 kernel's API from a user's point of view. The document starts by giving a brief overview of the seL4 microkernel design, followed by a reference of the high-level API exposed by the seL4 kernel to userspace.

While we have tried to ensure that this manual accurately reflects the behaviour of the seL4 kernel, this document is by no means a formal specification of the kernel. When the precise behaviour of the kernel under a particular circumstance needs to be known, users should refer to the seL4 abstract specification, which gives a formal description of the seL4 kernel.

Chapter 2

Kernel Services and Objects

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 seL4 provides are as follows:

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

Scheduling contexts are an abstraction of CPU execution time.

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

Inter-process communication (IPC) via *endpoints* allows threads to communicate using message passing;

Notifications provide a non-blocking signalling mechanism similar to binary semaphores;

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.

This chapter gives an overview of these services, describes how kernel objects are accessed by userspace applications, and describes how new objects can be created.

2.1 Capability-based Access Control

The seL4 microkernel provides a capability-based access-control model. Access control governs all kernel services; in order to perform an operation, 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 methods may be 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. This means, the seL4 capability model is an instance of a *segregated* (or *partitioned*) capability system, 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 of a capability in a capability space is the concatenation of the indices of slots within CNodes forming the path to the destination slot; we discuss CNode objects in detail in Chapter 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 to withdraw authority. Revocation recursively removes any capabilities that have been derived from the original capability being revoked. The propagation of capabilities through the system is controlled by a *take-grant*-based model [EKE08, Boy09].

2.2 System Calls

The seL4 kernel provides a message-passing service for communication between threads. This mechanism is also used for communication with kernel-provided services. There is a standard message format, each message containing a number of data words and possibly some capabilities. The structure and encoding of these messages are described in detail in Chapter 4.

Threads send messages by invoking capabilities within their capability space. When an endpoint capability is invoked in this way, the message will be transferred through the kernel to another thread. When capabilities to kernel objects are invoked, the message will be interpreted as a method invocation in a manner specific to the type of kernel object. For example, invoking a thread control block (TCB) capability with a correctly formatted message will suspend the target thread.

Logically, the kernel provides two system calls, *Send*, *Receive*. However, there are also combinations and variants of the basic *Send* and *Receive* calls, e.g. the *Call* operation, which consists of a send followed by a *Receive* from the same object. Methods on kernel objects other than endpoints and notifications are all mapped to *Send* or *Call*, depending on whether or not the method returns a result.

The complete set of system calls is:

`seL4_Send()` delivers a message through the named capability and the application to continue. If the invoked capability is an endpoint, and no receiver is ready to receive the message immediately, the sending thread will block until the message can be delivered. No error code or response will be returned by the receiving object.

`seL4_NBSend()` performs a polling send on an endpoint. It is similar to `seL4_Send()`, except that it is guaranteed not to block. If the message cannot be delivered immediately, i.e. there is no receiver waiting on the destination Endpoint, the message is silently dropped. Like `seL4_Send()`, no error code or response will be returned.

`seL4_Call()` combines `seL4_Send()` and `seL4_Recv()`. The call blocks the sending thread until its message is delivered and a reply message is received. When the sent message is delivered to another thread (via an Endpoint), the kernel adds an additional '*reply*' capability to the message that is delivered to the receiver, giving the latter the right to reply to the original sender. The reply capability is deposited in a dedicated slot in the receiver's TCB, and is a single-use right, meaning that the kernel invalidates it as soon as it has been invoked.

The `seL4_Call()` operation exists not only for efficiency reasons (combining two operations into a single system call). It differs from `seL4_Send()` immediately followed by `seL4_Recv()` in two ways:

1. the single-use reply capability is created to establish a reply channel with minimal trust;
2. the transition from send to recv phase is atomic, meaning it cannot be preempted, and the receiver can reply without any risk of blocking.

When invoking capabilities to kernel services, using `seL4_Call()` allows the kernel to return an error code or other response through the reply message.

`seL4_Recv()` is used by a thread to receive messages through endpoints or notifications. If no sender or notification is pending, the caller will block until a message or notification can be delivered. This system call works only on Endpoint or Notification capabilities, raising a fault (see section 6.2) when attempted with other capability types.

`seL4_Reply()` is used to respond to a `seL4_Call()`, using the reply capability generated by the `seL4_Call()` system call and stored in the replying thread's TCB. It delivers the message to the thread that invoked the `seL4_Call()`, waking it in the process.

There is space for only one reply capability in each thread's TCB, so the `seL4_Reply()` syscall can be used to reply to the most recent caller only. The `seL4_CNode_SwapCaller()` and `seL4_CNode_SwapTCBCaller()`ethods that will be described later can be used to swap the reply capability into regular capability space, where it can be used with `seL4_Send()` or swapped back in.

`seL4_ReplyRecv()` combines `seL4_Reply()` and `seL4_Recv()`. It exists mostly 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. The transition from the reply to the receive phase is also atomic.

`seL4_NBRecv()` is used by a thread to check for signals pending on a notification object or messages pending on an endpoint without blocking. This system call works only on endpoints and notification object capabilities, raising a fault (see section 6.2) when attempted with other capability types.

`seL4_SignalRecv()` combines a `seL4_NBSend()` with a `seL4_Recv()`, and can be called on two different objects (notification objects, reply caps or endpoints). Regardless of the success of the non-blocking send, the receive operation will continue. The send operation does not transmit a message.

2.3 Kernel Objects

In this section we give a brief overview of the kernel-implemented object types whose instances (also simply called *objects*) can be invoked by applications. The interface to these objects forms the interface to the kernel itself. The creation and use of kernel services is achieved by the creation, manipulation, and combination of these kernel objects:

CNodes (see Chapter 3) store capabilities, giving threads permission to invoke methods on particular objects. Each 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.

Thread Control Blocks (TCBs; see Chapter 6) represent a thread of execution in seL4. Threads are the unit of execution that is scheduled, blocked, unblocked, etc., depending on the application's interaction with other threads.

Scheduling contexts (SchedulingContexts; see Chapter 6) represent CPU time in seL4. Users can create scheduling contexts from untyped objects, however on creation scheduling contexts are *empty* and do not represent any time. Initially, there is a capability to SchedControl, which allows scheduling context to be populated with parameters, which combined with priority control threads access to CPU time.

Endpoints (see Chapter 4) facilitate message-passing communication between threads. IPC is synchronous: A thread trying to send or receive on an endpoint blocks until the message can be delivered. This means that message delivery only happens if a sender and a receiver rendezvous at the endpoint, and the kernel can deliver the message with a single copy (or without copying for short messages using only registers).

A capability to an endpoint can be restricted to be send-only or receive-only. Additionally, Endpoint capabilities can have the grant right, which allows sending capabilities as part of the message.

Notification Objects (see Chapter 5) provide a simple signalling mechanism. A Notification is a word-size array of flags, each of which behaves like a binary semaphore. Operations are *signalling* a subset of flags in a single operation, polling to check any flags, and blocking until any are signalled. Notification capabilities can be signal-only or wait-only.

Virtual Address Space Objects (see Chapter 7) are used to construct a virtual address space (or VSpace) for one or more threads. These objects largely directly correspond to those of the hardware, and as such are architecture-dependent. The kernel also includes ASID Pool and ASID Control objects for tracking the status of address spaces.

Interrupt Objects (see Chapter 8) give applications the ability to receive and acknowledge interrupts from hardware devices. Initially, there is a capability to IRQControl, which allows for the creation of IRQHandler capabilities. An IRQHandler capability permits the management of a specific interrupt source associated with a specific device. It is delegated to a device driver to access an interrupt source. The IRQHandler object allows threads to wait for and acknowledge individual interrupts.

Untyped Memory (see Section 2.4) is the foundation of memory allocation in the seL4 kernel. Untyped memory capabilities have a single method which allows the creation of new kernel objects. If the method succeeds, the calling thread gains access to capabilities to the newly-created objects. Additionally, untyped memory objects can be divided into a group of smaller untyped memory objects allowing delegation of part (or all) of the system's memory. We discuss memory management in general in the following sections.

2.4 Kernel Memory Allocation

The seL4 microkernel does not dynamically allocate memory for kernel objects. Instead, objects must be explicitly created from application-controlled memory regions via Untyped Memory capabilities. Applications must have explicit authority to memory (through these Untyped Memory capabilities) in order to create new objects, and all objects consume a fixed amount of memory once created. These 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. There are no arbitrary resource limits in the kernel apart from those dictated by the hardware[1], and so many denial-of-service attacks via resource exhaustion are avoided.

At boot time, seL4 pre-allocates the memory required for the kernel itself, including the code, data, and stack sections (seL4 is a single kernel-stack operating system). It then creates an initial user thread (with an appropriate address and capability space). The kernel than hands all remaining memory to the initial thread in the form of capabilities to Untyped Memory, and some additional capabilities to kernel objects that were required

[1]The treatment of virtual ASIDs imposes a fixed number of address spaces. This limitation is to be removed in future versions of seL4.

to bootstrap the initial thread. These Untyped Memory regions can then be split into smaller regions or other kernel objects using the `seL4_Untyped_Retype()` method; the created objects are termed *children* of the original untyped memory object.

The user-level application that creates an object using `seL4_Untyped_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.

2.4.1 Reusing 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 Untyped Memory regions to be reused. Reusing a region of memory is allowed only when there are no dangling references (i.e., capabilities) left to the objects inside that memory. The kernel tracks *capability derivations*, i.e., the children generated by the methods `seL4_Untyped_Retype()`, `seL4_CNode_Mint()`, `seL4_CNode_Copy()`, and `seL4_CNode_Mutate()`.

The tree structure so generated is termed the *capability derivation tree* (CDT).[2] 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.

For each Untyped Memory region, the kernel keeps a *watermark* recording how much of the region has previously been allocated. Whenever a user requests the kernel to create new objects in an untyped memory region, the kernel will carry out one of two actions: if there are already existing objects allocated in the region, the kernel will allocate the new objects at the current watermark level, and increase the watermark. If all objects previously allocated in the region have been deleted, the kernel will reset the watermark and start allocating new objects from the beginning of the region again.

Finally, the `seL4_CNode_Revoke()` method provided by CNode objects destroys all capabilities derived from the argument capability. Revoking the last capability to a kernel object triggers the *destroy* operation on the now unreferenced object. This simply cleans up any in-kernel dependencies between it, other objects and the kernel.

By calling `seL4_CNode_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 invocation there are no valid references to any object within the untyped region, and the region may be safely retyped and reused.

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

Chapter 3

Capability Spaces

Recall from Section 2.1 that seL4 implements a capability-based access control model. Each userspace thread has an associated *capability space* (CSpace) that contains the capabilities that the thread possesses, thereby governing which resources the thread can access.

Recall that capabilities reside within kernel-managed objects known as CNodes. A CNode is a table of slots, each of which may contain a capability. This may include capabilities to further CNodes, forming a directed graph. Conceptually a thread's CSpace is the portion of the directed graph that is reachable starting with the CNode capability that is its CSpace root.

A CSpace address refers to an individual slot (in some CNode in the CSpace), which may or may not contain a capability. Threads refer to capabilities in their CSpaces (e.g. when making system calls) using the address of the slot that holds the capability in question. An address in a CSpace is the concatenation of the indices of the CNode capabilities forming the path to the destination slot; we discuss this further in Section 3.3.

Recall that capabilities can be copied and moved within CSpaces, and also sent in messages (message sending will be described in detail in Section 4.2.2). Furthermore, new capabilities can be *minted* from old ones with a subset of their rights. Recall, from Section 2.4.1, that seL4 maintains a *capability derivation tree* (CDT) in which it tracks the relationship between these copied capabilities and the originals. The revoke method removes all capabilities (in all CSpaces) that were derived from a selected capability. This mechanism can be used by servers to restore sole authority to an object they have made available to clients, or by managers of untyped memory to destroy the objects in that memory so it can be retyped.

seL4 requires the programmer to manage all in-kernel data structures, including CSpaces, from userspace. This means that the userspace programmer is responsible for constructing CSpaces as well as addressing capabilities within them. This chapter first discusses capability and CSpace management, before discussing how capabilities are addressed within CSpaces, i.e. how applications can refer to individual capabilities within their CSpaces when invoking methods.

3.1 Capability and CSpace Management

3.1.1 CSpace Creation

CSpaces are created by creating and manipulating CNode objects. When creating a CNode the user must specify the number of slots that it will have, and this determines the amount of memory that it will use. Each slot requires 16 bytes of physical memory and has the capacity to hold exactly one capability. Like any other object, a CNode must be created by calling `seL4_Untyped_Retype()` on an appropriate amount of untyped memory (see Section 10.3.32). The caller must therefore have a capability to enough untyped memory as well as enough free capability slots available in existing CNodes for the `seL4_Untyped_Retype()` invocation to succeed.

3.1.2 CNode Methods

Capabilities are managed largely through invoking CNode methods.

CNodes support the following methods:

`seL4_CNode_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 and a different guard (see Section 3.3.1). `seL4_CNode_Mint()` can also create a badged capability (see Section 4.2.1) from an unbadged one.

`seL4_CNode_Copy()` is similar to `seL4_CNode_Mint()`, but the newly created capability has the same badge and guard as the original.

`seL4_CNode_Move()` moves a capability between two specified capability slots. You cannot move a capability to the slot in which it is currently.

`seL4_CNode_Mutate()` can move a capability similarly to `seL4_CNode_Move()` and also reduce its rights similarly to `seL4_CNode_Mint()`, although without an original copy remaining.

`seL4_CNode_Rotate()` moves two capabilities between three specified capability slots. It is essentially two `seL4_CNode_Move()` invocations: 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 with the capability in the second slot. The method is atomic; either both or neither capabilities are moved.

`seL4_CNode_Delete()` removes a capability from the specified slot.

`seL4_CNode_Revoke()` is equivalent to calling `seL4_CNode_Delete()` on each derived child of the specified capability. It has no effect on the capability itself, except in very specific circumstances outlined in Section 3.2.

`seL4_CNode_SwapCaller()` swaps a kernel-generated reply capability of the current thread from the special TCB slot it was created in, with the designated CSpace slot.

`seL4_CNode_SwapTCBCaller()` swaps a kernel-generated reply capability of a target thread from the special TCB slot it was created in, with the designated CSpace slot.

`seL4_CNode_Recycle()` is similar to `seL4_CNode_Revoke()`, except that it also resets some aspects of the object to its initial state.

3.1.3 Capability Rights

As mentioned previously, some capability types have *access rights* associated with them. Currently, access rights are associated with capabilities for Endpoints (see Chapter 4), Notifications (see Chapter 5) and Pages (see Chapter 7). The access rights associated with a capability determine the methods that can be invoked. seL4 supports three orthogonal access rights, which are Read, Write and Grant. The meaning of each right is interpreted relative to the various object types, as detailed in Table 3.1.

When an object is first created, the initial capability that refers to it carries the maximum set of access rights. Other, less-powerful capabilities may be manufactured from this original capability, using methods such as `seL4_CNode_Mint()` and `seL4_CNode_Mutate()`. If a greater set of rights than the source capability is specified for the destination capability in either of these invocations, the destination rights are silently downgraded to those of the source.

Type	Read	Write	Grant
Endpoint	Required to receive.	Required to send.	Required to send capabilities (including reply capabilities).
Notification	Required to wait.	Required to signal.	N/A
Page	Required to map the page readable.	Required to map the page writable.	N/A

Table 3.1: seL4 access rights.

3.1.4 Capability Derivation Tree

As mentioned in Section 2.4.1, seL4 keeps track of capability derivations in a capability derivation tree.

Various methods, such as `seL4_CNode_Copy()` or `seL4_CNode_Mint()`, may be used to create derived capabilities. Not all capabilities support derivation. In general, only *original* capabilities support derivation invocations, but there are exceptions. Table 3.2 summarises the conditions that must be met for capability derivation to succeed for the various capability types, and how capability-derivation failures are reported in each case. The capability types not listed can be derived once.

Figure 3.1 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

Cap Type	Conditions for Derivation	Error Code on Derivation Failure
ReplyCap	Cannot be derived	Dependent on syscall
IRQControl	Cannot be derived	Dependent on syscall
Untyped	Must not have children (Section 3.2)	`seL4_RevokeFirst`
Page Table	Must be mapped	`seL4_IllegalOperation`
Page Directory	Must be mapped	`seL4_IllegalOperation`
IO Page Table (IA-32 only)	Must be mapped	`seL4_IllegalOperation`

Table 3.2: Capability derivation.

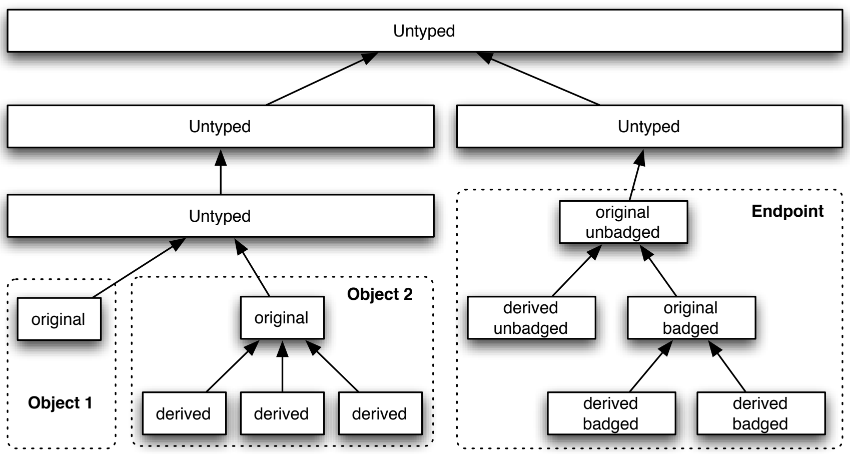

Figure 3.1: Example capability derivation tree.

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. Further copies of these derived capabilities will create siblings, in this case remaining on level 5. There is an exception to this scheme for Endpoint and Notification capabilities — they support an additional layer of depth though *badging*. The original Endpoint or *Notification* capability will be unbadged. Using the mint method, a copy of the capability with a specific *badge* can be created (see Section 4.2.1, Section 5.1). 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.

3.2 Deletion, Revocation, and Recycling

Capabilities in seL4 can be deleted and revoked. Both methods primarily affect capabilities, but they can have side effects on objects in the system where the deletion or revocation results in the destruction of the last capability to an object.

As described above, `seL4_CNode_Delete()` will remove a capability from the specified CNode slot. Usually, this is all that happens. If, however, it was the last typed capability to an object, this object will now be destroyed by the kernel, cleaning up all remaining in-kernel references and preparing the memory for re-use.

If the object to be destroyed was a capability container, i.e. a TCB or CNode, the destruction process will delete each capability held in the container, prior to destroying the container. This may result in the destruction of further objects if the contained capabilities are the last capabilities.[1]

The `seL4_CNode_Revoke()` method will `seL4_CNode_Delete()` all CDT children of the specified capability, but will leave the capability itself intact. If any of the revoked child capabilities were the last capabilities to an object, the appropriate destroy operation is triggered.

Note: `seL4_CNode_Revoke()` may only partially complete in two specific circumstances. The first being where a CNode containing the last capability to the TCB of the thread performing the revoke (or the last capability to the TCB itself) is deleted as a result of the revoke. In this case the thread performing the revoke is destroyed during the revoke and the revoke does not complete. The second circumstance is where the storage containing the capability that is the target of the revoke is deleted as a result of the revoke. In this case, the authority to perform the revoke is removed during the operation and the operation stops part way through. Both these scenarios can be and should be avoided at user-level by construction.

The `seL4_CNode_Recycle()` method can be used to partially reset an object without fully removing all capabilities to it. Invoking it will first revoke all child capabilities, but it will not remove siblings or parents. Only if, after revocation, the capability is the last typed capability to the object, the same destroy operation as in `seL4_CNode_Delete()` will be executed. Otherwise, not all aspects of the object will be reset: for badged endpoint capabilities, only IPC with this badge will be cancelled in the endpoint, for TCBs the capabilities will be reset, for CNodes, the guard on the capability will be reset.

Note that for page tables and page directories, neither `seL4_CNode_Revoke()` nor

[1]The recursion is limited as if the last capability to a CNode is found within the container, the found CNode is not destroyed. Instead, the found CNode is made unreachable by moving the capability pointing to the found CNode into the found cnode itself, by swapping the capability with the first capability in the found cnode, and then trying to delete the swapped capability instead. This breaks the recursion.

The result of this approach is that deleting the last cap to the root CNode of a CSpace does not recursively delete the entire CSpace. Instead, it deletes the root CNode, and the branches of the tree become unreachable, potentially including the deleting of some of the unreachable CNode's caps to make space for the self-referring capability. The practical consequence of this approach is that CSpace deletion requires user-level to delete the tree leaf first if unreachable CNodes are to be avoided. Alternatively, any resulting unreachable CNodes can be cleaned up via revoking a covering untyped capability, however this latter approach may be more complex to arrange by construction at user-level.

seL4_CNode_Recycle() will revoke frame capabilities mapped into the address space. They will only be unmapped from the space.

3.3 CSpace Addressing

When performing a system call, a thread specifies to the kernel the capability to be invoked by giving an address in its CSpace. This address refers to the specific slot in the caller's CSpace that contains the capability to be invoked.

CSpaces are designed to permit sparsity, and the process of looking-up a capability address must be efficient. Therefore, CSpaces are implemented as *guarded page tables*.

As explained earlier, a CSpace is a directed graph of CNode objects, and each CNode is a table of slots, where each slot can either be empty, or contain a capability, which may refer to another CNode. Recall from Section 2.3 that the number of slots in a CNode must be a power of two. A CNode is said to have a *radix*, which is the power to which two is raised in its size. That is, if a CNode has 2^k slots, its radix would be k. The kernel stores a capability to the root CNode of each thread's CSpace in the thread's TCB. Conceptually, a CNode capability stores not only a reference to the CNode to which it refers, but also carries a *guard* value, explained in Section 3.3.1.

3.3.1 Capability Address Lookup

Like a virtual memory address, a capability address is simply an integer. Rather than referring to a location of physical memory (as does a virtual memory address), a capability address refers to a capability slot. When looking up a capability address presented by a userspace thread, the kernel first consults the CNode capability in the thread's TCB that defines the root of the thread's CSpace. It then compares that CNode's *guard* value against the most significant bits of the capability address. If the two values are different, lookup fails. Otherwise, the kernel then uses the next most-significant *radix* bits of the capability address as an index into the CNode to which the CNode capability refers. The slot s identified by these next *radix* bits might contain another CNode capability or contain something else (including nothing). If s contains a CNode capability c and there are remaining bits (following the *radix* bits) in the capability address that have yet to be translated, the lookup process repeats, starting from the CNode capability c and using these remaining bits of the capability address. Otherwise, the lookup process terminates successfully; the capability address in question refers to the capability slot s.

Figure 3.2 demonstrates a valid CSpace with the following features:

- a top level CNode object with a 12-bit guard set to 0x000 and 256 slots;
- a top level CNode with direct object references;
- a top level CNode with two second-level CNode references;
- second level CNodes with different guards and slot counts;
- a second level CNode that contains a reference to a top level CNode;

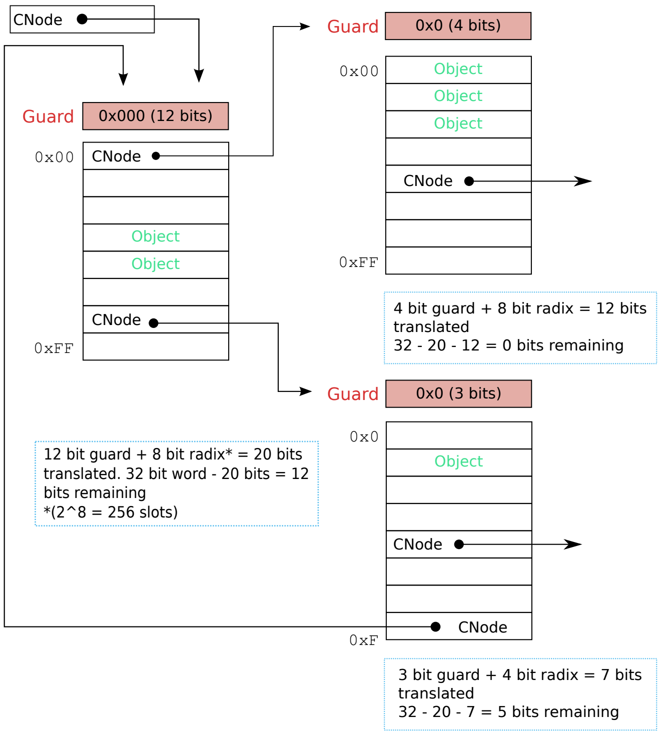

Figure 3.2: An example CSpace demonstrating object references at all levels, various guard and radix sizes and internal CNode references.

- a second level CNode that contains a reference to another CNode where there are some bits remaining to be translated;
- a second level CNode that contains a reference to another CNode where there are no bits remaining to be translated; and
- object references in the second level CNodes.

It should be noted that Figure 3.2 demonstrates only what is possible, not what is usually practical. Although the CSpace is legal, it would be reasonably difficult to work with due to the small number of slots and the circular references within it.

3.3.2 Addressing Capabilities

A capability address is stored in a CPointer (abbreviated CPTR), which is an unsigned integer variable. Capabilities are addressed in accordance with the translation algorithm described above. Two special cases involve addressing CNode capabilities themselves and addressing a range of capability slots.

Recall that the translation algorithm described above will traverse CNode capabilities while there are address bits remaining to be translated. Therefore, in order to address a CNode capability, the user must supply not only a capability address but also specify the maximum number of bits of the capability address that are to be translated, called the *depth limit*.

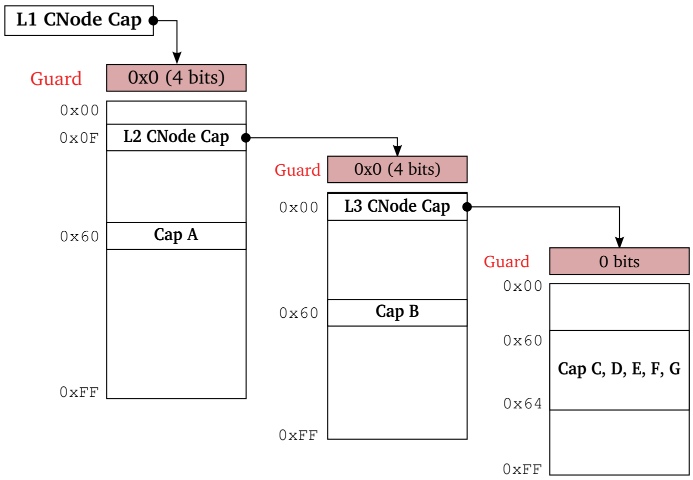

Figure 3.3: An arbitrary CSpace layout.

Certain methods, such as `seL4_Untyped_Retype()`, require the user to provide a range of capability slots. This is done by providing a base capability address, which refers to the first slot in the range, together with a window size parameter, specifying the number of slots (with consecutive addresses, following the base slot) in the range.

Figure 3.3 depicts an example CSpace. In order to illustrate these ideas, we determine the address of each of the 10 capabilities in this CSpace.

Cap A. The first CNode has a 4-bit guard set to 0x0, and an 8-bit radix. Cap A resides in slot 0x60 so it may be referred to by any address of the form 0x060xxxxx (where xxxxx is any number, because the translation process terminates after translating the first 12 bits of the address). For simplicity, we usually adopt the address 0x06000000.

Cap B. Again, the first CNode has a 4-bit guard set to 0x0, and an 8-bit radix. The second CNode is reached via the L2 CNode Cap. It also has a 4-bit guard of 0x0 and Cap B resides at index 0x60. Hence, Cap B's address is 0x00F06000. Translation of this address terminates after the first 24 bits.

Cap C. This capability is addressed via both CNodes. The third CNode is reached via the L3 CNode Cap, which resides at index 0x00 of the second CNode. The third CNode has no guard and Cap C is at index 0x60. Hence, its address is 0x00F00060. Translation of this address leaves 0 bits untranslated.

Caps C–G. This range of capability slots is addressed by providing a base address (which refers to the slot containing Cap C) of 0x00F00060 and a window size of 5.

L2 CNode Cap. Recall that to address a CNode capability, the user must supply not only a capability address but also specify the depth limit, which is the maximum number of bits to be translated. L2 CNode Cap resides at offset 0x0F of the first CNode, which has a 4-bit guard of 0x0. Hence, its address is 0x00F00000, with a depth limit of 12 bits.

L3 CNode Cap. This capability resides at index 0x00 of the second CNode, which is reached by the L2 CNode Cap. The second CNode has a 4-bit guard of 0x0. Hence, the capability's address is 0x00F00000 with a depth limit of 24 bits. Note that the addresses of the L2 and L3 CNode Caps are the same, but that their depth limits are different.

In summary, to refer to any capability (or slot) in a CSpace, the user must supply its address. When the capability might be a CNode, the user must also supply a depth limit. To specify a range of capability slots, the user supplies a starting address and a window size.

3.4 Lookup Failure Description

When a capability lookup fails, a description of the failure is given to either the calling thread or the thread's exception handler in its IPC buffer. The format of the description is always the same but may occur at varying offsets in the IPC buffer depending on how the error occurred. The description format is explained below. The first word indicates the type of lookup failure and the meaning of later words depend on this.

3.4.1 Invalid Root

A CSpace CPTR root (within which a capability was to be looked up) is invalid. For example, the capability is not a `CNode` cap.

Data	Meaning
`Offset + 0`	`seL4_InvalidRoot`

3.4.2 Missing Capability

A capability required for an invocation is not present or does not have sufficient rights.

Data	Meaning
`Offset + 0`	`seL4_MissingCapability`
`Offset + 1`	Bits left

3.4.3 Depth Mismatch

When resolving a capability, a CNode was traversed that resolved more bits than was left to decode in the CPTR or a non-CNode capability was encountered while there were still bits remaining to be looked up.

Data	Meaning
`Offset + 0`	`seL4_DepthMismatch`
`Offset + 1`	Bits of CPTR remaining to decode
`Offset + 2`	Bits that the current CNode being traversed resolved

3.4.4 Guard Mismatch

When resolving a capability, a CNode was traversed with a guard size larger than the number of bits remaining or the CNode's guard did not match the next bits of the CPTR being resolved.

Data	Meaning
`Offset + 0`	`seL4_GuardMismatch`
`Offset + 1`	Bits of CPTR remaining to decode
`Offset + 2`	The CNode's guard
`Offset + 3`	The CNode's guard size

Chapter 4

Message Passing (IPC)

The seL4 microkernel provides a message-passing IPC mechanism for communication between threads. The same mechanism is also used for communication with kernel-provided services. Messages are sent by invoking a capability to a kernel object. Messages sent to `Endpoint` are destined for other threads, while messages sent to other objects are processed by the kernel. This chapter describes the common message format, endpoints, and how they can be used for communication between applications.

4.1 Message Registers

Each message contains a number of message words and optionally a number of capabilities. The message words are sent to or received from a thread by placing them in its *message registers*. The message registers are numbered and the first few message registers are implemented using physical CPU registers, while the rest are backed by a fixed region of memory called the *IPC buffer*. The reason for this design is efficiency: very short messages need not use the memory. The physical CPU registers used for the message registers are described in Table 4.1 for x86 and Table 4.2 for ARM. The IPC buffer is assigned to the calling thread (see Section 6.1 and Section 10.3.23).

Role	CPU Register
Capability register *(in)*	`ebx`
Badge register *(out)*	`ebx`
Message tag *(in/out)*	`esi`
Message register 1 *(in/out)*	`edi`
Message register 2 *(in/out)*	`ebp`

Table 4.1: Physical register allocation for IPC messages on the x86 architecture.

Every IPC message also has a tag (structure `seL4_MessageInfo_t`). The tag consists of four fields: the label, message length, number of capabilities (the `extraCaps` field) and the `capsUnwrapped` field. The message length and number of capabilities determine either the number of message registers and capabilities that the sending thread wishes

Role	CPU Register
Capability register *(in)*	`r0`
Badge register *(out)*	`r0`
Message tag *(in/out)*	`r1`
Message register 1–4 *(in/out)*	`r2` – `r5`

Table 4.2: Physical register allocation for IPC messages on the ARM architecture.

to transfer, or the number of message registers and capabilities that were actually transferred. The label is not interpreted by the kernel and is passed unmodified as the first data payload of the message. The label may, for example, be used to specify a requested operation. The `capsUnwrapped` field is used only on the receive side, to indicate the manner in which capabilities were received. It is described in Section 4.2.2.

Type	**Name**	**Description**
`seL4_MessageInfo_t`	`tag`	Message tag
`seL4_Word[]`	`msg`	Message contents
`seL4_Word`	`userData`	Base address of the structure, used by supporting user libraries
`seL4_CPtr[]` *(in)*	`caps`	Capabilities to transfer
`seL4_CapData_t[]` *(out)*	`badges`	Badges for endpoint capabilities received
`seL4_CPtr`	`receiveCNode`	CPTR to a CNode from which to find the receive slot
`seL4_CPtr`	`receiveIndex`	CPTR to the receive slot relative to `receiveCNode`
`seL4_Word`	`receiveDepth`	Number of bits of `receiveIndex` to use

Table 4.3: Fields of the `seL4_IPCBuffer` structure. Note that `badges` and `caps` use the same area of memory in the structure.

The kernel assumes that the IPC buffer contains a structure of type `seL4_IPCBuffer` as defined in Table 4.3. The kernel uses as many physical registers as possible to transfer IPC messages. When more arguments are transferred than physical message registers are available, the kernel begins using the IPC buffer's `msg` field to transfer arguments. However, it leaves room in this array for the physical message registers. For example, if an IPC transfer or kernel object invocation required 4 message registers (and there are only 2 physical message registers available on this architecture) then arguments 1 and 2 would be transferred via message registers and arguments 3 and 4 would be in `msg[2]` and `msg[3]`. This allows the user-level object-invocation stubs to copy the arguments passed in physical registers to the space left in the `msg` array if desired. The situation is similar for the tag field. There is space for this field in the `seL4_IPCBuffer` structure, which the kernel ignores. User level stubs may wish to copy the message tag from its CPU register to this field, although the user level stubs provided with the kernel do not do this.

4.2 Endpoints

Endpoints allow a small amount of data and capabilities (namely the IPC buffer) to be transferred between two threads. Endpoint objects are invoked directly using the seL4 system calls described in Section 2.2.

IPC Endpoints uses a rendezvous model and as such is synchronous and blocking. An Endpoint object may queue threads either to send or to receive. If no receiver is ready, threads performing the `seL4_Send()` or `seL4_Call()` system calls will wait in a queue for the first available receiver. Likewise, if no sender is ready, threads performing the `seL4_Recv()` system call or the second half of `seL4_ReplyRecv()` will wait for the first available sender.

4.2.1 Endpoint Badges

Endpoint capabilities may be *minted* to create a new endpoint capability with a *badge* attached to it, a data word chosen by the invoker of the *mint* operation. When a message is sent to an endpoint using a badged capability, the badge is transferred to the receiving thread's `badge` register.

An endpoint capability with a zero badge is said to be *unbadged*. Such a capability can be badged with the `seL4_CNode_Mutate()` or `seL4_CNode_Mint()` invocations on the CNode containing the capability. Endpoint capabilities with badges cannot be unbadged, rebadged or used to create child capabilities with different badges.

4.2.2 Capability Transfer

Messages may contain capabilities, which will be transferred to the receiver, provided that the endpoint capability invoked by the sending thread has Grant rights. An attempt to send capabilities using an endpoint capability without the Grant right will result in transfer of the raw message, without any capability transfer.

Capabilities to be sent in a message are specified in the sending thread's IPC buffer in the `caps` field. Each entry in that array is interpreted as a CPTR in the sending thread's capability space. The number of capabilities to send is specified in the `extraCaps` field of the message tag.

The receiver specifies the slot in which it is willing to receive a capability, with three fields within the IPC buffer: `receiveCNode`, `receiveIndex` and `receiveDepth`. These fields specify the root CNode, capability address and number of bits to resolve, respectively, to find the slot in which to put the capability. Capability addressing is described in Section 3.3.2.

A received capability has the same rights as the original, except if the *receiving* endpoint capability lacks the Write right. In this case, the rights on the sent capability are *diminished*, by stripping the Write right from the received copy of the capability.

Note that receiving threads may specify only one receive slot, whereas a sending thread may include multiple capabilities in the message. Messages containing more than one capability may be interpreted by kernel objects. They may also be sent to receiving threads in the case where some of the extra capabilities in the message can be *unwrapped*.

If the n-th capability in the message refers to the endpoint through which the message is sent, the capability is *unwrapped*: its badge is placed into the n-th position of the receiver's badges array, and the kernel sets the n-th bit (counting from the least significant) in the `capsUnwrapped` field of the message tag. The capability itself is not transferred, so the receive slot may be used for another capability.

If a receiver gets a message whose tag has an `extraCaps` of 2 and a `capsUnwrapped` of 2, then the first capability in the message was transferred to the specified receive slot and the second capability was unwrapped, placing its badge in `badges[1]`. There may have been a third capability in the sender's message which could not be unwrapped.

4.2.3 Errors

Errors in capability transfers can occur at two places: in the send phase or in the receive phase. In the send phase, all capabilities that the caller is attempting to send are looked up to ensure that they exist before the send is initiated in the kernel. If the lookup fails for any reason, `seL4_Send()` and `seL4_Call()` system calls immediately abort and no IPC or capability transfer takes place. The system call will return a lookup failure error as described in Section 10.1.

In the receive phase, seL4 transfers capabilities in the order that they are found in the sending thread's IPC buffer `caps` array and terminates as soon as an error is encountered. Possible error conditions are:

- A source capability cannot be looked up. Although the presence of the source capabilities is checked when the sending thread performs the send system call, this error may still occur. The sending thread may have been blocked on the endpoint for some time before it was paired with a receiving thread. During this time, its CSpace may have changed and the source capability pointers may no longer be valid.
- The destination slot cannot be looked up. Unlike the send system call, the `seL4_Recv()` system call does not check that the destination slot exists and is empty before it initiates the receive. Hence, the `seL4_Recv()` system call will not fail with an error if the destination slot is invalid and will instead transfer badged capabilities until an attempt to save a capability to the destination slot is made.
- The capability being transferred cannot be derived. See Section 3.1.4 for details.

An error will not void the entire transfer, it will just end it prematurely. The capabilities processed before the failure are still transferred and the `extraCaps` field in the receiver's IPC buffer is set to the number of capabilities transferred up to failure. No error message will be returned to the receiving thread in any of the above cases.

Chapter 5

Notifications

Notifications are a simple, non-blocking signalling mechanism that logically represents a set of binary semaphores.

5.1 Notification Objects

A Notification object contains a single data word, called the *notification word*. Such an object supports two operations: `seL4_Signal()` and `seL4_Wait()`.

Notification capabilities can be badged, using `seL4_CNode_Mutate()` or `seL4_CNode_Mint()`, just like Endpoint capabilities (see Section 4.2.1). As with Endpoint capabilities, badged Notification capabilities cannot be unbadged, rebadged or used to create child capabilities with different badges.

5.2 Signalling, Polling and Waiting

The `seL4_Signal()` method updates the notification word by bit-wise `or`-ing it with the *badge* of the invoked notification capability. It also unblocks the first thread waiting on the notification (if any). As such, `seL4_Signal()` works like concurrently signalling multiple semaphores (those indicated by the bits set in the badge). If the signal sender capability was unbadged or 0-badged, the operation degrades to just waking up the first thread waiting on the notification (also see below).

The `seL4_Wait()` method works similarly to a select-style wait on the set of semaphores: If the notification word is zero at the time `seL4_Wait()` is called, the invoker blocks. Else, the call returns immediately, setting the notification word to zero and returning to the invoker the previous notification-word value.

The `seL4_Poll()` is the same as `seL4_Wait()`, except if no signals are pending (the notification word is 0) the call will return immediately without blocking.

If threads are waiting on the Notification object at the time `seL4_Signal()` is invoked, the first queued thread receives the notification. All other threads keep waiting until the next time the notification is signalled.

If seL4_Signal() is invoked with an unbadged or 0-badged capability, the first queued thread is unblocked with a zero return value. If no thread is waiting, the seL4_Signal() operation with an unbadged capability has no effect.

5.3 Binding Notifications

Notification objects and TCBs can be bound together in a 1-to-1 relationship through the seL4_TCB_BindNotification() invocation. When a Notification is bound to a TCB, signals to that notification object will be delivered even if the thread is receiving from an IPC endpoint. To distinguish whether the received message was a notification or an IPC, developers should check the badge value. By reserving a specific badge (or range of badges) for capabilities to the bound notification — distinct from endpoint badges — the message source can be determined.

Once a notification has been bound, the only thread that may perform seL4_Wait() on the notification is the bound thread.

Chapter 6

Threads and Execution

6.1 Threads & Scheduling Contexts

seL4 provides threads to represent an execution context, while scheduling contexts are used to manage processor time. A thread is represented in seL4 by its thread control block object (`TCB`) and a scheduling context by a scheduling context object. Threads cannot run unless they are bound to, or receive a scheduling context.

6.1.1 Thread control blocks

Each `TCB` has an associated CSpace (see Chapter 3) and VSpace (see Chapter 7) which may be shared with other threads. A `TCB` may also have an IPC buffer (see Chapter 4), which is used to pass extra arguments during IPC or kernel object invocation that do not fit in the architecture-defined message registers. While it is not compulsory that a thread has an IPC buffer, it will not be able to perform most kernel invocations, as they require cap transfer.

6.1.2 Thread Creation

Like other objects, `TCB`s are created with the `seL4_Untyped_Retype()` method (see Section 2.4). A newly created thread is initially inactive. It is configured by setting its CSpace and VSpace with the `seL4_TCB_SetSpace()` or `seL4_TCB_Configure()` methods and then calling `seL4_TCB_WriteRegisters()` with an initial stack pointer and instruction pointer. The thread can then be activated either by setting the `resume_target` parameter in the `seL4_TCB_WriteRegisters()` invocation to true or by seperately calling the `seL4_TCB_Resume()` method.

6.1.3 Thread Deactivation

The `seL4_TCB_Suspend()` method deactivates a thread. Suspended threads can later be resumed. Their suspended state can be retrieved with the `seL4_TCB_ReadRegisters()` and `seL4_TCB_CopyRegisters()` methods. They can also be reconfigured and reused or left suspended indefinitely if not needed. Threads will be automatically suspended when the last capability to their `TCB` is deleted. When threads are suspended, any active IPCs or signals are cancelled.

6.1.4 Scheduling Contexts

Access to CPU execution time is controlled through scheduling context objects. Scheduling contexts consist of a tuple of *budget (b)* and *period (p)*, both in microseconds, set by `seL4_SchedControl_Configure()` (see Section 6.1.6). The tuple (b, p) forms an upper bound on the thread's execution – the kernel will not permit a thread to run for more than b out of every p microseconds. However, $\frac{b}{p}$ does not represent a lower bound on execution, as a thread must have the highest or equal highest priority of all runnable threads to be guaranteed to be scheduled at all.

A scheduling context that has budget available is reffered to as *active*. Whenever a thread is executing it consumes the budget from its current scheduling context. Once the budget is exhausted the thread is preempted and will not be schedulable again until the period has passed, at which point the budget will be replenished. When a thread's budget is exhausted, the next runnable thread at that priority with an active scheduling context will be chosen by the scheduler. When $b = p$, b simply acts as a timeslice for a thread, as the budget is always replenished immediately after it expires, however the thread will be preempted.

The system call `seL4_SchedContext_Yield()` can be used to sacrifice any remaining budget and block until the budget is replenished.

Threads can be bound to scheduling contexts using `seL4_TCB_Configure()` or `seL4_-SchedContext_Bind()`, both invocations have the same effect although `seL4_TCB_-Configure()` allows more thread fields to be set with only one kernel entry. When a thread is bound to a scheduling context, if it is in a runnable state and the scheduling context is active, it will be added to the scheduler.

Threads can optionally generate exceptions when they attempt to run without available budget, see Section 6.1.11.

6.1.5 Passive Threads

Threads can be unbound from a scheduling context with `seL4_SchedContext_Un-bindObject()`. This is distinct from suspending a thread, in that threads that are blocked waiting in an endpoint or notification queue will remain in the queue and can still recieve messages and signals. However, the unbound thread will not be schedulable again until it receives a scheduling context. Threads without scheduling contexts are referred to as *passive* threads, as they cannot execute without the action of another thread.

6.1.6 Scheduling Context Creation

Like other objects, scheduling contexts are created from untyped memory using `seL4_-UntypedRetype()`. On creation, scheduling contexts are empty, representing 0% of

CPU execution time. To populate a scheduling context with parameters, one must invoke the SchedControl capability, which provides access to CPU time management and is provided to the initial task at run time. Scheduling context parameters can then be set and updated using `seL4_SchedControl_Configure()`, which allows the budget and period to be specified.

The kernel does not conduct any schedulability tests, as task admission is left to user-level policy and can be conducted online or offine, statically or dynamically or not at all.

6.1.7 Scheduling Context Donation and Borrowing

In addition to explictly binding and removing scheduling contexts through `seL4_SchedContext_Bind()` and `seL4_SchedContext_UnbindObject()`, scheduling contexts can move between threads over IPC. Scheduling contexts are donated implicitly when the system calls `seL4_Call()` and `seL4_SignalRecv()` are used to communicate with a passive thread. When `seL4_Call()` is used, the generated reply cap ensures that the callee is merely borrowing the scheduling context: when the reply cap is consumed by a reply message being sent the scheduling context will be returned to the caller. If the reply cap is revoked, and the callee holds the scheduling context, the scheduling context will be returned to the caller. However, if in a deep call chain and a reply cap in the middle of the call chain is revoked, such that the callee does not possess the scheduling context, the thread will be removed from the call chain and the scheduling context will remain where it is.

Consider an example where thread A calls thread B which calls thread C. If while C holds the scheduling context, B's reply cap to A is revoked, then the scheduling context will remain with C. However, a call chain will remain between A and C, such that if C's reply cap is revoked, or invoked, the scheduling context will return to A.

`seL4_SignalRecv()` only offers scheduling context donation: there is no guarantee that the scheduling context will return.

Scheduling contexts can also be bound to notification objects using `seL4_SchedContext_Bind()` and unbound using `seL4_SchedContext_UnbindObject()`. If a signal is delivered to a notification object with a passive thread blocked waiting on it, the passive thread will receive the scheduling context that is bound to the notification object. The scheduling context is returned when the thread blocks on the notification object. This feature allows for event-driven periodic threads which are triggered by events rather than the kernel time, and also allows for passive servers to use notification binding (See Section 5.3).

Scheduling contexts can be unbound from all objects (notification objects and TCBs that are bound or have received a scheduling context through donation) using `seL4_SchedContext_Unbind()`.

6.1.8 Scheduling algorithm

seL4 uses a preemptive, tickless, scheduler with 256 priority levels (0 — 255) and 0 — 4 criticality levels. Thread scheduling in seL4 is controlled via two distinct values:

priorities and criticalities, which facilitate mixed-criticality scheduling. The kernel maintains a criticality level and only threads of criticality higher or equal to the current kernel criticality level are eligible for scheduling. Additionally, threads are only eligible for scheduling if they have an active scheduling context. Of threads eligible for scheduling, the highest priority thread in a runnable state is chosen.

Thread priority (structure `seL4_Prio_t`) consists of four values as follows:

Priority the priority a thread will be scheduled with.

Maximum controlled priority (MCP) the highest priority a thread can set itself or another thread to.

Criticality the criticality of a thread.

Maximum controlled criticality the highest criticality a thread can set itself or another thread to.

Threads of sufficient maximum contrlled priority and with possession of the appropriate scheduling context capability can manipulate the scheduler and implement user-level schedulers using `seL4_SchedContext_YieldTo()` and `seL4_SchedContext_Consumed()`.

6.1.9 Priorities

Scheduling contexts provide access to and an upper bound on exection CPU time, however when a thread executes is determined by thread priority.

All threads have a maximum controlled priority (MCP) and a priority, the latter being the effective priority of the thread. When a thread creates or modifies another thread, it can only set the other thread's priority and MCP to be less than or equal to its own MCP. Thread priority and MCP can be set with `seL4_TCB_Configure()` and `seL4_TCB_SetPriority()`, `seL4_TCB_SetMCPriority()` methods.

Consequently, access to CPU is a function of thread MCPs, scheduling contexts and the SchedControl capability. The kernel will enforce that threads do not exceeed the budget in their scheduling context for any given period, and that the highest priority thread will always run, however it is up to the system designer to make sure the entire system is schedulable.

6.1.10 Criticalities

Thread criticality provides the mechanism for implementing mixed-criticality scheduling in an efficient way on seL4. Criticality allows the system to change operating mode, with the understanding that the highest priority thread is not neccessarily the most important thread. Should a high criticalty thread need more time, low criticality threads can be removed from the scheduler by changing the kernel criticality level with `seL4_SchedControl_SetCriticality()`, which is $O(n)$ in the number of threads with criticality greater than or equal to the criticality being set. Criticality can be restored with the same function.

Thread criticality can be set with `seL4_TCB_SetCriticality()`, and like thread priorities, criticality assignment is controlled by the maximum control criticality, set with `seL4_TCB_SetMCCriticality()`. Both fields can be set with `seL4_TCB_Configure()`.

6.1.11 Exceptions

Each thread has two associated exception-handler endpoints, a *standard* exception handler and a *temporal* exception handler. If the thread causes an exception, the kernel creates an IPC message with the relevant details and sends the appropriate endpoint. This thread can then take the appropriate action. Fault IPC messages are described in Section 6.2.

Exception-handler endpoints can be set with the `seL4_TCB_SetSpace()` or `seL4_TCB_Configure()` methods. With these methods, a capability address for the exception handler can be associated with a thread. This address is then used to lookup the handler endpoint, and the capability to the endpoint is installed into the threads' kernel CNode. For threads without an exception handler, a null capability can be used, however the consequences of are different per exception handler type.

Before raising an exception the handler capability is validated - the kernel does not perform another lookup, but checks that the capability is an endpoint with the correct rights.

The exception endpoint must have send and grant rights. Replying to the exception message restarts the thread. For certain exception types, the contents of the reply message may be used to set the values in the registers of the thread being restarted. See Section 6.2 for details.

Standard Exceptions

The standard exception handler is used when a fault is triggered by a thread which cannot be recovered from without action by another thread. For example, if a thread raises a fault due to an unmapped virtual memory page, the thread cannot make any more progress until the page is mapped. If a thread experiences a fault that would trigger the standard exception handler while it is set to a null capability, the kernel will pause the thread and it will not run again. This is because without action by another thread, standard exceptions cannot be recovered from. Consequently threads without standard exception handlers should be trusted not to fault at all.

Standard exception handlers can be passive, in which case they will run on the scheduling context of the faulting thread.

Temporal Exceptions

Temporal faults are raised when a thread attempts to run but has no available budget, and if that thread has a valid temporal exception handler capability. The handling of temporal faults is not compulsory: if a thread does not have a temporal fault handler, a fault will not be raised and the thread will continue running when it's budget is replenished. This allows temporally sensitive threads to handle budget overruns while other threads may ignore them.

Temporal faults are registered per thread, which means that while clients may not have a temporal fault handler, servers may, allowing single-threaded, time-sensitive, passive servers to use a temporal exception handler to recover from malicious or untrusted clients whose budget expires while the server is completing the request. Temporal faults handlers can use `seL4_CNode_SwapTCBCaller()` to save the servers reply capability and reply with an error to the client, then resetting the server to handle the next client request.

If a reply message is sent to a nested server and a scheduling context without available budget returned, another temporal fault will be generated if the nested server also has a temporal fault handler.

Additionally, if the system criticality is changed while a thread with higher criticality than the system criticality is running on a scheduling context that is bound to a thread with criticality lower than the system criticality, a temporal exception will be raised.

6.1.12 Message Layout of the Read-/Write-Registers Methods

The registers of a thread can be read and written with the `seL4_TCB_ReadRegisters()` and `seL4_TCB_WriteRegisters()` methods. The register contents are transferred via the IPC buffer. The IPC buffer locations that registers are copied to/from are given below.

IA-32

Register	IPC Buffer location
`EIP`	`IPCBuffer[0]`
`ESP`	`IPCBuffer[1]`
`EFLAGS`	`IPCBuffer[2]`
`EAX`	`IPCBuffer[3]`
`EBX`	`IPCBuffer[4]`
`ECX`	`IPCBuffer[5]`
`EDX`	`IPCBuffer[6]`
`ESI`	`IPCBuffer[7]`
`EDI`	`IPCBuffer[8]`
`EBP`	`IPCBuffer[9]`
`TLS_BASE`	`IPCBuffer[10]`
`FS`	`IPCBuffer[11]`
`GS`	`IPCBuffer[12]`

ARM

Register	IPC Buffer location
PC	IPCBuffer[0]
SP	IPCBuffer[1]
CPSR	IPCBuffer[2]
R0-R1	IPCBuffer[3-4]
R8-R12	IPCBuffer[5-9]
R2-R7	IPCBuffer[10-15]
R14	IPCBuffer[16]

6.2 Faults

A thread's actions may result in a fault. Faults are delivered to the thread's exception handler so that it can take the appropriate action. The fault type is specified in the message label and is one of: seL4_CapFault, seL4_VMFault, seL4_UnknownSyscall, seL4_UserException, seL4_Interrupt or seL4_TemporalFault.

6.2.1 Capability Faults

Capability faults may occur in two places. Firstly, a capability fault can occur when lookup of a capability referenced by a `seL4_Call()` or `seL4_Send()` system call failed (`seL4_NBSend()` calls on invalid capabilities silently fail). In this case, the capability on which the fault occurred may be the capability being invoked or an extra capability passed in the `caps` field in the IPC buffer.

Secondly, a capability fault can occur when `seL4_Recv()` or `seL4_NBRecv()` is called on a capability that does not exist, is not an endpoint or notification capability or does not have receive permissions.

Replying to the fault IPC will restart the faulting thread. The contents of the IPC message are given in Table 6.1.

Meaning	IPC buffer Location
Address at which to restart execution	IPCBuffer[0]
Capability address	IPCBuffer[1]
In receive phase (1 if the fault happened during a receive system call, 0 otherwise)	IPCBuffer[2]
Lookup failure description. As described in Section 3.4	IPCBuffer[3..]

Table 6.1: Contents of an IPC message.

6.2.2 Unknown Syscall

This fault occurs when a thread executes a system call with a syscall number that is unknown to seL4. The register set of the faulting thread is passed to the thread's exception handler so that it may, for example, emulate the system call if a thread is being virtualised.

Replying to the fault IPC allows the thread to be restarted and/or the thread's register set to be modified. If the reply has a label of zero, the thread will be restarted. Additionally, if the message length is non-zero, the faulting thread's register set will be updated as shown in Table 6.2 and Table 6.3. In this case, the number of registers updated is controlled with the length field of the message tag.

ARM

Value sent	Register set by reply	IPC buffer location
`R0-R7`	(same)	`IPCBuffer[0-7]`
`FaultInstruction`	(same)	`IPCBuffer[8]`
`SP`	(same)	`IPCBuffer[9]`
`LR`	(same)	`IPCBuffer[10]`
`CPSR`	(same)	`IPCBuffer[11]`
Syscall number	—	`IPCBuffer[12]`

Table 6.2: Unknown system call outcome on the ARM architecture.

IA-32

Value sent	Register set by reply	IPC buffer location
`EAX`	(same)	`IPCBuffer[0]`
`EBX`	(same)	`IPCBuffer[1]`
`ECX`	(same)	`IPCBuffer[2]`
`EDX`	(same)	`IPCBuffer[3]`
`ESI`	(same)	`IPCBuffer[4]`
`EDI`	(same)	`IPCBuffer[5]`
`EBP`	(same)	`IPCBuffer[6]`
`EIP`	(same)	`IPCBuffer[7]`
`ESP`	(same)	`IPCBuffer[8]`
`EFLAGS`	(same)	`IPCBuffer[9]`
Syscall number	—	`IPCBuffer[10]`

Table 6.3: Unknown system call outcome on the IA-32 architecture.

6.2.3 User Exception

User exceptions are used to deliver architecture-defined exceptions. For example, such an exception could occur if a user thread attempted to divide a number by zero.

Replying to the fault IPC allows the thread to be restarted and/or the thread's register set to be modified. If the reply has a label of zero, the thread will be restarted. Additionally, if the message length is non-zero, the faulting thread's register set will be updated as shown in Table 6.4 and Table 6.5. In this case, the number of registers updated is controlled with the length field of the message tag.

ARM

Value sent	Register set by reply	IPC buffer location
`FaultInstruction`	(same)	`IPCBuffer[0]`
`SP`	(same)	`IPCBuffer[1]`
`CPSR`	(same)	`IPCBuffer[2]`
Exception number	—	`IPCBuffer[3]`
Exception code	—	`IPCBuffer[4]`

Table 6.4: User exception outcome on the ARM architecture.

IA-32

Value sent	Register set by reply	IPC buffer location
`EIP`	(same)	`IPCBuffer[0]`
`ESP`	(same)	`IPCBuffer[1]`
`EFLAGS`	(same)	`IPCBuffer[2]`
Exception number	—	`IPCBuffer[3]`
Exception code	—	`IPCBuffer[4]`

Table 6.5: User exception outcome on the IA-32 architecture.

6.2.4 VM Fault

The thread caused a page fault. Replying to the fault IPC will restart the thread. The contents of the IPC message are given below.

Meaning	IPC buffer location
Program counter to restart execution at.	`IPCBuffer[0]`
Address that caused the fault.	`IPCBuffer[1]`
Instruction fault (1 if the fault was caused by an instruction fetch).	`IPCBuffer[2]`
Fault status register (FSR). Contains information about the cause of the fault. Architecture dependent.	`IPCBuffer[3]`

6.2.5 Temporal Fault

Temporal faults are raised when a thread consumes all of its budget and has a temporal fault handler that is not a null capability. They allow a temporal exception handler to take some action to restore the thread.

Meaning	IPC buffer location
Data word from the scheduling context object that the thread was running on when the fault occured.	`IPCBuffer[0]`

Chapter 7

Address Spaces and Virtual Memory

A virtual address space in seL4 is called a VSpace. In a similar way to a CSpace (see Chapter 3), a VSpace is composed of objects provided by the microkernel. Unlike CSpaces, these objects for managing virtual memory largely 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.

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.

7.1 Overview

IA-32 IA-32 processors have a two-level page-table structure. The top-level page directory covers a 4 GiB range and each page table covers a 4 MiB range. Frames can be 4 KiB or 4 MiB. Before a 4 KiB frame can be mapped, a page table covering the range that the frame will be mapped into must have been mapped, otherwise seL4 will return an error. 4 MiB frames are mapped directly into the page directory, thus, a page table does not need to be mapped first.

ARM ARM processors also have a two-level page-table structure. The top-level page directory covers a range of 4 GiB and each page table covers a 1 MiB range. Four page sizes are allowed: 4 KiB, 64 KiB, 1 MiB and 16 MiB. 4 KiB and 64 KiB pages are mapped into the second-level page table. Before they can be mapped, a page table covering the range that they will be mapped into must have been installed. 1 MiB and 16 MiB pages are installed directly into the page directory such that it is not necessary to map a page table first. Pages of 4 KiB and 1 MiB size occupy one slot in a page table and the page directory, respectively. Pages of 64 KiB and 16 MiB size occupy 16 slots in a page table and the page directory, respectively.

7.2 Objects

Page Directory The Page Directory (PD) is the top-level page table of the two-level page table structure. It has a hardware-defined format, but conceptually contains a number of page directory entries (PDEs). The Page Directory has no methods itself, but it is used as an argument to several other virtual-memory related object invocations.

Page Table The Page Table (PT) object forms the second level of the page-table structure. It contains a number of slots, each of which contains a page-table entry (PTE).

Page Table objects possess only two methods:

`seL4_ARM_PageTable_Map()`
`seL4_IA32_PageTable_Map()`

> Takes a Page Directory capability as an argument, and installs a reference to the invoked Page Table in a specified slot in the Page Directory.

`seL4_ARM_PageTable_Unmap()`
`seL4_IA32_PageTable_Unmap()`

> Removes the reference to the invoked Page Table from its containing 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:

`seL4_ARM_Page_Map()`
`seL4_IA32_Page_Map()`

> Takes a Page Directory capability as an argument and installs a reference to the given Page in the PD or PT slot corresponding to the given address.

`seL4_ARM_Page_Remap()`
`seL4_IA32_Page_Remap()`

> Changes the permissions of an existing mapping.

`seL4_ARM_Page_Unmap()`
`seL4_IA32_Page_Unmap()`

> Removes an existing mapping.

The virtual address for a Page mapping must be aligned to the size of the Page and must be mapped to a suitable Page Directory or Page Table. To map a page readable, the capability to the page that is being invoked must have read permissions. To map the page writable, the capability must have write permissions. The requested mapping permissions are specified with an argument of type `seL4_CapRights` given to the `seL4_ARM_Page_Map()` or `seL4_IA32_Page_Map()` method. `seL4_CanRead` and `seL4_CanWrite` are the only valid permissions on both ARM and IA-32 architectures. If the capability does not have sufficient permissions to authorise the given mapping, then the mapping permissions are silently downgraded.

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:

`seL4_ARM_ASIDControl_MakePool()`
`seL4_IA32_ASIDControl_MakePool()`

Together with a capability to Untyped Memory as argument creates an ASID Pool.

The untyped capability given to the `seL4_ARM_ASIDControl_MakePool()` call must represent a 4K memory object. This will create an ASID pool with enough space for 1024 VSpaces.

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:

`seL4_ARM_ASIDPool_Assign()`
`seL4_IA32_ASIDPool_Assign()`

Assigns an ASID to the VSpace associated with the Page Directory passed in as an argument.

7.3 Mapping Attributes

A parameter of type `seL4_ARM_VMAttributes` or `seL4_IA32_VMAttributes` is used to specify the cache behaviour of the page being mapped; possible values for ARM are shown in Table 7.1 and values for IA-32 are shown in Table 7.2.

Attribute	Meaning
`seL4_ARM_PageCacheable`	Enable data in this mapping to be cached
`seL4_ARM_ParityEnabled`	Enable parity checking for this mapping
`seL4_ARM_ExecuteNever`	Map this memory as non-executable

Table 7.1: Virtual memory attributes for ARM page table entries.

Attribute	Meaning
seL4_IA32_CacheDisabled	Prevent data in this mapping from being cached
seL4_IA32_WriteThrough	Enable write through cacheing for this mapping
seL4_IA32_WriteCombining	Enable write combining for this mapping

Table 7.2: Virtual memory attributes for IA32 page table entries.

7.4 Sharing Memory

seL4 does not allow Page Tables to be shared, but does allow pages to be shared between address spaces. To share a page, the capability to the Page must first be duplicated using the `seL4_CNode_Copy()` method and the new copy must be used in the `seL4_ARM_Page_Map()` or `seL4_IA32_Page_Map()` method that maps the page into the second address space. Attempting to map the same capability twice will result in an error.

7.5 Page Faults

Page faults are reported to the exception handler of the executed thread. See Section 6.2.4.

Chapter 8

Hardware I/O

8.1 Interrupt Delivery

Interrupts are delivered as notifications. A thread may configure the kernel to signal a particular Notification object each time a certain interrupt triggers. Threads may then wait for interrupts to occur by calling `seL4_Wait()` or `seL4_Poll()` on that Notification. In the notification word returned from either call, bit n (modulo word size) represents IRQ n, the bit will be set if the corresponding IRQ was raised. This supports the use of a single handler for multiple IRQs.

IRQHandler capabilities represent the ability of a thread to configure a certain interrupt. They have three methods:

`seL4_IRQHandler_SetNotification()` specifies the Notification the kernel should `signal()` when an interrupt occurs. A driver may then call `seL4_Wait()` or `seL4_Poll()` on this notification to wait for interrupts to arrive.

`seL4_IRQHandler_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.

`seL4_IRQHandler_Clear()` de-registers the Notification from the IRQHandler object.

When the system first starts, no IRQHandler capabilities are present. Instead, the initial thread's CSpace contains a single IRQControl capability. This capability may be used to produce a single IRQHandler capability for each interrupt available in the system. Typically, the initial thread of a system will determine which IRQs are required by other components in the system, produce an IRQHandler capability for each interrupt, and then delegate the resulting capabilities as appropriate. IRQControl has one method:

`seL4_IRQControl_Get()` creates an IRQHandler capability for the specified interrupt source.

8.2 IA-32-Specific I/O

8.2.1 I/O Ports

On IA-32 platforms, seL4 provides access to I/O ports to user-level threads. Access to I/O ports is controlled by IO Port capabilities. Each IO Port capability identifies a range of ports that can be accessed with it. Reading from I/O ports is accomplished with the `seL4_IA32_IOPort_In8()`, `seL4_IA32_IOPort_In16()`, and `seL4_IA32_IOPort_In32()` methods, which allow for reading of 8-, 16- and 32-bit quantities. Similarly, writing to I/O ports is accomplished with the `seL4_IA32_IOPort_Out8()`, `seL4_IA32_IOPort_Out16()`, and `seL4_IA32_IOPort_Out32()` methods. Each of these methods takes as arguments an IO Port capability and an unsigned integer `port`, which indicates the I/O port to read from or write to, respectively. In each case, `port` must be within the range of I/O ports identified by the given IO Port capability in order for the method to succeed.

At system initialisation, the initial thread's CSpace contains the master IO Port capability, which allows access to all I/O ports. Other IO Port capabilities, which authorise access to a specific range of I/O Ports, may be derived from this master capability using the `seL4_CNode_Mint()` method. The range of I/O ports that the newly created capability should identify are specified via the 32-bit `badge` argument provided to `seL4_CNode_Mint()`. The first port number in the range occupies the top 16 bits of `badge`, while the last port number in the range occupies the bottom 16 bits. The range is interpreted as being inclusive of these two numbers.

The I/O port methods return error codes upon failure. A `seL4_IllegalOperation` code is returned if port access is attempted outside the range allowed by the IO Port capability. Since invocations that read from I/O ports are required to return two values – the value read and the error code – a structure containing two members, `result` and `error`, is returned from these API calls.

8.2.2 I/O Space

I/O devices capable of DMA present a security risk because the CPU's MMU is bypassed when the device accesses memory. In seL4, device drivers run in user space to keep them out of the trusted computing base. A malicious or buggy device driver may, however, program the device to access or corrupt memory that is not part of its address space, thus subverting security. To mitigate this threat, seL4 provides support for the IOMMU on Intel IA-32-based platforms. An IOMMU allows memory to be remapped from the device's point of view. It acts as an MMU for the device, restricting the regions of system memory that it can access. More information can be obtained from Intel's IOMMU documentation [Int11].

seL4-based systems that wish to utilise DMA must have an IOMMU. This restriction results from the fact that seL4 provides no way to obtain the physical address of a Page from its capability. Hence, applications are unable to accurately instruct devices, at which address they should directly address the physical memory. Instead, frames of memory must be mapped into the device's address space using seL4's IOMMU primitives.

Two new objects are provided by the kernel to abstract the IOMMU:

IOSpace This object represents the address space associated with a hardware device on the PCI bus. It represents the right to modify a device's memory mappings.

IOPageTable This object represents a node in the multilevel page-table structure used by IOMMU hardware to translate hardware memory accesses.

Page capabilities are used to represent the actual frames that are mapped into the I/O address space. A Page can be mapped into either a VSpace or an IOSpace but never into both at the same time.

IOSpace and VSpace fault handling differ significantly. VSpace page faults are redirected to the thread's exception handler (see Section 6.2), which can take the appropriate action and restart the thread at the faulting instruction. There is no concept of an exception handler for an IOSpace. Instead, faulting transactions are simply aborted; the device driver must correct the cause of the fault and retry the DMA transaction.

An initial master IOSpace capability is provided in the initial thread's CSpace. An IOSpace capability for a specific device is created by using the `seL4_CNode_Mint()` method, passing the PCI identifier of the device as the low 16 bits of the `badge` argument, and a Domain ID as the high 16 bits of the `badge` argument. PCI identifiers are explained fully in the PCI specification [SA99], but are briefly described here. A PCI identifier is a 16-bit quantity. The first 8 bits identify the bus that the device is on. The next 5 bits are the device identifier: the number of the device on the bus. The last 3 bits are the function number. A single device may consist of several independent functions, each of which may be addressed by the PCI identifier. Domain IDs are explained fully in the Intel IOMMU documentation [Int11]. There is presently no way to query seL4 for how many Domain IDs are supported by the IOMMU and the `seL4_CNode_Mint()` method will fail if an unsupported value is chosen.

The IOMMU page-table structure has three levels. Page tables are mapped into an IOSpace using the `seL4_IA32_IOPageTable_Map()` method. This method takes the IOPageTable to map, the IOSpace to map into and the address to map at. Three levels of page tables must be mapped before a frame can be mapped successfully. A frame is mapped with the `seL4_IA32_Page_MapIO()` method whose parameters are analogous to the corresponding method that maps Pages into VSpaces (see Chapter 7), namely `seL4_IA32_Page_Map()`.

Unmapping is accomplished with the usual unmap (see Chapter 7) API call, `seL4_IA32_Page_Unmap()`.

More information about seL4's IOMMU abstractions can be found in [Pal09].

Chapter 9

System Bootstrapping

9.1 Initial Thread's Environment

The seL4 kernel creates a minimal boot environment for the initial thread. This environment consists of the initial thread's TCB, CSpace and VSpace, consisting of frames that contain the userland image (code/data of the initial thread) and the IPC buffer. The kernel creates a scheduling context for the root task with a configurable budget that defaults to 10 milliseconds. The initial thread's CSpace consists of exactly one CNode which contains capabilities to the initial thread's own resources was well as to all available global resources. The CNode size can be configured at compile time (default is 2^{12} slots), but the guard is always chosen so that the CNode resolves exactly 32 bits. This means, the first slot of the CNode has CPTR 0x0, the second slot has CPTR 0x1 etc.

The first 12 slots contain specific capabilities as listed in Table 9.1.

9.2 BootInfo Frame

CNode slots with CPTR `seL4_NumInitialCaps` (defined in the seL4 userland library) and above are filled dynamically during bootstrapping. Their exact contents depend on the userland image size, platform configuration (devices) etc. In order to tell the initial thread which capabilities are stored where in its CNode, the kernel provides a *BootInfo Frame* which is mapped into the initial thread's address space. The mapped address is chosen by the kernel and given to the initial thread via a CPU register. The initial thread can access this address through the function `seL4_GetBootInfo()`.

The BootInfo Frame contains the C struct described in Table 9.2. It is defined in the seL4 userland library. Besides talking about capabilities, it also informs the initial thread about the current platform's configuration.

The type `seL4_SlotRegion` is a C struct which contains `start` and `end` slot CPTRs. It denotes a region of slots in the initial thread's CNode, starting with CPTR `start` and with `end` being the CPTR of the first slot after the region ends, i.e. `end - 1` points to the last slot of the region.

Table 9.1: Initial thread's CNode content.

CPTR	Enum Constant	Capability
0x0	`seL4_CapNull`	null
0x1	`seL4_CapInitThreadTCB`	initial thread's TCB
0x2	`seL4_CapInitThreadCNode`	initial thread's CNode
0x3	`seL4_CapInitThreadPD`	initial thread's page directory
0x4	`seL4_CapIRQControl`	global IRQ controller (see Section 8.1)
0x5	`seL4_CapASIDControl`	global ASID controller (see Chapter 7)
0x6	`seL4_CapInitThreadASIDPool`	initial thread's ASID pool (see Chapter 7)
0x7	`seL4_CapIOPort`	global I/O port cap, null cap if unsupported (see Section 8.2.1)
0x8	`seL4_CapIOSpace`	global I/O space cap, null cap if unsupported (see Section 8.2.2)
0x9	`seL4_CapBootInfoFrame`	BootInfo frame (see Section 9.2)
0xa	`seL4_CapInitThreadIPCBuffer`	initial thread's IPC buffer (see Section 4.1)
0xb	`seL4_CapInitThreadSchedContext`	initial thread's scheduling scontext (see Section 6.1.4)
0xc	`seL4_CapSchedControl`	global scheduling cap for managing CPU time (see Section 6.1.6

The capabilities in `userImageFrames` are ordered such that the first capability references the first frame of the userland image and so on. The capabilities in `userImagePaging` are ordered in descending order of paging structure size. Within a given paging structure size, capabilities are ordered by the virtual address at which the corresponding objects are mapped into the initial thread's address space.

It is up to userland to infer the virtual address of frames referenced by the capabilities in `userImageFrames` and the virtual address and types of paging structures referenced by the capabilities in `userImagePaging`. Userland typically has a way of finding out to which virtual addresses its code and data is mapped (e.g. in GCC, with the standard linker script, the symbols `__executable_start` and `_end` are available). Additionally, the initial thread can assume that its address space is virtually contiguous, and is made up of the smallest frames available on the architecture. It's also assumed that the initial thread knows which paging structures are available on the architecture it's running on. This, along with knowledge of how capabilities in `userImageFrames` and `userImagePaging` are ordered, is sufficient information for userland to infer the virtual address of each frame capability, and the virtual address and type of each paging structure capability.

Untyped memory is given in no particular order. The array entry `untypedSizeBitsList[i]` stores the untyped-memory size (2^n bytes) of the i-th untyped cap of the slot region `untyped`. Therefore, the array length is at least `untyped.end - un-`

Table 9.2: BootInfo struct.

Field Type	Field Name	Description
`seL4_Word`	`nodeID`	node ID (see Section 9.4)
`seL4_Word`	`numNodes`	number of nodes (see Section 9.4)
`seL4_Word`	`numIOPTLevels`	number of I/O page-table levels (0 if no IOMMU)
`seL4_IPCBuffer*`	`ipcBuffer`	pointer to the initial thread's IPC buffer
`seL4_SlotRegion`	`empty`	empty slots (null caps)
`seL4_SlotRegion`	`sharedFrames`	see Section 9.4
`seL4_SlotRegion`	`userImageFrames`	frames containing the userland image
`seL4_SlotRegion`	`userImagePaging`	userland-image paging structure caps
`seL4_SlotRegion`	`untyped`	untyped-memory capabilities
`seL4_Word[]`	`untypedPaddrList`	array of untyped-memory physical addresses
`uint8_t[]`	`untypedSizeBitsList`	array of untyped-memory sizes (2^n bytes)
`uint8_t`	`initThreadCNodeSizeBits`	CNode size (2^n slots)
`seL4_Word`	`numDeviceRegions`	number of device memory regions
`seL4_DeviceRegion[]`	`deviceRegions`	device memory regions (see Table 9.3)

`typed.start`. The actual length is hardcoded in the kernel and irrelevant to the reader of the array. The same is true for the array `untypedPaddrList`. For each untyped-memory capability, it stores the physical addresses backing the untyped memory. This allows userland to infer physical memory addresses of retyped frames and use them to initiate DMA transfers when no IOMMU is available. The kernel makes no guarantees about certain sizes of untyped memory being available except that it provides a compile-time-configurable minimum number of 4K untyped capabilities (default is 12).

Table 9.3: DeviceRegion struct.

Field Type	Field Name	Description
`seL4_Word`	`basePaddr`	physical base address of the device region
`seL4_Word`	`frameSizeBits`	size (2^n bytes) of the frames used
`seL4_SlotRegion`	`frames`	capabilities to the frames covering the region

The kernel creates frames covering each physical memory region associated with a memory-mapped device. These device regions are either hardcoded (e.g. on embedded platforms) or discovered at boot time by the kernel through a PCI bus scan. The physical base address of each region is stored in `basePaddr`. The slot region `frames` identifies all frame caps used to back this region. They are ordered, so the first frame of the region is referenced by the first cap in this slot region and is backed by the physical address `basePaddr`. All frames have the same size: $2^{frameSizeBits}$ bytes. Hence, the size of the whole region is: `(frames.end - frames.start) << frameSizeBits`

The array `deviceRegions` of the BootInfo struct stores all available device regions (i.e. their structs). There are `numDeviceRegions` of them available in the array.

If the platform has an seL4-supported IOMMU, `numIOPTLevels` contains the number of IOMMU-page-table levels. This information is needed by userland when constructing an IOMMU address space (IOSpace). If there is no IOMMU support, `numIOPTLevels` is `0`.

9.3 Boot Command-line Arguments

On IA-32, seL4 accepts boot command-line arguments which are passed to the kernel via a multiboot-compliant bootloader (e.g. GRUB, syslinux). Multiple arguments are separated from each other by whitespace. Two forms of arguments are accepted: (1) key-value arguments of the form "key=value" and (2) single keys of the form "key". The value field of the key-value form may be a string, a decimal integer, a hexadecimal integer beginning with "0x", or an integer list where list elements are separated by commas. Keys and values can't have any whitespace in them and there can be no whitespace before or after an "=" or a comma either. Arguments are listed in Table 9.4 along with their default values (if left unspecified).

Table 9.4: IA-32 boot command-line arguments.

Key	Value	Default
`console_port`	I/O-port base of the serial port that the kernel prints to (if compiled in debug mode)	0x3f8
`debug_port`	I/O-port base of the serial port that is used for kernel debugging (if compiled in debug mode)	0x3f8
`disable_iommu`	none	The IOMMU is enabled by default on VT-d-capable platforms
`max_num_nodes`	Maximum number of seL4 nodes that can be started up (see Section 9.4)	1
`num_sh_frames`	Number of frames shared between seL4 nodes (see Section 9.4)	0

9.4 Multikernel Bootstrapping

On ARM, seL4 is uniprocessor only and does not support multikernel [BBD+09] bootstrapping. Therefore, the field `nodeID` of the BootInfo struct will always be `0`, `numNodes` will be `1` and `sharedFrames` will be an empty region.

On IA-32, seL4 can be bootstrapped as a multikernel by setting the boot command-line argument `max_num_nodes` to a value `>1`. Each available CPU core will then run one isolated *node* of seL4, up to the maximum number specified. The available physical memory is partitioned equally between nodes. All device frames are given to all nodes. The nodes' initial threads have to coordinate access to these device frames, e.g. by defining which node is responsible for which device. IOMMU management can only be performed by the first node, the other nodes are not given a global IOSpace capability. There is also a hard upper limit of number of nodes defined at compile time (default is 8, but it can be increased to 256).

Nodes are isolated from each other, except for *shared frames*. When bootstrapping, the kernel creates a number of 4K userland frames which are shared between nodes. This number can be configured via the boot command-line argument `num_sh_frames`. The shared frames will appear in the CNode of each node's initial thread. They are given to each initial thread in the same order. In the BootInfo struct, the field `sharedFrames` contains their slot region.

Shared frames can be used by userland to implement shared data structures, message passing and synchronisation mechanisms. Individual frames can, for example, be minted and handed out to subsystems. This allows connecting them across nodes in a fine-grained manner. Each node has a node ID (field `nodeID` in BootInfo) whereas the number of nodes can be obtained from the field `numNodes`.

Userland images are given to the kernel by a multiboot-compliant bootloader (e.g. GRUB, syslinux) via *boot modules*. Each boot module contains an ELF file of a userland image. If there is only one userland image, each node will get its own copy of that userland image. If multiple userland images are given, the first node gets the first image, the second node the second image etc. If there are more nodes than images, each remaining node gets a copy of the last image.

If the kernel is compiled in debug mode, each node can be assigned a separate console port and debug port (see Table 9.4) which is done by specifying an array of I/O-port base addresses. For example, `console_port=0x3f8,0x2f8,0x3e8,0x2e8` assigns port 0x3f8 to node 0, port 0x2f8 to node 1, etc. The remaining nodes have no assigned port and produce no console output. The argument `debug_port` works in exactly the same way.

Further details can be obtained from [vT10] which mainly talks about the formal aspects of seL4's multikernel version but also contains details about the bootstrapping. More recent additional information can be found in [vT12], which focusses less on the formal side and more on the OS side.

Chapter 10

seL4 API Reference

10.1 Error Codes

Invoking a capability with invalid parameters will result in an error. seL4 system calls return an error code in the message tag and a short error description in the message registers to aid the programmer in determining the cause of errors.

10.1.1 Invalid Argument

A non-capability argument is invalid.

Field	Meaning
`Label`	`seL4_InvalidArgument`
`IPCBuffer[0]`	Invalid argument number

10.1.2 Invalid Capability

A capability argument is invalid.

Field	Meaning
`Label`	`seL4_InvalidCapability`
`IPCBuffer[0]`	Invalid capability argument number

10.1.3 Illegal Operation

The requested operation is not permitted.

Field	Meaning
`Label`	`seL4_IllegalOperation`

10.1.4 Range Error

An argument is out of the allowed range.

Field	Meaning
`Label`	`seL4_RangeError`
`IPCBuffer[0]`	Minimum allowed value
`IPCBuffer[1]`	Maximum allowed value

10.1.5 Alignment Error

A supplied argument does not meet the alignment requirements.

Field	Meaning
`Label`	`seL4_AlignmentError`

10.1.6 Failed Lookup

A capability could not be looked up.

Field	Meaning
`Label`	`seL4_FailedLookup`
`IPCBuffer[0]`	1 if the lookup failed for a source capability, 0 otherwise
`IPCBuffer[1]`	Type of lookup failure
`IPCBuffer[2..]`	Lookup failure description as described in Section 3.4

10.1.7 Delete First

A destination slot specified in the syscall arguments is occupied.

Field	Meaning
`Label`	`seL4_DeleteFirst`

10.1.8 Revoke First

The object currently has other objects derived from it and the requested invocation cannot be performed until either these objects are deleted or the revoke invocation is performed on the capability.

Field	Meaning
`Label`	`seL4_RevokeFirst`

10.1.9 Not Enough Memory

The Untyped Memory object does not have enough unallocated space to complete the `seL4_Untyped_Retype()` request.

Field	Meaning
`Label`	`seL4_NotEnoughMemory`
`IPCBuffer[0]`	Amount of memory available in bytes

10.2 System Calls

10.2.1 Send

`static inline void seL4_Send`

Send to a capability

Type	**Name**	**Description**
`seL4_CPtr`	`dest`	The capability to be invoked.
`seL4_MessageInfo_t`	`msgInfo`	The messageinfo structure for the IPC.

Return value: This method does not return anything.

Description: See Section 2.2

10.2.2 Recv

`static inline seL4_MessageInfo_t seL4_Recv`

Block until a message is received on an endpoint

Type	**Name**	**Description**
`seL4_CPtr`	`src`	The capability to be invoked.
`seL4_Word*`	`sender`	The address to write sender information to. The sender information is the badge of the endpoint capability that was invoked by the sender, or the notification word of the notification object that was signalled. This parameter is ignored if `NULL`.

Return value: A `seL4_MessageInfo_t` structure as described in Section 4.1.

Description: See Section 2.2

10.2.3 Call

`static inline seL4_MessageInfo_t seL4_Call`

Call a capability

Type	Name	Description
`seL4_CPtr`	`dest`	The capability to be invoked.
`seL4_MessageInfo_t`	`msgInfo`	The messageinfo structure for the IPC.

Return value: A `seL4_MessageInfo_t` structure as described in Section 4.1.

Description: See Section 2.2

10.2.4 Reply

`static inline void seL4_Reply`

Perform a send to a one-off reply capability stored when the thread was last called

Type	Name	Description
`seL4_MessageInfo_t`	`msgInfo`	The messageinfo structure for the IPC.

Return value: This method does not return anything.

Description: See Section 2.2

10.2.5 Polling Send

`static inline void seL4_NBSend`

Perform a polling send to a capability

Type	Name	Description
`seL4_CPtr`	`dest`	The capability to be invoked.
`seL4_MessageInfo_t`	`msgInfo`	The messageinfo structure for the IPC.

Return value: This method does not return anything.

Description: See Section 2.2

10.2.6 Reply Recv

`static inline seL4_MessageInfo_t seL4_ReplyRecv`

Perform a reply followed by a receive in one system call

Type	Name	Description
`seL4_CPtr`	`dest`	The capability to be invoked.
`seL4_MessageInfo_t`	`msgInfo`	The messageinfo structure for the IPC.
`seL4_Word*`	`sender`	The address to write sender information to. The sender information is the badge of the endpoint capability that was invoked by the sender, or the notification word of the notification object that was signalled. This parameter is ignored if `NULL`.

Return value: A `seL4_MessageInfo_t` structure as described in Section 4.1.

Description: See Section 2.2

10.2.7 NBRecv

`static inline seL4_MessageInfo_t seL4_NBRecv`

Receive a message from an endpoint but do not block in the case that no messages are pending

Type	Name	Description
`seL4_CPtr`	`src`	The capability to be invoked.
`seL4_Word*`	`sender`	The address to write sender information to. The sender information is the badge of the endpoint capability that was invoked by the sender, or the notification word of the notification object that was signalled. This parameter is ignored if `NULL`.

Return value: A `seL4_MessageInfo_t` structure as described in Section 4.1.

Description: See Section 2.2

10.2.8 Signal

`static inline void seL4_Signal`

Signal a notification

Type	Name	Description
`seL4_CPtr`	`dest`	The capability to be invoked.

Return value: This method does not return anything.

Description: This is not a proper system call known by the kernel. Rather, it is a convenience wrapper provided by the seL4 userland library which calls `seL4_Send()`. It is useful for signalling a notification.

See the description of `seL4_Send()` in Section 2.2

10.2.9 Wait

`static inline void seL4_Wait`

Perform a receive on a notification object.

Type	Name	Description
`seL4_CPtr`	`src`	The capability to be invoked.
`seL4_Word*`	`sender`	The address to write sender information to. The sender information is the badge of the endpoint capability that was invoked by the sender, or the notification word of the notification object that was signalled. This parameter is ignored if `NULL`.

Return value: This method does not return anything.

Description: This is not a proper system call known by the kernel. Rather, it is a convenience wrapper provided by the seL4 userland library which calls `seL4_Recv()`.

See the description of `seL4_Recv()` in Section 2.2

10.2.10 Poll

`static inline void seL4_Poll`

Perform a non-blocking recv on a notification object.

Type	Name	Description
seL4_CPtr	src	The capability to be invoked.
seL4_Word*	sender	The address to write sender information to. The sender information is the badge of the endpoint capability that was invoked by the sender, or the notification word of the notification object that was signalled. This parameter is ignored if NULL.

Return value: This method does not return anything.

Description: This is not a proper system call known by the kernel. Rather, it is a convenience wrapper provided by the seL4 userland library which calls `seL4_NBRecv()`. It is useful for doing a non-blocking wait on a notification.

See the description of `seL4_NBRecv()` in Section 2.2

10.2.11 Signal, Blocking Recv

`static inline void seL4_SignalRecv`

Perform a non-blocking signal (a send with no message) to a capability, then wait on another capability.

Type	Name	Description
seL4_CPtr	dest	Endpoint, reply capability or notification object to signal
seL4_CPtr	source	Endpoint or notification object to block on.
seL4_Word*	badge	Badge to be applied to the new capability.

Return value: This method does not return anything.

Description: See Section 2.2

10.3 Architecture-Independent Object Methods

10.3.1 CNode - Copy

`static inline int seL4_CNode_Copy`

Copy a capability, setting its access rights whilst doing so

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
`seL4_Word`	`dest_index`	CPTR to the destination slot. Resolved from the root of the destination CSpace.
`uint8_t`	`dest_depth`	Number of bits of dest_index to resolve to find the destination slot.
`seL4_CNode`	`src_root`	CPTR to the CNode that forms the root of the source CSpace. Must be at a depth of 32.
`seL4_Word`	`src_index`	CPTR to the source slot. Resolved from the root of the source CSpace.
`uint8_t`	`src_depth`	Number of bits of src_index to resolve to find the source slot.
`seL4_CapRights`	`rights`	The rights inherited by the new capability. Possible values for this type are given in Section 3.1.3.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.2 CNode - Delete

`static inline int seL4_CNode_Delete`

Delete a capability

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode at the root of the CSpace where the capability will be found. Must be at a depth of 32.
`seL4_Word`	`index`	CPTR to the capability. Resolved from the root of the _service parameter.
`uint8_t`	`depth`	Number of bits of index to resolve to find the capability being operated on.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.3 CNode - Mint

`static inline int seL4_CNode_Mint`

Copy a capability, setting its access rights and badge whilst doing so

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
`seL4_Word`	`dest_index`	CPTR to the destination slot. Resolved from the root of the destination CSpace.
`uint8_t`	`dest_depth`	Number of bits of dest_index to resolve to find the destination slot.
`seL4_CNode`	`src_root`	CPTR to the CNode that forms the root of the source CSpace. Must be at a depth of 32.
`seL4_Word`	`src_index`	CPTR to the source slot. Resolved from the root of the source CSpace.
`uint8_t`	`src_depth`	Number of bits of src_index to resolve to find the source slot.
`seL4_CapRights`	`rights`	The rights inherited by the new capability. Possible values for this type are given in Section 3.1.3.
`seL4_CapData_t`	`badge`	Badge to be applied to the new capability.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.4 CNode - Move

`static inline int seL4_CNode_Move`

Move a capability

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
`seL4_Word`	`dest_index`	CPTR to the destination slot. Resolved from the root of the destination CSpace.
`uint8_t`	`dest_depth`	Number of bits of dest_index to resolve to find the destination slot.
`seL4_CNode`	`src_root`	CPTR to the CNode that forms the root of the source CSpace. Must be at a depth of 32.
`seL4_Word`	`src_index`	CPTR to the source slot. Resolved from the root of the source CSpace.
`uint8_t`	`src_depth`	Number of bits of src_index to resolve to find the source slot.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.5 CNode - Mutate

`static inline int seL4_CNode_Mutate`

Move a capability, setting its badge in the process

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
`seL4_Word`	`dest_index`	CPTR to the destination slot. Resolved from the root of the destination CSpace.
`uint8_t`	`dest_depth`	Number of bits of dest_index to resolve to find the destination slot.
`seL4_CNode`	`src_root`	CPTR to the CNode that forms the root of the source CSpace. Must be at a depth of 32.
`seL4_Word`	`src_index`	CPTR to the source slot. Resolved from the root of the source CSpace.
`uint8_t`	`src_depth`	Number of bits of src_index to resolve to find the source slot.
`seL4_CapData_t`	`badge`	Badge to be applied to the new capability.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.6 CNode - Recycle

`static inline int seL4_CNode_Recycle`

The recycle method is intended roughly as a shortcut for reusing an object within the same protection domain. The method will first revoke the capability and will then reset some but not necessarily all aspects of an object to its neutral state. Recycling badged endpoint caps will only cancel IPCs for this badge. Recycled frames, page tables and directories should only be re-used in the same protection domain, not all authority to them is rescinded. For full reuse in a different protection domain, revoke and retype the untyped cap that was used to create the object. For the precise behaviour of recycle, see the formal specification.

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode at the root of the CSpace where the capability will be found. Must be at a depth of 32.
`seL4_Word`	`index`	CPTR to the capability. Resolved from the root of the _service parameter.
`uint8_t`	`depth`	Number of bits of index to resolve to find the capability being operated on.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.7 CNode - Revoke

`static inline int seL4_CNode_Revoke`

Delete all child capabilities of a capability

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode at the root of the CSpace where the capability will be found. Must be at a depth of 32.
`seL4_Word`	`index`	CPTR to the capability. Resolved from the root of the _service parameter.
`uint8_t`	`depth`	Number of bits of index to resolve to find the capability being operated on.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.8 CNode - Rotate

`static inline int seL4_CNode_Rotate`

Given 3 capability slots - a destination, pivot and source - move the capability in the pivot slot to the destination slot and the capability in the source slot to the pivot slot

Type	Name	Description
seL4_CNode	_service	CPTR to the CNode at the root of the CSpace where the destination slot will be found. Must be at a depth of 32.
seL4_Word	dest_index	CPTR to the destination slot. Resolved relative to _service. Must be empty unless it refers to the same slot as the source slot.
uint8_t	dest_depth	Depth to resolve dest_index to.
seL4_CapData_t	dest_badge	The new capdata for the capability that ends up in the destination slot.
seL4_CNode	pivot_root	CPTR to the CNode at the root of the CSpace where the pivot slot will be found. Must be at a depth of 32.
seL4_Word	pivot_index	CPTR to the pivot slot. Resolved relative to pivot_root. The resolved slot must not refer to the source or destination slots.
uint8_t	pivot_depth	Depth to resolve pivot_index to.
seL4_CapData_t	pivot_badge	The new capdata for the capability that ends up in the pivot slot.
seL4_CNode	src_root	CPTR to the CNode at the root of the CSpace where the source slot will be found. Must be at a depth of 32.
seL4_Word	src_index	CPTR to the source slot. Resolved relative to src_root.
uint8_t	src_depth	Depth to resolve src_index to.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.9 CNode - Swap Caller

`static inline int seL4_CNode_SwapCaller`

Save the reply capability from the last time the thread was called in the given CSpace so that it can be invoked or swapped back later. If the provided CPTR points to a reply cap, save this into the threads reply slot. This can be used to restore a previously saved reply capability.

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode at the root of the CSpace where the capability is to be swapped. Must be at a depth of 32. Must either be an empty CPTR or reply capability.
`seL4_Word`	`index`	CPTR to the slot in which to swap the capability. Resolved from the root of the _service parameter.
`uint8_t`	`depth`	Number of bits of index to resolve to find the slot being targeted.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.10 CNode - Swap TCB Caller

`static inline int seL4_CNode_SwapTCBCaller`

Save the reply capability from the last time the target thread was called in the given CSpace so that it can be invoked or swapped back later. If the provided CPTR points to a reply cap, save this into the target threads reply slot. This can be used to restore a previously saved reply capability.

Type	Name	Description
`seL4_CNode`	`_service`	CPTR to the CNode at the root of the CSpace where the capability is to be swapped. Must be at a depth of 32. Must either be an empty CPTR or reply capability.
`seL4_Word`	`index`	CPTR to the slot in which to swap the capability. Resolved from the root of the _service parameter.
`uint8_t`	`depth`	Number of bits of index to resolve to find the slot being targeted.
`seL4_CPtr`	`tcb`	A capability to the target thread.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 3.1.2.

10.3.11 Debug - Halt

`static inline void seL4_DebugHalt`

Halt the system

Type	Name	Description
`void`		

Return value: This method does not return anything.

Description: Halts the system, if debugging is turned on.

10.3.12 Debug - Put Character

`static inline void seL4_DebugPutChar`

Print a character to the console

Type	Name	Description
`char`	`c`	The character to print.

Return value: This method does not return anything.

Description: Prints a character to the serial port, if debugging is turned on.

10.3.13 IRQ Control - Get

`static inline int seL4_IRQControl_Get`

Create an IRQ handler capability

Type	Name	Description
`seL4_IRQControl`	`_service`	An IRQControl capability. This gives you the authority to make this call.
`int`	`irq`	The IRQ that you want this capability to handle.
`seL4_CNode`	`root`	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
`seL4_Word`	`index`	CPTR to the destination slot. Resolved from the root of the destination CSpace.
`uint8_t`	`depth`	Number of bits of dest_index to resolve to find the destination slot.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.1

10.3.14 IRQ Handler - Acknowledge

`static inline int seL4_IRQHandler_Ack`

Acknowledge the receipt of an interrupt and re-enable it

Type	Name	Description
`seL4_IRQHandler`	`_service`	The IRQ handler capability.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.1

10.3.15 IRQ Handler - Clear

`static inline int seL4_IRQHandler_Clear`

Clear the handler capability from the IRQ slot

Type	Name	Description
`seL4_IRQHandler`	`_service`	The IRQ handler capability.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.1

10.3.16 IRQ Handler - Set Notification

`static inline int seL4_IRQHandler_SetNotification`

Set the notification which the kernel will signal on interrupts controlled by the supplied IRQ handler capability

Type	Name	Description
`seL4_IRQHandler`	`_service`	The IRQ handler capability.
`seL4_CPtr`	`notification`	The notification which the IRQs will signal.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.1

10.3.17 TCB - Bind Notification

`static inline int seL4_TCB_BindNotification`

Binds a notification object to a `TCB`

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_CPtr`	`notification`	Notification to bind.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 5.3

10.3.18 TCB - Unbind Notification

`static inline int seL4_TCB_UnbindNotification`

Unbinds any notification object from a `TCB`

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 5.3

10.3.19 TCB - Configure

`static inline int seL4_TCB_Configure`

Set the parameters of a TCB

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_CPtr`	`fault_ep`	CPTR to the endpoint which receives IPCs when this thread faults. This capability is in the caller's cspace.
`seL4_CPtr`	`temporal_fault_ep`	CPTR to the endpoint which receives IPCs when this thread triggers a temporal fault. This capability is in the caller's cspace.
`seL4_Prio_t`	`priority`	A `seL4_Prio_t` structure as described in Section 6.1.8.
`seL4_CPtr_t`	`schedcontext`	Capability to the scheduling context that the TCB should run on. If the scheduling context is already bound to a notification or TCB that is not this TCB this operation will fail. Similarly, if this TCB is already bound to a scheduling context that is not this scheduling context, this will also fail.
`seL4_CNode`	`cspace_root`	The new CSpace root.
`seL4_CapData_t`	`cspace_root_data`	Optionally set the guard and guard size of the new root CNode. If set to zero, this parameter has no effect.
`seL4_CNode`	`vspace_root`	The new VSpace root.
`seL4_CapData_t`	`vspace_root_data`	Has no effect on IA-32 or ARM processors.
`seL4_Word`	`buffer`	Location of the thread's IPC buffer. Must be 512-byte aligned. The IPC buffer may not cross a page boundary.
`seL4_CPtr`	`bufferFrame`	Capability to a page containing the thread's IPC buffer.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1

10.3.20 TCB - Copy Registers

`static inline int seL4_TCB_CopyRegisters`

Copy the registers from one thread to another

Type	Name	Description
seL4_TCB	_service	Capability to the TCB which is being operated on. This is the destination TCB.
seL4_TCB	source	Cap to the source TCB.
bool	suspend_source	The invocation should also suspend the source thread.
bool	resume_target	The invocation should also resume the destination thread.
bool	transfer_frame	Frame registers should be transferred.
bool	transfer_integer	Integer registers should be transferred.
uint8_t	arch_flags	Architecture dependent flags. These have no meaning on either IA-32 or ARM.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: In the context of this function, frame registers are those that are read, modified or preserved by a system call and integer registers are those that are not. Refer to the seL4 userland library source for specifics. Section 6.1.3

10.3.21 TCB - Read Registers

`static inline int seL4_TCB_ReadRegisters`

Read a thread's registers into the first `count` fields of a given seL4_UserContext

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`bool`	`suspend_source`	The invocation should also suspend the source thread.
`uint8_t`	`arch_flags`	Architecture dependent flags. These have no meaning on either IA-32 or ARM.
`seL4_Word`	`count`	The number of registers to read.
`seL4_UserContext*`	`regs`	The structure to read the registers into.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.12

10.3.22 TCB - Resume

`static inline int seL4_TCB_Resume`

Resume a thread

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1

10.3.23 TCB - Set IPC Buffer

`static inline int seL4_TCB_SetIPCBuffer`

Set a thread's IPC buffer

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_Word`	`buffer`	Location of the thread's IPC buffer. Must be 512-byte aligned. The IPC buffer may not cross a page boundary.
`seL4_CPtr`	`bufferFrame`	Capability to a page containing the thread's IPC buffer.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Sections 6.1 and 4.1

10.3.24 TCB - Set Priority

`static inline int seL4_TCB_SetPriority`

Change a thread's priority

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`uint8_t`	`priority`	The thread's new priority.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.8

10.3.25 TCB - Set Maximum Controlled Priority

`static inline int seL4_TCB_SetMCPriority`

Change a thread's maximum controlled priority

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`uint8_t`	`mcp`	The thread's new maximum controlled priority.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.8

10.3.26 TCB - Set Criticality

`static inline int seL4_TCB_SetCriticality`

Change a thread's criticality

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`uint8_t`	`criticality`	The thread's new criticality.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.10

10.3.27 TCB - Set Maximum Controlled Criticality

`static inline int seL4_TCB_SetMCCriticality`

Change a thread's maximum controlled criticality

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`uint8_t`	`mcc`	The thread's new maximum controlled criticality.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.10

10.3.28 TCB - Set Space

`static inline int seL4_TCB_SetSpace`

Set the fault endpoint, CSpace and VSpace of a thread

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_CPtr`	`fault_ep`	CPTR to the endpoint which receives IPCs when this thread faults. This capability is in the caller's cspace.
`seL4_CPtr`	`temporal_fault_ep`	CPTR to the endpoint which receives IPCs when this thread triggers a temporal fault. This capability is in the caller's cspace.
`seL4_CNode`	`cspace_root`	The new CSpace root.
`seL4_CapData_t`	`cspace_root_data`	Optionally set the guard and guard size of the new root CNode. If set to zero, this parameter has no effect.
`seL4_CNode`	`vspace_root`	The new VSpace root.
`seL4_CapData_t`	`vspace_root_data`	Has no effect on IA-32 or ARM processors.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1

10.3.29 TCB - Suspend

`static inline int seL4_TCB_Suspend`

Suspend a thread, cancelling any active messages or signals.

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.3

10.3.30 TCB - Write Registers

`static inline int seL4_TCB_WriteRegisters`

Set a thread's registers to the first `count` fields of a given seL4_UserContext

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`bool`	`resume_target`	The invocation should also resume the destination thread.
`uint8_t`	`arch_flags`	Architecture dependent flags. These have no meaning on either IA-32 or ARM.
`seL4_Word`	`count`	The number of registers to be set.
`seL4_UserContext*`	`regs`	Data structure containing the new register values.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.12

10.3.31 Untyped - Retype

`static inline int seL4_Untyped_Retype`

Retype an untyped object

Type	Name	Description
seL4_Untyped	_service	CPTR to an untyped object.
int	type	The seL4 object type that we are retyping to.
int	size_bits	Only valid for objects with various sizes. Explained below.
seL4_CNode	root	CPTR to the CNode at the root of the destination CSpace.
int	node_index	CPTR to the destination CNode. Resolved relative to the root parameter.
int	node_depth	Number of bits of node_index to translate when addressing the destination CNode.
int	node_offset	Number of slots into the node at which capabilities start being placed.
int	num_objects	Number of capabilities to create.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: Given a capability, `_service`, to an untyped object, creates `num_objects` of the requested type. Creates `num_objects` capabilities to the new objects starting at `node_offset` in the given CNode.

The retype method can be complex because multiple capabilities may be created and some objects such as CNodes may have varying sizes.

Most kernel objects have a fixed size, and hence no further information must be given to the kernel about them. CNodes and Untyped Memory however have a variable size, and so the user must additionally give a value in the `size_bits` parameter to specify the desired size for the objects to be created. For CNodes, the number of slots in each CNode is calculated as 2^{size_bits} and hence the required amount of memory for each is $16*2^{size\_bits}$. For the case where Untyped Memory is being split into smaller blocks of Untyped Memory, the size of each of the resulting Untyped Memory blocks is calculated as 2^{size_bits}. If the size of the memory area needed (calculated by the object size multiplied by `num_objects`) is greater than the remaining unallocated memory of the untyped memory region, an error will result. Otherwise object allocation will proceed.

To understand how memory from the untyped will be allocated when creating objects see Section 2.4.1.

The retype method places capabilities to the objects produced at consecutive locations in a CNode. The CNode is specified by the `root`, `node_index` and `node_depth` parameters (see Section 3.3.2). The `node_offset` parameter specifies the index in the CNode at which the first capability will be placed. The `num_objects` parameter specifies the

Object	Object Size
n-bit Untyped	2^n bytes (where $n \geq 4$)
n-slot CNode	$16n$ bytes (where $n \geq 2$)
Endpoint	16 bytes
Notification	16 bytes
IRQ Control	—
IRQ Handler	—

Table 10.1: Platform-independent object sizes.

number of capabilities (and, hence, objects) to create. All slots must be empty or an error will result. All resulting objects will be placed in the same CNode.

10.3.32 Summary of Object Sizes

When retyping untyped memory it is useful to know how much memory the object will require. Object sizes are summarised in Tables 10.1, 10.2 and 10.3.

IA-32 Object	Object Size
Thread Control Block	1KiB
IA32 4K Frame	4KiB
IA32 4M Frame	4MiB
IA32 Page Directory	4KiB
IA32 Page Table	4KiB
IA32 ASID Control	—
IA32 ASID Pool	4KiB
IA32 Port	—
IA32 IO Space	—
IA32 IO Page table	4KiB

Table 10.2: IA-32-specific object sizes.

ARM Object	Object Size
Thread Control Block	512 bytes
ARM Small Frame	4KiB
ARM Large Frame	64KiB
ARM Section	1MiB
ARM Supersection	16MiB
ARM Page Directory	16KiB
ARM Page Table	1KiB
ARM ASID Control	—
ARM ASID Pool	4KiB

Table 10.3: ARM-specific object sizes.

10.3.33 SchedControl - Configure

`static inline int seL4_SchedControl_Configure`

Set the parameters of a scheduling context by invoking the scheduling control capability. If the scheduling context is bound to a currently running thread, the parameters will take effect immediately: that is the current budget will be increased or reduced by the difference between the new and previous budget and the replenishment time will be updated according to any difference in the period. This can result in active threads being post-poned or released depending on the nature of the parameter change and the state of the thread. Additionally, if the scheduling context was previously empty (no budget) but bound to a runnable thread, this can result in a thread running for the first time since it now has access to CPU time. This call will return seL4_Invalid_Argument if the parameters are too small (smaller than the kernel WCET for this platform) or too large (will overflow the timer).

Type	Name	Description
`seL4_SchedControl`	`_service`	The SchedControl capability.
`seL4_CPtr_t`	`schedcontext`	Capability to the scheduling context which is being operated on.
`seL4_Time_t`	`budget`	Timeslice in microseconds, when the budget expires the thread will be pre-empted.
`seL4_Time_t`	`period`	Period in microseconds, the budget is replenished every time the period expires.
`seL4_Word_t`	`data`	A word of data to identify the owner of the scheduling context that a thread faulted on to a temporal fault handler.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.8

10.3.34 SchedControl - Set Criticality

`static inline int seL4_SchedControl_Configure`

Set the system criticality level.

Type	Name	Description
`seL4_SchedControl`	`_service`	The SchedControl capability.
`seL4_Uint32`	`criticality`	The system criticality level. Threads with criticalities below the system criticality level will not be chosen by the scheduler.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 6.1.8

10.3.35 SchedContext - Yield

`static inline void seL4_SchedContext_Yield`

Surrender remaining timeslice, which will be refilled when the budget is due to be replenished and place the thread bound to this scheduling context at the end of the scheduling queue for it's priority.

If this operation is invoked on the scheduling context of the currently running thread and the budget is already due to be replenished and there are no other threads at the caller's priority, the caller will be rescheduled immediately.

Type	Name	Description
`void`		

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: see Section 6.1.8

10.3.36 SchedContext - YieldTo

`static inline void seL4_SchedContext_YieldTo`

If a thread is currently runnable and running on this scheduling context and the scheduling context has available budget, place it at the head of the scheduling queue. If the caller is at an equal priority to the thread this will result in the thread being scheduled. If the caller is at a higher priority the thread will not run until the threads priority is the highest priority in the system. The caller must have a maximum control priority greater than or equal to the threads priority.

Type	Name	Description
`seL4_SchedContext`	`_service`	Capability to the scheduling context which is being operated on.

Return value: A `seL4_SchedContext_YieldTo_t` containing `consumed`, which is the amount of time that threads have been executing using this scheduling context since `seL4_SchedContext_YieldTo()` or `seL4_SchedContext_Consumed()` was called, and `error`, an error code, see Section 10.1.

Description: see Section 6.1.8

10.3.37 SchedContext - Consumed

`static inline void seL4_SchedContext_Consumed`

Return the amount of time the scheduling context has consumed.

Type	Name	Description
`seL4_SchedContext`	`_service`	Capability to the scheduling context which is being operated on.

Return value: A `seL4_SchedContext_Consumed_t` containing `consumed`, which is the amount of time that threads have been executing using this scheduling context since `seL4_SchedContext_YieldTo()` or `seL4_SchedContext_Consumed()` was called, and `error`, an error code, see Section 10.1.

Description: see Section 6.1.8

10.3.38 SchedContext - Bind

`static inline int seL4_SchedContext_Bind`

Bind an object to a scheduling context. The object can be a notification object or a thread.

If the object is a thread and the thread is in a runnable state and the scheduling context has available budget, this will start the thread running.

If the object is a notification, when passive threads wait on the notification object and a signal arrives, the passive thread will receive the scheduling context and possess it until it waits on the notification object again.

This operation will fail if the scheduling context is already bound to a thread or notification object.

Type	Name	Description
`seL4_SchedContext`	`_service`	Capability to the scheduling context which is being operated on.
`seL4_CPtr`	`cap`	Capability to a TCB or a notification object

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: see Section 6.1.8

10.3.39 SchedContext - UnbindObject

`static inline int seL4_SchedContext_UnbindObject`

Unbind an object from a scheduling context. The object can be either a thread or a notification.

If the thread being unbound is the thread that is bound to this scheduling context, this will render the thread passive, see Section 6.1.5. However if the thread being unbound received the scheduling context via scheduling context donation over IPC, the scheduling context will be returned to the thread that it was originally bound to.

If the object is a notification and it is bound to the scheduling context, unbind it.

Type	Name	Description
`seL4_SchedContext`	`_service`	Capability to the scheduling context which is being operated on.
`seL4_CPtr`	`cap`	Capability to a notification that is bound to the scheduling context or capability to a tcb that is bound to this scheduling context or has received it through scheduling context donation.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: see Section 6.1.5

10.3.40 SchedContext - Unbind

`static inline int seL4_SchedContext_Unbind`

Unbind any objects (threads or notification objects) from a scheduling context. This will render the bound thread passive, see Section 6.1.5.

Type	Name	Description
`seL4_SchedContext`	`_service`	Capability to the scheduling context which is being operated on.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: see Section 6.1.5

10.4 IA-32-Specific Object Methods

10.4.1 IA32 ASID Control - Make Pool

`static inline int seL4_IA32_ASIDControl_MakePool`

Create an IA-32 ASID pool

Type	Name	Description
`seL4_IA32_ASIDControl`	`_service`	The master ASIDControl capability.
`seL4_Untyped`	`untyped`	Capability to an untyped memory object that will become the pool. Must be 4K bytes.
`seL4_CNode`	`root`	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
`seL4_Word`	`index`	CPTR to the destination slot. Resolved from the root of the destination CSpace.
`uint8_t`	`depth`	Number of bits of index to resolve to find the destination slot.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.2 IA32 ASID Pool - Assign

`static inline int seL4_IA32_ASIDPool_Assign`

Assign an ASID pool

Type	Name	Description
`seL4_IA32_ASIDPool`	`_service`	The ASID pool which is being assigned to. Must not be full. Each ASID pool can contain 1024 entries.
`seL4_IA32_PageDirectory`	`vroot`	The page directory that is being assigned to an ASID pool. Must not already be assigned to an ASID pool.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.3 IA32 IO Port - In 8

`static inline seL4_IA32_IOPort_In8_t seL4_IA32_IOPort_In8`

Read 8 bits from an IO port

Type	Name	Description
`seL4_IA32_IOPort`	`_service`	An IO port capability.
`uint16_t`	`port`	The port to read from.

Return value: A seL4_IA32_IOPort_In8_t structure as described in Section 8.2.1

Description: See Section 8.2.1

10.4.4 IA32 IO Port - In 16

`static inline seL4_IA32_IOPort_In16_t seL4_IA32_IOPort_In16`

Read 16 bits from an IO port

Type	Name	Description
`seL4_IA32_IOPort`	`_service`	An IO port capability.
`uint16_t`	`port`	The port to read from.

Return value: A seL4_IA32_IOPort_In16_t structure as described in Section 8.2.1

Description: See Section 8.2.1

10.4.5 IA32 IO Port - In 32

`static inline seL4_IA32_IOPort_In32_t seL4_IA32_IOPort_In32`

Read 32 bits from an IO port

Type	Name	Description
`seL4_IA32_IOPort`	`_service`	An IO port capability.
`uint16_t`	`port`	The port to read from.

Return value: A seL4_IA32_IOPort_In32_t structure as described in Section 8.2.1

Description: See Section 8.2.1

10.4.6 IA32 IO Port - Out 8

`static inline int seL4_IA32_IOPort_Out8`

Write 8 bits to an IO port

Type	Name	Description
seL4_IA32_IOPort	_service	An IO port capability.
uint16_t	port	The port to write to.
uint8_t	data	Data to write to the IO port.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.2.1

10.4.7 IA32 IO Port - Out 16

`static inline int seL4_IA32_IOPort_Out16`

Write 16 bits to an IO port

Type	Name	Description
seL4_IA32_IOPort	_service	An IO port capability.
uint16_t	port	The port to write to.
uint16_t	data	Data to write to the IO port.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.2.1

10.4.8 IA32 IO Port - Out 32

`static inline int seL4_IA32_IOPort_Out32`

Write 32 bits to an IO port

Type	Name	Description
`seL4_IA32_IOPort`	`_service`	An IO port capability.
`uint16_t`	`port`	The port to write to.
`uint32_t`	`data`	Data to write to the IO port.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.2.1

10.4.9 IA32 IO Page Table - Map

`static inline int seL4_IA32_IOPageTable_Map`

Map a page table into an IOSpace

Type	Name	Description
`seL4_IA32_IOPageTable`	`_service`	The page table that is being mapped.
`seL4_IA32_IOSpace`	`iospace`	The IOSpace that the page table is being mapped into.
`seL4_Word`	`ioaddr`	The address that the page table is being mapped at.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.2.2

10.4.10 IA32 Page - Map IO

`static inline int seL4_IA32_Page_MapIO`

Map a page into an IOSpace

Type	Name	Description
seL4_IA32_Page	_service	The frame that is being mapped.
seL4_IA32_IOSpace	iospace	The IOSpace that the frame is being mapped into.
seL4_CapRights	rights	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
seL4_Word	ioaddr	The address that the frame is being mapped at.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Section 8.2.2

10.4.11 IA32 Page - Map

`static inline int seL4_IA32_Page_Map`

Map a page into an address space

Type	Name	Description
seL4_IA32_Page	_service	Capability to the page to map.
seL4_IA32_PageDirectory	pd	Capability to the VSpace which will contain the mapping.
seL4_Word	vaddr	Virtual address to map the page into.
seL4_CapRights	rights	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
seL4_IA32_VMAttributes	attr	VM Attributes for the mapping. Possible values for this type are given in Chapter 7.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.12 IA32 Page - Remap

`static inline int seL4_IA32_Page_Remap`

Remap a page

Type	Name	Description
`seL4_IA32_Page`	`_service`	Capability to the page to map.
`seL4_IA32_PageDirectory`	`pd`	Capability to the VSpace which will contain the mapping.
`seL4_CapRights`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
`seL4_IA32_VMAttributes`	`attr`	VM Attributes for the mapping. Possible values for this type are given in Chapter 7.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.13 IA32 Page - Unmap

`static inline int seL4_IA32_Page_Unmap`

Unmap a page

Type	Name	Description
`seL4_IA32_Page`	`_service`	Capability to the page to unmap.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.14 IA32 Page - Get Address

`static inline seL4_IA32_Page_GetAddress_t seL4_IA32_Page_GetAddress`

Get the physical address of the underlying frame

Type	Name	Description
`seL4_IA32_Page`	`_service`	Capability to the page to lookup.

Return value: A `seL4_IA32_Page_GetAddress_t` struct that contains `seL4_Word paddr`, which holds the physical address of the page, and `int error`. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.15 IA32 Page Table - Map

`static inline int seL4_IA32_PageTable_Map`

Map a page table into an address space

Type	Name	Description
`seL4_IA32_PageTable`	`_service`	Capability to the page table to map.
`seL4_IA32_PageDirectory`	`pd`	Capability to the VSpace which will contain the mapping.
`seL4_Word`	`vaddr`	Virtual address to map the page into.
`seL4_IA32_VMAttributes`	`attr`	VM Attributes for the mapping. Possible values for this type are given in Chapter 7.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.16 IA32 Page Table - Unmap

`static inline int seL4_IA32_PageTable_Unmap`

Unmap a page table from its address space and zero it out

Type	Name	Description
`seL4_IA32_PageTable`	`_service`	Capability to the page table to unmap.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.4.17 X86 Page Directory - Get Status Bits

`static inline int seL4_X86_PageDirectory_GetStatusBits`

Retrieved the accessed and dirty bits of a page mapped mapped into an address space

Type	Name	Description
`seL4_X86_PageDirectory`	`_service`	Capability to the address space to query.
`seL4_Word`	`vaddr`	Virtual address of the page to query

Return value: A seL4_X86_PageDirectory_GetStatusBits_t structure

Description: See Chapter 7

10.5 ARM-Specific Object Methods

10.5.1 ARM ASID Control - Make Pool

`static inline int seL4_ARM_ASIDControl_MakePool`

Create an ASID Pool

Type	Name	Description
`seL4_ARM_ASIDControl`	`_service`	The master ASIDControl capability.
`seL4_Untyped`	`untyped`	Capability to an untyped memory object that will become the pool. Must be 4K bytes.
`seL4_CNode`	`root`	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
`seL4_Word`	`index`	CPTR to the destination slot. Resolved from the root of the destination CSpace.
`uint8_t`	`depth`	Number of bits of index to resolve to find the destination slot.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.2 ARM ASID Pool - Assign

`static inline int seL4_ARM_ASIDPool_Assign`

Assign an ASID Pool

Type	Name	Description
`seL4_ARM_ASIDPool`	`_service`	The ASID pool which is being assigned to. Must not be full. Each ASID pool can contain 1024 entries.
`seL4_ARM_PageDirectory`	`vroot`	The page directory that is being assigned to an ASID pool. Must not already be assigned to an ASID pool.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.3 ARM Page - Clean Data

`static inline int seL4_ARM_Page_Clean_Data`

Cleans the data cache out to RAM. The start and end are relative to the page being serviced.

Type	Name	Description
`seL4_ARM_Page`	`_service`	The page whose contents will be flushed.
`seL4_Word`	`start_offset`	The offset, relative to the start of the page inclusive.
`seL4_Word`	`end_offset`	The offset, relative to the start of the page exclusive.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.4 ARM Page - Invalidate Data

`static inline int seL4_ARM_Page_Invalidate_Data`

Invalidates the cache range within the given page. The start and end are relative to the page being serviced and should be aligned to a cache line boundary where possible. An additional clean is performed on the outer cache lines if the start and end are not aligned, to clean out the bytes between the requested and the cache line boundary.

Type	Name	Description
`seL4_ARM_Page`	`_service`	The page whose contents will be flushed.
`seL4_Word`	`start_offset`	The offset, relative to the start of the page inclusive.
`seL4_Word`	`end_offset`	The offset, relative to the start of the page exclusive.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.5 ARM Page - Clean and Invalidate Data

`static inline int seL4_ARM_Page_CleanInvalidate_Data`

Clean and invalidates the cache range within the given page. The range will be flushed out to RAM. The start and end are relative to the page being serviced.

Type	Name	Description
`seL4_ARM_Page`	`_service`	The page whose contents will be flushed.
`seL4_Word`	`start_offset`	The offset, relative to the start of the page inclusive.
`seL4_Word`	`end_offset`	The offset, relative to the start of the page exclusive.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.6 ARM Page - Unify Instruction Cache

`static inline int seL4_ARM_Page_Unify_Instruction`

Unify Instruction Cache. Cleans data lines to point of unification, invalidate corresponding instruction lines to point of unification, then invalidates branch predictors. The start and end are relative to the page being serviced.

Type	Name	Description
`seL4_ARM_Page`	`_service`	The page whose contents will be flushed.
`seL4_Word`	`start_offset`	The offset, relative to the start of the page inclusive.
`seL4_Word`	`end_offset`	The offset, relative to the start of the page exclusive.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.7 ARM Page - Map

`static inline int seL4_ARM_Page_Map`

Map a page into an address space

Type	Name	Description
`seL4_ARM_Page`	`_service`	Capability to the page to map.
`seL4_ARM_PageDirectory`	`pd`	Capability to the VSpace which will contain the mapping.
`seL4_Word`	`vaddr`	Virtual address to map the page into.
`seL4_CapRights`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
`seL4_ARM_VMAttributes`	`attr`	VM Attributes for the mapping. Possible values for this type are given in Chapter 7.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.8 ARM Page - Remap

`static inline int seL4_ARM_Page_Remap`

Remap a page

Type	Name	Description
`seL4_ARM_Page`	`_service`	Capability to the page to remap.
`seL4_ARM_PageDirectory`	`pd`	Capability to the VSpace which will contain the mapping.
`seL4_CapRights`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
`seL4_ARM_VMAttributes`	`attr`	VM Attributes for the mapping. Possible values for this type are given in Chapter 7.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.9 ARM Page - Unmap

`static inline int seL4_ARM_Page_Unmap`

Unmap a page

Type	Name	Description
`seL4_ARM_Page`	`_service`	Capability to the page to unmap.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.10 ARM Page - Get Address

`static inline seL4_ARM_Page_GetAddress_t seL4_ARM_Page_GetAddress`

Get the physical address of the underlying frame

Type	Name	Description
`seL4_ARM_Page`	`_service`	Capability to the page to lookup.

Return value: A seL4_ARM_Page_GetAddress_t struct that contains `seL4_Word paddr`, which holds the physical address of the page, and `int error`. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.11 ARM Page Table - Map

`static inline int seL4_ARM_PageTable_Map`

Map a page table into an address space

Type	Name	Description
`seL4_ARM_PageTable`	`_service`	Capability to the page table that will be mapped.
`seL4_ARM_PageDirectory`	`pd`	Capability to the VSpace which will contain the mapping.
`seL4_Word`	`vaddr`	Virtual address to map the page into.
`seL4_ARM_VMAttributes`	`attr`	VM Attributes for the mapping. Possible values for this type are given in Chapter 7.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

10.5.12 ARM Page Table - Unmap

`static inline int seL4_ARM_PageTable_Unmap`

Unmap a page table from its address space and zero it out

Type	Name	Description
`seL4_ARM_PageTable`	`_service`	Capability to the page table that will be unmapped.

Return value: A return value of `0` indicates success. A non-zero value indicates that an error occurred. See Section 10.1 for a description of the message register and tag contents upon error.

Description: See Chapter 7

Bibliography

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009. ACM.

[Boy09] 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.

[BSC+11] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 2011.

[BSH12] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *EuroSys*, pages 323–336, Bern, Switzerland, April 2012.

[CKS08] 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-Verlag.

[DEK+06] 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.

[EKE08] 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-Verlag.

[Int11] Intel Corporation. *Intel Virtualization Technology for Directed I/O — Architecture Specification*, February 2011. http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.

[KEH+09] 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.

[MMB+13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security & Privacy*, pages 415–429, San Francisco, CA, May 2013.

[Pal09] Ameya Palande. Capability-based secure DMA in seL4. Masters thesis, Vrije Universiteit, Amsterdam, January 2009.

[SA99] Tom Shanley and Don Anderson. *PCI System Architecture*. Mindshare, Inc., 1999.

[SWG+11] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editor, *Interactive Theorem Proving (ITP)*, pages 325–340, Nijmegen, The Netherlands, August 2011.

[TKN07] 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. ACM.

[vT10] Michael von Tessin. Towards high-assurance multiprocessor virtualisation. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *Proceedings of the 6th International Verification Workshop*, volume 3 of *EasyChair Proceedings in Computing*, pages 110–125, Edinburgh, UK, July 2010. EasyChair.

[vT12] Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *Proceedings of the 2nd Workshop on Systems for Future Multi-core Architectures*, Bern, Switzerland, April 2012.

[WKS+09] 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-Verlag.