Data61 Trustworthy Systems
https://ts.data61.csiro.au/projects/TS/

seL4 MCS Reference Manual
Version 9.0.0-MCS

Trustworthy Systems Team, Data61
https://sel4.systems/contact/

12 April 2018

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@data61.csiro.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 execuion 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;

Reply objects are used to store single-use reply capabilities, and are provided by the receiver during 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 three system calls, *Send*, *Receive* and *Yield*. 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 *Yield* system call is not associated with any kernel object and is the only operation that does not invoke a capability. *Wait* is a variant of *Receive* that does not require a reply object to be provided.

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_Wait()` 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_NBWait()` is used by a thread to poll for messages through endpoints or notifications. If no sender or notification is pending, the system call returns immediately.

`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 giving the receiver the right to reply to the original sender. The reply capability is deposited in a dedicated reply object provided by the receiver, 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_ReplyRecv()` combines an invocation of the reply capability (`seL4_Send()`) and uses the same reply object for the `seL4_Recv()` phase. 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_NBSendWait()` combined an `seL4_NBSend()` and `seL4_Wait()` into one atomic system call.

`seL4_NBSendRecv()` combined an `seL4_NBSend()` and `seL4_Recv()` into one atomic system call.

`seL4_Yield()` is the only system call that does not require a capability to be used. It forfeits the remainder of the calling thread's timeslice and causes invocation of the kernel's scheduler. If there are no other runnable threads with the same priority as the caller, the calling thread will immediately be scheduled with a fresh timeslice, unless it is a sporadically scheduled thread, which will be scheduled when the next replenishment is due, see Section 6.1.4.

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 for each node, which allows scheduling context to be populated with parameters, which combined with priority control thread's 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.

Reply objects (see Chapter 4) track scheduling context donation and provide a container for single-use reply capabilities. They are provided by `seL4_Recv()`.

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 then 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 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.

Untyped memory objects represent two different types of memory: general purpose memory, or device memory. *General purpose* memory can be untyped into any other object type and used for any operation on untyped memory provided by the kernel. *Device memory* covers memory regions reserved for devices as determined by the hardware platform, and usage of these objects is restricted by the kernel in the following ways:

- Device untyped objects can only be retyped into frames or other untyped objects; developers cannot, for example, create an endpoint from device memory.
- Frame objects retyped from device untyped objects cannot be set as thread IPC buffers, or used in the creation of an ASID pool

The type attribute (whether it represents *general purpose* or *device* memory) of a child untyped object is inherited from its parent untyped object. That is, any child of a device untyped will also be a device untyped. Developers cannot change the type attribute of an untyped.

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()`.

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

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.

2.4.2 Summary of Object Sizes

When retyping untyped memory it is useful to know how much memory the object will require. Object sizes are defined in libsel4.

Note that CNodes and Untyped Objects have variables sizes. When retyping untyped memory into CNodes or breaking an Untyped Object into smaller Untyped Objects, the `size_bits` argument to `seL4_Untyped_Retype()` is used to specify the size of the resulting objects. For all other object types, the size is fixed, and the `size_bits` argument to `seL4_Untyped_Retype()` is ignored.

A single call to `seL4_Untyped_Retype()` can retype a single Untyped Object into multiple objects. The number of objects to create is specified by its `num_objects` argument. All created objects must be of the same type, specified by the `type` argument. In the case of variable-sized objects, each object must also be of the same size. 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 Object, an error will result.

[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 2.4.2). 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_CancelBadgedSends()` cancels any outstanding sends that use the same badge and object as the specified capability.

3.1.3 Capabilities to Newly-Retyped Objects

When retyping untyped memory into objects with `seL4_Untyped_Retype()`, capabilities to the newly-retyped objects are placed in consecutive slots in a CNode specified by its `root`, `node_index`, and `node_depth` arguments. The `node_offset` argument specifies the index into the CNode at which the first capability will be placed. The `num_objects` argument specifies the number of capabilities (and, hence, objects) to create. All slots must be empty or an error will result. All resulting capabilities will be placed in the same CNode.

3.1.4 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.

3.1.5 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.

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.

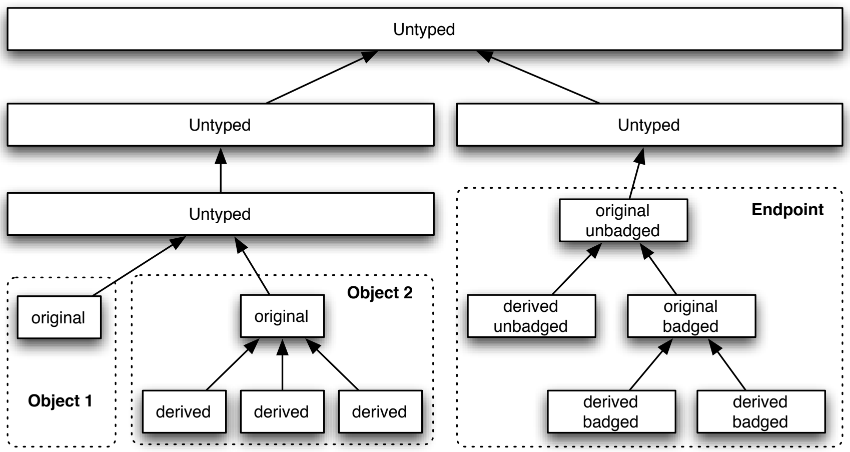

Figure 3.1: Example capability derivation tree.

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 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 and Revocation

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.

Note that for page tables and page directories `seL4_CNode_Revoke()` will not 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

[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.

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;
- 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.

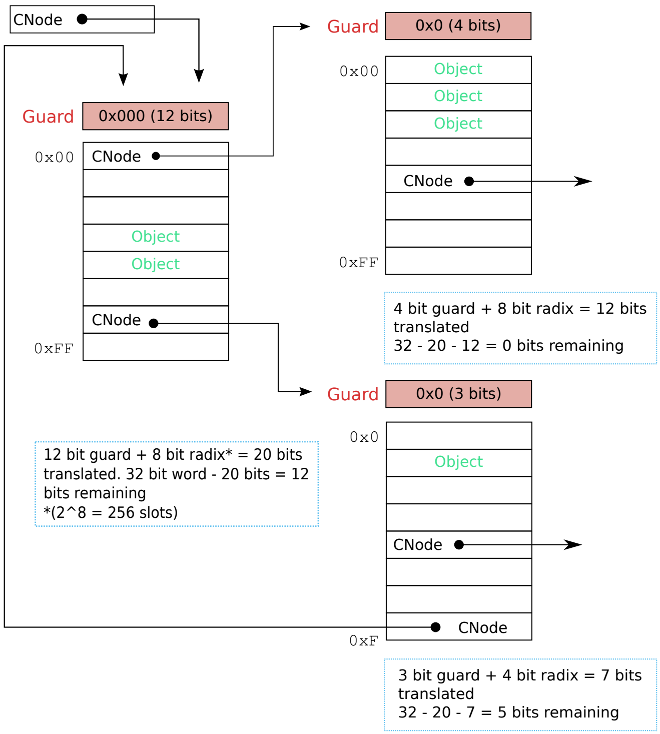

Figure 3.2: An example CSpace demonstrating object references at all levels, various guard and radix sizes and internal CNode references.

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*.

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.

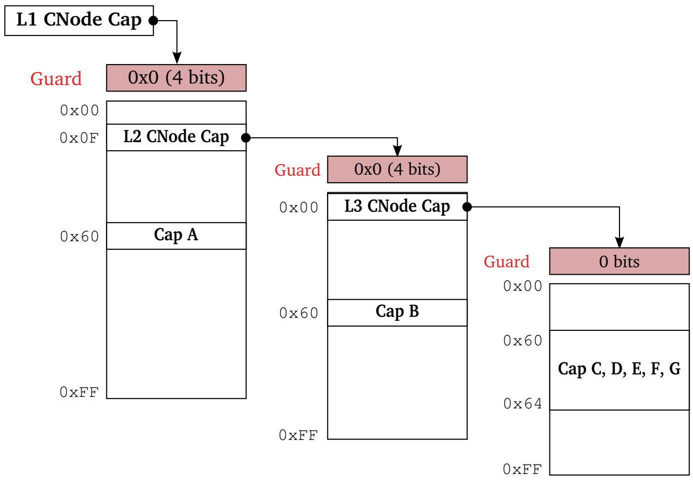

Figure 3.3: An arbitrary CSpace layout.

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 + seL4_CapFault_BitsLeft`	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 + seL4_CapFault_BitsLeft`	Bits of CPTR remaining to decode
`Offset + seL4_CapFault_DepthMismatch_BitsFound`	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 + seL4_CapFault_BitsLeft`	Bits of CPTR remaining to decode
`Offset + seL4_CapFault_GuardMismatch_GuardFound`	The CNode's guard
`Offset + seL4_CapFault_GuardMismatch_BitsFound`	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 Endpoints 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 IPC buffer is assigned to the calling thread (see Section 6.1 and Section 10.3.23).

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 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.

The kernel assumes that the IPC buffer contains a structure of type `seL4_IPCBuffer` as defined in Table 4.1. 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

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

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.

Only the low 28 bits of the badge are available for use. The kernel will silently ignore any usage of the high 4 bits.

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 3.1.5 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 Scheduling contexts

6.1 Threads

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 (SCO). 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. Each thread belongs to exactly one security domain (see Section 6.3).

6.1.2 Thread Creation

Like other objects, TCBs 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. Both of these methods place the thread in a runnable state. If the thread has a scheduling context it will begin running, and on multicore machines, the thread would be running on the core that the scheduling context is for.

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.

6.1.4 Scheduling Contexts

Access to CPU execution time is controlled through scheduling context objects. Scheduling contexts are configured with 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, and the kernel does not conduct an admission test. As a result the set of all parameters is not neccesserily schedulable. If multiple threads have available budget concurrently they are scheduled first-in first-out, and round-robin scheduling is applied once the budget is expired.

A scheduling context that is eligible to be picked by the scheduler, i.e has budget available, is referred to as *active*. Budget charging and replenishment rules are different for round-robin and sporadic threads. For round-robin threads, the budget is charged each time the current node's scheduling context is changed, until it is depleted and then refilled immediately.

Threads where $b == p$ are treated as round robin threads, where b acts as a timeslice. Otherwise the kernel uses *sporadic servers* to enforce temporal isolation, which enforce the property that $\frac{b}{p}$ cannot be exceeded for all possible p. In theory, sporadic servers provide temporal isolation – preventing threads from exceeding their allocated budget – by using the following algorithm:

- When a thread starts executing at current time T, record T_s
- When a thread stops executing (blocks or is preempted), schedule a replenishment at $T_s + p$ for the amount of time consumed $(T - T_s)$ and subtract that from the current replenishment being used.

seL4 implements this algorithm by maintaining an ordered list of sporadic replenishments – `refills` for brevity – in each scheduling context. Each replenishment contains a tuple of the time it is eligible for use (*rTime*) and the amount that replenishment is for (`rAmount`). While a thread is executing, it constantly drains the budget from the `rAmount` at the head of the replenishment list. If the `rTime` is in the future, the thread bound to that scheduling context is placed in a queue of threads waiting for more budget.

Round-robin threads are treated that same as sporadic threads except for one case: how the budget is charged. Round-robin threads have two refills only, both of which are always ready to be used. When a round-robin thread stops executing, budget is moved from the head to the tail replenishment. Once the head budget is consumed, the thread is removed from the scheduling queue for its priority and appended at the tail.

Sporadic threads behave differently depending on the amount of replenishments available, which must be bounded. Developers have two options to configure the size of the replenishment list:

- The maximum number of refills in a single scheduling context is determined by the size of the scheduling context when created by `seL4_Untyped_Retype()`.
- A per scheduling context parameter, `extra_refills` that limits the number of refills for that specific scheduling context. This value is added to the base value of 2 and is limited by the size of the scheduling context.

Threads that have short exection times (e.g interrupt handlers) and are not frequently preempted should have less refills, while longer running threads with long values of b should have a higher value. Threads bound to a scheduling context with 0 extra refills will behave periodically – tasks that use their head replenishment, or call yield, will not be scheduled again until the start of their next period.

Given the number of replenishments is limited, if a node's SC changes and the outgoing SC does not have enough space to store the new replenishment, space is created by removing the current replenishment which can result in preemption if the next replenishment is not yet available. Scheduling contexts with a higher number of configured refills will consume closer to their whole budget, as they can be preempted or switch threads more often without filling their replenishment queue. However, the scheduling overhead will be higher as the replenishment list is subject to fragmentation.

Whenever a thread is executing it consumes the budget from its current scheduling context. The system call `seL4_Yield()` can be used to sacrifice any remaining budget and block until the next replenishment is ready to be used.

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.

6.1.5 Passive Threads

Threads can be unbound from a scheduling context with `seL4_SchedContext_UnbindObject()`. 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 appropriate SchedControl capability, which provides access to CPU time management on a single node. A scheduling control cap for each node is provided to the initial task at run time. Threads run on the node that their scheduling context is configured for. 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 offline, statically or dynamically or not at all.

6.1.7 Scheduling Context Donation

In addition to explicitly 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_NBSendRecv()` are used to communicate with a passive thread. When an active thread invokes an endpoint with `seL4_Call()` and rendezvous with a passive thread, the active thread's scheduling context is donated to the passive thread. 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. If the receiver does not provide a reply object to track the donation in (i.e uses `seL4_Wait()` instead of `seL4_Recv()` scheduling context donation will not occur but the message will be delivered. The passive receiver will be set to inactive as it does not have a scheduling context.

Consider an example where thread A calls thread B which calls thread C. If whilst 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_NBSendRecv()` can also result in scheduling context donation. If the non-blocking send phase of the operation results in message delivery to a passive thread, the scheduling context will be donated to that passive thread and the thread making the system call becomes passive on the receiving endpoint in the receive phase. No reply capability generated, so there is no guarantee that the scheduling context will return, which increases book keeping complexity but allows for data-flow like architectures rather than remote-procedure calls. Note that `seL4_Call()` does not guarantee the return of a scheduling context: this is an inherently trusted operation as the server could never reply and return the scheduling context.

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 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()`.

Passive threads will run on the CPU node that the scheduling context was configured with, and will be migrated on IPC.

6.1.8 Scheduling algorithm

seL4 uses a preemptive, tickless, scheduler with 256 priority levels (0 — 255). All threads have a maximum controlled priority (MCP) and a priority, the latter being the effective priority of the thread. When a thread modifies a another threads priority (including itself) it must provide a thread capability from which to use the MCP from. Threads can only set priorities and MCPs to be less than or equal to the provided thread's MCP. The initial task starts with an MCP and priority as the highest priority in the system (`seL4_MaxPrio`). Thread priority and MCP can be set with `seL4_TCB_SetSchedParams()` and `seL4_TCB_SetPriority()`, `seL4_TCB_SetMCPriority()` methods.

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_PrioProps_t`) consists of two 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.

Threads of sufficient maximum controlled priority and with possession of the appropriate scheduling context capability can manipulate the scheduler and implement user-level schedulers using IPC.

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

6.1.9 Exceptions

Each thread has two associated exception-handler endpoints, a *standard* exception handler and a *timeout* exception handler. If the thread causes an exception, the kernel creates an IPC message with the relevant details and sends this to the endpoint. This thread can then take the appropriate action. Fault IPC messages are described in Section 6.2. Standard exception-handler endpoints can be set with the `seL4_TCB_SetSpace()` or `seL4_TCB_SetSchedParams()` methods while Timeout exception handlers an be set with `seL4_TCB_SetTimeoutHandler()`. 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 are different per exeception 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.

6.1.9.1 Standard Exceptions

The standard exception handler is used when a fault is triggered by a thread which cannot be recovered 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.

6.1.9.2 Timeout Exceptions

Timeout faults are raised when a thread attempts to run but has no available budget, and if that thread has a valid timeout exception handler capability. The handling of timeout faults is not compulsory: if a thread does not have a timeout 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.

Timeout faults are registered per thread, which means that while clients may not have a timeout fault handler, servers may, allowing single-threaded, time-sensitive, passive servers to use a timeout exception handler to recover from malicious or untrusted clients whose budget expires while the server is completing the request. Timeout fault handlers can access server reply objcets reply with an error to the client, then reset 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 timeout fault will be generated if the nested server also has a timeout 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 timeout exception will be raised.

6.1.10 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. For some registers, the kernel will silently mask certain bits or ranges of bits off, and force them to contain certain values to ensure that they cannot be maliciously set to values that would compromise the running system, or to respect values that the architecture specifications have mandated to be certain values. The register contents are transferred via the IPC buffer.

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_Fault_CapFault, seL4_Fault_VMFault, seL4_Fault_UnknownSyscall, seL4_Fault_UserException, seL4_Fault_DebugException, seL4_Fault_TimeoutFault, or seL4_Fault_NullFault (indicating no fault occured and this is a normal IPC message).

Faults are delivered in such a way as to imitate a Call from the faulting thread. This means that to send a fault message the fault endpoint must have both write and grant permissions.

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	`seL4_CapFault_IP`
Capability address	`seL4_CapFault_Addr`
In receive phase (1 if the fault happened during a receive system call, 0 otherwise)	`seL4_CapFault_InRecvPhase`
Lookup failure description. As described in Section 3.4	`seL4_CapFault_LookupFailureType`

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. In this case, the number of registers updated is controlled with the length field of the message tag.

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. In this case, the number of registers updated is controlled with the length field of the message tag.

6.2.4 Debug Exception: Breakpoints and Watchpoints

Debug exceptions are used to deliver trace and debug related events to threads. Breakpoints, watchpoints, trace-events and instruction-performance sampling events are examples. These events are supported for userspace threads when the kernel is configured to include them (when CONFIG_HARDWARE_DEBUG_API is set). The hardware debugging extensions API is supported on the following subset of the platforms that the kernel has been ported to:

- PC99: IA-32 and x86_64
- Sabrelite (i.MX6)
- Jetson TegraK1

- HiSilicon Hikey
- Raspberry Pi 3
- Odroid-X (Exynos4)
- Xilinx zynq7000

Information on the available hardware debugging resources is presented in the form of the following constants:

seL4_NumHWBreakpoints : Defines the total number of hardware break registers available, of all types available on the hardware platform. On the ARM Cortex A7 for example, there are 6 exclusive instruction breakpoint registers, and 4 exclusive data watchpoint registers, for a total of 10 monitor registers. On this platform therefore, `seL4_NumHWBreakpoints` is defined as 10. The instruction breakpoint registers will always be assigned the lower API-IDs, and the data watchpoints will always be assigned following them.

Additionally, `seL4_NumExclusiveBreakpoints`, `seL4_NumExclusiveWatchpoints` and `seL4_NumDualFunctionMonitors` are defined for each target platform to reflect the number of available hardware breakpoints/watchpoints of a certain type.

seL4_NumExclusiveBreakpoints : Defines the number of hardware registers capable of generating a fault **only** on instruction execution. Currently this will be set only on ARM platforms. The API-ID of the first exclusive breakpoint is given in `seL4_FirstBreakpoint`. If there are no instruction-break exclusive registers, `seL4_NumExclusiveBreakpoints` will be set to `0` and `seL4_FirstBreakpoint` will be set to -1.

seL4_NumExclusiveWatchpoints : Defines the number of hardware registers capable of generating a fault **only** on data access. Currently this will be set only on ARM platforms. The API-ID of the first exclusive watchpoint is given in `seL4_FirstWatchpoint`. If there are no data-break exclusive registers, `seL4_NumExclusiveWatchpoints` will be set to `0` and `seL4_FirstWatchpoint` will be set to -1.

seL4_NumDualFunctionMonitors : Defines the number of hardware registers capable of generating a fault on either type of access – i.e, the register supports both instruction and data breaks. Currently this will be set only on x86 platforms. The API-ID of the first dual-function monitor is given in `seL4_FirstDualFunctionMonitor`. If there are no dual-function break registers, `seL4_NumDualFunctionMonitors` will be set to `0` and `seL4_FirstDualFunctionMonitor` will be set to -1.

6.2.5 Debug Exception: Single-stepping

The kernel provides support for the use of hardware single-stepping of userspace threads when configured to do so (when CONFIG_HARDWARE_DEBUG_API is set). To this end it exposes the invocation, `seL4_TCB_ConfigureSingleStepping`.

Value sent	IPC buffer location
`Breakpoint instruction address`	`IPCBuffer[0]`
`Exception reason`	`IPCBuffer[1]`
`Watchpoint data access address`	`IPCBuffer[2]`
`Register API-ID`	`IPCBuffer[3]`

Table 6.2: Debug fault message layout. The register API-ID is not returned in the fault message from the kernel on single-step faults.

The caller is expected to select an API-ID that corresponds to an instruction breakpoint, to use when setting up the single-stepping functionality (i.e, API-ID from 0 to `seL4_NumExclusiveBreakpoints` - 1). However, not all hardware platforms require an actual hardware breakpoint register to provide single-stepping functionality. If the caller's hardware platform requires the use of a hardware breakpoint register, it will use the breakpoint register given to it in `bp_num`, and return `true` in `bp_was_consumed`. If the underlying platform does not need a hardware breakpoint to provide single-stepping, seL4 will return `false` in `bp_was_consumed` and leave `bp_num` unchanged.

If `bp_was_consumed` is `true`, the caller should not attempt to re-configure `bp_num` for Breakpoint or Watchpoint usage until the caller has disabled single-stepping and released that register, via a subsequent call to `seL4_TCB_ConfigureSingleStepping`, or a fault-reply with `n_instr` being 0. Setting `num_instructions` to `0` **disables single stepping**.

On architectures that require an actual hardware registers to be configured for single-stepping functionality, seL4 will restrict the number of registers that can be configured as single-steppers, to one at any given time. The register that is currently configured (if any) for single-stepping will be the implicit `bp_num` argument in a single-step debug fault reply.

The kernel's single-stepping, also supports skipping a certain number of instructions before delivering the single-step fault message. `Num_instructions` should be set to `1` when single-stepping, or any non-zero integer value to skip that many instructions before resuming single-stepping. This skip-count can also be set in the fault-reply to a single-step debug fault.

Value sent	Register set by reply	IPC buffer location
`Breakpoint instruction address`	`num_instructions` to skip	`IPCBuffer[0]`
`Exception reason`	—	`IPCBuffer[1]`

Table 6.3: Single-step fault message layout.

6.2.6 Timeout Fault

Timeout faults are raised when a thread consumes all of its budget and has a timeout fault handler that is not a null capability. They allow a timeout exception handler to take some action to restore the thread, and deliver a message containting the schedluing context data word, as well as the amount of time consumed since the last timeout fault occured on this scheduling context, or since `seL4_SchedContext_YieldTo()` or `seL4_SchedContext_Consumed()` was last called. Timeout exception handlers can reply to a temporal fault with the registers set in the same format as in outlined in Section 6.1.10.

Meaning	IPC buffer location
Data word from the scheduling context object that the thread was running on when the fault occured.	`seL4_TimeoutFault_Data`
Upper 32-bits of microseconds consumed since last reset	`seL4_TimeoutFault_Consumed`
Lower 32-bits of microseconds consumed since last reset	`seL4_TimeoutFault_Consumed_LowBits`

Table 6.4: Timeout fault outcome on 32-bit architectures.

6.2.7 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.	`seL4_VMFault_IP`
Address that caused the fault.	`seL4_VMFault_SP`
Instruction fault (1 if the fault was caused by an instruction fetch).	`seL4_VMFault_PrefetchFault`
Fault status register (FSR). Contains information about the cause of the fault. Architecture dependent.	`seL4_VMFault_FSR`

Table 6.5: VM Fault outcome on all architectures.

6.3 Domains

Domains are used to isolate independent subsystems, so as to limit information flow between them. The kernel switches between domains according to a fixed, time-triggered schedule. The fixed schedule is compiled into the kernel via the constant `CONFIG_NUM_DOMAINS` and the global variable `ksDomSchedule`.

A thread belongs to exactly one domain, and will only run when that domain is active. The `seL4_DomainSet_Set()` method changes the domain of a thread. The caller must possess a Domain cap and the thread's TCB cap. The initial thread starts with a Domain cap (see Section 4.1).

6.4 Virtualisation

Hardware execution virtualisation is supported on specific arm and x86 platforms. The interface is exposed through a series of kernel objects, invocations and syscalls that allow the user to take advantage of hardware virtualisation features.

Hardware virtualisation allows for a thread to perform instructions and operations as if it were running at a higher privilege level. As higher privilege levels typically have access to additional machine registers and other pieces of state a VCPU object is introduced to act as storage for this state. For simplicity we refer to this virtualised higher privileged level as 'guest mode'. VCPUs are bound in a one-to-one relationship with a TCB in order to provide a thread with this ability to run in higher privilege mode. See the section on ARM or x86 for more precise details.

VCPU objects also have additional, architecture specific, invocations for manipulating the additional state or other virtualisation controls provided by the hardware. Binding of a VCPU to a TCB is done by an invocation on the VCPU only, and not the TCB.

The provided objects and invocations are, generally speaking, the thinnest possible shim over the underlying hardware primitives and operations. As a result an in depth familiarity with the underlying architecture specific harwdare mechanisms is required to use these objects, and such familiarity is therefore assumed in description.

6.4.1 ARM

When a TCB has a bound VCPU it is allowed to have the mode portion of the `cpsr` register set to values other than `user`. Specifically it may have any value other than `hypervisor`.

TODO: this section needs more detail

6.4.2 x86

A TCB with a bound VCPU has two execution modes; one is the original thread just as if there was no bound VCPU, and the other is the guest mode execution using the VCPU. Switching from regular execution mode into the guest execution mode is done by using the `seL4_VMEnter()` syscall. Executing this syscall causes the thread, whenever it is scheduled thereafter, to execute using the higher privlidged mode controlled by the VCPU. Should the guest execution mode generate any kind of fault, or if a message arrives on the TCBs bound notification, the TCB will be switched back to regular mode and the `seL4_VMEnter()` syscall will return with a message indicating the reason for return.

VCPU state and execution is controlled through the `seL4_VCPU_ReadVMCS()` and `seL4_VCPU_WriteVMCS()` invocations. These are very thin wrappers around the hardware `vmread` and `vmwrite` instructions and the kernel merely does enough validation on the parameters to ensure the VCPU is not configured to run in such a way as to violate any kernel properties. For example, it is not possible to disable the use of External Interrupt Exiting, as this would prevent the kernel from receiving timer interrupts and allow the thread to monopolise CPU time.

Memory access of the guest execution mode is controlled by requiring the use of Extended Page Tables (EPT). A series of EPT related paging structure objects (EPTPML4, EPTPDPT, EPTPD, EPTPT) exist and are manipulated in exactly the same manner as the objects for the regular virtual address space. Once constructed a TCB can be given an EPTPML4 as an EPT root with `seL4_TCB_SetEPTRoot()`, which serves as the vspace root when executing in guest mode, with the vspace root set with `seL4_TCB_SetSPace()` or `seL4_TCB_Configure()` continuing to provide translation when the TCB is executing in its normal mode.

Direct access to I/O ports can be given to the privlidged execution mode through the `seL4_X86_VCPU_EnableIOPort()` invocation and allows the provided I/O port capability to be linked to the VCPU, and a subset of its I/O port range to be made accessible to the VCPU. Linking means that an I/O port capability can only be used in a single `seL4_X86_VCPU_EnableIOPort()` invocation and a second invocation will undo the previous one. The link also means that if the I/O port capability is deleted for any reason the access will be correspondingly removed from the VCPU.

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

7.1.0.0.1 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.

7.1.0.0.2 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

7.2.0.0.1 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). On x86, the Page Directory has no invocations itself, but it is used as an argument to several other virtual-memory related object invocations.

On ARM, the highest level page directory (Page Directory on aarch32 and global page directory on aarch64) objects possess a few invocations for the user to do cache maintenance operations:

`seL4_ARM_PageDirectory_Clean_Data()`

`seL4_ARM_PageDirectory_Invalidate_Data()`

`seL4_ARM_PageDirectory_CleanInvalidate_Data()`

`seL4_ARM_PageDirectory_Unify_Instruction()`

`seL4_ARM_PageGlobalDirectory_Clean_Data()`

`seL4_ARM_PageGlobalDirectory_Invalidate_Data()`

`seL4_ARM_PageGlobalDirectory_CleanInvalidate_Data()`

`seL4_ARM_PageGlobalDirectory_Unify_Instruction()`

By making these cache related operations invocations on page directory capabilities in addition to the page capabilities themselves, the API allows users more flexible policy options. For example, a process that has delegated a page directory can conduct cache operations on all frames mapped from that capability without access to those capabilities directly.

7.2.0.0.2 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_x86_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_x86_PageTable_Unmap()`

> Removes the reference to the invoked Page Table from its containing Page Directory.

7.2.0.0.3 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_x86_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_x86_Page_Remap()`

Changes the permissions of an existing mapping.

`seL4_ARM_Page_Unmap()`
`seL4_x86_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_x86_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.

7.2.0.0.4 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_x86_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.

7.2.0.0.5 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_x86_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_x86_VMAttributes` is used to specify the cache behaviour of the page being mapped; possible values for ARM that can be bitwise OR'd together are shown in Table 7.1 and an enumeration of valid 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_x86_WriteBack`	Read and writes are cached
`seL4_x86_CacheDisabled`	Prevent data in this mapping from being cached
`seL4_x86_WriteThrough`	Enable write through cacheing for this mapping
`seL4_x86_WriteCombining`	Enable write combining for this mapping

Table 7.2: Virtual memory attributes for x86 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_x86_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 6.2.7.

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.

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. Methods on IRQControl can be used for creating IRQHandler capabilities for interrupt sources.

8.2 x86-Specific I/O

8.2.1 Interrupts

In addition to managing IRQHandler capabilities, x86 platforms require the delivery location in the CPU vectors to be configured. Regardless of where an interrupt comes from (IOAPIC, MSI, etc) it must be assigned a unique vector for delivery, ranging from VECTOR_MIN to VECTOR_MAX. The rights to allocate a vector are effectively given through the IRQControl capability and can be considered as the kernel outsourcing the allocation of this namespace to user level.

`seL4_IRQControl_GetIOAPIC()` creates an IRQHandler capability for an IOAPIC interrupt

`seL4_IRQControl_GetMSI()` creates an IRQHandler capability for an MSI interrupt

8.2.2 I/O Ports

On x86 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_X86_IOPort_In8()`, `seL4_X86_IOPort_In16()`, and `seL4_X86_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_X86_IOPort_Out8()`, `seL4_X86_IOPort_Out16()`, and `seL4_X86_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.

8.2.3 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 x86-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].

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_X86_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_X86_Page_MapIO()` method whose parameters are analogous to the corresponding method that maps Pages into VSpaces (see Chapter 7), namely `seL4_X86_Page_Map()`.

Unmapping is accomplished with the usual unmap (see Chapter 7) API call, `seL4_X86_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, which is started at priority `seL4_MaxPrio` and maximum control priority `seL4_MaxPrio`. 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 starts the initial thread with a round-robin scheduling context with `CONFIG_BOOT_THREAD_TIME_SLICE` milliseconds timeslice. 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. On ARM this register is `r0`, on IA32 it is `ebx` and on x86-64 `rdi`

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.

Enum Constant	Capability
`seL4_CapNull`	null
`seL4_CapInitThreadTCB`	initial thread's TCB
`seL4_CapInitThreadCNode`	initial thread's CNode
`seL4_CapInitThreadVSpace`	initial thread's VSpace
`seL4_CapIRQControl`	global IRQ controller (see Section 8.1)
`seL4_CapASIDControl`	global ASID controller (see Chapter 7)
`seL4_CapInitThreadASIDPool`	initial thread's ASID pool (see Chapter 7)
`seL4_CapIOPort`	global I/O port cap, null cap if unsupported (see Section 8.2.2)
`seL4_CapIOSpace`	global I/O space cap, null cap if unsupported (see Section 8.2.3)
`seL4_CapBootInfoFrame`	BootInfo frame (see Section 9.2)
`seL4_CapInitThreadIPCBuffer`	initial thread's IPC buffer (see Section 4.1)
`seL4_CapDomain`	domain cap (see Section 6.3)
`seL4_CapInitThreadSC`	initial thread's scheduling context

Depending on the architecture and platform there might be additional pieces of boot information. If `extraLen` is greater then zero then 4K after the start of bootinfo is a region of extraLen bytes containing additional bootinfo structures. Each chunk starts with a `seL4_BootInfoHeader`, described in Table 9.3, that describes what the chunk is and how long it is, where the length includes the header. The length can be used to skip over chunks that you do not understand. The only generally defined chunk type is `SEL4_BOOTINFO_HEADER_PADDING` and describes an empty chunk that has no data, any other types are platform or architecture specific. The `extraBIPages` slot region gives the frames capabilities for the pages that make up the additional boot info region.

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.

Table 9.2: BootInfo struct.

Field Type	Field Name	Description
`seL4_Word`	`extraLen`	length of additional bootinfo information in bytes
`seL4_Word`	`nodeID`	node ID
`seL4_Word`	`numNodes`	number of nodes
`seL4_Word`	`numIOPTLevels`	number of I/O page-table levels (-1 if CONFIG_IOMMU unset)
`seL4_IPCBuffer*`	`ipcBuffer`	pointer to the initial thread's IPC buffer
`seL4_SlotRegion`	`empty`	empty slots (null caps)
`seL4_SlotRegion`	`sharedFrames`	reserved
`seL4_SlotRegion`	`userImageFrames`	frames containing the userland image
`seL4_SlotRegion`	`userImagePaging`	userland-image paging structure caps
`seL4_SlotRegion`	`ioSpaceCaps`	I/O space capabilities for ARM SMMU
`seL4_SlotRegion`	`extraBIPages`	frames backing additional bootinfo information
`seL4_UntypedDesc[]`	`untypedList`	array of information about each untyped
`seL4_Uint8`	`initThreadCNodeSizeBits`	CNode size (2^n slots)
`seL4_Word`	`initThreadDomain`	domain of the initial thread (see Section 6.3)
`seL4_SlotRegion`	`untyped`	untyped-memory capabilities
`seL4_SlotRegion`	`untyped`	seL4_SchedControl capabilities, one for each node.

Untyped memory is given in no particular order. The array entry `untypedList[i]` stores the untyped-memory information of the i-th untyped cap of the slot region `untyped`. Therefore, the array length is at least `untyped.end - untyped.start`. The actual length is hardcoded in the kernel and irrelevant to the reader of the array. The untyped memory information is stored in a `seL4_UntypedDesc` struct, described in Table 9.4, and details the address, size and kind of the memory backing the untyped. 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.

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`.

On ARM if the platform has any available SMMU units the capabilities for them will

Table 9.3: BootInfoHeader struct.

Field Type	Field Name	Description
`seL4_Word`	`id`	Identifier indicating the contents of the chunk
`seL4_Uint8`	`len`	Length in bytes of the chunk

Table 9.4: seL4_UntypedDesc struct

Field Type	Field Name	Description
`seL4_Word`	`paddr`	physical base address of the untyped object
`seL4_Uint8`	`padding1`	manual padding so final struct is a multiple of the word size
`seL4_Uint8`	`padding2`	manual padding so final struct is a multiple of the word size
`seL4_Uint8`	`sizeBits`	size (2^n bytes) of the untyped object
`seL4_Uint8`	`isDevice`	is this untyped a device or not (see Section 2.4)

be described by the `ioSpaceCaps` slot region. The mapping of a capability from this region to a specific SMMU is platform specific.

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 9.5 along with their default values (if left unspecified).

Table 9.5: 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

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 General System Calls

10.2.1.1 Send

`LIBSEL4_INLINE_FUNC 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

10.2.1.2 Recv

`LIBSEL4_INLINE_FUNC 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`.
`seL4_CPtr`	`reply`	The capability to the reply object to use on a call.

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

Description: See Section 2.2

10.2.1.3 Call

`LIBSEL4_INLINE_FUNC 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

10.2.1.4 Polling Send

`LIBSEL4_INLINE_FUNC 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

10.2.1.5 Polling Recv

`LIBSEL4_INLINE_FUNC seL4_MessageInfo_t seL4_NBRecv`

Poll on an endpoint, and receive a message if one is present.

Type	Name	Description
`seL4_CPtr`	`src`	The capability to receive on.
`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`.
`seL4_CPtr`	`reply`	The capability to the reply object to use on a call.

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

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

See Section 2.2

See Section 2.2

10.2.1.6 Reply Recv

`LIBSEL4_INLINE_FUNC seL4_MessageInfo_t seL4_ReplyRecv`

Perform a reply followed by a receive in one system call.

Type	Name	Description
`seL4_CPtr`	`src`	The capability to perform the receive on.
`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`.
`seL4_CPtr`	`reply`	The capability to the reply object, which is first invoked and then used for the recv phase to store a new reply capability.

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

Description: See Section 2.2

10.2.1.7 NBSend Recv

`LIBSEL4_INLINE_FUNC seL4_MessageInfo_t seL4_NBSendRecv`

Non-blocking send on one capability, and a blocking recieve on another in a single system call.

Type	Name	Description
`seL4_CPtr`	`dest`	The capability to be invoked.
`seL4_MessageInfo_t`	`msgInfo`	The messageinfo structure for the IPC.
`seL4_CPtr`	`src`	The capability to receive on.
`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`.
`seL4_CPtr`	`reply`	The capability to the reply object, which is first invoked and then used for the recv phase to store a new reply capability.

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

Description: See Section 2.2

10.2.1.8 NBSend Wait

`LIBSEL4_INLINE_FUNC seL4_MessageInfo_t seL4_NBSendWait`

Non-blocking invoke of a capability and wait on another 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_CPtr`	`src`	The capability to receive on.
`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

10.2.1.9 Yield

`LIBSEL4_INLINE_FUNC void seL4_Yield`

Yield the remaining timeslice. Periodic threads will not be scheduled again until their next sporadic replenishment.

Type	Name	Description
void		

Return value: This method does not return anything.

Description: See Section 2.2

10.2.1.10 Wait

`LIBSEL4_INLINE_FUNC seL4_MessageInfo_t seL4_Wait`

Perform a wait on an endpoint or 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: TODO

Description: Block on a notification or endpoint waiting for a message. No reply object is required for a Wait. Wait should not be paired with Call, as it does not provide a reply object. If Wait is paired with a Call the waiter will block after recieving the message.

See the description of `seL4_Wait()` in Section 2.2.

10.2.1.11 NBWait

`LIBSEL4_INLINE_FUNC seL4_MessageInfo_t seL4_NBWait`

Perform a polling wait on an endpoint or 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: TODO

Description: Poll a notification or endpoint waiting for a message. No reply object is required for a Wait. Wait should not be paired with Call.

See the description of `seL4_NBWait()` in Section 2.2.

10.2.1.12 Poll

`LIBSEL4_INLINE_FUNC seL4_MessageInfo_t 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: A `seL4_MessageInfo_t` structure as described in Section 4.1

Description: This is not a proper system call known by the kernel. Rather, it is a convenience wrapper which calls `seL4_NBWait()`. It is useful for doing a non-blocking wait on a notification.

See the description of `seL4_NBWait()` in Section 2.2.

10.2.1.13 Signal

`LIBSEL4_INLINE_FUNC 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 which calls `seL4_Send()`. It is useful for signalling a notification.

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

10.2.2 Debugging System Calls

This section documents debugging system calls available when the kernel is build with the `DEBUG_BUILD` configuration. For any system calls that rely on a kernel serial driver, `PRINTING` must also be enabled.

10.2.2.1 Put Char

`LIBSEL4_INLINE_FUNC void seL4_DebugPutChar`

Output a single char through the kernel.

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

Return value: This method does not return anything.

Description: Use the kernel serial driver to output a single character. This is useful for debugging when a user level serial driver is not available.

10.2.2.2 Dump scheduler

`LIBSEL4_INLINE_FUNC void seL4_DebugDumpScheduler`

Output the contents of the kernel scheduler.

Type	Name	Description
`void`		

Return value: This method does not return anything.

Description: Dump the state of the all TCB objects to kernel serial output. This system call will output a table containing:

- Address: the address of the TCB object for that thread,
- Name: the name of the thread (if set),
- IP: the contents of the instruction pointer the thread is at,
- Priority: the priority of that thread,
- State : the state of the thread.

10.2.2.3 Halt

`LIBSEL4_INLINE_FUNC void seL4_DebugHalt`

Halt the system.

Type	Name	Description
`void`		

Return value: This method does not return anything.

Description: This debugging system call will cause the kernel immediately cease responding to system calls. The kernel will switch permanently to the idle thread with interrupts disabled. Depending on the platform, the kernel may switch the hardware into a low-power state.

10.2.2.4 Snapshot

`LIBSEL4_INLINE_FUNC void seL4_DebugSnapshot`

Output a capDL dump of the current kernel state.

Type	Name	Description
`void`		

Return value: This method does not return anything.

Description: This debugging system call will output all of the capabilities in the current kernel using capDL.

10.2.2.5 Cap Identify

`LIBSEL4_INLINE_FUNC seL4_Uint32 seL4_DebugCapIdentify`

Identify the type of a capability in the current cspace.

Type	Name	Description
`seL4_CPtr`	`cap`	A capability slot in the current cspace.

Return value: The type of capability passed in.

Description: This debugging system call returns the type of capability in a capability slot in the current cspace. The type returned is not a libsel4 type, but refers to an internal seL4 type. This can be looked up in a built kernel by looking for the (generated) `enum cap_tag`, type `cap_tag_t`.

10.2.2.6 Name Thread

`LIBSEL4_INLINE_FUNC void seL4_DebugNameThread`

Name a thread.

Type	Name	Description
`seL4_CPtr`	`tcb`	A capability to the tcb object for the thread to name.
`const char *`	`name`	The name for the thread.

Return value: This method does not return anything.

Description: Name a thread. This name will then be output by the kernel in all debugging output. Note that the max name length that can be passed to this function is limited by the number of chars that will fit in an IPC message (`seL4_MsgMaxLength` multiplied by the amount of chars that fit in a word). However the name is also truncated in order to fit into a TCB object. For some platforms you may need to increase `seL4_TCBBits` by 1 in a debug build in order to fit a long enough name.

10.2.2.7 Run

`LIBSEL4_INLINE_FUNC void seL4_DebugRun`

Run a user level function in kernel mode.

Type	Name	Description
`void(*)(void *)`	`userfn`	The address in userspace of the function to run.
`void *`	`userarg`	A single argument to pass to the function.

Return value: This method does not return anything.

Description: This extremely dangerous function is for running benchmarking and debugging code that needs to be executed in kernel mode from userlevel. It should never be used in a release kernel. This works because the kernel can access all user mappings of device memory, and does not switch page directories on kernel entry.

Unlike the other system calls in this section, `seL4_DebugRun` does not depend on the `DEBUG_BUILD` configuration option, but its own config variable `DANGEROUS_CODE_INJECTION`.

10.2.3 Benchmarking System Calls

This section documents system calls available when the kernel is configured with benchmarking enabled. There are several different benchmarking modes which can be configured when building the kernel:

1. `BENCHMARK_TRACEPOINTS`: Enable using tracepoints in the kernel and timing code.
2. `BENCHMARK_TRACK_KERNEL_ENTRIES`: Keep track of information on kernel entries.
3. `BENCHMARK_TRACK_UTILISATION`: Allow users to get CPU timing info for the system, threads and/or idle thread.

10.2.3.1 Reset Log

`LIBSEL4_INLINE_FUNC seL4_Error seL4_BenchmarkResetLog`

Reset benchmark logging.

Type	Name	Description
`void`		

Return value: A `seL4_Error` error if the user-level log buffer has not been set by the user (`BENCHMARK_TRACEPOINTS`/`BENCHMARK_TRACK_KERNEL_ENTRIES`).

Description: The behaviour of this system call depends on benchmarking mode in action while invoking this system call:

1. `BENCHMARK_TRACEPOINTS`: resets the log index to 0,
2. `BENCHMARK_TRACK_KERNEL_ENTRIES`: as above,
3. `BENCHMARK_TRACK_UTILISATION`: resets benchmark and current thread start time (to the time of invoking this syscall), resets idle thread utilisation to 0, and starts tracking utilisation.

10.2.3.2 Finalize Log

`LIBSEL4_INLINE_FUNC seL4_Word seL4_BenchmarkFinalizeLog`

Stop benchmark logging.

Type	Name	Description
`void`		

Return value: The index of the final entry in the log buffer (if `BENCHMARK_TRACEPOINTS`/`BENCHMARK_TRACK_KERNEL_ENTRIES` are enabled).

Description: The behaviour of this system call depends on benchmarking mode in action while invoking this system call:

1. `BENCHMARK_TRACEPOINTS`: Sets the final log buffer index to the current index,
2. `BENCHMARK_TRACK_KERNEL_ENTRIES`: as above,
3. `BENCHMARK_TRACK_UTILISATION`: sets benchmark end time to current time, stops tracking utilisation.

10.2.3.3 Set Log Buffer

`LIBSEL4_INLINE_FUNC seL4_Error seL4_BenchmarkSetLogBuffer`

Set log buffer.

Type	Name	Description
`seL4_Word`	`frame_cptr`	A capability pointer to a user allocated frame of seL4_LargePage size.

Return value: A `seL4_IllegalOperation` error if `frame_cptr` is not valid and couldn't set the buffer.

Description: Provide a large frame object for the kernel to use as a log-buffer. The object must not be device memory, and must be seL4_LargePageBits in size.

10.2.3.4 Null Syscall

`LIBSEL4_INLINE_FUNC void seL4_BenchmarkNullSyscall`

Null system call that enters and exits the kernel immediately, for timing kernel traps in microbenchmarks.

Type	Name	Description
`void`		

Return value: This method does not return anything.

Description: Used to time kernel traps (in and out).

10.2.3.5 Flush Caches

`LIBSEL4_INLINE_FUNC void seL4_BenchmarkFlushCaches`

Flush hardware caches.

Type	Name	Description
`void`		

Return value: This method does not return anything.

Description: Flush all possible hardware caches for this platform.

10.2.3.6 Get Thread Utilisation

`LIBSEL4_INLINE_FUNC void seL4_BenchmarkGetThreadUtilisation`

Get utilisation timing information.

Type	Name	Description
`seL4_Word`	`tcb_cptr`	TCB cap pointer to a thread to get CPU utilisation for.

Return value: This method does not return anything.

Description: Get timing information for the system, requested thread and idle thread. Such information is written into the caller's IPC buffer; see the definition of `benchmark_track_util_ipc_index` enum for more details on the data/format returned on the IPC buffer.

10.2.3.7 Reset Thread Utilisation

`LIBSEL4_INLINE_FUNC void seL4_BenchmarkResetThreadUtilisation`

Reset utilisation timing for a specific thread.

Type	Name	Description
`seL4_Word`	`tcb_cptr`	TCB cap pointer to a thread to get CPU utilisation for.

Return value: This method does not return anything.

Description: Reset the kernel's timing information data (start time and utilisation) for a specific thread.

10.2.4 X86 System Calls

10.2.4.1 VMEnter

`LIBSEL4_INLINE_FUNC seL4_Word seL4_VMEnter`

Change current thread to execute from its bound VCPU.

Type	Name	Description
`seL4_Word *`	`sender`	The address to write sender information to. If the syscall returns due to receiving a notification on the bound notification then the sender information is the badge of the notification capability that was invoked. This parameter is ignored if `NULL`.

Return value: `SEL4_VMENTER_RESULT_NOTIF` if a notification was received or `SEL4_VMENTER_RESULT_FAULT` if the guest mode execution faulted for any reason

Description: Changes the execution mode of the current thread from normal TCB execution, to guest execution using its bound VCPU. For details on VCPUs and execution modes see Section 6.4.

Invoking `seL4_VMEnter` is similar to replying to a fault in that updates to the registers can be given in the message, but unlike a fault no message info (see Section 4.1) is sent as the registers are not optional and the number that must be sent is fixed. The mapping of hardware register to message register is

- `SEL4_VMENTER_CALL_EIP_MR` Address to start executing instructions at in the guest mode
- `SEL4_VMENTER_CALL_CONTROL_PPC_MR` New value for the Primary Processor Based VM Execution Controls
- `SEL4_VMENTER_CALL_CONTROL_ENTRY_MR` New value for the VM Entry Controls

On return these same three message registers will be filled with the values at the point that the privlidged mode ceased executing. If this function returns with `SEL4_VMENTER_RESULT_FAULT` then the following additional message registers will be filled out

- `SEL4_VMENTER_FAULT_REASON_MR`
- `SEL4_VMENTER_FAULT_QUALIFICATION_MR`
- `SEL4_VMENTER_FAULT_INSTRUCTION_LEN_MR`
- `SEL4_VMENTER_FAULT_GUEST_PHYSICAL_MR`
- `SEL4_VMENTER_FAULT_RFLAGS_MR`
- `SEL4_VMENTER_FAULT_GUEST_INT_MR`

- SEL4_VMENTER_FAULT_CR3_MR
- SEL4_VMENTER_FAULT_EAX
- SEL4_VMENTER_FAULT_EBX
- SEL4_VMENTER_FAULT_ECX
- SEL4_VMENTER_FAULT_EDX
- SEL4_VMENTER_FAULT_ESI
- SEL4_VMENTER_FAULT_EDI
- SEL4_VMENTER_FAULT_EBP

10.3 Architecture-Independent Object Methods

10.3.1 CNode - Cancel Badged Sends

`static inline int seL4_CNode_CancelBadgedSends`

The cancel badged sends method is intend to allow for the reuse of badges by an authority. When used with a badged endpoint capability it will cancel any outstanding send operations for that endpoint and badge. This operation has no effect on un-badged or other objects.

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.
`seL4_Uint8`	`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.

10.3.2 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.
`seL4_Uint8`	`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.
`seL4_Uint8`	`src_depth`	Number of bits of src_index to resolve to find the source slot.
`seL4_CapRights_t`	`rights`	The rights inherited by the new capability. Possible values for this type are given in Section 3.1.4 .

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 - 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.
`seL4_Uint8`	`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.

10.3.4 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.
`seL4_Uint8`	`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.
`seL4_Uint8`	`src_depth`	Number of bits of src_index to resolve to find the source slot.
`seL4_CapRights_t`	`rights`	The rights inherited by the new capability. Possible values for this type are given in Section 3.1.4 .
`seL4_Word`	`badge`	Badge or guard to be applied to the new capability. For badges the high 4 bits are ignored.

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 - 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.
`seL4_Uint8`	`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.
`seL4_Uint8`	`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.

10.3.6 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.
`seL4_Uint8`	`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.
`seL4_Uint8`	`src_depth`	Number of bits of src_index to resolve to find the source slot.
`seL4_Word`	`badge`	Badge or guard to be applied to the new capability. For badges the high 4 bits are ignored.

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.
`seL4_Uint8`	`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.
`seL4_Uint8`	`dest_depth`	Depth to resolve dest_index to.
`seL4_Word`	`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.
`seL4_Uint8`	`pivot_depth`	Depth to resolve pivot_index to.
`seL4_Word`	`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.
`seL4_Uint8`	`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 Domain Set - Set

`static inline int seL4_DomainSet_Set`

Change the domain of a thread.

Type	Name	Description
`seL4_DomainSet`	`_service`	Capability allowing domain configuration.
`seL4_Uint8`	`domain`	The thread's new domain.
`seL4_TCB`	`thread`	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 6.3.

10.3.10 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.
`seL4_Uint8`	`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.

10.3.11 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.

10.3.12 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.

10.3.13 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.

10.3.14 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

10.3.15 TCB - Configure Single Stepping

`static inline seL4_TCB_ConfigureSingleStepping_t seL4_TCB_ConfigureSingleStepping`

Set or modify single stepping options for the target TCB. Subsequent calls to this function overwrite previous configuration. Depending on your processor architecture, this may or may not require the consumption of a hardware register.

Type	**Name**	**Description**
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_Uint16`	`bp_num`	The API-ID of a target breakpoint. This ID will be a positive integer, with values ranging from 0 to seL4_NumHWBreakpoints - 1.
`seL4_Word`	`num_instructions`	Number of instructions to step over before delivering a fault to the target thread's fault endpoint. Setting this to 0 disables single-stepping.

Return value: A `seL4_TCB_ConfigureSingleStepping_t`: Struct that contains '`seL4_Error error`', an seL4 API error value, '`seL4_Bool bp_was_consumed`', a boolean which indicates whether or not the `bp_num` breakpoint ID that was passed to the function, was consumed in the setup of the single-stepping functionality: if this is `true`, the caller should not attempt to re-use `bp_num` until it has disabled the single-stepping functionality via a subsequent call to seL4_TCB_ConfigureSingleStepping with an `num_instructions` argument of 0.

Description: See Sections 6.2.5 and 6.2.4

10.3.16 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_CNode`	`cspace_root`	The new CSpace root.
`seL4_Word`	`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_Word`	`vspace_root_data`	Has no effect on x86 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

10.3.17 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.
seL4_Bool	suspend_source	The invocation should also suspend the source thread.
seL4_Bool	resume_target	The invocation should also resume the destination thread.
seL4_Bool	transfer_frame	Frame registers should be transferred.
seL4_Bool	transfer_integer	Integer registers should be transferred.
seL4_Uint8	arch_flags	Architecture dependent flags. These have no mearing on either x86 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

10.3.18 TCB - Get Breakpoint

`static inline seL4_TCB_GetBreakpoint_t seL4_TCB_GetBreakpoint`

Read a breakpoint or watchpoint's current configuration.

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_Uint16`	`bp_num`	The API-ID of a target breakpoint. This ID will be a positive integer, with values ranging from 0 to seL4_NumHWBreakpoints - 1.

Return value: A `seL4_TCB_GetBreakpoint_t`: Struct that contains '`seL4_Error error`', an seL4 API error value, '`seL4_Word vaddr`', the virtual address at which the breakpoint will currently be triggered; '`seL4_Word type`', the type of operation which will currently trigger the breakpoint, whether instruction execution, or data access; '`seL4_Word size`', integer value for the span-size of the breakpoint. Usually a power of two (1, 2, 4, etc.); '`seL4_Word rw`', the access direction that will currently trigger the breakpoint, whether read, write, or both and '`seL4_Bool is_enabled`', which indicates whether or not the breakpoint will currently be triggered if the match conditions are met.

Description: See Section 6.2.4

10.3.19 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.
`seL4_Bool`	`suspend_source`	The invocation should also suspend the source thread.
`seL4_Uint8`	`arch_flags`	Architecture dependent flags. These have no mearing on either x86 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 6.1.10

10.3.20 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 6.1.3

10.3.21 TCB - Set Breakpoint

`static inline int seL4_TCB_SetBreakpoint`

Set or modify a thread's breakpoints or watchpoints. Calls to this function overwrite previous configurations for the target breakpoint. Do not use this with seL4_SingleStep: the API will reject the call and return an error. Instead, use seL4_TCB_ConfigureSingleStepping to configure single-stepping.

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_Uint16`	`bp_num`	The API-ID of a target breakpoint. This ID will be a positive integer, with values ranging from 0 to seL4_NumHWBreakpoints - 1.
`seL4_Word`	`vaddr`	A virtual address which forms part of the match conditions for the triggering of the breakpoint.
`seL4_Word`	`type`	One of: seL4_InstructionBreakpoint, which specifies that the breakpoint should occur on instruction execution at the specified vaddr or seL4_DataBreakpoint, which states that the breakpoint should occur on data access at the specified vaddr.
`seL4_Word`	`size`	A positive integer indicating the trigger-span of the watchpoint. Must be zero when 'type' is seL4_InstructionBreakpoint.
`seL4_Word`	`rw`	One of seL4_BreakOnRead, meaning the breakpoint will only be triggered on read-access; seL4_BreakOnWrite meaning the breakpoint will only be triggered on write-access, and seL4_BreakOnReadWrite meaning the breakpoint will be triggered on any access.

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 6.2.4

10.3.22 TCB - Set CPU Affinity

`static inline int seL4_TCB_SetAffinity`

Change a thread's current CPU in multicore machine

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_Word`	`affinity`	The thread's new CPU to run.

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 6.1.2

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 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.
`seL4_CPtr`	`authority`	Capability to the TCB to use the MCP from when setting the MCP.
`seL4_Word`	`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

10.3.25 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.
`seL4_CPtr`	`authority`	Capability to the TCB to use the MCP from when setting the priority.
`seL4_Word`	`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

10.3.26 TCB - Set Sched Params

`static inline int seL4_TCB_SetSchedParams`

Change a thread's priority, maximum controlled priority, scheduling context and fault handler.

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_CPtr`	`authority`	Capability to the TCB to use the MCP from when setting the priority and MCP.
`seL4_Word`	`mcp`	The thread's new maximum controlled priority.
`seL4_Word`	`priority`	The thread's new priority.
`seL4_CPtr`	`sched_context`	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_CPtr`	`fault_ep`	CPTR to the endpoint which receives IPCs when this thread faults.

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

10.3.27 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.
seL4_CNode	cspace_root	The new CSpace root.
seL4_Word	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_Word	vspace_root_data	Has no effect on x86 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

10.3.28 TCB - Set Timeout Endpoint

`static inline int seL4_TCB_SetTimeoutEndpoint`

Set a thread's timeout endpoint.

Type	Name	Description
seL4_TCB	_service	Capability to the TCB which is being operated on.
seL4_CPtr	timeout_fault_ep	CPTR to the endpoint which receives IPCs when this thread triggers timeout faults. Can be null.

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: Timeout exception messages will be delivered to this endpoint if it is not a null capability.

10.3.29 TCB - Suspend

`static inline int seL4_TCB_Suspend`

Suspend 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 6.1.3

10.3.30 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

10.3.31 TCB - Unset Breakpoint

`static inline int seL4_TCB_UnsetBreakpoint`

Disables a hardware breakpoint or watchpoint. The caller should assume that the underlying configuration of the hardware registers has also been cleared. Do not use this to clear single-stepping: the API will reject the call and return an error. Instead, use seL4_TCB_ConfigureSingleStepping to disable single-stepping.

Type	Name	Description
`seL4_TCB`	`_service`	Capability to the TCB which is being operated on.
`seL4_Uint16`	`bp_num`	The API-ID of a target breakpoint. This ID will be a positive integer, with values ranging from 0 to seL4_NumHWBreakpoints - 1.

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 6.2.4

10.3.32 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.
`seL4_Bool`	`resume_target`	The invocation should also resume the destination thread.
`seL4_Uint8`	`arch_flags`	Architecture dependent flags. These have no mearing on either x86 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 6.1.10

10.3.33 Untyped - Retype

`static inline int seL4_Untyped_Retype`

Retype an untyped object

Type	Name	Description
`seL4_Untyped`	`_service`	CPTR to an untyped object.
`seL4_Word`	`type`	The seL4 object type that we are retyping to.
`seL4_Word`	`size_bits`	Used to determine the size of variable-sized objects.
`seL4_CNode`	`root`	CPTR to the CNode at the root of the destination CSpace.
`seL4_Word`	`node_index`	CPTR to the destination CNode. Resolved relative to the root parameter.
`seL4_Word`	`node_depth`	Number of bits of node_index to translate when addressing the destination CNode.
`seL4_Word`	`node_offset`	Number of slots into the node at which capabilities start being placed.
`seL4_Word`	`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 CNode specified by `root`, `node_index`, and `node_depth`.

For variable-sized kernel objects, the `size_bits` argument is used to determine the size of objects to create. The relationship between `size_bits` and object size depends on the type of object being created. See Section 2.4.2 for more information about object sizes.

See Section 2.4 for more information about how untyped memory is retyped.

See Section 3.1.3 for more information about the placement of capabilities to created objects.

10.3.34 seL4_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`	TODO
`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

10.3.35 seL4_SchedContext - Consumed

```
static inline seL4_SchedContext_Consumed_t seL4_SchedContext_Consumed
```

Return the amount of time used by this scheduling context since this function was last called or a timeout exception triggered.

Type	Name	Description
`seL4_SchedContext`	`_service`	TODO

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

10.3.36 seL4_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`	TODO

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

10.3.37 seL4_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. 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`	TODO
`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

10.3.38 seL4_SchedContext - YieldTo

`static inline seL4_SchedContext_YieldTo_t 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`	TODO

Return value: TODO

Description: TODO

10.3.39 seL4_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`	TODO
`seL4_SchedContext`	`schedcontext`	Capability to the scheduling context which is being operated on.
`seL4_Time`	`budget`	Timeslice in microseconds, when the budget expires the thread will be pre-empted.
`seL4_Time`	`period`	Period in microseconds, if equal to budget, this thread will be treated as a round-robin thread. Otherwise, sporadic servers will be used to assure the scheduling context does not exceed the budget over the specified period.
`seL4_Word`	`extra_refills`	Number of extra sporadic replenishments this scheduling context should use. Ignored for round-robin threads.
`seL4_Word`	`badge`	Identifier for this scheduling context. Delivered to timeout exception handler. Can be used to determine which scheduling context triggered the timeout.

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

10.4 x86-Specific Object Methods

10.4.1 General x86 Object Methods

10.4.1.1 ASID Control - Make Pool

`static inline int seL4_X86_ASIDControl_MakePool`

Create an X86 ASID pool.

Type	Name	Description
`seL4_X86_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.
`seL4_Uint8`	`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

10.4.1.2 ASID Pool - Assign

`static inline int seL4_X86_ASIDPool_Assign`

Assign an ASID pool.

Type	Name	Description
`seL4_X86_ASIDPool`	`_service`	The ASID pool which is being assigned to. Must not be full. Each ASID pool can contain 1024 entries.
`seL4_CPtr`	`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

10.4.1.3 Extended Page Table Page Directory - Map

`static inline int seL4_X86_EPTPD_Map`

Map an EPT page directory.

Type	Name	Description
`seL4_X86_EPTPD`	`_service`	Capability to the EPT PD being operated on.
`seL4_X86_EPTPML4`	`pml4`	Capability to the EPT root which will contain the mapping
`seL4_Word`	`gpa`	Guest physical address to map the page into.
`seL4_X86_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

10.4.1.4 Extended Page Table Page Directory - Unmap

`static inline int seL4_X86_EPTPD_Unmap`

Unmap an EPT page directory.

Type	Name	Description
`seL4_X86_EPTPD`	`_service`	Capability to the EPT PD 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 Chapter 7

10.4.1.5 Extended Page Table Page Directory Page Table - Map

`static inline int seL4_X86_EPTPDPT_Map`

Map an EPT page directory page table.

Type	Name	Description
`seL4_X86_EPTPDPT`	`_service`	Capability to the EPT PDPT being operated on.
`seL4_X86_EPTPML4`	`pml4`	Capability to the EPT root which will contain the mapping
`seL4_Word`	`gpa`	Guest physical address to map the page into.
`seL4_X86_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

10.4.1.6 Extended Page Table Page Directory Page Table - Unmap

`static inline int seL4_X86_EPTPDPT_Unmap`

Unmap an EPT page directory page table.

Type	Name	Description
`seL4_X86_EPTPDPT`	`_service`	Capability to the EPT PDPT 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 Chapter 7

10.4.1.7 Extended Page Table Page Table - Map

`static inline int seL4_X86_EPTPT_Map`

Map an EPT page table.

Type	Name	Description
`seL4_X86_EPTPT`	`_service`	Capability to the EPT PT being operated on.
`seL4_X86_EPTPML4`	`pml4`	Capability to the EPT root which will contain the mapping
`seL4_Word`	`gpa`	Guest physical address to map the page into.
`seL4_X86_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

10.4.1.8 Extended Page Table Page Table - Unmap

`static inline int seL4_X86_EPTPT_Unmap`

Unmap an EPT page table.

Type	Name	Description
`seL4_X86_EPTPT`	`_service`	Capability to the EPT PT 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 Chapter 7

10.4.1.9 I/O Page Table - Map

`static inline int seL4_X86_IOPageTable_Map`

Map an IO page table into an IOSpace.

Type	Name	Description
`seL4_X86_IOPageTable`	`_service`	Capability to the I/O page table being operated on.
`seL4_X86_IOSpace`	`iospace`	The IOSpace to map the page table into.
`seL4_Word`	`ioaddr`	The address to map the page table 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 8.2.3

10.4.1.10 I/O Page Table - Unmap

`static inline int seL4_X86_IOPageTable_Unmap`

Unmap an IO page table from an IOSpace.

Type	Name	Description
`seL4_X86_IOPageTable`	`_service`	Capability to the I/O page table being operated on.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 Section 8.2.3

10.4.1.11 I/O Port - In16

`static inline seL4_X86_IOPort_In16_t seL4_X86_IOPort_In16`

Read 16 bits from an IO port.

Type	Name	Description
`seL4_X86_IOPort`	`_service`	An I/O Port capability.
`seL4_Uint16`	`port`	The port to read from.

Return value: A `seL4_X86_IOPort_In16_t` structure as described in Section 8.2.2.

Description: See Section 8.2.2

10.4.1.12 I/O Port - In32

`static inline seL4_X86_IOPort_In32_t seL4_X86_IOPort_In32`

Read 32 bits from an IO port.

Type	Name	Description
`seL4_X86_IOPort`	`_service`	An I/O Port capability.
`seL4_Uint16`	`port`	The port to read from.

Return value: A `seL4_X86_IOPort_In32_t` structure as described in Section 8.2.2.

Description: See Section 8.2.2

10.4.1.13 I/O Port - In8

`static inline seL4_X86_IOPort_In8_t seL4_X86_IOPort_In8`

Read 8 bits from an IO port.

Type	Name	Description
seL4_X86_IOPort	_service	An I/O Port capability.
seL4_Uint16	port	The port to read from.

Return value: A `seL4_X86_IOPort_In8_t` structure as described in Section 8.2.2.

Description: See Section 8.2.2

10.4.1.14 I/O Port - Out16

`static inline int seL4_X86_IOPort_Out16`

Write 16 bits to an IO port.

Type	Name	Description
seL4_X86_IOPort	_service	An I/O Port capability.
seL4_Word	port	The port to write to.
seL4_Word	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 8.2.2

10.4.1.15 I/O Port - Out32

`static inline int seL4_X86_IOPort_Out32`

Write 32 bits to an IO port.

Type	Name	Description
`seL4_X86_IOPort`	`_service`	An I/O Port capability.
`seL4_Word`	`port`	The port to write to.
`seL4_Word`	`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 8.2.2

10.4.1.16 I/O Port - Out8

`static inline int seL4_X86_IOPort_Out8`

Write 8 bits to an IO port.

Type	Name	Description
`seL4_X86_IOPort`	`_service`	An I/O Port capability.
`seL4_Word`	`port`	The port to write to.
`seL4_Word`	`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 8.2.2

10.4.1.17 IRQ Control - Get I/O APIC

`static inline int seL4_IRQControl_GetIOAPIC`

Create an IRQ handler capability for an interrupt from an IOAPIC.

Type	Name	Description
`seL4_IRQControl`	`_service`	An IRQControl capability. This gives you the authority to make this call.
`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.
`seL4_Uint8`	`depth`	Number of bits of index to resolve to find the destination slot.
`seL4_Word`	`ioapic`	Zero based index of IOAPIC to get interrupt from, ordered the same as in ACPI tables
`seL4_Word`	`pin`	IOAPIC pin that generates the interrupt.
`seL4_Word`	`level`	Indicates whether the IOAPIC should be programmed to treat this interrupt as level triggered.
`seL4_Word`	`polarity`	Indicates whether the IOAPIC should be programmed to treat this interrupt as high or low triggered
`seL4_Word`	`vector`	CPU vector to deliver the interrupt 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 8.1 and Section 8.2.1.

10.4.1.18 IRQ Control - Get MSI

`static inline int seL4_IRQControl_GetMSI`

Create an IRQ handler capability for an interrupt from an MSI.

Type	Name	Description
`seL4_IRQControl`	`_service`	An IRQControl capability. This gives you the authority to make this call.
`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.
`seL4_Uint8`	`depth`	Number of bits of index to resolve to find the destination slot.
`seL4_Word`	`pci_bus`	PCI bus ID of the device that will generate the interrupt.
`seL4_Word`	`pci_dev`	PCI device ID of the device that will generate the interrupt.
`seL4_Word`	`pci_func`	PCI function ID of the device that will generate the interrupt.
`seL4_Word`	`handle`	Value of the handle programmed into the data portion of the MSI.
`seL4_Word`	`vector`	CPU vector to deliver the interrupt 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 8.1 and Section 8.2.1.

10.4.1.19 Page - Get Address

`static inline seL4_X86_Page_GetAddress_t seL4_X86_Page_GetAddress`

Get the physical address of the underlying frame.

Type	Name	Description
`seL4_X86_Page`	`_service`	Capability to the page being operated on.

Return value: A `seL4_IA32_Page_GetAddress_t` struct that contains a `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

10.4.1.20 Page - Map EPT

`static inline int seL4_X86_Page_MapEPT`

TODO

Type	Name	Description
`seL4_X86_Page`	`_service`	Capability to the page being operated on.
`seL4_X86_EPTPML4`	`vroot`	TODO
`seL4_Word`	`vaddr`	TODO
`seL4_CapRights_t`	`rights`	TODO
`seL4_X86_VMAttributes`	`attr`	TODO

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: TODO

10.4.1.21 Page - Map I/O

`static inline int seL4_X86_Page_MapIO`

Map a page into an IOSpace.

Type	Name	Description
`seL4_X86_Page`	`_service`	Capability to the page being operated on.
`seL4_X86_IOSpace`	`iospace`	The IOSpace that the frame is being mapped into
`seL4_CapRights_t`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.4
`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 Chapter 7

10.4.1.22 Page - Map

`static inline int seL4_X86_Page_Map`

Map a page into an address space.

Type	Name	Description
`seL4_X86_Page`	`_service`	Capability to the page being operated on.
`seL4_CPtr`	`vroot`	Capability to the VSpace which will contain the mapping
`seL4_Word`	`vaddr`	Virtual address to map the page into.
`seL4_CapRights_t`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.4
`seL4_X86_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

10.4.1.23 Page - Remap

`static inline int seL4_X86_Page_Remap`

Remap a page.

Type	Name	Description
`seL4_X86_Page`	`_service`	Capability to the page being operated on.
`seL4_CPtr`	`vroot`	Capability to the VSpace which will contain the mapping
`seL4_CapRights_t`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.4
`seL4_X86_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

10.4.1.24 Page - Unmap

`static inline int seL4_X86_Page_Unmap`

Unmap a page.

Type	Name	Description
`seL4_X86_Page`	`_service`	Capability to the page 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 Chapter 7

10.4.1.25 Page Directory - Get Status Bits

`static inline seL4_X86_PageDirectory_GetStatusBits_t seL4_X86_PageDirectory_-GetStatusBits`

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

Type	Name	Description
`seL4_X86_PageDirectory`	`_service`	Capability to the page directory being operated on.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

10.4.1.26 Page Directory - Map

`static inline int seL4_X86_PageDirectory_Map`

Map a page directory.

Type	Name	Description
`seL4_X86_PageDirectory`	`_service`	Capability to the page directory being operated on.
`seL4_CPtr`	`vroot`	Capability to the VSpace which will contain the mapping
`seL4_Word`	`vaddr`	Virtual address to map the page into.
`seL4_X86_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

10.4.1.27 Page Directory - Unmap

`static inline int seL4_X86_PageDirectory_Unmap`

Unmap a page directory.

Type	Name	Description
`seL4_X86_PageDirectory`	`_service`	Capability to the page directory 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 Chapter 7

10.4.1.28 Page Table - Map

`static inline int seL4_X86_PageTable_Map`

Map a page table into an address space.

Type	Name	Description
`seL4_X86_PageTable`	`_service`	Capability to the page table being operated on.
`seL4_CPtr`	`vroot`	Capability to the VSpace which will contain the mapping
`seL4_Word`	`vaddr`	Virtual address to map the page into.
`seL4_X86_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

10.4.1.29 Page Table - Unmap

`static inline int seL4_X86_PageTable_Unmap`

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

Type	Name	Description
`seL4_X86_PageTable`	`_service`	Capability to the page table 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 Chapter 7

10.4.1.30 TCB - Set EPT Root

`static inline int seL4_TCB_SetEPTRoot`

Set the EPT root of a thread

Type	Name	Description
`seL4_TCB`	`_service`	TODO
`seL4_CPtr`	`eptpml4`	CPTR to an EPT PML4 object to act as the guest mode vspace root

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 6.4.

10.4.1.31 VCPU - Disable IO Port

`static inline int seL4_X86_VCPU_DisableIOPort`

Disable I/O port range in privileged execution

Type	Name	Description
`seL4_X86_VCPU`	`_service`	VCPU object to operate on
`seL4_Word`	`low`	Start of the I/O port range to disable
`seL4_Word`	`high`	Last I/O port in the range to disable

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: Disable a range of I/O ports for direct access by the execution mode in the `VCPU`.

10.4.1.32 VCPU - Enable IO Port

`static inline int seL4_X86_VCPU_EnableIOPort`

Enable I/O port range in guest execution

Type	Name	Description
`seL4_X86_VCPU`	`_service`	VCPU object to operate on
`seL4_CPtr`	`ioPort`	I/O port capability whose authority is being delegating
`seL4_Word`	`low`	Start of the I/O port range to enable
`seL4_Word`	`high`	Last I/O port in the range to enable

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: Enables a range of I/O ports for direct access by the execution mode in the `VCPU`. The requested port range must be a sub range of the provided I/O port capability.

This also establishes a link between the provided I/O port capability and the `VCPU`, see Section 6.4 for details.

10.4.1.33 VCPU - Read VMCS

`static inline seL4_X86_VCPU_ReadVMCS_t seL4_X86_VCPU_ReadVMCS`

Read VMCS field from the hardware

Type	Name	Description
`seL4_X86_VCPU`	`_service`	VCPU object to operate on
`seL4_Word`	`field`	Field to give to vmread instruction

Return value: A `seL4_X86_VCPU_ReadVMCS_t` struct that contains a `seL4_Word value`, which holds the return result of the `vmread` instruction, and `int error`. See Section 10.1 for a description of the message register and tag contents upon error.

Description: Thin wrapper around the `vmread` instruction that is performed on the VMCS region that is part of the VCPU object. After validating that a legal field is requested the value of ‘vmread‘ is returned directly in the result.

10.4.1.34 VCPU - Set TCB

`static inline int seL4_X86_VCPU_SetTCB`

Bind TCB to VCPU

Type	Name	Description
`seL4_X86_VCPU`	`_service`	VCPU object to operate on
`seL4_CNode`	`tcb`	CPTR of the TCB to bind 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: Configures the one-to-one binding of a VCPU and TCB, overwriting any previous binding in both. See Section 6.4.

10.4.1.35 VCPU - Write Registers

`static inline int seL4_X86_VCPU_WriteRegisters`

Set guest mode registers to the fields of a given `seL4_VCPUContext`

Type	Name	Description
`seL4_X86_VCPU`	`_service`	VCPU object to operate on
`seL4_VCPUContext *`	`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: Sets the guest mode registers, which is any registers not already part of the VMCS.

10.4.1.36 VCPU - Write VMCS

`static inline seL4_X86_VCPU_WriteVMCS_t seL4_X86_VCPU_WriteVMCS`

Write VMCS field to the hardware

Type	Name	Description
`seL4_X86_VCPU`	`_service`	VCPU object to operate on
`seL4_Word`	`field`	Field to give to `vmwrite` instruction
`seL4_Word`	`value`	Value to write using `vmwrite` instruction

Return value: A `seL4_X86_VCPU_WriteVMCS_t` struct that contains a `seL4_Word writen`, which holds the final value written with the `vmwrite` instruction, and `int error`. See Section 10.1 for a description of the message register and tag contents upon error.

Description: Thin wrapper around the ‘vmwrite‘ instruction that is performed on the VMCS region that is part of the VCPU object. As well as validating that a legal field is requested, the value may be modified to ensure any bits that are fixed in the hardware are correct, and that any features required for kernel correctness are not disabled (see Section 6.4).

The final value written to the hardware is returned and can be compared to the input parameter to determine what bits the kernel changed.

10.4.2 IA32-Specific Object Methods

No methods.

10.4.3 x86_64-Specific Object Methods

10.4.3.1 PDPT - Map

`static inline int seL4_X86_PDPT_Map`

TODO

Type	Name	Description
`seL4_X86_PDPT`	`_service`	TODO
`seL4_X64_PML4`	`pml4`	TODO
`seL4_Word`	`vaddr`	TODO
`seL4_X86_VMAttributes`	`attr`	TODO

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: TODO

10.4.3.2 PDPT - Unmap

`static inline int seL4_X86_PDPT_Unmap`

TODO

Type	Name	Description
`seL4_X86_PDPT`	`_service`	TODO

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: TODO

10.5 ARM-Specific Object Methods

10.5.1 General ARM Object Methods

10.5.1.1 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 being operated on.
`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.
`seL4_Uint8`	`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.

10.5.1.2 ASID Pool - 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_CPtr`	`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.

10.5.1.3 I/O Page Table - Map

`static inline int seL4_ARM_IOPageTable_Map`

TODO

Type	Name	Description
`seL4_ARM_IOPageTable`	`_service`	TODO
`seL4_ARM_IOSpace`	`iospace`	TODO
`seL4_Word`	`ioaddr`	TODO

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: TODO

10.5.1.4 I/O Page Table - Unmap

`static inline int seL4_ARM_IOPageTable_Unmap`

TODO

Type	Name	Description
`seL4_ARM_IOPageTable`	`_service`	TODO

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: TODO

10.5.1.5 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`	Capability to the page being operated on.
`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.

10.5.1.6 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`	Capability to the page being operated on.
`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.

10.5.1.7 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 being operated on.

Return value: A `seL4_ARM_Page_GetAddress_t` struct that contains a `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.

10.5.1.8 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`	Capability to the page being operated on.
`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.

10.5.1.9 Page - Map I/O

`static inline int seL4_ARM_Page_MapIO`

Type	Name	Description
`seL4_ARM_Page`	`_service`	Capability to the page being operated on.
`seL4_ARM_IOSpace`	`iospace`	TODO
`seL4_CapRights_t`	`rights`	TODO
`seL4_Word`	`ioaddr`	TODO

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: TODO

10.5.1.10 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 being operated on.
`seL4_CPtr`	`pd`	Capability to the VSpace which will contain the mapping.
`seL4_Word`	`vaddr`	Virtual address to map the page into.
`seL4_CapRights_t`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.4 .
`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.

10.5.1.11 Page - Remap

`static inline int seL4_ARM_Page_Remap`

Remap a page.

Type	Name	Description
`seL4_ARM_Page`	`_service`	Capability to the page being operated on.
`seL4_CPtr`	`pd`	Capability to the VSpace which will contain the mapping.
`seL4_CapRights_t`	`rights`	Rights for the mapping. Possible values for this type are given in Section 3.1.4 .
`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.

10.5.1.12 Page - Unify Instruction

`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`	Capability to the page being operated on.
`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.

10.5.1.13 Page - Unmap

`static inline int seL4_ARM_Page_Unmap`

Unmap a page.

Type	Name	Description
`seL4_ARM_Page`	`_service`	Capability to the page 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 Chapter 7.

10.5.1.14 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 being operated on.
`seL4_CPtr`	`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

10.5.1.15 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 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 Chapter 7.

10.5.1.16 VCPU - Inject IRQ

`static inline int seL4_ARM_VCPU_InjectIRQ`

Inject an IRQ to a virtual CPU

Type	Name	Description
`seL4_ARM_VCPU`	`_service`	TODO
`seL4_Uint16`	`virq`	Virtual IRQ ID
`seL4_Uint8`	`priority`	Priority of the IRQ to be injected
`seL4_Uint8`	`group`	IRQ group
`seL4_Uint8`	`index`	IRQ index

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: TODO

10.5.1.17 VCPU - Read Registers

`static inline seL4_ARM_VCPU_ReadRegs_t seL4_ARM_VCPU_ReadRegs`

Read a virtual CPU register

Type	Name	Description
`seL4_ARM_VCPU`	`_service`	TODO
`seL4_Word`	`field`	Register to read from a VCPU

Return value: TODO

Description: TODO

10.5.1.18 VCPU - Set TCB

`static inline int seL4_ARM_VCPU_SetTCB`

Bind a TCB to a virtual CPU

Type	Name	Description
`seL4_ARM_VCPU`	`_service`	TODO
`seL4_TCB`	`tcb`	Capability to TCB to bind to a virtual CPU

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: There is a 1:1 relationship between a virtual CPU and a TCB. If either (or both) of them is associated with another one, they will be dissociated, and then associated to the ones called in this system calls.

10.5.1.19 VCPU - Write Registers

`static inline int seL4_ARM_VCPU_WriteRegs`

Write a virtual CPU register

Type	Name	Description
`seL4_ARM_VCPU`	`_service`	TODO
`seL4_Word`	`field`	Register ID to write to a VCPU
`seL4_Word`	`value`	Value to be written to the VCPU register

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: TODO

10.5.2 Aarch32-Specific Object Methods

10.5.2.1 Page Directory - Clean Data

`static inline int seL4_ARM_PageDirectory_Clean_Data`

Clean cached pages within a page directory

Type	Name	Description
`seL4_ARM_PageDirectory`	`_service`	TODO
`seL4_Word`	`start`	Start address
`seL4_Word`	`end`	End address

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.

10.5.2.2 Page Directory - Clean and Invalidate Data

`static inline int seL4_ARM_PageDirectory_CleanInvalidate_Data`

Clean and invalidate cached pages within a page directory

Type	Name	Description
`seL4_ARM_PageDirectory`	`_service`	TODO
`seL4_Word`	`start`	Start address
`seL4_Word`	`end`	End address

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.

10.5.2.3 Page Directory - Invalidate Data

`static inline int seL4_ARM_PageDirectory_Invalidate_Data`

Invalidate cached pages within a page directory

Type	Name	Description
`seL4_ARM_PageDirectory`	`_service`	TODO
`seL4_Word`	`start`	Start address
`seL4_Word`	`end`	End address

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.

10.5.2.4 Page Directory - Unify Instruction

`static inline int seL4_ARM_PageDirectory_Unify_Instruction`

Clean and invalidate cached instruction pages to point of unification

Type	Name	Description
`seL4_ARM_PageDirectory`	`_service`	TODO
`seL4_Word`	`start`	Start address
`seL4_Word`	`end`	End address

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 Aarch64-Specific Object Methods

10.5.3.1 Page Directory - Map

`static inline int seL4_ARM_PageDirectory_Map`

Map a page directory

Type	Name	Description
`seL4_ARM_PageDirectory`	`_service`	TODO
`seL4_CPtr`	`pud`	Upper page directory
`seL4_Word`	`vaddr`	Virtual adress
`seL4_ARM_VMAttributes`	`attr`	Memory attributes

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: Map a page directory (level 2) to an upper page directory (level 1)

10.5.3.2 Page Directory - Unmap

`static inline int seL4_ARM_PageDirectory_Unmap`

Unmap a page directory

Type	Name	Description
`seL4_ARM_PageDirectory`	`_service`	TODO

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: Unmap a page directory (level 2) from an upper page directory (level 1)

10.5.3.3 Page Global Directory - Clean Data

`static inline int seL4_ARM_PageGlobalDirectory_Clean_Data`

Clean cached pages within a global page directory

Type	Name	Description
`seL4_ARM_PageGlobalDirectory`	`_service`	TODO
`seL4_Word`	`start`	Start address
`seL4_Word`	`end`	End address

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.

10.5.3.4 Page Global Directory - Clean and Invalidate Data

`static inline int seL4_ARM_PageGlobalDirectory_CleanInvalidate_Data`

Clean and invalidate cached pages within a global page directory

Type	Name	Description
`seL4_ARM_PageGlobalDirectory`	`_service`	TODO
`seL4_Word`	`start`	Start address
`seL4_Word`	`end`	End address

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.

10.5.3.5 Page Global Directory - Invalidate Data

`static inline int seL4_ARM_PageGlobalDirectory_Invalidate_Data`

Invalidate cached pages within a global page directory

Type	Name	Description
`seL4_ARM_PageGlobalDirectory`	`_service`	TODO
`seL4_Word`	`start`	Start address
`seL4_Word`	`end`	End address

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.

10.5.3.6 Page Global Directory - Unify Instruction

`static inline int seL4_ARM_PageGlobalDirectory_Unify_Instruction`

Clean and invalidate cached instruction pages to point of unification

Type	Name	Description
`seL4_ARM_PageGlobalDirectory`	`_service`	TODO
`seL4_Word`	`start`	TODO
`seL4_Word`	`end`	TODO

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.

10.5.3.7 Page Upper Directory - Map

`static inline int seL4_ARM_PageUpperDirectory_Map`

Map an upper page directory

Type	Name	Description
`seL4_ARM_PageUpperDirectory`	`_service`	TODO
`seL4_CPtr`	`pgd`	Global page directory
`seL4_Word`	`vaddr`	Virtual address
`seL4_ARM_VMAttributes`	`attr`	Memory attributes

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: Map an upper page directory (level 1) to a global page directory (level 0)

10.5.3.8 Page Upper Directory - Unmap

`static inline int seL4_ARM_PageUpperDirectory_Unmap`

TODO

Type	Name	Description
`seL4_ARM_PageUpperDirectory`	`_service`	TODO

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: TODO

Bibliography

[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.

[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.