seL4 Core Platform User Manual (v1.2-pre)

Contents

1 Introduction

seL4 Core Platform (sel4cp) is a small and simple operating system (OS) built on the seL4 microkernel. sel4cp is designed for building system with a *static architecture*. A static architecture is one where system resources are assigned up-front at system initialisation time.

1.1 Purpose

The seL4 Core Platform is intended to:

- provide a small and simple OS for a wide range of IoT, cyberphysical and other embedded use cases;
- provide a reasonable degree of application portability appropriate for the targeted use cases;
- make seL4-based systems easy to develop and deploy within the target areas;
- leverage seL4's strong isolation properties to support a near-minimal *trusted computing base* (TCB);
- retain seL4's trademark performance for systems built with it;
- be, in principle, amenable to formal analysis of system safety and security properties (although such analysis is beyond the initial scope).

1.2 Overview

An sel4cp system is built from a set of individual programs that are isolated from each other, and the system, in *protection domains*. Protection domains can interact by calling *protected procedures* or sending *notifications*.

sel4cp is distributed as a software development kit (SDK). The SDK includes the tools, libraries and binaries required to build an sel4cp system. The sel4cp source is also available which allows you to customize or extend sel4cp and produce your own SDK.

To build an sel4cp system you will write some programs that use `libsel4cp`. sel4cp programs are a little different to a typical progress on a Linux-like operating system. Rather than a single `main` entry point, a program has three distinct entry points: `init`, `notified` and, optionally, `protected`.

The individual programs are combined to produce a single bootable *system image*. The format of the image is suitable for loading by the target board's bootloader. The sel4cp tool, which is provided as part of the SDK, is used to generate the system image.

The sel4cp tool takes a *system description* as input. The system description is an XML file that specifies all the objects that make up the system.

Note: sel4cp does **not** impose any specific build system; you are free to choose the build system that works best for you.

1.3 Document Structure

The Concepts chapter describes the various concepts that make up sel4cp. It is recommended that you familiarise yourself with these concepts before trying to build a system.

The SDK chapter describes the software development kit, including its components and system requirements for use.

The sel4cp tool chapter describes the host system tool used for generating a firmware image from the system description.

The libsel4cp chapter describes the interfaces to the core platform library.

The System Description Format chapter describes the format of the system description XML file.

The Board Support Packages chapter describes each of the board support packages included in the SDK.

The Rationale chapter documents the rationale for some of the key design choices of in sel4cp.

2 Concepts

This chapter describes the key concepts provided by sel4cp.

As with any set of concepts there are words that take on special meanings. This document attempts to clearly describe all of these terms, however as the concepts are inter-related it is sometimes necessary to use a term prior to its formal introduction.

- system
- protection domain (PD)
- channel
- memory region
- notification
- protected procedure

2.1 System

At the most basic level sel4cp provides the platform for running a *system* on a specific board. As a *user* of sel4cp you use the platform to create a softare system that implements your use case. The system is described in a declarative configuration file, and the sel4cp tool takes this system description as an input and produces an appropriate system image that can be loaded on the target board.

The key elements that make up a system are *protection domains*, *memory regions* and *channels*.

2.2 Protection Domains

A **protection domain** (PD) is the fundamental runtime abstraction in the seL4 platform. It is analogous, but very different in detail, to a process on a UNIX system.

A PD provides a thread of control that executes within a fixed virtual address space. The isolation provided by the virtual address space is enforced by the underlying hardware MMU.

The virtual address space for a PD has mappings for the PD's *program image* along with any memory regions that the PD can access. The program image is an ELF file containing the code and data which implements the isolated component.

The platform supports a maximum of 63 protection domains.

2.2.1 Entry points

Although a protection domain is somewhat analogous to a process, it has a considerably different program structure and life-cycle. A process on a typical operating system will have a `main` function which is invoked by the system when the process is created. When the `main` function returns the process is destroyed. By comparison a protection domain has three entry points: `init`, `notify` and, optionally, `protected`.

When an seL4 Core Platform system is booted, all PDs in the system execute the `init` entry point.

The `notified` entry point will be invoked whenever the protection domain receives a *notification* on a *channel*. The `protected` entry point is invoked when a PD's *protected procedure* is called by another PD. A PD does not have to provide a protected procedure, therefore the `protected` entry point is optional. These entry points are described in more detail in subsequent sections.

Note: The processing of `init` entry points is **not** synchronised across protection domains. Specifically, it is possible for a high priority PD's `notified` or `protected` entry point to be called prior to the completion of a low priority PD's `init` entry point.

The overall computational model for an sel4cp system is a set of isolated components reacting to incoming events.

2.2.2 Scheduling

The PD has a number of scheduling attrbites that are configured in the system description:

- priority (0 – 254)
- period (microseconds)
- budget (microseconds)

The budget and period bound the fraction of CPU time that a PD can consume. Specifically, the **budget** specifies the amount of time for which the PD is allowed to execute. Once the PD has consumed its budget, it is no longer runnable until the budget is replenished; replenishment happens once every **period** and resets the budget to its initial value. This means that the maximum fraction of CPU time the PD can consume is budget/period.

The budget cannot be larger than the period. A budget that equals the period (aka. a "full" budget) behaves like a traditional time slice: After executing for a full period, the PD is preempted and put at the end of the scheduling queue of its priority. On other words, PDs with full budgets are scheduled round-robin with a time slice defined by the period.

The **priority** determines which of the runnable PDs to schedule. A PD is runnable if one of its entry points have been invoked and it has budget remaining in the current period. Runnable PDs of the same priority are scheduled in a round-robin manner.

2.3 Memory Regions

A *memory region* is a contiguous range of physical memory. A memory region may have a *fixed* physical address. For memory regions without a fixed physical address the physical address is allocated as part of the build process. Typically, memory regions with a fixed physical address represents memory-mapped device registers.

The size of a memory region must be a multiple of a supported page size. The supported page sizes are architecture dependent. For example, on AArch64 architectures, sel4cp support 4KiB and 2MiB pages. The page size for a memory region may be specified explicitly in the system description. If page size is not specified, the smallest supported page size is used.

Note: The page size also restricts the alignment of the memory region's physical address. A fixed physical address must be a multiple of the specified page size.

A memory region can be *mapped* into one or more protection domains. The mapping has a number of attributes, which include:

- the virtual address at which the region is mapped in the PD
- caching attributes (mostly relevant for device memory)
- permissions (read, write and execute)

Note: When a memory region is mapped into multiple protection domains, the attributes used for different mapping may vary.

2.4 Channels

A *channel* enables two protection domains to interact using protected procedures or notifications. Each connects connects exactly two PDs; there are no multi-party channels.

When a channel is created between two PDs, a *channel identifier* is configured for each PD. The *channel identifier* is used by the PD to reference the channel. Each PD can refer to the channel with a different identifier. For example if PDs **A** and **B** are connected by a channel, **A** may refer to the channel using an identifier of **37** while **B** may use **42** to refer to the same channel.

Note: There is no way for a PD to directly refer to another PD in the system. PDs can only refer to other PDs indirectly if there is a channel between them. In this case the channel identifier is effectively a proxy identifier for the other PD. So, to extend the prior example, **A** can indirectly refer to **B** via the channel identifier **37**. Similary, **B** can refer to **A** via the channel identifier **42**.

The system supports a maximum up 64 channels and interrupts per protection domain.

2.4.1 Protected procedure

A protection domain may provide a *protected procedure* (PP) which can be invoked from another protection domain. Up to 64 words of data may be passed as arguments when calling a protected procedure. The protected procedure return value may also be up to 64 words.

When a protection domain calls a protected procedure, the procedure executes within the context of the providing protection domain.

A protected call is only possible if the callee has strictly higher priority than the caller. Transitive calls are possible, and as such a PD may call a *protected procedure* in another PD from a `protected` entry point. However the overall call graph between PDs forms a directed, acyclic graph. It follows that a PD can not call itself, even indirectly. For example, `A calls B calls C` is valid (subject to the priority constraint), while `A calls B calls A` is not valid.

When a protection domain is called, the `protected` entry point is invoked. The control returns to the caller when the `protected` entry point returns.

The caller is blocked until the callee returns. Protected procedures must execute in bounded time. It is intended that future version of the platform will enforce this condition through static analysis. In the present version the callee must trust the callee to conform.

In general, PPs are provided by services for use by clients that trust the protection domain to provide that service.

To call a PP, a PD calls `sel4cp_ppcall` passing the channel identifier and a *message* structure. A *message* structure is returned from this function.

When a PD's protected procedure is invoked, the `protected` entry point is invoked with the channel identifier and message structure passed as arguments. The `protected` entry point must return a message structure.

2.4.2 Notification

A notification is a (binary) semaphore-like synchronisation mechanism. A PD can *notify* another PD to indicate availability of data in a shared memory region if they share a channel.

To notify another PD, a PD calls `sel4cp_notify`, passing the channel identifier. When a PD receives a notification, the `notified` entry point is invoked with the appropriate channel identifier passed as an argument.

Unlike protected procedures, notifications can be sent in either direction on a channel regardless of priority.

Note: Notifications provide a mechanism for synchronisation between PDs, however this is not a blocking operation. If a PD notifies another PD, that PD will become scheduled to run (if it is not already), but the current PD does **not** block. Of course, if the notified PD has a higher priority than the current PD, then the current PD will be preempted (but not blocked) by the other PD.

2.5 Interrupts

Hardware interrupts can be used to notify a protection domain. The system description specifies if a protection domain receives notifications for any hardware interrupt sources. Each hardware interrupt is assigned a channel identifier. In this way the protection domain can distinguish the hardware interrupt from other notifications. A specific hardware interrupt can only be associated with at most one protection domain.

Although interrupts are the final concept to be described here, they are in some ways the most important. Without interrupts a system would not do much after system initialisation.

sel4cp does not provides timers, nor any *sleep* API. After initialisation, activity in the system is initiated by an interrupt causing a `notified` entry point to be invoked. That notified function may in turn notify or call other protection domains that cause other system activity, but eventually all activity indirectly initiated from that interrupt will complete, at which point the system is inactive again until another interrupt occurs.

3 SDK

sel4cp is distributed as a software development kit (SDK).

The SDK includes support for one or more *boards*. Two *configurations* are supported for each board: *debug* and *release*. The *debug* configuration includes a debug build of the seL4 kernel to allow console debug output using the kernel's UART driver.

The SDK contains:

- sel4cp user manual (this document)
- sel4cp tool

Additionally, for each supported board configuration the following are provided:

- `libsel4cp`
- `loader.elf`
- `kernel.elf`
- `monitor.elf`

For some boards there are also examples provided in the `examples` directory.

The sel4cp SDK does **not** provide, nor require, any specific build system. The user is free to build their system using whatever build system is deemed most appropriate for their specific use case.

The sel4cp tool should be invoked by the system build process to transform a system description (and any referenced program images) into an image file which can be loaded by the target board's bootloader.

The ELF files provided as program images should be standard ELF files and have been linked against the provided libsel4cp.

3.1 System Requirements

The sel4cp tool requires Linux x86_64. The sel4cp tool is statically linked and should run on any Linux distribution. The sel4cp tool does not depend on any additional system binaries.

4 sel4cp tool

The sel4cp tool is available in `bin/sel4cp`.

The sel4cp tool takes as input a system description. The format of the system description is described in a subsequent chapter.

Usage:

```
sel4cp [-h] [-o OUTPUT] [-r REPORT] system
```

The path to the sytem description file must be provided.

In the case of errors, a diagnostic message shall be output to `stderr` and a non-zero code returned.

In the case of success, a loadable image file and a report shall be produced. The output paths for these can be specified by `-o` and `-r` respectively. The default output paths are `loader.img` and `report.txt`.

The loadable image will be a binary that can be loaded by the board's bootloader.

The report is a plain text file describing important information about the system. The report can be useful when debugging potential system problems. This report does not have a fixed format and may change between versions. It is not intended to be machine readable.

5 libsel4cp

All program images should link against `libsel4cp.a`.

The library provides the C runtime for the protection domain, along with interfaces for the sel4cp APIs.

The component must provide the following functions:

```
void init(void);
void notified(sel4cp_channel ch);
```

Additionally, if the protection domain provides a protected procedure it must also implement:

```
sel4cp_msginfo protected(sel4cp_channel ch, sel4cp_msginfo msginfo);
```

`libsel4cp` provides the following functions:

```
sel4cp_msginfo sel4cp_ppcall(sel4cp_channel ch, sel4cp_msginfo msginfo);
void sel4cp_notify(sel4cp_channel ch);
sel4cp_msginfo sel4cp_msginfo_new(uint64_t label, uint16_t count);
uint64_t sel4cp_msginfo_get_label(sel4cp_msginfo msginfo);
void sel4cp_irq_ack(sel4cp_channel ch);
void sel4cp_mr_set(uint8_t mr, uint64_t value);
uint64_t sel4cp_mr_get(uint8_t mr);
```

5.1 `void init(void)`

Every PD must expose an `init` entry point. This is called by the system at boot time.

5.2 `sel4cp_message protected(sel4cp_channel channel, sel4cp_message message)`

The `protected` entry point is optional. This is called when another PD call `sel4cp_ppcall` on a channel shared with the PD.

The `channel` argument identifies the channel on which the PP was invoked. Indirectly this identifies the PD performing the call. Channel identifiers are specified in the system configuration. **Note:** The channel argument is passed by the system and is unforgable.

The `message` argument is the argument passed to the PP and is provided by the calling PD. The contents of the message is up to a pre-arranged protocol between the PDs. The message contents are opaque to the system. Note: The message is *copied* from the caller.

The returned `message` is the return value of the protected procedure. As with arguments this is *copied* to the caller.

5.3 `void notified(sel4cp_channel channel)`

The `notified` entry point is called by the system when a PD has received a notification on a channel.

`channel` identifies the channel which has been notified (and indirectly the PD that performed the notification).

Note: `channel` could identify an interrupt.

Channel identifiers are specified in the system configuration.

5.4 `sel4cp_message sel4cp_ppcall(sel4cp_channel channel, sel4cp_message message)`

Performs a call to a protected procedure in a different PD. The `channel` argument identifies the protected procedure to be called. `message` is passed as argument to the protected procedure. Channel identifiers are specified in the system configuration.

The protected procedure's return data is returned in the `sel4cp_message`.

5.5 `void sel4cp_notify(sel4cp_channel channel)`

Notify the `channel`. Channel identifiers are specified in the system configuration.

5.6 `void sel4cp_irq_ack(sel4cp_channel ch)`

Acknowledge the interrupt identified by the specified channel.

5.7 `sel4cp_message sel4cp_msginfo_new(uint64_t label, uint16_t count)`

Creates a new message structure.

The message can be passed to `sel4cp_ppcall` or returned from `protected`.

5.8 `uint64_t sel4cp_msginfo_get_label(sel4cp_message message)`

Returns the label from a message.

5.9 `uint64_t sel4cp_mr_get(uint8_t mr)`

Get a message register.

5.10 `void sel4cp_mr_set(uint8_t mr, uint64_t)`

Set a message register.

6 System Description Format

This section describes the format of the system description file. This file is provided as the input to the `sel4cp` tool.

The system description file is an XML file.

The root element of the XML file is `system`.

Within the `system` root element the following child elements are supported:

- `protection_domain`
- `memory_region`
- `channel`

6.1 `protection_domain`

The `protection_domain` element describes a protection domain.

It supports the following attributes:

- `name`: a unique name for the protection domain
- `pp`: (optional) indicates that the protection domain has a protected procedure; defaults to false.
- `priority`: the priority of the protection domain (integer 0 to 254).
- `budget`: (optional) the PD's budget in microseconds; defaults to 1,000.
- `period`: (optional) the PD's period in microseconds; must not be smaller than the budget; defaults to the budget.

Additionally, it supports the following child elements:

- `program_image`: (exactly one) describes the program image for the protection domain.
- `map`: (zero or more) describes mapping of memory regions into the protection domain.
- `irq`: (zero or more) describes hardware interrupt associations.
- `setvar`: (zero or more) describes variable rewriting.

The `program_image` element has a single `path` attribute describing the path to an ELF file.

The `map` element has the following attributes:

- `mr`: Identifies the memory region to map.
- `vaddr`: Identifies the virtual address at which to map the memory region.
- `perms`: Identifies the permissions with which to map the memory region. Can be a combination of `r` (read), `w` (write), and `x` (eXecute).
- `cached`: Determines if mapped with caching enabled or disabled. Defaults to `true`.
- `setvar_vaddr`: Specifies a symbol in the program image. This symbol will be rewritten with the virtual address of the memory region.

The `irq` element has the following attributes:

- `irq`: The hardware interrupt number.
- `id`: The channel identifier.

The `setvar` element has the following attributes:

- `symbol`: Name of a symbol in the ELF file.
- `region_paddr`: Name of an MR. The symbol's value shall be updated to this MRs physical address.

6.2 `memory_region`

The `memory_region` element describes a memory region.

It supports the following attributes:

- `name`: a unique name for the memory region
- `size`: size of the memory region in bytes (must be a multiple of the page size)
- `page_size`: (optional) size of the pages used in the memory region; must be a supported page size if provided.
- `phys_addr`: (optional) the physical address for the start of the memory region.

The `memory_region` element does not support any child elements.

6.3 `channel`

The `channel` element has exactly two `end` children elements for specifying the two PDs associated with the channel.

The `end` element has the following attributes:

- `pd`: Name of the protection domain for this end.
- `id`: Channel identifier in the context of the named protection domain.

The `id` is passed to the PD in the `notified` and `protected` entry points. The `id` should be passed to the `sel4cp_notify` and `sel4cp_ppcall` functions.

7 Board Support Packages

This chapter describes the board support packages that are available in the SDK.

7.1 TQMa8XQP 1GB

The TQMa8XQP is a system-on-module designed by TQ-Systems GmbH. The modules incorporates an NXP i.MX8X Quad Plus system-on-chip and 1GiB ECC memory.

TQ-Systems provide the MBa8Xx carrier board for development purposes. The instructions provided assume the use of the MBa8Xx carrier board. If you are using a different carrier board please refer to the appropriate documentation.

The MBa8Xx provides access to the TQMa8XQP UART via UART-USB bridge. To access the UART connect a USB micro cable to port **X13**. The UART-USB bridge supports 4 individual UARTs; the UART is connected to the 2nd port.

By default the SoM will autoboot using U-boot. Hit any key during the boot process to stop the autoboot.

A new board will autoboot to Linux. You will likely want to disable autoboot:

```
=> env set bootdelay -1
=> env save
```

The board can be reset by pressing switch **S4** (located next to the Ethernet port). Alternatively, you can use the `reset` command from the U-Boot prompt.

During development the most convenient way to boot an sel4cp image is via network booting. U-boot support booting via the *tftp* protocol. To support this you'll want to configure the network. U-Boot supports DHCP, however it is often more reliable to explicitly set an IP address. For example:

```
=> env set ipaddr 10.1.1.2
=> env set netmask 255.255.255.0
=> env set serverip 10.1.1.1
=> env save
```

To use tftp you also need to set the file to load and the memory address to load it to:

```
=> env set bootfile loader.img
=> env set loadaddr 0x80280000
=> env save
```

The system image generated by the sel4cp tool is a raw binary file.

An example sequence of commands for booting is:

```
=> tftpboot
=> dcache flush
=> icache flush
=> go ${loadaddr}
```

Rather than typing these each time you can create a U-Boot script:

```
=> env set sel4cp 'tftpboot; dcache flush; icache flush; go ${loadaddr}'
=> env save
=> run sel4cp
```

When debugging is enabled the kernel will use the same UART as U-Boot.

7.2 ZCU102

Initial support is available for the ZCU102.

FIXME: Additional documentation required here.

The ZCU102 can run on a physical board or on an appropriate QEMU based emulator.

An QEMU command line:

```
$ qemu-system-aarch64 \
   -m 4G  \
   -M arm-generic-fdt \
   -nographic \
   -hw-dtb [PATH TO zcu102-arm.dtb] \
   -device loader,file=[SYSTEM IMAGE],addr=0x40000000,cpu-num=0 \
   -device loader,addr=0xfd1a0104,data=0x0000000e,data-len=4 \
   -serial mon:stdio
```

8 Rationale

This section describes the rationales driving the sel4cp design choices.

8.1 Overview

The seL4 microkernel provides a set of powerful and flexible mechanisms that can be used for building almost arbitrary systems. While minimising constraints on the nature of system designs and scope of deployments, this flexibility makes it challenging to design the best system for a particular use case, requiring extensive seL4 experience from developers.

The seL4 Core Platform addresses this challenge by constraining the system architecture to one that provides enough features and power for its target usage class (IoT, cyberphysical and other embedded systems with a static architecture), enabling a much simpler set of developer-visible abstractions.

8.2 Protection Domains

PDs are single-threaded to keep the programming model and implementations simple, and because this serves the needs of most present use cases in the target domains. Extending the model to multithreaded applications (clients) is straightforward and can be done if needed. Extending to multithreaded services is possible but requires additional infrastructure for which we see no need in the near future.

8.3 Protected Procedure Priorities

The restriction of only calling to higher priority prevents deadlocks and reflects the notion that the callee operates on behalf of the caller, and it should not be possible to preempt execution of the callee unless the caller could be preempted as well.

This greatly simplifies reasoning about real-time properties in the system; in particular, it means that PPs can be used to implement *resource servers*, where shared resources are encapsulated in a component that ensures mutual exclusion, while avoiding unbounded priority inversions through the *immediate priority ceiling protocol*.

While it would be possible to achieve the same by allowing PPs between PDs of the same priority, this would be much harder to statically analyse for loop-freedom (and thus deadlock-freedom). The drawback is that we waste a part of the priority space where a logical entity is split into multiple PDs, eg to separate out a particularly critical component to formally verify it, when the complete entity would be too complex for formal verification. For the kinds of systems targeted by the seL4 Core Platform, this reduction of the usable priority space is unlikely to cause problems.

8.4 Protected Procedure Argument Size

The limitation on the size of by-value arguments is forced by the (architecture-dependent) limits on the payload size of the underlying seL4 operations, as well as by efficiency considerations. The protected procedure payload should be considered as analogous to function arguments in the C language; similar limitations exist in the C ABIs (Application Binary Interfaces) of various platforms.

8.5 Limits

The limitation on the number of protection domains in the system is relatively arbitrary. Based on experience with the system and the types of systems being built it is possible for this to be increased in the future.

The limitation on the number of channels for a protection domain is based on the size of the notification object in seL4. Changing this to be large than 64 would most likely require changes to seL4.