Setting up your environment

This page outlines how to set up your environment for completing the workshop and how to test that you've done so correctly. If you have any issues feel free to message me on Mattermost or email me at i.velickovic@unsw.edu.au!

Acquiring the dependencies

Here are two options for building and running the workshop material:

1. The seL4 Docker container. If you already have the Docker container used for seL4 development set up, that is sufficient for this workshop.
2. Install dependencies and use the SDK natively, however this option is only available on Linux x86_64.

Option 1 - seL4 Docker container

Follow the instructions on the seL4 website.

Option 2 - Native (Linux x86_64)

You'll need the following dependencies:

- make
- AAarch64 cross compiler (specifically, the `aarch64-linux-gnu-` toolchain)
- QEMU for AAarch64 emulation

If you use `apt` you can install these with the following command:

```
sudo apt-get update && sudo apt-get install make gcc-aarch64-linux-gnu qemu-system-arm
```

Acquiring the SDK

Regardless of how you set up your machine, you'll need to acquire the specific SDK for the workshop from here. Alternatively, run the following commands:

```
mkdir sel4cp_workshop
cd sel4cp_workshop
wget summit.ivanvelickovic.com/downloads/sdk.tar.gz
tar xvf sdk.tar.gz
```

Testing your environment

After setting up everything, you should attempt to run a simple "hello world" system to make sure that your environment is working correctly. Note that if you're using the Docker container, you'll want to do the following steps inside the container.

Build the "hello world" image:

```
# Inside the "sel4cp_workshop" directory
mkdir hello_world_build
make -C sel4cp_workshop_sdk/board/qemu_arm_virt/example/hello/
BUILD_DIR=$(pwd)/hello_world_build SEL4CP_SDK=$(pwd)/sel4cp_workshop_sdk
SEL4CP_BOARD=qemu_arm_virt SEL4CP_CONFIG=debug
```

Run the built image:

```
qemu-system-aarch64 -machine virt -cpu cortex-a53 -serial mon:stdio -device
loader,file=hello_world_build/loader.img,addr=0x70000000,cpu-num=0 -m size=1G -
display none
```

After running the hello world example, you should get the following output:

```
LDR|INFO: altloader for seL4 starting
LDR|INFO: Flags:                0x0000000000000000
LDR|INFO: Kernel:      entry:   0xffffff8040000000
LDR|INFO: Root server: physmem: 0x000000004024d000 -- 0x0000000040254000
LDR|INFO:              virtmem: 0x000000008a000000 -- 0x000000008a007000
LDR|INFO:              entry  : 0x000000008a000000
LDR|INFO: region: 0x00000000   addr: 0x0000000040000000   size:
0x0000000000249000   offset: 0x0000000000000000   type: 0x0000000000000001
LDR|INFO: region: 0x00000001   addr: 0x000000004024d000   size:
0x00000000000060d0   offset: 0x0000000000249000   type: 0x0000000000000001
LDR|INFO: region: 0x00000002   addr: 0x0000000040249000   size:
0x0000000000000960   offset: 0x000000000024f0d0   type: 0x0000000000000001
LDR|INFO: region: 0x00000003   addr: 0x000000004024a000   size:
0x0000000000000318   offset: 0x000000000024fa30   type: 0x0000000000000001
LDR|INFO: region: 0x00000004   addr: 0x000000004024b000   size:
0x0000000000001020   offset: 0x000000000024fd48   type: 0x0000000000000001
LDR|INFO: copying region 0x00000000
LDR|INFO: copying region 0x00000001
LDR|INFO: copying region 0x00000002
LDR|INFO: copying region 0x00000003
LDR|INFO: copying region 0x00000004
LDR|INFO: Setting all interrupts to Group 1
LDR|INFO: GICv2 ITLinesNumber: 0x00000000
LDR|INFO: CurrentEL=EL1
LDR|INFO: enabling MMU
LDR|INFO: jumping to kernel
Bootstrapping kernel
Warning: Could not infer GIC interrupt target ID, assuming 0.
reserved virt address space regions: 3
  [ffffff8040000000..ffffff8040249000]
  [ffffff8040249000..ffffff804024d000]
  [ffffff804024d000..ffffff8040254000]
available phys memory regions: 1
  [40000000..80000000]
Booting all finished, dropped to user space
MON|INFO: seL4 Core Platform Bootstrap
MON|INFO: bootinfo untyped list matches expected list
MON|INFO: Number of bootstrap invocations: 0x00000009
MON|INFO: Number of system invocations:    0x00000022
MON|INFO: completed bootstrap invocations
MON|INFO: completed system invocations
hello, world
```

To exit QEMU, press `CTRL + a` then `x`.

If you manage to get the final "hello, world", you have set up your machine correctly and can move on to the exercises, once they're released. That's all you need to know for now, the details of building and running a system will be explained in the workshop.

Welcome

Welcome to the dry-run of the seL4 Core Platform workshop for the 2022 seL4 Summit! Thank you for participating, any and all feedback is appreciated.

In this workshop you will be learning, with the guide of this book, to implement a game called Wordle on top of the seL4 Core Platform (seL4CP). Wordle is a simple game that became very popular this year, if you aren't familiar I suggest quickly playing around with it here.

The Core Platform is a minimal operating system aimed for secure and performant embedded systems built on seL4. It provides abstractions over seL4 in order to allow users to build systems without significant friction. By implementing Wordle, you will be exposed to seL4CP abstractions such as:

- Protection Domains
- Channels
- Memory Regions
- Interrupts
- Basic dynamicism

If you haven't already, please first set up your machine.

Workshop material

You can download the workshop material from here. Alternatively, run the following commands:

```
wget summit.ivanvelickovic.com/downloads/workshop.tar.gz
tar xvf workshop.tar.gz
```

While the exercises should explain everything, it might also be a good idea to keep the seL4CP manual (at `/path/to/sdk/doc/manual.pdf`) as well as the seL4 manual handy. You'll notice that some of the seL4CP specific documentation in the exercises was taken from the manual.

Now that you have the material, you can proceed to the first exercise.

Part 1 - Serial server

In implementing Wordle we will need to be able to both receive serial input and produce serial output. Our initial goal will be to have a "serial server" that will facitilate reading and writing via the UART.

Creating a serial server

System description

In the Core Platform, you describe all the parts of your system in an XML format. Below is a basic example of how you would make a simple "Hello world!" system. Here we define a protection domain with a name and priority as well as the image that will be run once the protection domain is started.

```
<?xml version="1.0" encoding="UTF-8"?>
<system>
    <protection_domain name="hello_world" priority="254">
        <program_image path="hello_world.elf" />
    </protection_domain>
</system>
```

Everything necessary is defined inside the `<system>` tag. Here we have a simple PD called "hello_world".

Protection Domains (PDs)

Each protection domain is a single thread of execution. It is has its own TCB, address space (VSpace) and capabilities (CSpace). It is somewhat analogous to a "process" in other systems such as Unix or Linux.

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`, `notified` and, optionally, `protected`.

When a Platform system is booted, all PDs in the system execute the init entry point. However, there is nothing in-place to synchronise the `init` entry point across protection domains. There could be a case where a high priority PD's `notified` or `protected` entry point is called before the `init` entry point of another PD has completed.

Here is a full description of the `protection_domain` element:

- `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.

Serial server PD

Now that you have some understanding of how protection domains work and are specified, try to edit `wordle.system` to have a protection domain for the serial server. Consider whether the priority should be the default or explicitly defined. If successful, you should see a message being printed from the `init()` entry point.

Accessing the UART

Our serial driver will be useless unless it can actually access the UART device registers. But since we're in user space, we will need to map in the physical address into our PD's virtual address space. The Core Platform provides an abstraction for this:

Memory Regions (MRs)

We can create memory regions at a specific physical address, or somewhere in RAM, that we can then map into the address space of whichever PD we want to have access to it. Here is a full description of the `memory_region` element which you would use to create a memory region:

- `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 (4KiB or 2MiB on AArch64).
- `phys_addr`: (optional) the physical address for the start of the memory region.

In order to access the memory region, it needs to be mapped using the `map` element:

- `mr`: Identifies the memory region to map.
- `vaddr`: Identifies the virtual address at which to map the memory region to.
- `perms`: Identifies the permissions with which to map the memory region with. Can be any combination of r (read), w (write), and x (eXecute).
- `cached`: Determines if reigon is 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.

Example

If I wanted to have two PDs that shared a buffer to avoid costly IPC to send data between the PDs, my system description might look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<system>
    <!-- Create a page sized buffer to share -->
    <memory_region name="shared_data_buffer" size="0x1000" />
    <protection_domain name="sender">
        <program_image path="sender.elf" />
        <!-- "sender" simply writes to the buffer. The seL4CP tool will do ELF
symbol patching to set the
             virtual address of the mapping to a global variable called
"buffer". -->
        <map mr="shared_data_buffer" vaddr="0x2_000_000" perms="w"
setvar_vaddr="buffer"/>
    </protection_domain>
    <protection_domain name="receiver">
        <program_image path="receiver.elf" />
        <!-- "receiver" simply reads from the buffer. The seL4CP tool will do
ELF symbol patching to set the
             virtual address of the mapping to a global variable called
"buffer". -->
        <map mr="shared_data_buffer" vaddr="0x2_000_000" perms="r"
setvar_vaddr="buffer"/>
    </protection_domain>
</system>
```

From reading the above, you should be able to create the mapping to the UART for the serial server UART. We'll be using PL011 UART which has a physical address of `0x9000000` on the QEMU board. Think about what might be different for a mapping to a device compared to a regular region of memory.

Your task now is to:

- Map in the UART to the serial server PD.
- Test using the provided code that the mapping works correctly instead of using `sel4cp_dbg_puts`.

Interrupts (IRQs)

In order to handle serial input, we need to set up the serial server to receive hardware interrupts and handle them accordingly. Interrupts are delivered as notifications via the `notified(sel4cp_channel ch)` entry point. Receiving an interrupt or notification causes this entry point to be invoked, after it finishes the PD goes back to waiting to receive other messages.

The `irq` element has the following attributes:

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

The channel identifier is what's used by a PD to distiniguish a particular interrupt from other interrupts or other notification sources.

Your next task is to edit `wordle.system` to specify the interrupt for serial input. On the QEMU board we are using the IRQ number is 33. Inspect the provided code to see what is necessary to handle the interrupt, also don't forget to acknowledge it (see `sel4cp_irq_ack`). Check to see that you are actually receiving interrupts and try use the provided functions to get a character and then print it to the screen!

Building

If we build part 1 and then look inside the build directory:

```
$ make part1 SEL4CP_SDK=/path/to/sdk
$ ls build
serial_server.elf serial_server.o wordle.img report.txt
```

If you look inside the Makefile, you'll see that the last command that gets run is an invocation of the `sel4cp` tool that comes with the SDK. It is this tool that takes in the ELF files for the PDs along with a system description and produces a raw image that you can boot from and have your system start. I would like to stress that the `sel4cp` tool only requires valid ELF files, it does not matter how you produce the ELFs, as long as they are valid and linked with the `libsel4cp` library from the SDK. For the purposes of this workshop, using make is fairly simple and sufficient, however you are free to use whatever build system you want when making a Core Platform system.

I'll also make a note about the report file, `report.txt`. This report file is worth taking a brief look at as it contains a lot of information about the system, and can be especially helpful when debugging.

Running

You can invoke QEMU via make:

```
make run
```

▼ Explanation of QEMU command
The command that is actually being run by make is:

```
qemu-system-aarch64 -machine virt -cpu cortex-a53 -serial mon:stdio -device
loader,file=hello_world_build/loader.img,addr=0x70000000,cpu-num=0 -m size=1G -
display none
```

For those not overly familiar with QEMU, I will break down the arguments.

- `-machine virt -cpu cortex-a53`: This just specifies the board we are emulating, in this case we are using QEMU's AArch64 "virt" board which has no physical counterpart, and we're specifying to use one ARM Cortex-A53 CPU.
- `-serial mon:stdio`: Send virtual serial and QEMU monitor ouitput onto `stdio`.
- `-device loader,file=wordle.img,addr=0x70000000,cpu-num=0`: QEMU won't know how to automatically load this image. Typically with seL4 projects the image is passed via the `-kernel` flag, however in this case we have to specify the address that the seL4CP bootloader is expecting to be loaded at.
- `-m size=1G`: Have 1G of RAM. This is plenty for this workshop, but depending on what you're doing, might need to be increased.
- `-display none`: TODO

Part 2 - Client

Now that we have a serial server working, our next goal will be to have a client PD that serves a CLI interface to play Wordle. For this you want to have a way for the client to get the inputted character from the serial server. A relatively simple way to do this is to share a buffer and have the serial server notify the client that there's a new character to read.[1]

[1] While it would be easier to just IPC five characters at a time, since the server has higher priority than the client, doing a protected procedure call (same thing as IPC, to be explained later), is restricted in the Core Platform. See this section in the manual for details.

First, you'll have to learn how communication between PDs is done:

Channels

A channel allows two protection domains to interact with each other, either via notifications (asynchronous) or protected procedures (synchronous). In this part of the workshop, you'll be using notifications (protected procedure calls will be covered later).

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.

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` function which allows you to notify the PD on the other end of the channel.

Example

Here's a simple example of two PDs that have a channel between them for notifying each other.

```
<?xml version="1.0" encoding="UTF-8"?>
<system>
    <protection_domain name="sender">
        <program_image path="sender.elf" />
    </protection_domain>
    <protection_domain name="receiver">
        <program_image path="receiver.elf" />
    </protection_domain>

    <channel>
        <end pd="sender" id="1" />
        <end pd="receiver" id="2" />
    </channel>
</system>
```

On the sender's side, their `notified` and `init` entry points might look something like this:

```
#define RECEIVER_CHANNEL_ID 1

void init() {
    sel4cp_notify(RECEIVER_CHANNEL_ID);
}

void notified(sel4cp_channel ch) {
    switch (ch) {
        case RECEIVER_CHANNEL_ID:
            sel4cp_dbg_puts("Got message from receiver! Sending notification
back\n");
            sel4cp_notify(RECEIVER_CHANNEL_ID);
            break;
        default:
            sel4cp_dbg_puts("panic!\n");
    }
}
```

On the receiver's side, their `notified` entry point might look something like this:

```
#define SENDER_CHANNEL_ID 2

void notified(sel4cp_channel ch) {
    switch (ch) {
        case SENDER_CHANNEL_ID:
            sel4cp_dbg_puts("Got message from sender! Sending notification
back\n");
            sel4cp_notify(1);
            break;
        default:
            sel4cp_dbg_puts("panic!\n");
    }
}
```

▼ Can you guess what this system will do?
Answer: They will be playing ping pong with notifications indefinitely!

Part 2 goal

Ultimately, you want to be able to get some output like this:

```
SERIAL|INFO: starting
CLIENT|INFO: starting
Welcome to the Wordle client!
[h] [e] [l] [l] [o]
[t] [h] [e] [r] [e]
[ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ]
```

Pressing `ENTER` should move to the next line if they've entered enough characters. Similarly, you should be able to delete a character by pressing `BACKSPACE`. Once you have that working, we can start communicating with the Wordle server to get it to check whether we have a correct guess.

Building

```
make part2 /path/to/sdk
```

Running

```
make run
```

Part 3 - Wordle server

This part invovles hooking up the client with a Wordle server PD that facilitates.

Exercise Part 4 - Dynamicism