Capabilities

This tutorial provides a basic introduction to seL4 capabilities.

Prerequisites

1. Set up your machine.
2. Hello world

Initialising

```
# For instructions about obtaining the tutorial sources see
https://docs.sel4.systems/Tutorials/#get-the-code
#
# Follow these instructions to initialise the tutorial
# initialising the build directory with a tutorial exercise
./init --tut capabilities
# building the tutorial exercise
cd capabilities_build
ninja
```

Outcomes

By the end of this tutorial, you should be familiar with:

1. The jargon CNode, CSpace, CSlot.
2. Know how to invoke a capability.
3. Know how to delete and copy CSlots.

Background

What is a capability?

A *capability* is a unique, unforgeable token that gives the possessor permission to access an entity or object in system. In seL4, capabilities to all resources controlled by seL4 are given to the root task on initialisation. To change the state of any resources, users use the kernel API, available in `libsel4` to request an operation on a specific capability.

For example, the root task is provided with a capability to its own thread control block (TCB), `seL4_CapInitThreadTCB`, a constant defined by `libsel4`. To change the properties of the initial TCB, one can use any of the TCB API methods on this capability. Below is an example which changes the stack pointer of the root task's TCB, a common operation in the root task if a larger stack is needed:

```
    seL4_UserContext registers;
    seL4_Word num_registers = sizeof(seL4_UserContext)/sizeof(seL4_Word);
```

```
    /* Read the registers of the TCB that the capability in seL4_CapInitThreadTCB
grants access to. */
    seL4_Error error = seL4_TCB_ReadRegisters(seL4_CapInitThreadTCB, 0, 0,
num_registers, &registers);
    assert(error == seL4_NoError);

    /* set new register values */
    registers.sp = new_sp; // the new stack pointer, derived by prior code.

    /* Write new values */
    error = seL4_TCB_WriteRegisters(seL4_CapInitThreadTCB, 0, 0, num_registers,
registers);
    assert(error == seL4_NoError);
```

Further documentation is available on TCB_ReadRegisters and TCB_WriteRegisters.

CNodes

A *CNode* (capability-node) is an object full of capabilities: you can think of a CNode as an array of capabilities. We refer to slots as *CSlots* (capability-slots). In the example above, `seL4_CapInitThreadTCB` is the slot in the root task's CNode that contains the capability to the root task's TCB. Each CSlot in a CNode can be in the following state:

- empty: the CNode slot contains a null capability,
- full: the slot contains a capability to a kernel resource.

By convention the 0th CSlot is kept empty, for the same reasons as keeping NULL unmapped in process virtual address spaces: to avoid errors when uninitialised slots are used unintentionally.

CSlots are `1u << seL4_SlotBits` in size, and as a result the number of slots in a CNode can be calculated by `CNodeSize / (1u << seL4_SlotBits)`.

CSpaces

A *CSpace* (capability-space) is the full range of capabilities accessible to a thread, which may be formed of one or more CNodes. In this tutorial, we focus on the CSpace constructed for the root task by seL4's initialisation protocol, which consists of one CNode.

CSpace addressing

In order to refer to a capability, to perform operations on it, you must address the capability. There are two ways to address capabilities in the seL4 API. First is by invocation, the second is by direct addressing. Invocation is what we used to manipulate the registers of the root task's TCB, which we now explain in further detail.

Invocation

On boot, the root task has a CNode capability installed as its *CSpace root*. An *invocation* is when a CSlot is addressed by implicitly invoking a thread's installed CSpace root. In the code example above, we use an

invocation on the `seL4_CapInitThreadTCB` CSlot to read and write to the registers of the TCB represented by the capability in that specific CSlot.

```
seL4_TCB_WriteRegisters(seL4_CapInitThreadTCB, 0, 0, num_registers, registers);
```

This implicity looks up the `seL4_CapInitThreadTCB` CSlot in the cspace root of the calling thread, which in this case is the root task.

Direct CSpace addressing

Direct addressing allows you to specify the CNode to address, rather than implicitly using the CSpace root, and is used to construct and manipulate the shape of CSpaces. Note that direct addressing requires invocation: the operation occurs by invoking a CNode capability, which itself is indexed from the CSpace root.

The following fields are used when directly addressing CSlots:

- *_service/root* A capability to the CNode to operate on.
- *index* The index of the CSlot in the CNode to address.
- *depth* How far to traverse the CNode before resolving the CSlot. For the initial, single-level CSpace, the *depth* value is always `seL4_WordBits`. For invocations, the depth is always implicitly `seL4_WordBits`. More on CSpace depth will be discussed in future tutorials.

In the example below, we directly address the root task's TCB to make a copy of it in the 0th slot in the cspace root. CNode copy requires two CSlots to be directly addressed: the destination CSlot, and the source CSlot. Because we are copying in the same CNode, the root used in both addresses is the same: `seL4_CapInitThreadCNode`, which is the slot where seL4 places a capability to the root task's CSpace root.

```
    seL4_Error error = seL4_CNode_Copy(seL4_CapInitThreadCNode, 0, seL4_WordBits,
                                       seL4_CapInitThreadCNode,
seL4_CapInitThreadTCB, seL4_WordBits,
                                       seL4_AllRights);
    assert(error == seL4_NoError);
```

All CNode invocations, require direct CSpace addressing.

Initial CSpace

The root task has a CSpace, set up by seL4 during boot, which contains capabilities to all resources manages by seL4. We have already seen several capabilities in the root cspace: `seL4_CapInitThreadTCB`, and `seL4_CapInitThreadCNode`. Both of these are specified by constants in `libsel4`, however not all initial capabilities are statically specified. Other capabilities are described by the `seL4_BootInfo` data structure, described in `libsel4` and initialised by seL4. `seL4_BootInfo` describes ranges of initial capabilities, including free slots available in the initial CSpace.

Exercises

The initial state of this tutorial provides you with the BootInfo structure, and calculates the size (in bytes) of the initial CNode object.

```
int main(int argc, char *argv[]) {

    /* parse the location of the seL4_BootInfo data structure from
    the environment variables set up by the default crt0.S */
    seL4_BootInfo *info = platsupport_get_bootinfo();

    size_t initial_cnode_object_size = BIT(info->initThreadCNodeSizeBits);
    printf("Initial CNode is %zu bytes in size\n", initial_cnode_object_size);
```

When you run the tutorial without changes, you will see something like the following output:

```
Booting all finished, dropped to user space
Initial CNode is 4096 bytes in size
The CSpace has 0 CSlots
<<seL4(CPU 0) [decodeInvocation/530 T0xffffff801ffb5400 "rootserver" @401397]:
Attempted to invoke a null cap #4095.>>
main@main.c:33 [Cond failed: error]
    Failed to set priority
```

By the end of the tutorial all of the output will make sense. For now, the first line is from the kernel. The second is the `printf`, telling you the size of the initial CNode. The third line stating the number of slots in the CSpace, is incorrect, and your first task is to fix that.

How big is your CSpace?

Exercise: refer to the background above, and calculate the number of slots in the initial thread's CSpace.

```
    size_t num_initial_cnode_slots = 0; // TODO calculate this.
    printf("The CSpace has %zu CSlots\n", num_initial_cnode_slots);
```

Copy a capability between CSlots

After the output showing the number of CSlots in the CSpace, you will see an error:

```
<<seL4(CPU 0) [decodeInvocation/530 T0xffffff801ffb5400 "rootserver" @401397]:
Attempted to invoke a null cap #4095.>>
main@main.c:33 [Cond failed: error]
    Failed to set priority
```

The error occurs as the existing code tries to set the priority of the initial thread's TCB by invoking the last CSlot in the CSpace, which is currently empty. seL4 then returns an error code, and our check that the

operation succeeded fails.

Exercise: fix this problem by making another copy of the TCB capability into the last slot in the CNode.

```
    seL4_CPtr first_free_slot = info->empty.start;
    seL4_Error error = seL4_CNode_Copy(seL4_CapInitThreadCNode, first_free_slot,
seL4_WordBits,
                                       seL4_CapInitThreadCNode,
seL4_CapInitThreadTCB, seL4_WordBits,
                                       seL4_AllRights);
    ZF_LOGF_IF(error, "Failed to copy cap!");
    seL4_CPtr last_slot = info->empty.end - 1;
    /* TODO use seL4_CNode_Copy to make another copy of the initial TCB capability
to the last slot in the CSpace */

    /* set the priority of the root task */
    error = seL4_TCB_SetPriority(last_slot, last_slot, 10);
    ZF_LOGF_IF(error, "Failed to set priority");
```

On success, you will now see the output:

```
<<seL4(CPU 0) [decodeCNodeInvocation/94 T0xffffff801ffb5400 "rootserver" @401397]:
CNode Copy/Mint/Move/Mutate: Destination not empty.>>
main@main.c:44 [Cond failed: error != seL4_FailedLookup]
    first_free_slot is not empty
```

Which will be fixed in the next exercise.

How do you delete capabilities?

The provided code checks that both `first_free_slot` and `last_slot` are empty, which of course is not true, as you copied TCB capabilities into those CSlots. Checking if CSlots are empty is done by a neat hack: by attempting to move the CSlots onto themselves. This should fail with an error code `seL4_FailedLookup` if the source CSLot is empty, and an `seL4_DeleteFirst` if not.

Exercise: delete both copies of the TCB capability.

- You can either use `seL4_CNode_Delete` on the copies, or
- `seL4_CNode_Revoke` on the original capability to achieve this.

```
    // TODO delete the created TCB capabilities

    // check first_free_slot is empty
    error = seL4_CNode_Move(seL4_CapInitThreadCNode, first_free_slot,
seL4_WordBits,
                            seL4_CapInitThreadCNode, first_free_slot,
seL4_WordBits);
    ZF_LOGF_IF(error != seL4_FailedLookup, "first_free_slot is not empty");
```

```
    // check last_slot is empty
    error = seL4_CNode_Move(seL4_CapInitThreadCNode, last_slot, seL4_WordBits,
                            seL4_CapInitThreadCNode, last_slot, seL4_WordBits);
    ZF_LOGF_IF(error != seL4_FailedLookup, "last_slot is not empty");
```

On success, the output will now show:

```
<<seL4(CPU 0) [decodeCNodeInvocation/106 T0xffffff801ffb5400 "rootserver"
@401397]: CNode Copy/Mint/Move/Mutate: Source slot invalid or empty.>>
<<seL4(CPU 0) [decodeCNodeInvocation/106 T0xffffff801ffb5400 "rootserver"
@401397]: CNode Copy/Mint/Move/Mutate: Source slot invalid or empty.>>
Suspending current thread
main@main.c:56 Failed to suspend current thread
```

Invoking capabilities

Exercise Use `seL4_TCB_Suspend` to try and suspend the current thread.

```
    printf("Suspending current thread\n");
    // TODO suspend the current thread
    ZF_LOGF("Failed to suspend current thread\n");
```

On success, the output will be as follows:

```
<<seL4(CPU 0) [decodeCNodeInvocation/106 T0xffffff801ffb5400 "rootserver"
@401397]: CNode Copy/Mint/Move/Mutate: Source slot invalid or empty.>>
<<seL4(CPU 0) [decodeCNodeInvocation/106 T0xffffff801ffb5400 "rootserver"
@401397]: CNode Copy/Mint/Move/Mutate: Source slot invalid or empty.>>
Suspending current thread
```

Further exercises

That's all for the detailed content of this tutorial. Below we list other ideas for exercises you can try, to become more familiar with cspaces.

- Use a data structure to track which CSlots in a CSpace are free.
- Make copies of the entire cspace described by `seL4_BootInfo`
- Experiment with other CNode invocations.

Getting help

Stuck? See the resources below.

- FAQ
- seL4 Manual
- Debugging guide
- IRC Channel
- Developer's mailing list