Mapping

This tutorial provides an introduction to virtual memory management on seL4.

Prerequisites

1. Set up your machine.
2. Capabilities tutorial

Outcomes

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

1. How to map and unmap virtual memory pages in seL4.

Background

Virtual memory

seL4 does not provide virtual memory management, beyond kernel primitives for manipulating hardware paging structures. User-level must provide services for creating intermediate paging structures, mapping and unmapping pages.

Users are free to define their own address space layout with one restriction: the seL4 kernel claims the high part of the virtual memory range. On most 32-bit platforms, this is 0xe0000000 and above. This variable is set per platform, and can be found by finding the `kernelBase` variable in the seL4 source.

Paging structures

As part of the boot process, seL4 initialises the root task with a top-level hardware virtual memory object, which is referred to as a *VSpace*. A capability to this structure is made available in the `seL4_CapInitThreadVSpace` slot in the root tasks CSpace. For each architecture, this capability is to a different object type corresponding to the hardware, top-level paging structure. The table below lists the VSpace object type for each supported architecture.

Architecture	VSpace Object
aarch32	`seL4_PageDirectory`
aarch64	`seL4_PageGlobalDirectory`
ia32	`seL4_PageDirectory`
x86_64	`seL4_PML4`
RISC-V	`seL4_PageTable`

In addition to the top-level paging structure, intermediate hardware virtual memory objects are required to map pages. The table below lists those objects, in order, for each architecture.

Architecture	Objects
aarch32	`seL4_PageTable`
aarch64	`seL4_PageUpperDirectory`, `seL4_PageDirectory`, `seL4_PageTable`
ia32	`seL4_PageTable`
x86_64	`seL4_PDPT`, `seL4_PageDirectory`, `seL4_PageTable`
RISC-V	`seL4_PageTable`

This tutorial covers the x86_64 architecture, but should contain sufficient information on the virtual memory API provided by seL4 to generalise to other architectures.

Each paging structure can be invoked in order to map or unmap it. Below is an example of mapping an x86_64 `seL4_PDPT` object:

```
    /* map a PDPT at TEST_VADDR */
    error = seL4_X86_PDPT_Map(pdpt, seL4_CapInitThreadVSpace, TEST_VADDR,
seL4_X86_Default_VMAttributes);
```

All mapping functions take three arguments:

- the VSpace to map the object into,
- the virtual address to map the object at,
- and virtual memory attributes.

If the virtual memory address provided is not aligned to the size of the paging object, seL4 will mask out any unused bits e.g a 4KiB page mapped at 0xDEADBEEF will end up mapped at 0xDEADB000.

Virtual memory attributes determine the caching attributes of the mapping, which is architecture dependent. Alongside the attributes used in this tutorial (`seL4_X86_Default_VMAttributes`) you can find alternative values, in `libsel4`.

Pages

Once all of the intermediate paging structures have been mapped for a specific virtual address range, physical frames can be mapped into that range by invoking the frame capability. The code snippet below shows an example of mapping a frame at address `TEST_VADDR`.

```
    /* map a read-only page at TEST_VADDR */
    error = seL4_X86_Page_Map(frame, seL4_CapInitThreadVSpace, TEST_VADDR,
seL4_CanRead, seL4_X86_Default_VMAttributes);
```

For a page mapping to succeed, all mid-level paging structures must be mapped. The `libsel4` function `seL4_MappingFailedLookupLevel()` can be used to determine at which level paging structures are missing. Note that to map a frame multiple times, one must make copies of the frame capability: each frame capability can only track one mapping.

In addition to the arguments taken by the map methods for intermediate paging structures, page mapping takes a `rights` argument which determines the mapping type. In the example above, we map the page read only.

Types and sizes

Page types and sizes are architecture dependent. For both x86 and ARM architectures, each size of page is a different object type with a specific size. On RISC-V, pages are the same object type and variably sized. Configuration and hardware settings alter the available page sizes.

Exercises

This tutorial uses several helper functions to allocate objects and capabilities. All object and CSlot allocations have already been done for you using these functions. For more information see the on these mechanisms, see the capabilities and untyped tutorials.

Map a page directory

On starting the tutorial, you will see the following output:

```
Missing intermediate paging structure at level 30
main@main.c:34 [Cond failed: error != seL4_NoError]
    Failed to map page
```

This is because while the provided code maps in a `seL4_PDPT` object, there are two missing levels of paging structures. The value corresponds to the `libsel4` constant `SEL4_MAPPING_LOOKUP_NO_PD` which is the number of bits in the virtual address that could not be resolved due to missing paging structures.

Exercise Map in the `pd` structure using (`seL4_PageDirectory_Map`) [https://docs.sel4.systems/ApiDoc.html#map-5].

```
    // TODO map a page directory object
```

On success, you should see the following:

```
Missing intermediate paging structure at level 21
main@main.c:34 [Cond failed: error != seL4_NoError]
    Failed to map page
```

Map a page table

Note that in the above output, the number of failed bits has changed from `30` to `21`: this is because another 9 bits could be resolved from the newly mapped page directory.

Exercise Map in the `pt` structure using `seL4_PageTable_Map`.

```
    // TODO map a page table object
```

On success, you should see the following:

```
Read x: 0
Set x to 5
Caught cap fault in send phase at address (nil)
while trying to handle:
vm fault on data at address 0xa000000000 with status 0x7
in thread 0xffffff801ffb5400 "rootserver" at address 0x401afe
With stack:
0x41c920: 0x41c9d0
...
```

The page is successfully mapped and we read a value of 0: as seL4 zeros all non-device pages when they are retyped from untyped memory. However, then the code attempts to write to the page. Since you mapped the page as read-only, this fails and the kernel raises a VM fault.

Since the initial task does not have a fault handler (more details in the upcoming threads tutorial), this triggers a subsequent capability fault (cap fault) on slot 0 (nil). Because you are running a debug kernel for the tutorial, the kernel outputs the details of both faults.

The vm fault is the most interesting: it shows the fault address, fault status register, and the instruction pointer that the fault occured on (address).

Remap a page

Exercise Fix the fault by remapping the page with `seL4_ReadWrite` permissions, using the seL4_X86_Page_Map invocation.

```
    // TODO remap the page
```

Unmapping pages

Pages can be unmapped by either using `Unmap` invocations on the page or any intermediate paging structure, or deleting the final capability to any of the paging structure.

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 virtual memory management on seL4.

- Try unmapping the structures you just mapped in.
- Port this tutorial to another architecture (ARM, RISCV).

- Create a generic function for converting from `seL4_MappingFailedLookupLevel` to the required seL4 object.

Getting help

Stuck? See the resources below.

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