Untyped

This tutorial provides an introduction to physical 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. The jargon *untyped* and *device untyped*.
2. Know how to create objects from untyped memory in seL4.
3. Know how to reclaim objects.

Background

Physical memory

Apart from a small, static amount of kernel memory, all physical memory is managed by user-level in an seL4 system. Capabilities to objects created by seL4 at boot, as well as the rest of the physical resources managed by seL4, are passed to the root task on start up.

Untyped

Excluding the objects used to create the root task, capabilities to all available physical memory are passed to the root task as capabilities to *untyped* memory. Untyped memory is a block of contiguous physical memory with a specific size. Untyped objects have a boolean property *device* which indicates whether the memory is writable by the kernel or not: if memory is *device* memory, it is not backed by RAM but some other device. Device untyped can only be retyped as frame objects (physical memory frames, which can be mapped into virtual memory), and cannot be written to by the kernel.

Initial state

The `seL4_BootInfo` structure provided to the root task describes all of the untyped capabilities, including their size, if they are a device untyped, and the physical address of the untyped. The code example below shows how to print out the initial untyped capabilities provided from `seL4_BootInfo`.

```
    printf("    CSlot   \tPaddr           \tSize\tType\n");
    for (seL4_CPtr slot = info->untyped.start; slot != info->untyped.end; slot++)
{
        seL4_UntypedDesc *desc = &info->untypedList[slot - info->untyped.start];
        printf("%8p\t%16p\t2^%d\t%s\n", (void *) slot, (void *) desc->paddr, desc-
>sizeBits, desc->isDevice ? "device untyped" : "untyped");
    }
```

Retyping

Untyped capabilities have a single invocation: seL4_Untyped_Retype which is used to create a new capability from an untyped. Specifically, the new capability created by a retype invocation provides access to the a subset of the memory range granted by the original untyped, with a specific type. New capabilities created by retyping an untyped object are referred to as *children* of that untyped object.

Untyped objects are retyped incrementally in a greedy fashion from the invoked untyped, which is important to understand in order to gain efficient memory utilisation in seL4 systems. Each untyped object maintains a single watermark, with addresses before the watermark being unavailable (already retyped) and after the watermark being free (not retyped yet). On a retype operation, the watermark is moved first to the alignment of the object being created, and then to the end of the size of the object. For example, if a 4KiB object is created from an untyped, then a 16KiB object is created, the 12KiB between the end of the 4KiB object and the start of the 16KiB object is wasted due to alignment. The memory cannot be reclaimed until both children are revoked.

TL;DR: Untyped objects should be allocated in order of size, largest first, to avoid wasting memory.

```
    error = seL4_Untyped_Retype(parent_untyped, // the untyped capability to
retype
                                seL4_UntypedObject, // type
                                untyped_size_bits,  //size
                                seL4_CapInitThreadCNode, // root
                                0, // node_index
                                0, // node_depth
                                child_untyped, // node_offset
                                1 // num_caps
                                );
```

The above code snippet shows an example of retyping an untyped into a smaller, 4KiB untyped object. We'll refer to this snippet as we discuss each parameter in the retype invocation. Recall that the first parameter is the untyped capability being invoked to do the retype.

Types

Each object in seL4 has a specific type, and all of the type constants can be found in `libsel4`. Some types are architecture specific, while others are general across architectures. In the example above, we create a new untyped object, which can be further retyped. The type of the created object determines the invocations that can be made on that object. For example, were we to retype the object into a thread control block (TCB -- `seL4_TCBObject`), we could then perform TCB invocations on the object, including `seL4_TCB_SetPriority`.

Size

The size argument determines the size of the new object and is specified in `size_bits`, where the corresponding object size is the exponentiation, where `size_bits` is the exponent and the base is 2. In other words, the object size is $2^{\texttt{size\_bits}}$ in bytes.

All seL4 objects must be powers of 2 in size, however some objects are fixed-size while other objects are variable in size. The size parameter is *ignored* for fixed-sized objects and only respected by the kernel for variably-sized objects. Variably sized objects include untypeds (`seL4_UntypedObject`), CNodes (`seL4_CapTableObject`) and frame objects, of which the latter are architecture dependent.

Root, node_index & node_depth

The `root`, `node_index` and `node_depth` parameters are used to specify the CNode in which to place the new capabilities. Depending on the `depth` parameter, the CSlot to use is addressed by invocation or by direct addressing.

In the example above, `node_depth` is set to 0, which means that the `root` parameter is looked up implicitly using the CSpace root of the current thread at a depth of `seL4_WordBits`. So the example code specifies the root task's CNode (`seL4_CapInitThreadCNode`). The `node_index` parameter in this case is ignored.

If the `node_depth` value is not set to 0, then direct addressing is used with the current thread's cspace root as the root. Then the `node_index` parameter is used to locate the CNode capability to place new capabilities in, at the specified `node_depth`. This is designed for managing multi-level CSpaces, and is not covered in this tutorial.

Node_offset

The node_offset is the CSlot to start creating new capabilities at, in the CNode selected by the previous parameters. In this case, the first empty CSlot in the initial CNode is selected.

Num_caps

The retype invocation can be used to create more than 1 capability at a time -- the amount of capabilities is specified using this argument. Note that there are two constraints on this value:

1. The untyped must be big enough to fit all of the memory being retyped (`num_caps * (1u << size_bits)`).
2. The CNode must have enough consecutive free CSlots to fit all of the new capabilities.

Exercises

Create an untyped

When you first run this tutorial, you will see something like the following output, which lists all of the untyped capabilities provided to the root task on boot:

```
Booting all finished, dropped to user space
    CSlot       Paddr               Size    Type
   0x12d            0x100000    2^20    device untyped
   0x12e            0x200000    2^21    device untyped
   0x12f            0x400000    2^22    device untyped
   0x130            0xb2e000    2^13    device untyped
```

At the end of the output, there is an error:

```
<<seL4(CPU 0) [decodeUntypedInvocation/105 T0xffffff801ffb5400 "rootserver"
@4012e1]: Untyped Retype: Requested UntypedItem size too small.>>
main@main.c:49 [Cond failed: error != seL4_NoError]
    Failed to retype
```

This error is because we are trying to create an untyped of size 0.

Exercise Calculate the size of the child untyped required, such that the child untyped can be used to create all of the objects lists in the `objects` array.

```
    // list of general seL4 objects
    seL4_Word objects[] = {seL4_TCBObject, seL4_EndpointObject,
seL4_NotificationObject};
    // list of general seL4 object size_bits
    seL4_Word sizes[] = {seL4_TCBBits, seL4_EndpointBits, seL4_NotificationBits};

    // TODO work out what size object we need to create to be able to create all
of the objects
    // listed above
    seL4_Word untyped_size_bits = 0;
    seL4_CPtr parent_untyped = 0;
    seL4_CPtr child_untyped = info->empty.start;

    // First, find an untyped big enough to fit all of our objects
    for (int i = 0; i < (info->untyped.end - info->untyped.start); i++) {
        if (info->untypedList[i].sizeBits >= untyped_size_bits && !info-
>untypedList[i].isDevice) {
            parent_untyped = info->untyped.start + i;
            break;
        }
    }
```

On success, the tutorial will progress further, printing "Failed to set priority"

Create a TCB Object

The priority check is failing as `child_tcb` is an empty CSlot.

Exercise fix this by creating a TCB object from `child_untyped` placed in `child_tcb` CSlot.

```
    seL4_CPtr child_tcb = child_untyped + 1;
    /* TODO create a TCB in CSlot child_tcb */

    // try to set the TCB priority
    error = seL4_TCB_SetPriority(child_tcb, seL4_CapInitThreadTCB, 10);
    ZF_LOGF_IF(error != seL4_NoError, "Failed to set priority");
```

On success, the tutorial will progress further, printing "Endpoint cap is null cap".

Create an endpoint object

The error you see now is caused be an invalid endpoint capability.

Exercise Create an endpoint object from `child_untyped` and place it in the `child_ep` CSlot.

```
    seL4_CPtr child_ep = child_tcb + 1;
    /* TODO create an endpoint in CSlot child_ep */

    // identify the type of child_ep
    uint32_t cap_id = seL4_DebugCapIdentify(child_ep);
    ZF_LOGF_IF(cap_id == 0, "Endpoint cap is null cap");
```

On success, 'Failed to bind notification' should be output.

Create a notification object

The next part of the tutorial attempts to use a notification object that does not yet exist.

Exercise create a notification object from `child_untyped` and place it in the `child_ntfn` CSlot.

```
    seL4_CPtr child_ntfn = child_ep + 1;
    // TODO create a notification object in CSlot child_ntfn

    // try to use child_ntfn
    error = seL4_TCB_BindNotification(child_tcb, child_ntfn);
    ZF_LOGF_IF(error != seL4_NoError, "Failed to bind notification.");
```

Delete the objects

The final part of the tutorial attempts to create enough endpoint objects from `child_untyped` to consume the entire untyped object. However, this fails, because the untyped is already completely consumed by the previous allocations.

Exercise revoke the child untyped, so we can create new objects from it that use up the whole thing.

```
    // TODO revoke the child untyped

    // allocate the whole child_untyped as endpoints
    seL4_Word num_eps = BIT(untyped_size_bits - seL4_EndpointBits);
    error = seL4_Untyped_Retype(child_untyped, seL4_EndpointObject, 0,
seL4_CapInitThreadCNode, 0, 0, child_tcb, num_eps);
    ZF_LOGF_IF(error != seL4_NoError, "Failed to create endpoints.");

    printf("Success\n");
```

Once the tutorial is completed successfully, you should see the message "Success".

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 untyped objects and memory allocation in seL4.

- Allocate objects at specific physical addresses.
- Create a simple object allocator for allocating seL4 objects.

Getting help

Stuck? See the resources below.

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