Threads

This is a tutorial for using threads on seL4.

Prerequisites

1. Set up your machine.
2. Capabilities tutorial
3. Mapping tutorial

Outcomes

1. Know the jargon TCB.
2. Learn how to start a thread in the same address space.
3. Understand how to read and update TCB register state.
4. Learn how to suspend and resume a thread.
5. Understand thread priorities and their interaction with the seL4 scheduler.
6. Gain a basic understanding of exceptions and debug fault handlers.

Background

Thread Control Blocks

seL4 provides threads to represent an execution context and manage processor time. Threads in seL4 are realised by *thread control block* objects (TCBs), one for each kernel thread.

TCBs contain the following information:

- a priority and maximum control priority,
- register state and floating-point context,
- CSpace capability,
- VSpace capability,
- endpoint capability to send fault messages to,
- and the reply capability slot.

Scheduling model

The seL4 scheduler chooses the next thread to run on a specific processing core, and is a priority-based round-robin scheduler. The scheduler picks threads that are runnable: that is, resumed, and not blocked on any IPC operation.

Priorities

The scheduler picks the highest-priority, runnable thread. seL4 provides a priority range of 0-255, where 255 is the maximum priority (encoded in `libsel4` as `seL4_MinPrio` and `seL4_MaxPrio`).

TCBs also have a *maximum control priority* (MCP), which acts as an informal capability over priorities. When setting the priority of a TCB, an explicit TCB capability must be provided to derive the authority from to set the

priority. The priority being set is checked against the authority TCB's MCP and the targer priority is greater, the operation fails. The root task starts with both priority and MCP set to `seL4_MaxPrio`.

Round robin

When multiple TCBs are runnable and have the same priority, the are scheduled in a first-in first-out round-robin fashion. In more detail, kernel time is accounted for in fixed-time quanta referred to as ticks, and each TCB has a timeslice field which represents the number of ticks that TCB is eligible to execute until preempted. The kernel timer driver is configured to fire a periodic interrupt which marks each tick, and when the timeslice is exhausted round robin scheduling is applied. Threads can surrender their current tick using the `seL4_Yield` system call.

Domain scheduling

In order to provide confidentiality seL4 provides a top-level hierarchical scheduler which provides static, cyclical scheduling of scheduling partitions known as domains. Domains are statically configured at compile time with a cyclic schedule, and are non-preemptible resulting in completely deterministic scheduling of domains.

Threads can be assigned to domains, and threads are only scheduled when their domain is active. Cross-domain IPC is delayed until a domain switch, and seL4_Yield between domains is not possible. When there are no threads to run while a domain is scheduled, a domain-specific idle thread will run until a switch occurs.

Assigning a thread to a domain requires access to the `seL4_DomainSet` capability. This allows a thread to be added to any domain.

```
/* Set thread's domain */
seL4_Error seL4_DomainSet_Set(seL4_DomainSet _service, seL4_Uint8 domain, seL4_TCB
thread);
```

Thread Attributes

seL4 threads are configured by invocations on the TCB object.

Exercises

This tutorial will guide you through using TCB invocations to create a new thread in the same address space and pass arguments to the new thread. Additionally, you will learn about how to debug a virtual memory fault.

By the end of this tutorial you want to spawn a new thread to call the function in the code example below.

```
int new_thread(void *arg1, void *arg2, void *arg3) {
    printf("Hello2: arg1 %p, arg2 %p, arg3 %p\n", arg1, arg2, arg3);
    void (*func)(int) = arg1;
    func(*(int *)arg2);
```

```
    while(1);
}
```

CapDL Loader

Previous tutorials have taken place in the root task where the starting CSpace layout is set by the seL4 boot protocol. This tutorial uses a the *capDL loader*, a root task which allocates statically configured objects and capabilities.

The capDL loader parses a static description of the system and the relevant ELF binaries. It is primarily used in Camkes projects but we also use it in the tutorials to reduce redundant code. The program that you construct will end up with its own CSpace and VSpace, which are separate from the root task, meaning CSlots like `seL4_CapInitThreadVSpace` have no meaning in applications loaded by the capDL loader.

Information about CapDL projects can be found here.

Configure a TCB

When you first build and run the tutorial, you should see something like the following:

```
Hello, World!
Dumping all tcbs!
Name                                            State           IP
Prio    Core
--------------------------------------------------------------------------------
----
tcb_threads                                    running         0x4012ef    254
0
idle_thread                                     idle            (nil)   0
0
rootserver                                      inactive        0x4024c2    255
0
<<seL4(CPU 0) [decodeInvocation/530 T0xffffff8008140c00 "tcb_threads" @4012ef]:
Attempted to invoke a >
main@threads.c:42 [Cond failed: result]
Failed to retype thread: 2
```

`Dumping all tcbs!` and the following table is generated by a debug syscall called `seL4_DebugDumpScheduler()`. seL4 has a series of debug syscalls that are available in debug kernel builds. The available debug syscalls can be found in libsel4. `seL4_DebugDumpScheduler()` is used to dump the current state of the scheduler and can be useful to debug situations where a system seems to have hung.

After the TCB table, you can see the `seL4_Untyped_Retype` invocation is failing due to invalid arguments. The loader has been configured to set up the following capabilities and symbols:

```
// the root CNode of the current thread
```

```
extern seL4_CPtr root_cnode;
// VSpace of the current thread
extern seL4_CPtr root_vspace;
// TCB of the current thread
extern seL4_CPtr root_tcb;
// Untyped object large enough to create a new TCB object

extern seL4_CPtr tcb_untyped;
extern seL4_CPtr buf2_frame_cap;
extern const char buf2_frame[4096];

// Empty slot for the new TCB object
extern seL4_CPtr tcb_cap_slot;
// Symbol for the IPC buffer mapping in the VSpace, and capability to the mapping
extern seL4_CPtr tcb_ipc_frame;
extern const char thread_ipc_buff_sym[4096];
// Symbol for the top of a 16 * 4KiB stack mapping, and capability to the mapping
extern const char tcb_stack_base[65536];
static const uintptr_t tcb_stack_top = (const uintptr_t)&tcb_stack_base +
sizeof(tcb_stack_base);
```

Exercise Fix the `seL4_Untyped_Retype` call (shown below) using the capabilities provided above, such that a new TCB object is created in `tcb_cap_slot`.

```
int main(int c, char* arbv[]) {

    printf("Hello, World!\n");

    seL4_DebugDumpScheduler();
    // TODO fix the parameters in this invocation
    seL4_Error result = seL4_Untyped_Retype(seL4_CapNull, seL4_TCBObject,
seL4_TCBBits, seL4_CapNull, 0, 0, seL4_CapNull, 1);
    ZF_LOGF_IF(result, "Failed to retype thread: %d", result);
    seL4_DebugDumpScheduler();
```

Once the TCB has been created it will show up in the `seL4_DebugDumpScheduler()` output as `child of: 'tcb_threads'`. Throughout the tutorial you can use this syscall to debug some of the TCB attributes that you set.

After the scheduler table, you should see a another error:

```
  <<seL4(CPU 0) [decodeInvocation/530 T0xffffff800813fc00 "tcb_threads" @4004bf]:
Attempted to invoke a null cap #0.>>
main@threads.c:46 [Cond failed: result]
    Failed to configure thread: 2
```

Exercise Now that you have a TCB object, configure it to have the same CSpace and VSpace as the current thread. Use the IPC buffer we have provided, but don't set a fault handler, as the kernel will print any fault we

receive with a debug build.

```
    //TODO fix the parameters in this invocation
    result = seL4_TCB_Configure(seL4_CapNull, seL4_CapNull, 0, seL4_CapNull, 0, 0,
(seL4_Word) NULL, seL4_CapNull);
    ZF_LOGF_IF(result, "Failed to configure thread: %d", result);
```

You should now be getting the following error:

```
<<seL4(CPU 0) [decodeSetPriority/1035 T0xffffff8008140c00 "tcb_threads" @4012ef]:
Set priority: author>
main@threads.c:51 [Cond failed: result]
Failed to set the priority for the new TCB object.
```

Change priority via `seL4_TCB_SetPriority`

A newly created thread will have a priority of 0, while the thread created by the loader is at a priority of 254. You need to change the priority of your new thread such that it will be scheduled round-robin with the current thread.

Exercise use `seL4_TCB_SetPriority` to set the priority. Remember that to set a thread's priority, the calling thread must have the authority to do so. In this case, the main thread can use its own TCB capability, which has an MCP of 254.

```
    // TODO fix the call to set priority using the authority of the current thread
    // and change the priority to 254
    result = seL4_TCB_SetPriority(tcb_cap_slot, seL4_CapNull, 0);
    ZF_LOGF_IF(result, "Failed to set the priority for the new TCB object.\n");
    seL4_DebugDumpScheduler();
```

Fixing up the `seL4_TCB_SetPriority` call should allow you to see that the thread's priority is now set to the same as the main thread in the next `seL4_DebugDumpScheduler()` call.

```
Name                                     State           IP
Prio    Core
-------------------------------------------------------------------------------
----
child of: 'tcb_threads'                  inactive        (nil)   254
0
tcb_threads                              running         0x4012ef    254
0
idle_thread                               idle            (nil)   0
0
rootserver                                inactive        0x4024c2    255
0
```

```
<<seL4(CPU 0) [decodeInvocation/530 T0xffffff8008140c00 "tcb_threads" @4012ef]:
Attempted to invoke a >
main@threads.c:57 [Err seL4_InvalidCapability]:
Failed to write the new thread's register set.
```

Set initial register state

The TCB is nearly ready to run, except for its initial registers. You need to set the program counter and stack pointer to valid values, otherwise your thread will crash immediately.

`libsel4utils` contains some functions for setting register contents in a platform agnostic manner. You can use these methods to set the program counter (instruction pointer) and stack pointer in this way. *Note: It is assumed that the stack grows downwards on all platforms.*

Exercise Set up the new thread to call the function `new_thread`. You can use the debug syscall to verify that you have at least set the instruction pointer (IP) correctly.

```
    seL4_UserContext regs = {0};
    int error = seL4_TCB_ReadRegisters(tcb_cap_slot, 0, 0,
sizeof(regs)/sizeof(seL4_Word), &regs);
    ZF_LOGF_IFERR(error, "Failed to read the new thread's register set.\n");

    // TODO use valid instruction pointer
    sel4utils_set_instruction_pointer(&regs, (seL4_Word)NULL);
    // TODO use valid stack pointer
    sel4utils_set_stack_pointer(&regs, NULL);
    // TODO fix parameters to this invocation
    error = seL4_TCB_WriteRegisters(seL4_CapNull, 0, 0, 0, &regs);
    ZF_LOGF_IFERR(error, "Failed to write the new thread's register set.\n"
                  "\tDid you write the correct number of registers? See arg4.\n");
    seL4_DebugDumpScheduler();
```

On success, you will see the following output:

```
<<seL4(CPU 0) [decodeInvocation/530 T0xffffff800813fc00 "tcb_threads" @4004bf]:
Attempted to invoke a null cap #0.>>
main@threads.c:63 [Err seL4_InvalidCapability]:
    Failed to start new thread.
```

Start the thread

Finally you are ready to start the thread, which makes the TCB runnable and eligible to be picked by the seL4 scheduler. This can be done by changing the second argument of `seL4_TCB_WriteRegisters` to 1 and removing the `seL4_TCB_Resume` call, or by fixing the resume call below.

Exercise resume the new thread.

```
    // TODO resume the new thread
    error = seL4_TCB_Resume(seL4_CapNull);
    ZF_LOGF_IFERR(error, "Failed to start new thread.\n");
```

If everything has been configured correctly, resuming the thread should result in the string `Hello2: arg1 0, arg2 0, arg3 0` followed by a fault.

Passing arguments

You will notice that all of the arguments to the new thread are 0. You can set the arguments by using the helper function `sel4utils_arch_init_local_context` or by directly manipulating the registers for your target architecture.

Exercise update the values written with `seL4_TCB_WriteRegisters` to pass the values 1, 2, 3 as arg1, arg2, and arg3 respectively.

```
    UNUSED seL4_UserContext regs = {0};
    int error = seL4_TCB_ReadRegisters(tcb_cap_slot, 0, 0,
sizeof(regs)/sizeof(seL4_Word), &regs);
    ZF_LOGF_IFERR(error, "Failed to write the new thread's register set.\n"
                  "\tDid you write the correct number of registers? See arg4.\n");

    sel4utils_arch_init_local_context((void*)new_thread,
                                      (void *)1, (void *)2, (void *)3,
                                      (void *)tcb_stack_top, &regs);
    error = seL4_TCB_WriteRegisters(tcb_cap_slot, 0, 0,
sizeof(regs)/sizeof(seL4_Word), &regs);
    ZF_LOGF_IFERR(error, "Failed to write the new thread's register set.\n"
                  "\tDid you write the correct number of registers? See arg4.\n");
```

Resolving a fault

At this point, you have created and configured a new thread, and given it initial arguments. The last part of this tutorial is what to do when your thread faults. We provide further detail on fault handling in a future tutorial, but for now you can rely on the kernel printing a fault message, as the thread you have created does not have a fault handler.

In the output below you can see a cap fault has occurred. The first part of the error is that the kernel was unable to send a fault to a fault handler as it is set to `(nil)`. The kernel then prints out the fault it was trying to send. In this case, the fault is a virtual memory fault. The new thread has tried to access data at address `0x2` which is an invalid and unmapped address. The output shows that the program counter of the thread when it faulted was `0x401e66`.

The fault status register is also output, which can be decoded by using the relevant architecture manual. Additionally, the kernel prints a raw stack dump from the current stack pointer. The size of the stack dump is configurable, using the `KernelUserStackTraceLength` cmake variable.

```
Caught cap fault in send phase at address (nil)
while trying to handle:
vm fault on data at address 0x2 with status 0x4
in thread 0xffffff8008140400 "child of: 'tcb_threads'" at address 0x401e66
With stack:
0x439fc0: 0x0
0x439fc8: 0x3
0x439fd0: 0x2
0x439fd8: 0x1
0x439fe0: 0x0
0x439fe8: 0x1
0x439ff0: 0x0
0x439ff8: 0x0
0x43a000: 0x404fb3
0x43a008: 0x0
0x43a010: 0x0
0x43a018: 0x0
0x43a020: 0x0
0x43a028: 0x0
0x43a030: 0x0
0x43a038: 0x0
```

To investigate the fault, you can use a tool such as `objdump` on the ELF file that was loaded to inspect the instruction that caused the fault. In this case, the ELF file is located at `./<BUILD_DIR>/<TUTORIAL_BUILD_DIR>/threads`.

You should be able to see that `arg2` is being dereferenced, but does not point to valid memory.

Exercise pass a valid arg2, by passing the address of a global variable.

Next, another fault will occur as the new thread expects `arg1` to be a pointer to a function.

Exercise Pass the address of a function which outputs the argument which is passed to it, as `arg2`.

Now you should have a new thread, which immediately calls the function passed in `arg2`.

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 TCBs and threading in seL4.

- Using different TCB invocations to change the new thread's attributes or objects
- Investigate how setting different priorities affects when the threads are scheduled to run
- Implementing synchronisation primitives using global memory.
- Trying to repeat this tutorial in the root task where there are more resources available to create more thread objects.
- Another tutorial...

Getting help

Stuck? See the resources below.

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