seL4 Dynamic Libraries: initialisation & threading

This tutorial provides code examples and exercises for using the dynamic libraries found in `seL4_libs` to bootstrap a system and start a thread.

The tutorial is useful in that it addresses conceptual problems for two different types of developers:

- Experienced kernel developers whose minds are pre-programmed to think in terms of "One address space equals one process", and begins to introduce the seL4 CSpace vs VSpace model.
- New kernel developers, for whom the tutorial will prompt them on what to read about.

Don't gloss over the globals declared before `main()` -- they're declared for your benefit so you can grasp some of the basic data structures.

Prerequisites

1. Set up your machine.
2. Hello world

Outcomes:

- Understand the kernel's startup procedure.
- Understand that the kernel centers around certain objects and capabilities to those objects.
- Understand that libraries exist to automate the very fine-grained nature of the seL4 API, and get a rough idea of some of those libraries.
- Learn how the kernel hands over control to userspace.
- Get a feel for how the seL4 API enables developers to manipulate the objects referred to by the capabilities they encounter.
- Understand the how to spawn new threads in seL4, and the basic idea that a thread has a TCB, VSpace and CSpace, and that you must fill these out.

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 dynamic-1
# building the tutorial exercise
cd dynamic-1_build
ninja
```

Exercises

When you first run the tutorial, you should see the following output:

```
Booting all finished, dropped to user space
main@main.c:89 [Cond failed: info == NULL]
Failed to get bootinfo.
```

Obtain BootInfo

After bootstrapping the system, the seL4 kernel hands over control to the root task. to an init thread. This thread receives a structure from the kernel that describes all the resources available on the machine. This structure is called the BootInfo structure. It includes information on all IRQs, memory, and IO-Ports (x86). This structure also tells the init thread where certain important capability references are. This step is teaching you how to obtain that structure.

`seL4_BootInfo* platsupport_get_bootinfo(void)` is a function that returns the BootInfo structure. It also sets up the IPC buffer so that it can perform some syscalls such as `seL4_DebugNameThread` used by `name_thread`.

- https://github.com/seL4/seL4/blob/master/libsel4/include/sel4/bootinfo_types.h
- https://github.com/seL4/seL4_libs/blob/master/libsel4platsupport/src/bootinfo.c

```
    /* TASK 1: get boot info */
    /* hint: platsupport_get_bootinfo()
     * seL4_BootInfo* platsupport_get_bootinfo(void);
     * @return Pointer to the bootinfo, NULL on failure
     */
}
```

On success, you should see the following:

```
dynamic-1: main@main.c:124 [Cond failed: allocman == NULL]
    Failed to initialize alloc manager.
    Memory pool sufficiently sized?
    Memory pool pointer valid?
```

Initialise simple

`libsel4simple` provides an abstraction for the boot environment of a thread. You need to initialize it with some default state before using it.

- https://github.com/seL4/seL4_libs/blob/master/libsel4simple-default/include/simple-default/simple-default.h

```
   /* TASK 2: initialise simple object */
   /* hint: simple_default_init_bootinfo()
```

```
     * void simple_default_init_bootinfo(simple_t *simple, seL4_BootInfo *bi);
     * @param simple Structure for the simple interface object. This gets
initialised.
     * @param bi Pointer to the bootinfo describing what resources are available
     */
```

On successful completion of this task, the output should not change.

Use simple to print BootInfo

Use a `simple` function to print out the contents of the `seL4_BootInfo` function.

```
    /* TASK 3: print out bootinfo and other info about simple */
    /* hint: simple_print()
     * void simple_print(simple_t *simple);
     * @param simple Pointer to simple interface.
     */
```

- https://github.com/seL4/seL4_libs/blob/master/libsel4simple/include/simple/simple.h

The error message should remain, but your output should now also contain something like:

```
Node 0 of 1
IOPT levels:     4294967295
IPC buffer:      0x52c000
Empty slots:     [406 --> 4096)
sharedFrames:    [0 --> 0)
userImageFrames: [16 --> 316)
userImagePaging: [12 --> 15)
untypeds:        [316 --> 406)
Initial thread domain: 0
Initial thread cnode size:
dynamic-1: main@main.c:126 [Cond failed: allocman == NULL]
```

Initialise an allocator

In seL4, memory management is delegated in large part to userspace, and each thread manages its own page faults with a custom pager. Without the use of the `allocman` library and the `VKA` library, you would have to manually allocate a frame, then map the frame into a page-table, before you could use new memory in your address space. In this tutorial you don't go through that procedure, but you'll encounter it later. For now, use the allocman and VKA allocation system. The allocman library requires some initial memory to bootstrap its metadata. Complete this step.

- https://github.com/seL4/seL4_libs/blob/master/libsel4allocman/include/allocman/bootstrap.h

```
    /* TASK 4: create an allocator */
    /* hint: bootstrap_use_current_simple()
     * allocman_t *bootstrap_use_current_simple(simple_t *simple, uint32_t
pool_size, char *pool);
     * @param simple Pointer to simple interface.
     * @param pool_size Size of the initial memory pool.
     * @param pool Initial memory pool.
     * @return returns NULL on error
     */
```

The output should now be as follows:

```
<<seL4(CPU 0) [decodeInvocation/530 T0xffffff801ffb5400 "dynamic-1" @401303]:
Attempted to invoke a null cap #0.>>
dynamic-1: main@main.c:199 [Err seL4_InvalidCapability]:
    Failed to set the priority for the new TCB object.
```

Obtain a generic allocation interface (vka)

`libsel4vka` is an seL4 type-aware object allocator that will allocate new kernel objects for you. The term "allocate new kernel objects" in seL4 is a more detailed process of "retyping" previously un-typed memory. seL4 considers all memory that hasn't been explicitly earmarked for a purpose to be "untyped", and in order to repurpose any memory into a useful object, you must give it an seL4-specific type. This is retyping, and the VKA library simplifies this for you, among other things.

- https://github.com/seL4/seL4_libs/blob/master/libsel4allocman/include/allocman/vka.h

```
    /* TASK 5: create a vka (interface for interacting with the underlying
allocator) */
    /* hint: allocman_make_vka()
     * void allocman_make_vka(vka_t *vka, allocman_t *alloc);
     * @param vka Structure for the vka interface object.  This gets initialised.
     * @param alloc allocator to be used with this vka
     */
```

On successful completion this task, the output should not change.

Find the CSpace root cap

```
    /* TASK 6: get our cspace root cnode */
    /* hint: simple_get_cnode()
     * seL4_CPtr simple_get_cnode(simple_t *simple);
     * @param simple Pointer to simple interface.
```

```
     * @return The cnode backing the simple interface. no failure.
     */
    seL4_CPtr cspace_cap;
```

This is where the differences between seL4 and contemporary kernels begin to start playing out. Every kernel-object that you "retype" will be handed to you using a capability reference. The seL4 kernel keeps multiple trees of these capabilities. Each separate tree of capabilities is called a "CSpace". Each thread can have its own CSpace, or a CSpace can be shared among multiple threads. The delineations between "Processes" aren't well-defined, since seL4 doesn't have any real concept of "processes". It deals with threads. Sharing and isolation is based on CSpaces (shared vs not-shared) and VSpaces (shared vs not-shared). The "process" idea goes as far as perhaps the fact that at the hardware level, threads sharing the same VSpace are in a traditional sense, siblings, but logically in seL4, there is no concept of "Processes" really.

So you're being made to grab a reference to your thread's CSpace's root "CNode". A CNode is one of the many blocks of capabilities that make up a CSpace.

- https://github.com/seL4/seL4_libs/blob/master/libsel4simple/include/simple/simple.h

On successful completion this task, the output should not change.

Find the VSpace root cap

```
    /* TASK 7: get our vspace root page diretory */
    /* hint: simple_get_pd()
     * seL4_CPtr simple_get_pd(simple_t *simple);
     * @param simple Pointer to simple interface.
     * @return The vspace (PD) backing the simple interface. no failure.
     */
    seL4_CPtr pd_cap;
```

Just as in the previous step, you were made to grab a reference to the root of your thread's CSpace, now you're being made to grab a reference to the root of your thread's VSpace.

- https://github.com/seL4/seL4_libs/blob/master/libsel4simple/include/simple/simple.h

On successful completion this task, the output should not change.

Allocate a TCB Object

```
    /* TASK 8: create a new TCB */
    /* hint: vka_alloc_tcb()
     * int vka_alloc_tcb(vka_t *vka, vka_object_t *result);
     * @param vka Pointer to vka interface.
     * @param result Structure for the TCB object.  This gets initialised.
     * @return 0 on success
```

```
     */
    vka_object_t tcb_object = {0};
```

In order to manage the threads that are created in seL4, the seL4 kernel keeps track of TCB (Thread Control Block) objects. Each of these represents a schedulable executable resource. Unlike other contemporary kernels, seL4 **doesn't** allocate a stack, virtual-address space (VSpace) and other metadata on your behalf. This step creates a TCB, which is a very bare-bones, primitive resource, which requires you to still manually fill it out.

- https://github.com/seL4/seL4_libs/blob/master/libsel4vka/include/vka/object.h

After completing this task, the errors should disappear, and you should see the following output:

```
main: hello world
```

Configure the new TCB

```
   /* TASK 9: initialise the new TCB */
    /* hint 1: seL4_TCB_Configure()
     * int seL4_TCB_Configure(seL4_TCB _service, seL4_Word fault_ep, seL4_CNode
cspace_root, seL4_Word cspace_root_data, seL4_CNode vspace_root, seL4_Word
vspace_root_data, seL4_Word buffer, seL4_CPtr bufferFrame)
     * @param service Capability to the TCB which is being operated on.
     * @param fault_ep Endpoint which receives IPCs when this thread faults (must
be in TCB's cspace).
     * @param cspace_root The new CSpace root.
     * @param cspace_root_data Optionally set the guard and guard size of the new
root CNode. If set to zero, this parameter has no effect.
     * @param vspace_root The new VSpace root.
     * @param vspace_root_data Has no effect on IA-32 or ARM processors.
     * @param buffer Address of the thread's IPC buffer. Must be 512-byte aligned.
The IPC buffer may not cross a page boundary.
     * @param bufferFrame Capability to a page containing the thread?s IPC buffer.
     * @return 0 on success.
     * Note: this function is generated during build.  It is generated from the
following definition:
     *
     * hint 2: use seL4_CapNull for the fault endpoint
     * hint 3: use seL4_NilData for cspace and vspace data
     * hint 4: we don't need an IPC buffer frame or address yet
     */
```

You must create a new VSpace for your new thread if you need it to execute in its own isolated address space, and tell the kernel which VSpace you plan for the new thread to execute in. This opens up the option for threads to share VSpaces. In similar fashion, you must also tell the kernel which CSpace your new thread will

use -- whether it will share a currently existing one, or whether you've created a new one for it. That's what you're doing now.

In this particular example, you're allowing the new thread to share your main thread's CSpace and VSpace.

In addition, a thread needs to have a priority set on it in order for it to run. `seL4_TCB_SetPriority(tcb_object.cptr, seL4_CapInitThreadTCB, seL4_MaxPrio);` will give your new thread the same priority as the current thread, allowing it to be run the next time the seL4 scheduler is invoked. The seL4 scheduler is invoked everytime there is a kernel timer tick.

- https://github.com/seL4/seL4/blob/master/libsel4/include/interfaces/sel4.xml

On successful completion this task, the output should not change.

Name the new TCB

```
    /* TASK 10: give the new thread a name */
    /* hint: we've done thread naming before */
```

This is a convenience function -- sets a name string for the TCB object.

On successful completion this task, the output should not change.

Set the instruction pointer

```
    /*
     * set start up registers for the new thread:
     */
    UNUSED seL4_UserContext regs = {0};

    /* TASK 11: set instruction pointer where the thread shoud start running */
    /* hint 1: sel4utils_set_instruction_pointer()
     * void sel4utils_set_instruction_pointer(seL4_UserContext *regs, seL4_Word
value);
     * @param regs Data structure in which to set the instruction pointer value
     * @param value New instruction pointer value
     *
     * hint 2: we want the new thread to run the function "thread_2"
     */
```

Pay attention to the line that precedes this particular task -- the line that zeroes out a new "seL4_UserContext" object. As we previously explained, seL4 requires you to fill out the Thread Control Block manually. That includes the new thread's initial register contents. You can set the value of the stack pointer, the instruction pointer, and if you want to get a little creative, you can pass some initial data to your new thread through its registers.

- https://github.com/seL4/seL4_libs/blob/master/libsel4utils/sel4_arch_include/x86_64/sel4utils/sel4_arch/util.h

On successful completion this task, the output should not change.

Set the stack pointer

```
    /* TASK 12: set stack pointer for the new thread */
    /* hint 1: sel4utils_set_stack_pointer()
     * void sel4utils_set_stack_pointer(seL4_UserContext *regs, seL4_Word value);
     * @param regs  Data structure in which to set the stack pointer value
     * @param value New stack pointer value
     *
     * hint 2: remember the stack grows down!
     */
```

This TASK is just some pointer arithmetic. The cautionary note that the stack grows down is meant to make you think about the arithmetic. Processor stacks push new values toward decreasing addresses, so give it some thought.

- https://github.com/seL4/seL4_libs/blob/master/libsel4utils/sel4_arch_include/x86_64/sel4utils/sel4_arch/util.h

On successful completion this task, the output should not change.

Write the registers

```
    /* TASK 13: actually write the TCB registers.  We write 2 registers:
     * instruction pointer is first, stack pointer is second. */
    /* hint: seL4_TCB_WriteRegisters()
     * int seL4_TCB_WriteRegisters(seL4_TCB service, seL4_Bool resume_target,
seL4_Uint8 arch_flags, seL4_Word count, seL4_UserContext *regs)
     * @param service Capability to the TCB which is being operated on.
     * @param resume_target The invocation should also resume the destination
thread.
     * @param arch_flags Architecture dependent flags. These have no meaning on
either IA-32 or ARM.
     * @param count The number of registers to be set.
     * @param regs Data structure containing the new register values.
     * @return 0 on success
     */
```

As explained above, we've been filling out our new thread's TCB for the last few operations, so now we're writing the values we've chosen, to the TCB object in the kernel.

- https://github.com/seL4/seL4/blob/master/libsel4/include/interfaces/sel4.xml

On successful completion this task, the output should not change.

Start the new thread

```
    /* TASK 14: start the new thread running */
    /* hint: seL4_TCB_Resume()
     * int seL4_TCB_Resume(seL4_TCB service)
     * @param service Capability to the TCB which is being operated on.
     * @return 0 on success
     */
```

Finally, we tell the kernel that our new thread is runnable. From here, the kernel itself will choose when to run the thread based on the priority we gave it, and according to the kernel's configured scheduling policy.

- https://github.com/seL4/seL4/blob/master/libsel4/include/interfaces/sel4.xml

On successful completion this task, the output should not change.

Print something

```
    /* TASK 15: print something */
    /* hint: printf() */
}
```

For the sake of confirmation that our new thread was executed by the kernel successfully, we cause it to print something to the screen.

On success, you should see output from your new thread.

Links to source

- `sel4_BootInfo`: https://github.com/seL4/seL4/blob/master/libsel4/include/sel4/bootinfo_types.h
- `simple_t`: https://github.com/seL4/seL4_libs/blob/master/libsel4simple/include/simple/simple.h
- `vka_t`: https://github.com/seL4/seL4_libs/blob/master//libsel4vka/include/vka/vka.h
- `allocman_t`: https://github.com/seL4/seL4_libs/blob/master/libsel4allocman/include/allocman/allocman.h
- `name_thread()`: https://github.com/SEL4PROJ/sel4-tutorials/blob/master/exercises/dynamic-1/src/util.c

That's it for this tutorial.

Getting help

Stuck? See the resources below.

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