Lab 4: Preemptive Multitasking

- **Handed out:** Tuesday, March 3, 2020
- **Due:** Monday, April 13, 2020

Introduction

In this assignment, you will enable user-level applications by implementing processes and related infrastructure. You will write privilege level switching code, context-switching code, a simple round-robin scheduler, system call handlers, and a virtual memory subsystem. You will also write several user programs and load them to start new processes.

Phase 0: Getting Started

Fetch the update for lab 4 from our git repository to your development machine.

```
$ git fetch skeleton
$ git merge skeleton/lab4
```

This is the directory structure of our repository. The directories you will be working on this assignment are marked with *.

```
.
├── bin : common binaries/utilities
├── doc : reference documents
├── ext : external files (e.g., resources for testing)
├── tut : tutorial/practices
│    ├── 0-rustlings
│    ├── 1-blinky
│    ├── 2-shell
│    ├── 3-fs
│    └── 4-spawn : questions for lab4 *
├── boot : bootloader
├── kern : the main os kernel *
├── lib  : required libraries
│     ├── aarch *
│     ├── kernel_api *
│     ├── fat32
│     ├── pi
│     ├── shim
│     ├── stack-vec
│     ├── ttywrite
│     ├── volatile
│     └── xmodem
└── user  : user level program *
      ├── fib *
      └── sleep *
```

You may need to resolve conflicts before continuing. For example, if you see a message that looks like:

```
Auto-merging kern/src/main.rs
CONFLICT (content): Merge conflict in kern/src/main.rs
Automatic merge failed; fix conflicts and then commit the result.
```

You will need to manually modify the `main.rs` file to resolve the conflict. Ensure you keep all of your changes from lab 3. Once all conflicts are resolved, add the resolved files with `git add` and commit. For more information on resolving merge conflicts, see this tutorial on githowto.com.

ARM Documentation

Throughout this assignment, we will be referring to three official ARM documents. They are:

- ARMv8 Reference Manual

  This is the official reference manual for the ARMv8 architecture. This is a wholistic manual covering the entire architecture in a general manner. For the specific implementation of the architecture for the Raspberry Pi 3, see the ARM Cortex-A53 Manual. We will be referring to sections from this manual with notes of the form (ref: C5.2) which indicates that you should refer to section C5.2 of the ARMv8 Reference Manual.

- ARM Cortex-A53 Manual

  Manual for the specific implementation of the ARMv8 (v8.0-A) architecture as used by the Raspberry Pi 3. We will be referring to sections from this manual with notes of the form (A53: 4.3.30) which indicates that you should refer to section 4.3.30 of the ARM Cortex-A53 Manual.

- ARMv8-A Programmer Guide

  A high-level guide on how to program an ARMv8-A process. We will be referring to sections from this manual with notes of the form (guide: 10.1) which indicates that you should refer to section 10.1 of the ARMv8-A Programmer Guide.

You can find all those three documents under `doc/` subdirectory of our lab repo. We recommend that you download these three documents now and maintain them within easy reach.

Phase 1: ARM and a Leg

In this phase, you will learn about the ARMv8 architecture, switch to a lower privilege level, install exception vectors, enable timer interrupts, and handle breakpoint exceptions by starting a debug shell. You will learn about exception levels in the ARM architecture and how the architecture handles exceptions, interrupts, and privilege levels.

Subphase A: ARMv8 Overview

In this subphase, you will learn about the ARMv8 architecture. You will not be writing any code, but you will be answering several questions about the architecture.

The ARM (Acorn RISC Machine) CPU architecture has a history spanning over 30 years. There are eight major revisions, the latest being ARMv8-A, introduced in 2011. The Broadcom BCM2837 SOC contains an ARM Cortex-A53, an ARMv8.0-A based CPU. The Cortex-A53 (and other specific CPUs) are referred to as *implementations* of the architecture. This is the CPU you have been programming in the last three assignments.

ARM CPUs dominate the mobile market.

ARM CPUs dominate the mobile market, appearing in over 95% of all smartphones sold worldwide and almost 100% of all flagship smartphones including Apple's iPhone and Google's Pixel.

Thus far, we've avoided the details of the underlying architecture, allowing the Rust compiler to handle them for us. To enable running processes in user-space, however, we'll need to program the CPU directly at the lowest of levels. Programming the CPU directly requires familiarization with the CPUs native assembly language and overall concepts. We'll start with an overview of the architecture and then proceed to describe a few key assembly instructions.

Registers

The ARMv8 architecture includes the following registers (ref: B1.2.1):

- `r0` ... `r30` - 64-bit general purpose registers

  These registers are accessed via aliases. The registers `x0` ... `x30` alias all 64-bits of these registers. The registers `w0` ... `w30` alias the least-significant 32-bits of these registers.

- `lr` - 64-bit link register; aliases `x30`

  Used to store the *link address*. The `bl <addr>` instruction stores the address of the next instruction in `lr` and branches to `addr`. The `ret` instruction sets the PC to the address in `lr`.

- `sp` - a dedicated stack pointer

  The lower 32 bits of the stack-pointer can be accessed via `wsp`. The stack pointer must always be 16-byte aligned.

- `pc` - the program counter

  This register can be read but not written. The `pc` is updated on branching instructions and exception entry/return.

- `v0` ... `v31` - 128-bit SIMD and FP point registers

  These registers are used for vectorizing SIMD instruction and floating point operations. These registers are accessed via aliases. The registers `q0` ... `q31` alias all 128-bits of these registers. The registers `d0` ... `d31` alias the lower 64-bits of these registers. There are also alias for the lower 32, 16, and 8 bits of these registers prefixed with `s`, `h`, and `b`, respectively.

- `xzr` - read-only zero register

  This pseudo-register, which may or may not be a hardware register, always holds the value `0`.

There are also *many special-purpose* registers. We'll describe these as needed, as in the next section.

PSTATE

At any point in time, an ARMv8 CPU captures the *program state* in a pseudo-register named PSTATE (ref: D1.7). PSTATE isn't a real register; there's no way to read or write it directly. Instead, there are special purpose registers that can be used to read or write the various fields of the PSTATE pseudo-register. On ARMv8.0, these are:

- `NZCV` - condition flags
- `DAIF` - exception mask bits, used to prevent exceptions from being issued
- `CurrentEL` - the current exception level (explained later)
- `SPSel` - stack pointer selector

These registers belong to the class of registers known as *system registers* or *special registers* (ref: C5.2). Typically, registers can be loaded into from memory (written) using the `ldr` instruction and stored in memory (read) using the `str` instruction. System registers cannot be read/written with these instructions. Instead, the special purpose instructions `mrs` and `msr` must be used (ref: C6.2.162 - C6.2.164). For example to read `NZCV` into `x1`, you can issue the instruction:

```
mrs x1, NZCV
```

Execution State

At any point in time, an ARMv8 CPU is executing in a given *execution state*. There are two such execution states: AArch32, corresponding to 32-bit ARMv7 compatibility mode, and AArch64, 64-bit ARMv8 mode (guide: 3.1). We'll always be executing in AArch64.

Secure Mode

At any point in time, an ARMv8 CPU is executing in a given *security state*, otherwise known as a *security mode* or *security world*. There are two security states: *secure*, and *non-secure*, also known as *normal*. We'll always be executing in non-secure (normal) mode.

Exception Levels

At any point in time, an ARMv8 CPU is executing at a given *exception level* (guide: 3). Each exception level corresponds to a *privilege level*: the higher the exception level, the greater the privileges programs running at that level have. There are 4 exception levels:

- **EL0 (user)** - Typically used to run untrusted user applications.
- **EL1 (kernel)** - Typically used to run privileged operating system kernels.
- **EL2 (hypervisor)** - Typically used to run virtual machine hypervisors.
- **EL3 (monitor)** - Typically used to run low-level firmware.

The Raspberry Pi's CPU boots into EL3. At that point, the firmware provided by the Raspberry Pi foundation runs, switches to EL2, and runs our `kernel8.img` file. Thus, *our* kernel starts executing in EL2. Later, you'll switch from EL2 to EL1 so that our kernel is running in the appropriate exception level.

ELx Registers

Several system registers such as `ELR`, `SPSR`, and `SP` are duplicated for each exception level. The register names are suffixed with `_ELn` to indicate the register for exception level `n`. For instance, `ELR_EL1` is the *exception link register* for EL1, while `ELR_EL2` is the exception link register for EL2.

We use the suffix `x`, such as in `ELR_ELx`, when we refer to a register from the *target* exception level `x`. The *target* exception level is the exception level the CPU will switch to, if necessary, to run the exception vector. We use the suffix `s`, such as in `SP_ELs`, when we refer to a register in the *source* exception level `s`. The *source* exception level is the exception level in which the CPU was executing when the exception occurred.

Switching Exception Levels

There is exactly one mechanism to increase the exception level and exactly one mechanism to decrease the exception level.

To switch from a higher level to a lower level (a privilege decrease), the running program must *return* from the exception level using the `eret` instruction (ref: D1.11). On executing an `eret` instruction when the current exception level is `ELx`, the CPU:

- Sets the PC to the value in `ELR_ELx`, a special purpose system-register.
- Sets the PSTATE to the values in `SPSR_ELx`, a special purpose system-register.

The `SPSR_ELx` register (ref: C5.2.18) also contains the exception level to return to. Note that changing exception levels also has the following implications:

- On return to `ELs`, the `sp` is set to `SP_ELs` if `SPSR_ELx[0] == 1` or `SP_EL0` if `SPSR_ELx[0] == 0`.

Switching from a lower level to a higher level only occurs as a result of an exception (guide: 10). Unless otherwise configured, the CPU will trap exceptions to the next exception level. For instance, if an interrupt is received while running in EL0, the CPU will switch to EL1 to handle the exception. When a switch to `ELx` occurs, the CPU will:

- Mask all exceptions and interrupts by setting `PSTATE.DAIF = 0b1111`.
- Save `PSTATE` and other fields to `SPSR_ELx`.
- Save the *preferred exception link address* to `ELR_ELx` (ref: D1.10.1).
- Set `sp` to `SP_ELx` if `SPSel` was set to `1`.
- Save the *exception syndrome* (described later) to `ESR_ELx` (ref: D1.10.4).
- Set `pc` to address corresponding to the exception vector (described later).

Note that the exception syndrome register is only valid when the exception was *synchronous* (described next). All general purpose and SIMD/FP registers will maintain the value they had when the exception occurred.

Exception Vectors

When an exception occurs, the CPU jumps to the *exception vector* for that exception (ref: D1.10.2). There are 4 types of exceptions each with 4 possible exception sources for a total of 16 exception vectors. The four types of exceptions are:

- **Synchronous** - an exception resulting from an instruction like `svc` or `brk`
- **IRQ** - an asynchronous interrupt request from an external source
- **FIQ** - an asynchronous *fast* interrupt request from an external source
- **SError** - a “system error” interrupt

The four sources are:

- Same exception level when source `SP = SP_EL0`
- Same exception level when source `SP = SP_ELx`
- Lower exception level running on AArch64
- Lower exception level running on AArch32

As described in (guide: 10.4):

> When an exception occurs, the processor must execute handler code which corresponds to the exception. The location in memory where [an exception] handler is stored is called the *exception vector*. In the ARM architecture, exception vectors are stored in a table, called the *exception vector table*. Each exception level has its own vector table, that is, there is one for each of EL3, EL2 and EL1. The table contains instructions to be executed, rather than a set of addresses [as in x86]. Each entry in the vector table is 16 instructions long. Vectors for individual exceptions are located at fixed offsets from the beginning of the table. The virtual address of each table base is set by the [special-purpose] Vector Based Address Registers `VBAR_EL3`, `VBAR_EL2` and `VBAR_EL1`.

The vectors are physically laid out as follows:

- Target and source at same exception level with source `SP = SP_EL0`:

`VBAR_ELx` Offset	Exception
0x000	Synchronous exception
0x080	IRQ
0x100	FIQ
0x180	SError

- Target and source at same exception level with source `SP = SP_ELx`:

`VBAR_ELx` Offset	Exception
0x200	Synchronous exception
0x280	IRQ
0x300	FIQ
0x380	SError

- Source is at lower exception level running on AArch64

`VBAR_ELx` Offset	Exception
0x400	Synchronous exception
0x480	IRQ
0x500	FIQ
0x580	SError

- Source is at lower exception level running on AArch32

`VBAR_ELx` Offset	Exception
0x600	Synchronous exception
0x680	IRQ
0x700	FIQ
0x780	SError

The vector table is contiguous.

Interface with Rust

We've provided `aarch64` library (`lib/aarch64`) to provide Rusty interface in order to access low-level details about the system. Before moving to next subphase, please compare your understanding with registers defined in `regs.rs`. The other files in the library will be revisited at the end of the next subphase.

Recap

For now, this is all of the ARMv8 architecture that you need to know. Before continuing, answer the following questions:

Which registers alias `x30`? (arm-x30)

If a value of `0xFFFF` is written to `x30`, which two other registers names can be used to retrieve that value?

How would you set the PC to a specific address? (arm-pc)

How would you set the PC to the address `A` using the `ret` instruction? How would you set the PC to the address `A` using the `eret` instruction? Be specific about which registers you would set to which values.

How would you determine the current exception level? (arm-el)

Which instructions, exactly, would you run to determine the current exception level?

How would you change the stack pointer on exception return? (arm-sp-el)

The stack pointer of a running program is `A` when an exception occurs. After handling the exception, you'd like to return back to where the program was executing but want to change its stack pointer to `B`. How would you do so?

Which vector is used for system calls from a lower EL? (arm-svc)

A process is running in `EL0` when it issues an `svc` instruction. To which address, exactly, does the CPU jump to?

Which vector is used for interrupts from a lower EL? (arm-int)

A process is running in `EL0` when a timer interrupt occurs. To which address, exactly, does the CPU jump to?

How do you unmask IRQ exceptions? (arm-mask)

Which values would you write to which register to unmask IRQ interrupts only?

How would you `eret` into an AArch32 execution state? (arm-aarch32)

An exception has occurred with the source running in the AArch64 state. The target is also running in AArch64. Which values in which registers would you change so that on return from the exception via `eret`, the CPU switches to the AArch32 execution state?

Hint

See (guide: 10.1).

Subphase B: Instructions

In this subphase, you will learn about the ARMv8 instruction set. You will not be writing any code, but you will be answering several questions about the instruction set.

Accessing Memory

ARMv8 is a load/store RISC (reduced instruction set computer) instruction set. Perhaps the defining feature of such an instruction set is that memory can only be accessed through specific instructions. In particular, memory can only be read by reading into a register with a load instruction and written to memory by storing from a register using a store instruction.

There are many load and store instructions and variations of particular instructions. We'll start with the simplest:

- `ldr <ra>, [<rb>]`: load value from address in `<rb>` into `<ra>`
- `str <ra>, [<rb>]`: store value in `<ra>` to address in `<rb>`

The register `<rb>` is known as the *base register*. Thus, if `r3 = 0x1234`, then:

```
ldr r0, [r3]      // r0 = *r3 (i.e., r0 = *(0x1234))
str r0, [r3]      // *r3 = r0 (i.e., *(0x1234) = r0)
```

You can also provide an offset in the range `[-256, 255]`:

```
ldr r0, [r3, #64]      // r0 = *(r3 + 64)
str r0, [r3, #-12]     // *(r3 - 12) = r0
```

You can also provide a post-index that changes the value in the base register *after* the load or store has been applied:

```
ldr r0, [r3], #30      // r0 = *r3; r3 += 30
str r0, [r3], #-12     // *r3 = r0; r3 -= 12
```

You can also provide a pre-index that changes the value in the base register *before* the load or store has been applied:

```
ldr r0, [r3, #30]!     // r3 += 30; r0 = *r3
str r0, [r3, #-12]!    // r3 -= 12; *r3 = r0
```

Offset, post-index, and pre-index are known as *addressing modes*.

Finally, you can load and store from two registers at once using the `ldp` and `stp` (load pair, store pair) instructions. These instructions can be used with all of the same addressing modes as `ldr` and `str`:

```
// push `x0` and `x1` onto the stack. after this operation the stack is:
//
//    |------| <x (original SP)
//    |  x1  |
//    |------|
//    |  x0  |
//    |------| <- SP
//
stp x0, x1, [SP, #-16]!

// pop `x0` and `x1` from the stack. after this operation, the stack is:
//
//    |------| <- SP
//    |  x1  |
//    |------|
//    |  x0  |
//    |------| <x (original SP)
//
ldp x0, x1, [SP], #16

// these four operations perform the same thing as the previous two
sub SP, SP, #16
stp x0, x1, [SP]
ldp x0, x1, [SP]
add SP, SP, #16

// same as before, but we are saving and restoring all of x0, x1, x2, and x3.
sub SP, SP, #32
stp x0, x1, [SP]
stp x2, x3, [SP, #16]

ldp x0, x1, [SP]
ldp x2, x3, [SP, #16]
add SP, SP, #32
```

Loading Immediates

An *immediate* is another name for an integer whose value is known without any computation. To load a 16-bit immediate into a register, optionally shifted to the left by a multiple of 16-bits, use `mov` (move). To load a 16-bit immediate shifted by left some number of bits without replacing any of the other bits, use the `movk` (move/keep) instruction. An example of their usage:

```
mov   x0, #0xABCD, LSL #32  // x0 = 0xABCD00000000
mov   x0, #0x1234, LSL #16  // x0 = 0x12340000

mov   x1, #0xBEEF           // x1 = 0xBEEF
movk  x1, #0xDEAD, LSL #16  // x1 = 0xDEADBEEF
movk  x1, #0xF00D, LSL #32  // x1 = 0xF00DDEADBEEF
movk  x1, #0xFEED, LSL #48  // x1 = 0xFEEDF00DDEADBEEF
```

Note that immediates are prefixed with a `#`, that the destination register appears to the left, and that `LSL` specifies the left shift.

Only 16-bit immediates with optional shifts can be loaded into a register. The assembler is able to figure out the right shift value in many cases. For instance, the assembler is able to convert `mov x12, #(1 << 21)` into a `mov x12, 0x20, LSL #16` automatically.

Loading Addresses from Labels

Sections of assembly code can be *labled* using `<label>:`:

```
add_30:
    add x1, x1, #10
    add x1, x1, #20
```

To load the address of the first instruction after the label, you can either use the `adr` or `ldr` instructions:

```
adr x0, add_30    // x0 = address of first instruction of add_30
ldr x0, =add_30   // x0 = address of first instruction of add_30
```

You *must* use `ldr` if the label is not within the same linker section as the instruction. If the label *is* within the same section, you should use `adr`.

Moving Between Registers

You can move values between registers with the `mov` instruction as well:

```
mov  x13, #23    //          x13 = 23
mov  sp, x13     // sp = 23, x13 = 23
```

Loading from Special Registers

Special and system registers such as `ELR_EL1` can only be loaded/stored from other registers using the `mrs` and `msr` instruction.

To write to a special register from another register, use `msr`:

```
msr ELR_EL1, x1  // ELR_EL1 = x1
```

To read from a special register into another register, use `mrs`:

```
mrs x0, CurrentEL // x0 = CurrentEL
```

Arithmetic

The `add` and `sub` instruction can be used to perform arithmetic. The syntax is:

```
add <dest> <a> <b> // dest = a + b
sub <dest> <a> <b> // dest = a - b
```

For example:

```
mov x2, #24
mov x3, #36
add x1, x2, x3  // x1 = 24 + 36 = 60
sub x4, x3, x2  // x4 = 36 - 24 = 12
```

The parameter `<b>` can also be an immediate:

```
sub sp, sp, #120 // sp -= 120
add x3, x1, #120 // x3 = x1 + 120
add x3, x3, #88  // x3 += 88
```

Logical Instructions

The `and` and `orr` instruction perform bitwise `AND` and `OR`. Their usage is identical to that of `add` and `sub`:

```
mov x1, 0b11001
mov x2, 0b10101

and x3, x1, x2  // x3 = x1 & x2 = 0b10001
orr x3, x1, x2  // x3 = x1 | x2 = 0b11101
orr x1, x1, x2  // x1 |= x2
and x2, x2, x1  // x2 &= x1

and x1, x1, #0b110  // x1 &= 0b110
orr x1, x1, #0b101  // x1 |= 0b101
```

Branching

Branching is another term for jumping to an address. A *branch* changes the PC to a given address or address of a label. To unconditionally jump to a label, use the `b` instruction:

```
b label // jump to label
```

To jump to a label while storing the next address in the link register, use `bl`. The `ret` instruction jumps to the address in `lr`:

```
my_function:
    add x0, x0, x1
    ret

mov  x0, #4
mov  x1, #30
bl   my_function  // lr = address of `mov x3, x0`
mov  x3, x0       // x3 = x0 = 4 + 30 = 34
```

The `br` and `blr` instruction are the same as `b` and `bl`, respectively, but jump to an address contained in a register:

```
ldr  x0, =label
blr  x0          // identical to bl label
br   x0          // identical to b  label
```

Conditional Branching

The `cmp` instruction compares values in two registers or a register and an immediate and sets flags for future conditional branching instructions such as `bne` (branch not equal), `beq` (branch if equal), `blt` (branch if less than), and so on (ref: C1.2.4):

```
// add 1 to x0 until it equals x1, then call `function_when_eq`, then exit
not_equal:
    add  x0, x0, #1
    cmp  x0, x1
    bne  not_equal
    bl   function_when_eq

exit:
    ...

// called when x0 == x1
function_when_eq:
    ret
```

Using an immediate:

```
cmp  x1, #0
beq  x1_is_eq_to_zero
```

Note that if the branch is not taken, execution simply continues forward.

Interface with Rust

Now go to the `aarch64` subdirectory again and start reviewing the remaining files. Here is brief description of each files in the library.

- `asm.rs` - Internally use of inline assembly to wrap as a function.
- `macros.rs` - Define macros used in this library
- `regs.rs` - Define registers and provide their interface using macros
- `sp.rs` - Access to the stack pointer
- `vmsa.rs` - Support virtual memory (described later)
- `lib.rs` - Include above modules; Has some useful functions such as `current_el()`.

As an example, the `get()` and `set()` function in `sp.rs` allows you retrieve and change the current stack pointer at any point in time. Similarly, the `current_el()` function in `lib.rs` returns the exception level the CPU is currently executing in, otherwise known as the *current exception level*.

Recap

There are many more instructions in the ARMv8 instruction set. With these as a basis, you should be able to pick up most of the remaining instructions with ease. The instructions are documented in (ref: C3). For a concise reference of the instructions presented above, see this ISA cheat sheet by Griffin Dietz. Before continuing, answer the following questions:

How would you write `memcpy` in ARMv8 assembly? (asm-memcpy)

Assuming that the source address is in `x0`, the destination address is in `x1`, and the number of bytes to copy is in `x2`, which is guaranteed to be a non-zero multiple of 8, how would you implement `memcpy` in ARMv8 assembly? Ensure you `ret`.

Hint

You can implement this in just 6 or 7 lines.

How would you write `0xABCDE` to `ELR_EL1`? (asm-movk)

Assume you're running in `EL1`, how would you write the immediate `0xABCDE` to `ELR_EL1` using ARMv8 assembly?

Hint

You'll need three instruction.

What does the `cbz` instruction do? (asm-cbz)

Read the documentation for the `cbz` instruction in (ref: C6.2.36). What does the instruction do? How would you use it?

What does `init.rs` do? (init)

Up until lab 3, we used `kern/src/init/init.s` as the first piece of code that runs in our kernel. `kern/src/init.rs` now replaced that role. In particular, the `_start` function will be at address `0x80000` when the Raspberry Pi boots, and the firmware will jump to this address as soon as it is done initializing the system. Soon, you'll modify this file to switch to EL1 and setup exception vectors.

Read all of the code in `kern/src/init.rs`. Then, for every function in the file, explain what the code is doing. For example, to explain the `_start` function, we might say:

"The [7:0] bits of the `MPIDR_EL1` register (ref: D7.2.74) are read (`Aff0`), yielding the core number that's currently executing code. If the number is zero, KERN_STACK_BASE(0x80_000) is set to stack pointer and kinit() is called.

Hint

Refer to the manual for any instruction/register you're not familiar with yet.

Hint

`aarch64` library and `kern/src/param.rs` will be needed while reviewing `init.rs`

Subphase C: Switching to EL1

In this subphase, you will write the Rust code to switch from EL2 to EL1. You will be working primarily in `kern/src/init.rs` and `kern/src/main.rs`. We recommend that you only proceed to this subphase after you have answered the questions from the previous subphases.

Current Exception Level

As mentioned before, the CPU should be running in EL2 when our kernel is called. Confirm this now by printing the current exception level in `kmain()`. Note that you'll need to use unsafe to call `current_el()`; we'll remove this call once we've confirmed that we've switched to EL1 successfully. `current_el()` is in the `lib/aarch64` library and you need to add it to solve dependency issue.

Switching

Now, you'll finish writing the assembly to switch to EL1. Find the line in `kern/src/init.rs` marked:

```
// FIXME: eret to itself, expecting current_el() == 1 this time
```

Above this line, you'll see following code.

```
SPSR_EL2.set(
    (SPSR_EL2::M & 0b0101)
    | SPSR_EL2::F
    | SPSR_EL2::I
    | SPSR_EL2::D
    | SPSR_EL2::A,
);
```

From the previous subphase, you should know what these do. In particular, you should know which bits are being set in `SPSR_EL2` and what the implications will be if an `eret` occurs thereafter.

Complete the switching routine now by replacing the `FIXME` with the proper code. Ensure that on the switch to EL1, the CPU jumps to `switch_to_el1` recursively with the proper exception level to bypass the internal check `current_el() == 2` and just move on to `kmain()`. You'll need exactly two lines of code to complete the routine. Recall that the only way to decrease the exception level is via an `eret`. Once you have completed the routine, ensure that `current_el()` now returns `1`.

Hint

Which register is used to set the PC on exception return?

Subphase D: Exception Vectors

In this subphase, you'll setup and install exception vectors and an exception handler. This will be the first step towards enabling your kernel to handle arbitrary exceptions and interrupts. You'll test your exception vector and exception handling code by implementing a tiny debugger that starts in response to a `brk #n` instruction. You will be working primarily in `kernel/src/init/vectors.s`, `kernel/src/traps.rs` and the `kernel/src/traps` directory.

Overview

Recall that the vector table consists of 16 vectors, where each vector is a series of at most 16 instructions. We've set apart space in `vectors.s` for these vectors and have placed the `vectors` label at the base of the table. This `vectors.s` file will be included to `init.rs` with `global_asm!` macro. Your task will be to populate the table with the 16 vectors such that, ultimately, the `handle_exception` Rust function in `kernel/src/traps.rs` is called with the proper arguments when an exception occurs. All exceptions will be routed to the `handle_exception` function. The function will determine why an exception has occurred and dispatch the exception to higher-level handlers as needed.

Calling Convention

In order to properly call the `handle_exception` function declared in Rust, we must know *how* the function expects to be called. In particular, we must know where the function should expect to find the values for its parameters `info`, `esr`, and `tf`, what it promises about the state of the machine after the function is called, and how it will return to the site of the function call.

This problem of knowing how to call foreign functions arises whenever one language calls into another (as in lab 3 between C and Rust). Instead of having to know how every language expects its functions to be called, *calling conventions* are established. A *calling convention*, or *procedure call standard*, is a set of rules that dictates the following:

- **How to pass parameters to a function.**

On AArch64, the first 8 parameters are passed in registers `r0` ... `r7`, in that order from left-to-right.

- **How to return values from a function.**

  On AArch64, the first 8 return values are passed in registers `r0` ... `r7`.

- **Which state (registers, stack, etc.) the function must preserve.**

  Registers are usually categorized as either *caller-saved* or *callee-saved*.

  A *caller-saved* register is not guaranteed to be preserved across a function call. Thus, if the caller requires the value in the register to be preserved, it must save the register's value before calling the function.

  On the contrary, a *callee-saved* register is guaranteed to be preserved across a function call. Thus, if a callee wishes to use the register during the function call, it must save the register's value before doing so and restore it before returning.

  Register values are typically saved and restored by pushing and popping from the stack.

  On AArch64, registers `r19` ... `r29` and `SP` are callee-saved. The remaining general purpose registers are caller-saved. Note that this includes `lr` (`x30`). SIMD/FP registers have complicated saving rules. For our purposes, it suffices to say that they are all caller-saved.

- **How to return to the caller.**

  On AArch64, the `lr` register holds the *link address*: the address the callee should jump to when it returns. The `ret` instruction branches to `lr`, so it often terminates a function.

The AArch64 calling convention is described in (guide: 9) as well as in the official procedure call standard documentation. When you call the `handle_exception` Rust function from assembly, you'll need to ensure that you adhere to this calling convention.

How does Rust know which convention to use?

Strictly adhering to a calling convention precludes all kinds of optimizations on function calls and bodies. As a result, Rust's functions are not guaranteed to abide to a particular calling convention by default. To force Rust to compile a function exactly according to the calling convention of the target platform, the `extern` function qualifier can be used. We've already declared `handle_exception` as `extern`, so we can be sure that Rust will compile the function appropriately.

Vector Table

To help you populate the vector table, we've provided the `HANDLER source, kind` macro which expands to a series of 8 instructions aligned to the next valid vector entry. When `HANDLER a, b` is used as an "instruction", it expands to the lines that reside between `.macro` and `.endm`. In other words, this:

```
vectors:
    HANDLER 32, 39
```

Expands to the following:

```
vectors:
    .align 7
    stp     lr, xzr, [SP, #-16]!
    stp     x28, x29, [SP, #-16]!

    mov     x29, #32
    movk    x29, #39, LSL #16
    bl      context_save

    ldp     x28, x29, [SP], #16
    ldp     lr, xzr, [SP], #16
    eret
```

The expanded code pushes `lr`, `xzr`, `x28` and `x29` to the stack, creates a 32-bit value in `x29` where the lower 16-bits are `source` and the upper 16-bits are `kind`, and calls the `context_save` assembly function (declared above `vectors` in `vectors.s`). When that function returns, it restores saved four registers from the stack and finally returns from the exception.

The `context_save` function currently does nothing: it simply falls through to a `ret` from `context_restore` below. Soon, you will modify the `context_save` function so that it correctly calls the `handle_exception` Rust function.

Syndrome

When a *synchronous* exception occurs (an exception caused by the execution or attempted execution of an instruction), the CPU sets a value in a syndrome register (`ESR_ELx`) that describes the cause of the exception (ref: D1.10.4). We've set up structures in `kernel/src/traps/syndrome.rs` that should parse the syndrome value into a meaningful `Syndrome` `enum`. You will soon write code that passes the `ESR_ELx` value to the Rust function as the `esr` parameter. You'll then use `Syndrome::from(esr)` to parse the syndrome value which determines what to do next.

Info

The `handle_exception` Rust function takes an `Info` structure as the first parameter. The structure has two `16-bit` fields: the first corresponds to the source, and the second corresponds to the `kind` of exception. As you may have guessed, this is exactly the `32-bit` value that the `HANDLE` macro sets up in `x29`. You'll need to move this `x29` to `x0` within `context_save` code block in order to pass it as the first parameter. In addition, please ensure that you use the correct `HANDLE` invocations for the correct entries so that the `Info` structure is correctly created.

Implementation

You're now ready to implement preliminary exception handling code. The first exception you will handle is the `brk` exception (a software breakpoint). When such an exception occurs, you'll start up a shell that would theoretically allow you explore the state of the machine at that point in its execution.

Start by inserting a call to `brk` in `main.rs`. Instead of using inline assembly, you can call `brk` macro defined in `asm.rs` of `aarch64` library.

Then, proceed as follows:

1. **Populate the `vectors` table using the `HANDLE` macro.**

   Ensure that your entries would correctly create the `Info` structure. Refer to (guide: 10.4) to check the entry order. The `source` and `kind` of handler should be matched with `Source` and `Kind` enum in `src/traps.rs`.

2. **Call the `handle_exception` function in `context_save`.**

   Ensure that you save/restore any caller-saved registers as needed and that you pass the appropriate parameters. For now, you can pass in `0` for the `tf` parameter; we'll be using this later. Refer `src/traps.rs` to see what to pass for arguments.

   Note: AArch64 requires the `SP` register to be 16-byte aligned whenever it is used as part of a load or store. Ensure that you keep `SP` 16-byte aligned at all times.

3. **Setup the correct `VBAR` register at the comment marked in `init.rs`.**

   ```
   // FIXME: load `vectors` addr into appropriate register (guide: 10.4)
   ```

4. **At this point, your `handle_exception` function should be called whenever an exception occurs.**

In `handle_exception`, print the value of the `info` and `esr` parameters and ensure that they are what you expect. Then, loop endlessly in the handler. You'll want to call `aarch64::nop()` in the loop to ensure it doesn't get optimized away. We will need to write more code to properly return from the exception handler, so we'll simply loop for now. We will fix this in the next subphase.

5. **Implement the `Syndrome::from()` and the `Fault::from()` methods.**

   The former should call the latter. You'll need to refer to (ref: D1.10.4, ref: Table D1-8) to implement these correctly. Clicking on the "ISS encoding description" in the table gives you details about how to decode the syndrome for a particular exception class as well as decode for `Fault`. You should ensure, for example, that a `brk 12` is decoded as `Syndrome::Brk(12)`. Similarly, a `svc 77` should be parsed as a `Syndrome::Svc(77)`. Note that we have excluded the 32-bit variants of some exceptions and coalesced exceptions when they are identical but occur with differing exception classes.

   Use aarch64 library!

   Instead of using inline assembly and raw bit operation for each register, use pre-defined registers and functions in `aarch64` library to your advantage. You will find `REG_NAME::get_value(raw_value, REG_NAME::MASK)` function useful.

6. **Start a shell when a `brk` exception occurs.**

   Use your `Syndrome::from()` method in `handle_exception` to detect a `brk` exception. When such an exception occurs, start a shell. You may wish to use a different shell prefix to differentiate between shells. Note that you should only call `Syndrome::from()` for synchronous exceptions. The `ESR_ELx` register is not guaranteed to hold a valid value otherwise.

   At this point, you'll also need to modify your shell to implement a new command: `exit`. When `exit` is called, your shell should end its loop and return. This will allow us to exit from a `brk` exception later. Because of this change, you'll also need to wrap your invocation to `shell()` in kmain in a `loop { }` to prevent your kernel from exiting and crashing.

Once you are finished, the `brk 2` instruction in `kmain` should result in an exception with syndrome `Brk(2)`, source `CurrentSpElx`, and kind `Synchronous` being routed to the `handle_exception` function. At that point, a debug shell should start. When `exit` is called from the shell, the shell should terminate, and the exception handler should begin to loop endlessly.

Before proceeding, you should ensure that you detect other synchronous exceptions correctly. You should try calling other exception-causing instructions such as `svc 3`. You should also try purposefully causing a data or instruction abort by jumping to an address outside of the physical memory range.

Once everything works as expected, you're ready to proceed to the next phase.

Subphase E: Exception Return

In this subphase, you will write the code to enable correct returns from an exception of any kind. You will be working primarily in `kern/init/vectors.s`, `kern/src/traps.rs` and the `kern/src/traps` directory.

Overview

If you try removing the endless loop in `handle_exception` now, your Raspberry Pi will likely enter an exception loop, where it repeatedly enters the exception handler, or crash entirely when you `exit` from your debug shell. This is because when your exception handler returns to whatever code was running previously, the state of the processor (its registers, primarily) has changed without the code accounting for it.

As an example, consider the following assembly:

```
1: mov x3, #127
2: mov x4, #127
3: brk 10
4: cmp x3, x4
5: beq safety
6: b   oh_no
```

When the `brk` exception occurs, your exception vector will be called, eventually calling `handle_exception`. The `handle_exception` function, as compiled by Rust, will make use of the `x3` and `x4` registers (among others) for processing. If your exception handler returns to the site of the `brk` call, the state of `x3` and `x4` is unknown, and the `beq safety` instruction on line 5 is not guaranteed to branch to `safety`.

As a result, in order for our exception handler to be able to use the machine as it desires, we'll need to ensure that we save all of the processing context (the registers, etc.) before we call our exception handler. Then, when the handler returns, we'll need to restore the processing context so that the previously executing code continues to execute flawlessly. This process of saving and restoring a processing context is known as a **context switch**.

What makes it a context *switch*?

The inclusion of the word *switch* can be a bit deceiving. After all, aren't we simply returning to the same context?

In fact, we rarely want to return to the same context. Instead, we typically want to modify the context before we return so that the CPU executes things just a little bit differently. For example, when we implement process switching, we'll swap out the context of one process for another's, multiplexing CPU time. When we implement system calls, we'll modify the values of registers to inject return values. And finally, when we return from a breakpoint exception, we'll need to modify the return address in `ELR` so that the *next* instruction executes instead of the same one.

Soon, you'll write the code to save all of the processing context into a structure known as a *trap frame*. You'll finish the definition of the `TrapFrame` structure in `kern/src/traps/frame.rs` so that you can access and modify the trap frame from Rust, and you'll write the assembly to save and restore the trap frame as well as pass a pointer to the trap frame to the `handle_exception` function in the `tf` parameter.

Trap Frame

The *trap frame* is the name we give to the structure that holds all of the processing context. The name "trap frame" comes from the term "trap" which is a generic term used to describe the mechanism by which a processor invokes a higher privilege level when an event occurs. We say that the processor *traps* to the higher privilege level.

There are many ways to create a trap frame, but all approaches are effectively the same: they save all of the state necessary for execution in memory. Most implementations push all of the state onto the stack. After pushing all of the state, the stack pointer itself becomes a pointer to the trap frame. We'll be taking exactly this approach.

For now, the complete execution state of our Cortex-A53 consists of:

- **x0…x30 - all 64-bits of all 31 general purpose registers**
- **q0…q31 - all 128-bits of all SIMD/FP registers**
- **TPIDR - the 64-bit "thread ID" register**

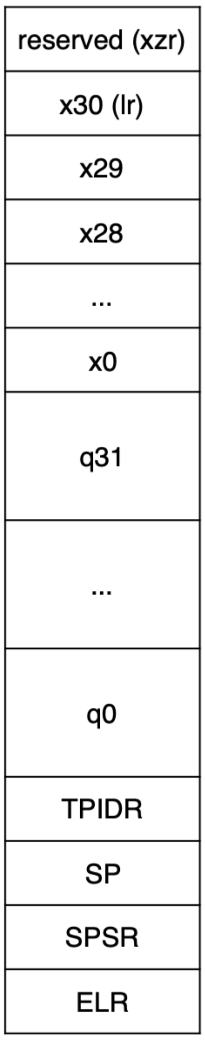
  This is stored in `TPIDR_ELs` when the source of the exception is at level `s`.
- **sp - the stack pointer**

  This is stored in `SP_ELs` when the source of the exception is at level `s`.
- **PSTATE - the program state**

  Recall that this is stored in `SPSR_ELx` when an exception is taken to `ELx`.
- **pc - the program counter**

  The register `ELR_ELx` stores the *preferred link address*, which may or may not be the PC that the CPU had when the exception is taken. Typically, the `ELR_ELx` is either the PC when the exception is taken, or `PC + 4`.

We'll need to save *all* of this context in the trap frame by pushing the relevant registers onto the stack before calling the exception handler and then restore the trap frame by popping from the stack when the handler returns. After saving all of the state, the stack should look as follows:



Please double check each register whether it uses `_ELx` or `_ELs`. Note that `SP` and `TPIDR` in the trap frame should be the stack pointer and thread ID of the source, not the target. Since the only eventual source of exception will be `EL0`, you should save/restore the `SP_EL0` and `TPIDR_EL0` registers. When all state has been pushed, the CPU's true `SP` (the one used by the exception vector) will point to the beginning of the trap frame.

Finally, you'll pass a pointer to the trap frame as the third argument to `handle_exception`. The type of the argument is `&mut TrapFrame`; `TrapFrame` is declared in `kern/src/traps/frame.rs`. You'll need to define the `TrapFrame` struct so that it *exactly* matches the trap frame's layout.

What's a thread ID?

The `TPIDR` register (`TPIDR_ELx`) allows the operating system to store some identifying information about what's currently executing. Later, when we implement process, we'll store the process's ID in this register. For now, we'll save and restore this register for posterity.

Preferred Exception Return Address

When an exception is taken to `ELx`, the CPU stores a *preferred link address*, or *preferred exception return address* in `ELR_ELx`. This value is defined in (ref: D1.10.1) as follows:

1. For asynchronous exceptions, it is the address of the first instruction that did not execute, or did not complete execution, as a result of taking the interrupt.
2. For synchronous exceptions other than system calls, it is the address of the instruction that generates the exception.
3. For exception generating instructions, it is the address of the instruction that follows the exception generating instruction.

A `brk` instruction falls into the second category. As such, if we want to continue execution after a `brk` instruction, we'll need to ensure that `ELR_ELx` contains the address of the *next* instruction before returning. Since all instructions are 32-bits wide on AArch64, this is simply `ELR_ELx + 4`.

Implementation

Start by implementing the `context_save` and `context_restore` routines in `kern/init/vectors.s`. The `context_save` routine should push all of the relevant registers onto the stack and then call `handle_exception`, passing a pointer to the trap frame as the third argument. Then implement `context_restore`, which should do nothing more than restore the context.

Note that the instructions generated by the `HANDLER` macro already save and restore `x28`, `x29`, `x30(lr)`, and `x31(xzr)`. You should not save and restore these registers in your `context_{save,restore}` routines, but your trap frame must still contain these registers. Technically there is no need to save and restore `xzr` register since it always contains zero. We save it just to make SP 16-byte aligned.

To minimize the impact on performance for the context switch, you should push/pop registers from the stack as follows:

```
// pushing registers `x1`, `x5`, `x12`, and `x13`
stp  x1, x5, [SP, #-16]!
stp  x12, x13, [SP, #-16]!

// popping registers `x1`, `x5`, `x12`, and `x13`
ldp  x1, x5, [SP], #16
ldp  x12, x13, [SP], #16
```

Once you have implemented these routines, finish defining `TrapFrame` in `kern/src/traps/frame.rs`. Ensure that the order and size of the fields exactly match the trap frame you create and pass a pointer to in `context_save`.

Finally, in `handle_exception`, remove the endless loop and increment the `ELR` in the trap frame by `4` before returning from a `brk` exception. Once you have successfully implemented the context switch, your kernel should continue to run as normal after `exit`ing from the debug shell. When you are ready, proceed to the next phase.

Trap Frame Layout

Your trap frame doesn't need to *exactly* match the diagram, but it should contain all of the same data.

`q0` .. `q31`

Don't forget that the `qn` registers are 128-bits wide!

Hint

To call `handle_exception`, you'll need to save/restore a register that's *not* part of the trap frame.

Hint

Rust has two 128-bit integer types: `u128` and `i128`.

Hint

Use the `mrs` and `msr` instruction to read/write special registers.

Hint

Our `context_save` routine is exactly 42 instructions.

Hint

Our `context_restore` routine is exactly 37 instructions.

Hint

Our `TrapFrame` contains 6 fields (two of them are arrays) and is 792 bytes in size without `xzr` register.

How could you lazy-load floating point registers? (lazy-float)

Saving and restoring the 128-bit SIMD/FP registers is *very* expensive; they account for 512 of the 792 bytes in the `TrapFrame`! It would be ideal if we saved/restored these registers only if they were actually in use by the source of the exception or the target of a context switch.

The AArch64 architecture allows the use of these registers to be selectively enabled and disabled. When SIMD/FP is disabled, an instruction that uses the registers traps. How could you use this functionality to implement lazy-loading of SIMD/FP registers so that they're only saved/restored on context switches if they're being used while continuing to allow the registers and SIMD/FP instructions to be used freely? Be specific about what you would do when specific exceptions occur, whether you would need to modify the `TrapFrame` struct, and what additional state you'd need to maintain.

Phase 2: It's a Process

In this phase, you will implement user-level processes. You'll start by implementing a `Process` struct that will maintain a process's state. You'll then bootstrap the system by starting the first process. Then, you'll implement a tick-based, round-robin scheduler. To do so, you'll first implement an interrupt controller driver and enable timer interrupts. Then, you'll invoke your scheduler when a timer interrupt occurs, performing a context switch to the next process. Finally, you'll implement your first system call: `sleep`.

After completing this subphase, you'll have built a minimal but complete multitasking operating system. For now, processes will be sharing physical memory with the kernel and other processes. In the next phase, we will enable virtual memory to isolate processes from one another and protect the kernel's memory from untrusted processes.

Subphase A: Processes

In this subphase, you'll complete the implementation of the `Process` structure in `kern/src/process/process.rs`. You'll use your implementation to start the first process in the next subphase.

What's a Process?

A process is a container for code and data that's executed, managed, and protected by the kernel. They are the singular unit by which non-kernel code executes: if code is executing, it is either executing as part of a process or executing as part of the kernel. There are many operating system architectures, especially in the research world, but they all have a concept that largely mirrors that of a process.

Processes typically run with a reduced set of privileges (`EL0` for our OS) so that the kernel can ensure system stability and security. If one process crashes, we don't want other processes to crash or for the entire machine to crash with it. We also don't want processes to be able to interfere with one another. If one process hangs, we don't want other processes to be unable to make progress. Processes provide *isolation*: they operate largely independently of one another. You likely see these properties of processes every day: when your web browser crashes or hangs, does the rest of the machine crash or hang as well?

Implementing processes, then, is about creating the structures and algorithms to protect, isolate, execute, and manage untrusted code and data.

What's in a Process?

To implement processes, we'll need to keep track of a process's code and data as well as auxiliary information to allow us to properly manage and isolate multiple processes. This means keeping track of a process's:

- **Stack**

  Each process needs a unique stack to execute on. When you implement processes, you'll need to allocate a section of memory suitable for use as the process's stack. You'll then need to bootstrap the process's stack pointer to point to this region of memory.

- **Heap**

  To enable disjoint dynamic memory allocation, each process will also have its own heap. The heap will start empty but can be expanded on request via a system call. In lab 4, we won't implement the heap and will support user programs that only require the stack.

- **Code**

  A process isn't very useful unless it's executing code, so the kernel will need to load the process's code into memory and execute it when appropriate.

- **Virtual address space**

  Because we don't want processes to have access to the kernel's memory or the memory of other processes, each process will be confined to a separate virtual address space using virtual memory.

- **Scheduling state**

  There are typically many more processes than there are CPU cores. The CPU can only execute one instruction stream at a time, so the kernel will need to multiplex the CPUs time (and thus, instruction stream) to execute processes concurrently. It is the scheduler's job to determine which process gets to run when and where. To do so correctly, the scheduler needs to know if a process is ready to be scheduled. The *scheduling state* keeps track of this.

- **Execution state**

> To correctly multiplex the CPUs time amongst several processes, we'll need to ensure that we save a process's execution state when we switch it off the CPU and restore it when we switch it back on. You've already seen the structure we use to maintain execution state: the trap frame. Each process maintains a trap frame to properly maintain its execution state.

A process's stack, heap, and code make up all of the physical state of a process. The rest of the state is necessary for the isolation, management, and protection of processes.

The `Process` structure in `kernel/src/process/process.rs` will maintain all of this information. Because all processes will be sharing memory for the time being, you won't see any fields for the process's heap, code, or virtual address space; you'll handle these later in the assignment.

Does a process have to trust the kernel? (kernel-distrust)

It should be clear that a kernel is distinctively distrustful of processes, but does a process have to trust the kernel? If so, what is it expecting from the kernel?

What could go wrong if two processes shared stacks? (isolated-stacks)

Imagine that two processes are executing concurrently and are sharing a stack. First: what would it mean for two processes to share a stack? Second: why would it be very likely that the processes would crash fairly quickly into their lives? Third: define a property of processes that, even if they were sharing a stack, would never crash as a result of sharing a stack. In other words, what would two processes that run concurrently and share a stack but never crash as a result of this sharing look like?

Implementation

You'll now start the implementation of the `Process` structure in `kern/src/process/process.rs`. Before you begin, read the implementation of the `Stack` structure that we provided for you in `kern/src/process/stack.rs`. Ensure that you know how to use the structure to allocate a new stack and retrieve a pointer to the stack for a new process. Then, read the implementation of the `State` structure, which will be used to keep track of the scheduling state, that we have provided for you in `kern/src/process/state.rs`. Try to reason about how you'd interpret the different variants when scheduling processes.

Finally, only implement the `Process::new()` method. The implementation will be simple; there's nothing complex about keeping track of state! You will finish the implementation of the process structure later on. When you're ready, proceed to the next subphase.

How is the stack's memory reclaimed? (stack-drop)

The `Stack` structure allocates a 16-byte aligned 1MiB block of memory when it is created. What ensures that this memory is freed when the `Process` that owns it is no longer around?

How could you lazily allocate memory for the stack? (lazy-stacks)

The `Stack` structure allocates 1MiB of memory for the stack regardless of whether or how much of the stack the process actually uses. Thinking ahead to virtual memory, how might we use virtual memory to lazily allocate memory for the stack so that no or minimal memory is used by the stack until it's needed?

How could a process increase its stack size? (stack-size)

Some processes will require significantly more stack space than 1MiB, but our simple design allocates exactly 1MiB of stack space for all processes. Assuming processes have access to dynamic memory allocation, how could a process increase its stack size? Be specific about which instructions the process would execute.

Subphase B: The First Process

In this subphase, we'll start the first user-space (`EL0`) process. You will be working primarily in `kern/src/process/scheduler.rs` and `kern/src/main.rs`.

Context Switching Processes

You've already done most of the work that will allow you to context switch between processes. To context switch between processes in response to an exception, you will:

1. Save the trap frame as the current process's trap frame in its `context` field.
2. Restore the trap frame of the next process to execute from its `context` field.
3. Modify the scheduling state to keep track of which process is executing.

Unfortunately, to context switch into the first process, we'll need to deviate from this plan a bit. It would be incorrect to execute one of the steps above before the first process. Can you tell which?

Let's see what would happen if we followed these steps before the first process. First, an exception occurs which prompts a context switch. We'll later see that this will be a timer interrupt which drives the process scheduler. Then we follow step 1: in response to the exception, we store the current trap frame in the current process's `context` field. Note, however, that there is no current process yet! And so Later, as part of step 2, we restore the next process's `context` and return.

Because the exception wasn't taken while the process running, the trap frame we save, and later restore, will have little in relation to the process itself. In other words, we've clobbered the process's trap frame with an unrelated one. Thus, we can't possibly run step 1 without a valid process's trap frame first. In other words, to properly context switch into the first process, it seems that the process needs to already be running. Said yet another way, we can't properly context switch until after the first context switch has occurred: catch-22!

To work around this, we're going to bootstrap context switching by *faking* the first context switch. Instead of the trap frame for the first process coming from the `context_save` routine you wrote previously, we will manually create the trap frame on the new process's stack and call `context_restore` ourselves, avoiding step 1 above entirely. Once the first process is running, all other context switching will work normally.

Kernel Threads

We haven't yet built a mechanism to load code from the disk into memory. Once we enable virtual memory, we'll need to implement the procedures to do so. For now, while we're sharing memory with the kernel, we can simply reuse the kernel's code and data. As long as the kernel and the processes don't share local data (the stack), which we've ensured they don't by allocating a new stack for each process, they will be able to execute concurrently without issue. What's more, Rust ensures that there is no possibility of a data race between the processes.

Sharing memory and other resources between processes is such a common occurrence that these types of processes have a special name: *threads*. Indeed, a thread is nothing more than a process that shares memory and other resources with another process.

Soon, you'll start the first process. Because that process will be sharing memory with the kernel, it will be a *kernel thread*. As such, the extent of the work required to start this first process is minimal since all of the code and data is already in memory:

1. Bootstrap context switching by setting up the "fake" saved trap frame.
2. Call `context_restore`
3. Switch to `EL0`.

While requiring very few lines of code, you'll find that it requires careful implementation for correctness.

The term *kernel thread* is overloaded.

The term *kernel thread* is used to refer both to threads implemented by the kernel (as opposed to threads implemented in user space) and threads running *in* the kernel. It's an unfortunate name clash, but context typically clarifies which is meant. As a quick heuristic, unless the discussion is about OS development, you should assume that the discussion is about threads implemented by the kernel.

Implementation

There is a new global variable in `kern/src/main.rs`, `SCHEDULER`, of type `GlobalScheduler`, which is simply a wrapper around a `Scheduler`. Both of these types are defined in `kern/src/process/scheduler.rs`. The `SCHEDULER` variable will serve as the handle to the scheduler for the entire system.

To initialize the scheduler and start executing the first process, the `start()` method on `GlobalScheduler` should be called. Your task is to implement the `start()` method.

To do so, you will need to:

1. **Write an `extern` function that takes no parameters and starts a shell.**

   You will arrange for this function to be called when the process first executes. You can write this function wherever you'd like. We'll remove it once we're able to start processes backed by binaries on the disk.

2. **In `start()`, create a new `Process` and set-up the saved trap-frame.**

   You'll need to set up the process's trap frame so that when it is restored to the CPU by `context_restore` later, your `extern` function executes, the process's stack pointer points to the top of the process's stack, the process is executing in `EL0` in the AArch64 execution state, and IRQ interrupts are unmasked for current `EL1` so that we can handle timer interrupts from `EL0` in the next section.

3. **Setup the necessary registers, call `context_restore`, and `eret` into EL0.**

Once you've set up the trap frame, you can bootstrap a context switch to that process by:

- Calling `context_restore` with the appropriate register(s) set to the appropriate values.

  Note: we are being vague here on purpose! If this feels opaque, consider what `context_restore` does, what you *want* it to do, and how you can make it do that.
- Setting the current stack pointer (`sp`) to its initial value (the address of `_start`). This is necessary so that we can use the entire `EL1` stack when we take exceptions later. **Note:** You cannot `ldr` or `adr` into `sp` directly. You must first load into a different register and then `mov` from that register into `sp` .
- Resetting any registers that may no longer contain `0` . You should not leak any information to user-level processes.
- Returning to `EL0` via `eret` .

You'll need to use inline assembly to implement this. As an example, if a variable `tf` is a pointer to the trap frame, the following sets the value of `x0` to that address and then copies it to `x1` :

```
unsafe {
    asm!("mov x0, $0
          mov x1, x0"
         :: "r"(tf)
         :: "volatile");
}
```

You may wish to add an infinite loop at the end of the `start()` function to satisfy compiler since the function should not be returned. Once you've implemented the method, add a call to `SCHEDULER.start()` in `kmain` and remove any shell or breakpoint invocations. You don't need to call `SCHEDULER.initialize()` yet which will be implemented in Subphase D. Your `kmain` should now simply be a series of two initialization calls and a scheduler starting call such as below.

```
unsafe fn kmain() -> ! {

    ALLOCATOR.initialize();
    FILESYSTEM.initialize();
    SCHEDULER.start()

}
```

If all is well, your `extern` function will be called from `EL0` when the kernel starts, running the shell as a user-level process.

Before continuing, you should also ensure that a context switch back to the same process works correctly at this point. Try adding a few calls to `brk` in your `extern` function before and after you start a shell:

```
extern fn run_shell() {
    unsafe { asm!("brk 1" :::: "volatile"); }
    unsafe { asm!("brk 2" :::: "volatile"); }
    shell::shell("user0> ");
    unsafe { asm!("brk 3" :::: "volatile"); }
    loop { shell::shell("user1> "); }
}
```

You should be able to return from each of the break point exceptions successfully. The source for each of the breakpoint exception should be `LowerAArch64`, indicating a successful switch to user-space. Once everything works as you expect, proceed to the next subphase.

Hint

Our inline assembly consists of exactly 6 instructions.

Hint

Besides the inline assembly, you do not need `unsafe`.

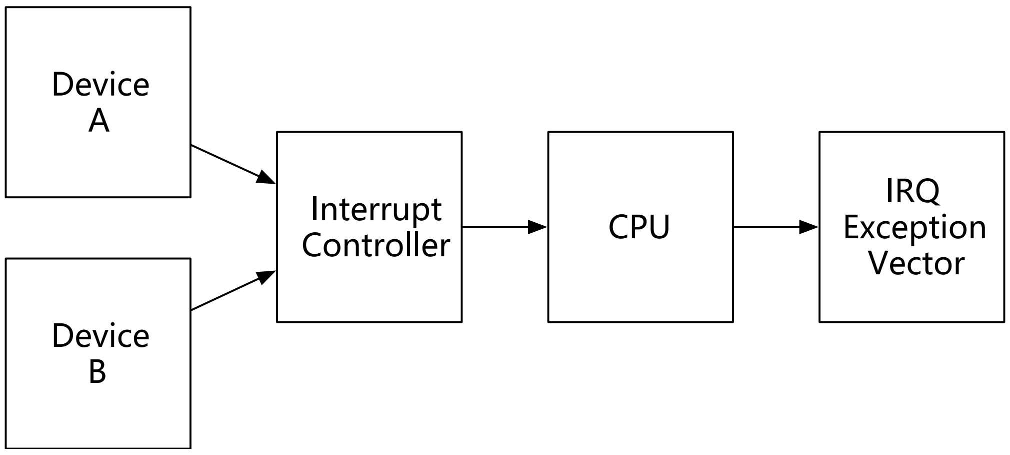
Subphase C: Timer Interrupts

In this subphase, you will implement a driver for the interrupt controller on the BCM2837. You'll also modify your existing system timer driver to enable timer interrupts to be configured. Finally, you'll enable periodic timer interrupts to act as the spring-board for scheduling based context switches. You will be working primarily in `lib/pi/src/interrupt.rs`, `lib/pi/src/timer.rs`, and `kern/src/traps`.

Interrupt Handling

On AArch64, interrupts are nothing more than exceptions of a particular class. The key differentiator between the two is that interrupts occur asynchronously: they are generated by an external source in response to external events.

The diagram below illustrates the path an interrupt takes from the source, an external device, to the sink, an exception vector:



Interrupts can be selectively disabled at each point along the path. In order for an interrupt to be delivered to an exception vector, the external device, the interrupt controller, and the CPU must all be configured to accept the interrupt.

What is an interrupt controller?

An interrupt controller as another external device that acts as a proxy and gate between interrupt generating devices, like the system timer, and the CPU. The interrupt controller is physically connected to the CPU's interrupt pins. When an input pin on the interrupt controller is signaled, the interrupt controller forwards the signal to the CPU.

The extra layer of indirection allows for interrupts to be selectively enabled and disabled. It also allows CPU manufacturers to choose which, if any, interrupt controller they want to bundle with the CPU.

External Device

You've already written a device driver for the system timer. In this subphase, you will extend your driver to enable configuration of the timer's compare registers. The system timer continuously compares the current time to the values in the compare registers and generates an interrupt when the values equal.

Interrupt Controller

The system timer delivers interrupts to the interrupt controller, which must then be configured to deliver interrupts to the CPU. You will write a device driver for the interrupt controller to do exactly this.

When the interrupt controller receives an interrupt, it marks the interrupt as pending and forwards it to the CPU by holding a physical interrupt pin on the CPU logically high. For some interrupts, including system timer interrupts, the pin is held high until the interrupt is acknowledged. This means that the interrupt will be continuously delivered until it is acknowledged. Once the interrupt is acknowledged, the interrupt pin is released, and the pending flag is unset.

CPU

Interrupts must be *unmasked* for the CPU to deliver them to exception vectors. By default, interrupts are *masked* by the CPU, so they will not be delivered. The CPU may deliver interrupts that were received while interrupts were masked as soon as interrupts are unmasked. When the CPU invokes an exception vector, it also automatically masks all interrupts. This is so that interrupts which are held high until they are handled, like system timer interrupts, don't immediately result in an exception loop.

In the previous subphase, you configured interrupts to be delivered when processes are executing in `EL0`, so there's no additional work to do on this front.

When would you unmask IRQs while handling an IRQ? (reentrant-irq)

Although our kernel keeps IRQs masked in exception handling routine, thus does not support nested interrupt handling, it turns out that unmasking IRQs while handling IRQs is a fairly common occurrence in commodity operating systems. Can you come up with a scenario in which you'd want to do this? Further, would doing so without first acknowledging pending IRQs result in an exception loop? Why or why not?

Exception Vector

You've already configured exception vectors. As such, all that's left is to properly handle IRQ (interrupt request) exceptions. To handle interrupts, we have a global `Irq` struct which holds a list of handler functions for corresponding interrupt. You can set a handler function for each interrupt by calling `register()` function and execute the registered handler with `invoke()` function in `kern/src/traps/irq.rs`. You'll modify your `handle_exception` function in `kern/src/traps.rs` so that it forwards all known interrupt requests to the `invoke()` function.

To determine which interrupt has occurred, you will need to check which interrupts are pending at the interrupt controller. The `handle_irq` function will then acknowledge the interrupt and process it.

Implementation

Start by implementing the interrupt controller driver in `lib/pi/src/interrupt.rs`. The documentation for the interrupt controller is on chapter 7 of the BCM2837 ARM Peripherals Manual. You only need to handle enabling, disabling, and checking the status of the regular IRQs described by the `Interrupt` enum; you needn't worry about FIQs or Basic IRQs. We are providing four methods for `Interrupt` struct. `iter()` methods can be used when iterating interrupts to check which interrupts are pending. `to_index()` and `from_index()` would be useful when implementing `register()` and `invoke()` method.

Then, implement the `tick_in()` method and function for your system timer driver in `lib/pi/src/timer.rs`. The documentation for the system timer is on chapter 12 of the BCM2837 ARM Peripherals Manual. You will need write to two registers to implement `tick_in()` correctly.

Now go to the `src/traps/irq.rs` and implement `register()` and `invoke()` methods. The `Irq` struct internally holds a list of `IrqHandler` which is a smart pointer for a handler function. You can set a new handler function for an interrupt with `register()` method, and executes it with `invoke()` method.

Then, enable timer interrupts and set a timer interrupt to occur in `TICK` microseconds just before you start the first process in `GlobalScheduler::start()` in `kern/src/process/scheduler.rs`. The `TICK` variable is declared in `kern/src/param.rs`. In addition, register an handler function for the timer. The handler function should set a new timer interrupt to occur in `TICK` microseconds, ensuring that timer interrupts occur every `TICK` microseconds indefinitely. To test your implementation, you might want to print a message in here.

Modify your `handle_exception` function in `kern/src/traps.rs` so that it forwards known interrupts to the `invoke()` function in `kern/src/traps/irq.rs`.

Finally, add `IRQ.initialize()` in your `kern/src/main.rs` before calling `SCHEDULER.start()`.

When you are finished, you should see a timer interrupt occur every `TICK` microseconds with a source of `LowerAArch64` and kind of `Irq`. You should be able to interact with the process normally between timer interrupts. When everything works as you expect, proceed to the next subphase.

❗ We'll change the `TICK` setting later on!

We're currently using an absurdly slow `TICK` setting of 2 seconds to ensure that everything works as we expect. Typically, this number is between 1 and 10 milliseconds. We'll decrease the `TICK` to a more reasonable 10 ms later on.

Subphase D: Scheduler

In this subphase, you will implement a simple round-robin preemptive scheduler. You will be working primarily in `kern/src/process/scheduler.rs` and `kern/src/process/process.rs`.

Scheduling

The scheduler's primary responsibility is to determine which task to execute next, where a task is defined as anything that requires execution on the CPU. Our operating system is relatively simple, and so the scheduler's idea of a task will be constrained to processes. As such, our scheduler will be responsible for determining which process to run next, if any.

There are many scheduling algorithms with a myriad of properties. One of the simplest is known as “round-robin” scheduling. A round-robin scheduler maintains a queue of tasks. The next task to execute is chosen from the front of the queue. The scheduler executes the task for a fixed time slice (the `TICK`), also known as a *quantum*. When the task has executed for at most its full quantum, the scheduler moves it to the back of the queue. Thus, a round-robin scheduler simply cycles through a queue of tasks.

In our operating system, the scheduler marks a task as being in one of four states:

- **Ready**

  A task that is ready to be executed. The scheduler will execute the task when its turn comes up.

- **Running**

  A task that is currently executing.

- **Waiting**

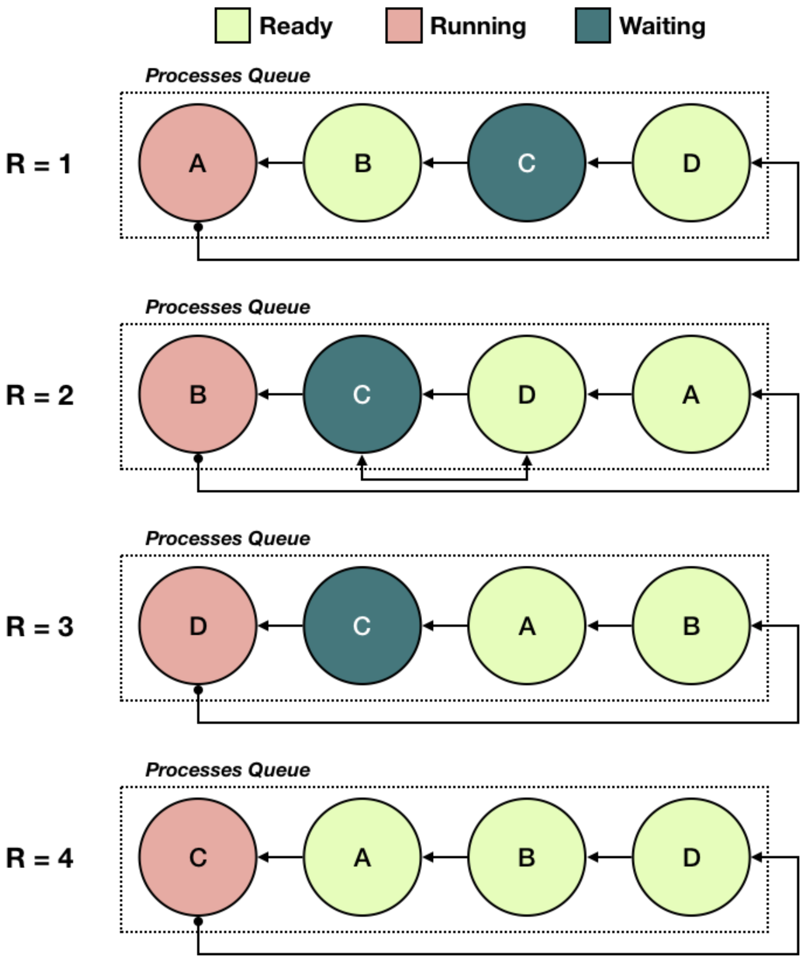
  A task that is waiting on an event and is not ready to be executed until that event occurs. The scheduler will check if the event has occurred when the task’s turn comes up. If the event has occurred, the task is executed. Otherwise, the task loses its turn and is checked again in the future.

- **Dead**

  A task is currently dead (not running nor eligible to run) and ready to be reclaimed.

The `State` enum in `kern/src/process/state.rs` represents these states. Each process struct is associated with a `State` which the scheduler will manage. Note that the `Waiting` state contains a function that the scheduler can use to determine if the event being waited on has occurred.

The diagram below depicts four scheduling rounds of a round-robin scheduler. The task `C` is waiting on an event that occurs some time between rounds 3 and 4.



The rounds are:

1. In round 1, there are four tasks in the queue, `A`, `B`, `C`, and `D`. `A`, the process at the head of the queue is currently running on the CPU. `C` is in a waiting state, while the others are ready or running. When `A`'s quantum is used up, it is moved to the back of the queue.
2. The task from the front of the queue, `B`, is executed. It is moved to the back of the queue when its quantum expires.
3. Since `C` is waiting for an event, the scheduler checks to see if the event being waited on has occurred. At this point it has not, so `C` is skipped and `D` is chosen to run next. `D` is moved to the front and executed. At the end of the quantum, `D` is moved to the back of the queue.
4. Since `C` is still in waiting state, the scheduler checks to see if the event has occurred. At this point it has, so `C` is scheduled. After its time quantum, `C` is moved to the back of the queue.

Would separating ready and waiting tasks be beneficial? (wait-queue)

An alternative implementation of a round-robin scheduler maintains two queues: a *ready* queue, consisting of only ready tasks, and a *wait* queue, consisting only of waiting tasks. How would you make use of the queues in the round-robin scheduler? Do you expect performance (average task latency/throughput) to be better or worse?

Code Structure

The `Scheduler` structure in `kern/src/process/scheduler.rs` maintains a queue of processes to execute. Processes are added to the queue via the the `Scheduler::add()` method. The method is also responsible for assigning unique IDs to processes. IDs are stored in the process's `TPIDR` register.

When a scheduling change is required, the `Scheduler::schedule_out()` and `Scheduler::switch_to()` method is invoked. As their name suggests, `schedule_out()` method changes the current process's state to `new_state`, saves the current trap frame in the current process, and push the current process back to the end of scheduling queue. Recalling that our Raspberry Pi board has 4 cores, there might exist multiple process with `Running` state up to the number of cores. Thus, you have to find current process based on the process id within current trapframe. `switch_to()` method finds the next process to execute, moves the next process to the front of the queue, restores the next process's trap frame, and marks the next process as `Running`.

To determine if a process is ready to execute, the scheduler should call the `process.is_ready()` method, defined in `kern/src/process/process.rs`. The method returns `true` if either the state is `Ready` or if an event being waited on has occurred.

The scheduler should be invoked every `TICK` microseconds. Timer interrupts, set up in the previous subphase, will be one of the primary sources of a scheduling change. The `GlobalScheduler` type provides thread-safe method `switch()` as well as wrappers around the `add()` and `kill()` methods of `Scheduler`. `switch()` method first gets the lock and schedules out the current process. Then, it repeatedly tries to switch to a next process. If `switch_to()` method fails to find a next process to switch to, such as when all processes are in the `Waiting` state, it executes `wfe` instruction to enter into a low-power state and wait until an event comes without executing further instruction.

Finally, to kill a currently running process, `Scheduler::kill()` method is invoked. This method will be called by system call in the next subphase. The method internally calls `Scheduler::schedule_out()` method with `Dead` state as a parameter to schedule out the current process. Then, it removes the dead process from the end of the queue and drop the dead process`s instance. Since `GlobalScheduler` type provides thread-safe wrappers around the `Scheduler::kill()`, it is guaranteed that the dead process is the last entry of the processes queue.

❗ Why doesn't the scheduler know the new state? (new-state)

The `scheduler.switch()` method requires the caller to pass in the new state of the current process. This implies that the scheduler does not know what the new state of the process should be. Why might it not?

Implementation

You're now ready to implement the round-robin scheduler. We recommend the following approach:

1. **Implement the `Process::is_ready()` method in `kernel/src/process/process.rs`.**

   The mem::replace() function will prove useful here. Please note that the `Waiting` state contains a function to check if the event being waited on has occurred. Only if it returns `true`, `Waiting` process can be scheduled.

2. **Implement the `Scheduler` struct in `kern/src/process/scheduler.rs`.**

   There are five functions you need to implement; `new()`, `add()`, `schedule_out()`, `switch_to()` and `kill()`.

3. **Initialize the scheduler in `GlobalScheduler::initialize()`.**

   The global scheduler should be created and initialized before the first process executes. The first process should be present in scheduler's queue before it executes.

4. **Modify the `GlobalScheduler::start()`.**

   Instead of starting a process from an address of a extern function, now make use of `switch_to` function to start a process from the scheduling queue.

5. **Invoke the scheduler when a timer interrupt occurs.**

   Invoke `SCHEDULER.switch()` on a timer interrupt to context switch between the current process and the next process.

Test your scheduler by adding more than one process in `GlobalScheduler::initialize()`. You'll need to allocate new processes and set up their trap frames appropriately. You'll likely want to create a new `extern` function for each new process so that you can differentiate between them. Ensure that you add the processes to the scheduler's queue in the correct order.

When you are finished, you should see a different process execute every `TICK` microseconds. You should be able to interact with each process normally between timer interrupts. When everything works as you expect, proceed to the next subphase.

Overflow

Don't overflow when generating a process ID!

Unsafe

You should not use `unsafe` to implement any of these routines!

Hint

Use mem::replace() to get an owned version of the process's `state`.

Why is it correct to wait for events when no process is ready? (wfe)

Using the `wfe` instruction to wait when no process is ready means that the CPU stalls until an event arrives. If no event arrives after a `wfe` is executed, scheduling never resumes. Why is this the correct behavior?

Hint

Think about the scenarios in which a process is in the waiting state.

Note about `wfe`

When there are no processes ready to be executed, the processor will go into Wait-For-Event `wfe` state. Event != interrupt in this context. Meaning that, when our processor goes into `wfe`, it will not be waken up by our timer interrupt, and so the scheduling will not resume. If you wish to have the processor waiting for interrupts instead, change `wfe` instruction to `wfi` instruction.

Subphase E: Sleep

In this subphase, you will implement the `sleep` system call and shell command. You will be working primarily in `kern/src/shell.rs` and `kern/src/traps`.

System Calls

A system call is nothing more than a particular kind of exception. When the `svc #n` instruction is executed, a synchronous exception with syndrome `Svc(n)` will be generated corresponding to system call `n`. This is similar to how `brk #n` generates a `Brk(n)` exception except that the preferred link address is the instruction after the `svc` instruction instead of the instruction itself. System calls are the mechanism that user processes use to request services from the operating system that they would otherwise have insufficient permissions to carry out.

A typical operating system exposes 100s of system calls ranging from file system operations to getting information about the underlying hardware. In this subphase you will implement the `sleep` system call. The `sleep` system call asks the scheduler not to schedule the process for some amount of time. In other words, it asks the operating system to put the process to sleep.

Syscall Convention

Just as we need a convention for function calls, we require a convention for system calls. Our operating system will adopt a modified version of the system call convention used by other Unix-based operating systems. The rules are:

- System call `n` is invoked with `svc #n`.
- Up to 7 parameters can be passed to a system call in registers `x0` ... `x6`.
- Up to 7 parameters can be returned from a system call in registers `x0` ... `x6`.
- Register `x7` is used to indicate an error. See `lib/kernel_api/src/lib.rs` for possible values.
  - If `x7` is `1`, there was no error.
  - If `x7` is `0`, unknown error has occurred.
  - If `x7` is any other value, it represents an error code specific to the system call.
- All other registers and program state are preserved by the kernel.

As such, to invoke an imaginary system call `7` that takes two parameters, a `u32` and a `u64`, and returns two values, two `u64`s, we might write the following using Rust's inline assembly

```
fn syscall_7(a: u32, b: u64) -> Result<(u64, u64), Error> {
    let ecode: u64;
    let result_one: u64;
    let result_two: u64;
    unsafe {
        asm!("mov w0, $3
              mov x1, $4
              svc 7
              mov $0, x0
              mov $1, x1
              mov $2, x7"
              : "=r"(result_one), "=r"(result_two), "=r"(ecode)
              : "r"(a), "r"(b)
              : "x0", "x1", "x7")
    }

    let e = OsError::from(ecode);
    if let OsError::Ok = e {
        Ok((result_one, result_two))
    }
    else {
        Err(e)
    }
}
```

Notice that the wrapper around the system call checks the error value before returning the result value. You can find pre-defined system call number and enum of `OsError` in `lib/kernel-api/src/lib.rs`.

❗ Why do we use a separate register to pass the error value? (syscall-error)

Most Unix operating systems, including Linux, overload the first result register (`x0` , in our case) as the error value register. In these conventions, negative values with a certain range represent error codes; all other values are interpreted as successful return values. What is the advantage to the approach that we have taken? What is the disadvantage?

Sleep Syscall

The `sleep` system call will be system call number `1` in our operating system. The call takes one parameter: a `u32` corresponding to the number of milliseconds that the calling process should be suspended for. Besides the possible error value, it returns one parameter: a `u32` corresponding to the number of milliseconds that elapsed between the process's initial request to sleep and the process being woken up. Its pseudocode signature would be:

```
fn sleep(t: u32) -> u32
```

When does the elapsed time differ from the requested time? (sleep-elapsed)

In which situations, if any, will the return value from `sleep` differ from the input value? In which situations, if any, will they be identical? What do you think the relative probability of each case is?

Implementation

Implement the `sleep` system call now. Start by modifying your `handle_exception` function in `kern/src/traps.rs` so that it recognizes system call exceptions and forwards them to the `handle_syscall` function in `kern/src/traps/syscalls.rs` .

Then implement the `handle_syscall` function. The function should recognize the `sleep` system call and calls its handler function `sys_sleep()` . Inside the `sys_sleep()` , modify the currently executing process as required. You will likely need to create a `Box<FnMut>` using a closure to complete your implementation. This should look as follows:

```
let boxed_fnmut = Box::new(move |p| {
    // use `p`
});
```

You can read more about closures in TRPLv2.

Finally, add a `sleep <ms>` command to your shell that invokes the sleep system call, passing in `ms` milliseconds as the sleep time.

Test your implementation by calling `sleep` in user-level shells. Ensure that a process is not scheduled while it is sleeping. All other processes should continue to be scheduled correctly. Then, ensure that *no* process is scheduled if all processes are sleeping. Once your implementation works as you expect, proceed to the next subphase.

Hint

The `sleep` system call handler will need to interact with the scheduler.

Hint

Recall that closures can capture values from their environment.

Hint

We are providing `kernel_api::syscall::sleep(span: Duration)` function. Feel free to use it in your shell to invoke sleep systemcall. System call number and error code can also be found in `kernel_api` library.

Hint

The `u32` type implement FromStr.

Phase 3: Memory Management Unit

In this phase, you'll enable the support of the memory management unit so that our system can run multiple processes independently with each running in their own private virtual memory space. You'll start by reviewing `VirtualAddr` and `PhysicalAddr` structs and several traits to support translation between addresses. Then, you'll implement a two level page table indexing 64KB aligned page. With this `PageTable` struct, you'll implement kernel page table and user page table. Finally, you'll revisit code you have partially built in the previous phases, such as `init.rs` or `scheduler.rs`, in order to support context switch between processes having virtual memory space.

Subphase A: Virtual Memory

In this subphase, you'll review `VirtualAddr` and `PhysicalAddr` structs to represent 64bit address values under `kern/src/vm/address.rs`. Then, you'll add two more registers in your TrapFrame under `kern/src/traps/frame.rs` and `kern/src/init/vectors.s`

Why Do We Need Virtual Memory?

Instead of sharing physical memory between kernel and user processes, each user process will have its own virtual memory space. This will isolate processes from each other and protect the kernel's memory from untrusted processes.

By supporting virtual memory space, user processes do not need any knowledge of the physical memory space anymore. That is, processes will be unaware of the actual hardware addresses, or about other processes' virtual spaces that might execute at the same time.

Because of the isolation between virtual spaces of each process, two different processes might use the exact same virtual address, but each will translate into two different physical addresses.

Also, because of the isolation between virtual and physical space, we can translate contiguous virtual memory pages into fragmented physical memory pages. Hence, we can utilize fragmented pages in the physical space to form one contiguous virtual space. User processes can write, compile, and link applications to run in the virtual memory space, and the operating system will take care of mapping all memory accesses to the physical memory addresses.

For the operating system to achieve this, it needs the memory management unit, commonly referred to as MMU, that handle all translations between virtual spaces and physical space.

How To Work With The MMU?

The memory management unit is a hardware unit that makes the translation between virtual space addresses and physical addresses. To make use of it, we need to enable it first, and provide the page mapping table which it will use for translations. That is, the MMU will perform all the translations, but the operating system is responsible for creating and upkeeping of the page tables.

With the MMU enabled, all processor instruction fetch or memory access will go through the MMU for translation, and so we will need to provide it with the proper page tables for both the kernel and user space.

A consequence of turning on the MMU is that the kernel now have to be prepared to use the virtual memory and maintain its own set of page tables. We will see how to set that up in later parts of this phase.

Separation of Kernel And User Process

Operating systems typically have a number of processes that are running concurrently. Since each of those processes have its own virtual memory space, the operating system needs to keep track of each process translation table. The translation table of each process will contain information that maps each virtual page address into a physical page address. Since these page tables are used for translation between user processes and physical addresses, they are hence called *User Page Tables*.

In the same way, since now we are enabling the MMU, we need to provide a *Kernel Page Table* that the MMU will use to continue to map the kernel addresses correctly. The MMU will also enforce the permissions of the kernel pages stated by the kernel page table we provide. Hence, we will have a permission isolation between kernel memory pages and processes memory pages.

Our ARM architecture have two registers to keep track of the page table the MMU should use for address translations: *Translation Table Base Register* `TTBR0_EL1` and `TTBR1_EL1`. In our design, we will use `TTBR1_EL1` to point to the *User Page Table* base address, and `TTBR0_EL1` to point to the *Kernel Page Table* base address. We will need to add both registers to the `TrapFrame` for bookkeeping of each process and the kernel page tables.

Since now we have the kernel and user level using the virtual space address, we will need to split the virtual address space, so that we have a clear address isolation between kernel and user addresses in the virtual space. To do so, will use addresses starting from `0xffff_ffff_c000_0000` for user level virtual address, and addresses starting from `0x0` for kernel space virtual address.

Thus, the maximum size of the virtual memory space for user level would be `0x4000_0000`, which is 1GB memory. You can find these variables defined in `kern/src/param.rs`.

How does the MMU know which table to use? (which-table)

Given a virtual address, the MMU will use either the user page table based at `TTBR1_EL1`, or the kernel page table based at `TTBR0_EL1` in order to correctly translate to the physical address. But how does the MMU know which of the two tables to use for a given address?

Hint

We did not split the kernel and user level virtual space at arbitrary address! What happens if we make the kernel level at the upper virtual space part? The following reference might help: ARMv8-A Address Translation

Implementation

Review the `VirtualAddr` and `PhysicalAddr` in `kern/src/vm/address.rs`. Although they have different name and different purpose, there is no difference in their structure, required traits or functions.

Now add `TTBR0` and `TTBR1` registers to the trap frame. You need to modify `kern/src/traps/frame.rs` and `kern/src/init/vectors.s` to do so. You should add instructions to save and restore `TTBR0` and `TTBR1`, just like we did with other system registers: saved in `context_save` before calling `handle_exception` and restored in `context_restore` after returning back. Also, you should add `TTBR0` and `TTBR1` in `TrapFrame` structs accordingly. Note that your `TrapFrame` struct should *exactly* match the trap frame's layout.

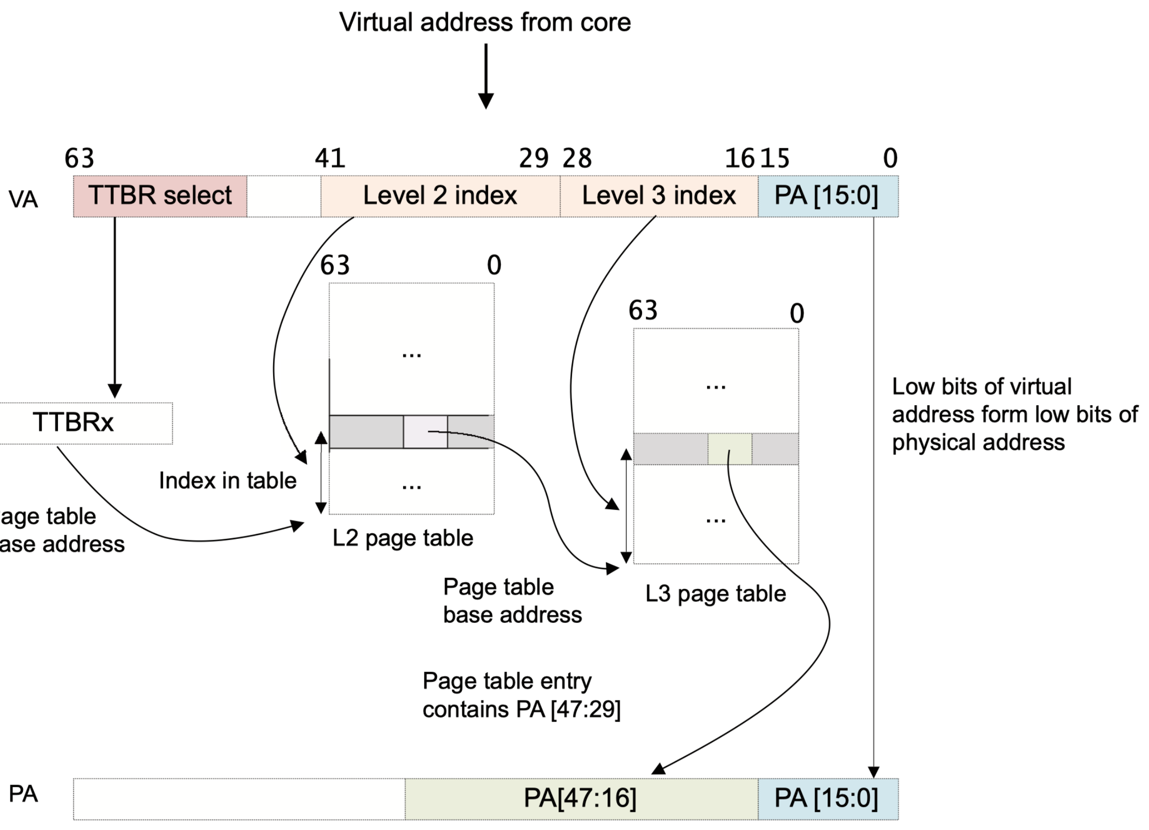
In addition, insert the following four lines after restoring `TTBR0` and `TTBR1` from the stack in `vectors.s`. This code block is required to ensure that memory accesses that occur before the `dsb` have completed before the completion of the `dsb` instruction.

```
dsb     ishst
tlbi    vmalle1
dsb     ish
isb
```

Test that your modification doesn't affect your trap frame and context switch functionality. If everything still works well, you can proceed to the next subphase.

Subphase B: Page Table

When the processor issues a 64-bit virtual address for an instruction fetch, or data access, the memory management unit(MMU) translates the virtual address to the corresponding physical address.



The above diagram depicts an example of virtual to physical address translation for a 64KB page. This assumes a 64KB granule and 42-bit virtual address space and we are going to stick to this setting. You could find more detail diagram with bits specified in Figure K6-13 in (ref: K6.1.2).

Lets walk through what is happening in the diagram above. The 64-bit virtual address -shown on the top- is split into 4 regions:

- **Bits [63-42]** are used to decide which page table to use, the one based at `TTBR0` or `TTBR1`, this is the first step of translation.
- **Bits [41-29]** are used to index into the L2 table based at the chosen `TTBRx`. Since 13 bits are used, this level of the page table can have 8192 entries, with each having 64-bit. We will later discover how each bit of the 64-bits is used, for now, we will just state that bits [47-16] of these entries are used to provide the base of the L3 page table.
- **Bits [28-16]** are used to index into the chosen L3 page table. Since we are using 13 bits, we can have 8192 entries for each L3 page table, with each entry being 64-bits. Out of these 64-bits, we will use the bits [47-16] in the final physical address.
- **Bits [15-0]** are used as a byte offset within the page addressed by bits [47-16] from L3 entry. Since we are having 64KB pages, 16 bits is enough to address each byte within.

Having known the different parts of the virtual address, lets detail the steps taken by the MMU to perform the address translation:

1. If it is a user page mapping, `TTBR1` is used for the base address for the first page table. When it is a mapping for kernel or I/O peripherals, `TTBR0` is used for the base address for the first page table.
2. The page table contains 8192 64-bit page table entries, and [41:29] bits of virtual address are used as an index to find an entry in the table.
3. Check the validity of the L2 page table entry and whether or not the requested memory access is allowed.
4. If the entry is valid and allowed, the [47:16] bits of L2 page table entry is used to find the address of the level 3 page table.
5. Bits [28:16] of the virtual address are used to index the level 3 page table entry.
6. Check the validity of the L3 page table entry and whether or not the requested memory access is allowed.
7. If the entry is valid and allowed, the L3 page table entry refers to a 64KB physical page. Bits [47:16] are taken from the level 3 page table entry and used to form bits [47:16] of a physical address.
8. Because we have 64KB page, virtual address [15:0] is taken to form physical address [15:0].

The number of page tables within each level is up to the operating system, and how much memory it needs to map. The fact is, if we fully utilize this hierarchical model, we will end up with much bigger virtual space than our need. In our design, however, we are sufficed with only 1GB of virtual memory in the user level. Hence, for the user page table, we will have only one L2 page table, with two entries, each pointing to different L3 page table. Each L3 table of the two will have 8192 entries, each pointing to a 64KB page. So the calculation becomes:

1 (L2 table) * 2 (L3 tables) * 8192 (L3 entries) * 64KB (page size) = 1GB

How is our design different? (translation-control)

The figure you see above does not exactly match our design. Particularly, the number of bits of `TTBR select` and `Level 2 index` are not the same in our design. Can you figure out the number of bits we allocated for each? and how exactly did we configure the MMU to apply our design?

Hint

Exactly the same hint of the previous question: how did we enforce the address split between the kernel and the user virtual spaces? Where in our code did we configure the MMU such that addresses above `0xffff_ffff_c000_0000` are for user virtual space?

Page Table Entry

The Page table entry PDF contains the specific details about L2 and L3 page table entry including their bit locations, names, and field descriptions. You will be referring to this document when you implement your page table. For more details, see the (ref: D4.3)

`ADDR` field contains 32-bit output address. That address is used differently depending on if it is a L2 table entry or a L3 table entry. For L2 table, the output address will be combined with Level 3 index that is located in [28:16] bits of virtual address to point the L3 page table. For L3 table, the output address will be the [47:16] bits of the translated physical address.

The low 10 bits of each entry represent the memory attributes.

- **VALID** : Identifies whether the descriptor is valid. You should set the entry as valid to use it.
- **TYPE** : L2 entry can be interpreted as a descriptor pointing a block of memory or a descriptor pointing next level of translation table (L3). Note that our L2 entries needs to point the L3 table, while our L3 entries will point to the memory pages.
- **ATTR** : Describes memory region attributes. When user page table allocates a new page, its L3 entry should be normal memory. Likewise, when kernel page table sets L3 entries, they should be normal memory. On the other hand, for the memory range from `IO_BASE` to `IO_BASE_END`, they should have device-memory entries in L3 page table.
- **NS** : We don’t consider this for our system
- **AP** : Sets data access permission of the entry. The kernel page table should have `KERN_RW` permission while the user page table should have `USER_RW` permission.

- **SH** : Shareability field. Normal memory space should set their entries as inner shareable while device memory should set theirs as outer shareable.
- **AF** : Should be set when first accessed. Make sure you set this bit whenever you make a page table entry; in our implementation, we will assume all pages are being used.

Now open the `lib/aarch64/src/vmsa.rs` file and review the pre-defined attributes, `RawL2Entry` bits and `RawL3Entry` bits. You can find `defbit!` macro defined in `lib/aarch64/src/macros.rs` file. Match your understanding with the library codes and learn how to use methods implemented for each entry.

Our OS Memory Space Layout

Now that you have knowledge about the small details of the MMU and the page tables. It is time to discuss how we are designing our operating system memory layout.

As a quick recap, in the virtual space, the kernel addresses will start from virtual address `0x0`, while the user addresses will start from virtual address `0xffff_ffff_c000_0000`. This is how we divided the virtual space, but what about the physical space?

We will design our operating system as follows: Once our kernel start running, we will create the kernel page table, and map all available physical memory to the kernel virtual memory. That is, we will use our `memory_map` function, that we implemented in Lab 3, to give us the end of the available physical memory. Then, we will start at address `0x0` in both address spaces, virtual and physical, and insert a page table entry for every page until the end limit we got from `memory_map`. This way, the kernel can continue to use the physical addresses we have been using without change, and we made the MMU happy by providing the table it needs!

We will also do the same for memory mapped IO devices. We will map the region between `IO_BASE` and `IO_BASE_END` to the same respective virtual address for every page. That way, we can continue to use the IO devices with the same addresses we have. You may find the definition of `IO_BASE` and `IO_BASE_END` in `pi/src/common.rs`.

Note that since our rpi memory is 1GB, we can design the kernel page table exactly as we did in the user page table: One L2 table and two L3 tables.

But what about the user virtual memory? If we map all physical memory to the kernel virtual memory then what will we map to the user virtual space?

Actually, mapping the pages to the kernel virtual memory does NOT mean they are allocated! We are only creating this mapping so that the kernel can manage the physical memory using the virtual space.

So now, how does the memory allocation look like? If you remember, our allocator is managing the memory given back by `memory_map`, which uses a physical address.

In case the kernel needs to allocate memory, we can continue to use the allocator as usual, since the physical address space the allocator uses and the kernel's virtual space are identical; thanks to the kernel page table we built in the beginning.

In case the user needs to allocate memory, we will use our allocator too, but then we will need to insert a proper page table entry in that context's user page table, so that the translation happen properly.

With this design, the allocator ensures that the kernel and the user have their demand of memory, without the kernel accidentally using a page that is allocated to the user. The MMU ensures that the user does not have access to the kernel pages since we are flagging the kernel permission for kernel pages, and we do not need to change any of the physical addresses we have been using in the kernel since the virtual address for the kernel is no different, pretty cool!

Implementation

You're now ready to implement a two level page table with 64KB granule starting at level 2. You'll be primarily working in `kern/src/vm/pagetable.rs`, `kern/src/process.rs` and `kern/src/vm.rs`.

We recommend the following approach:

1. **Implement `L2PageTable`, `L3Entry` and `L3PageTable` struct.**

   Feel free to add useful traits to convert {usize, u64, i32,.. } types to `PhysicalAddr` and `VirtualAddr`, or some functions to convert `PhysicalAddr` and `VirtualAddr` to {usize, u64, *mut u8,...} such as `as_usize()`, `as_u64()`, and `as_mut_ptr()`.

2. **Implement** `PageTable` struct.

   There are six methods needs to be implemented for `PageTable`. The requirements for each methods are specified in doc-comments within a skeleton code. Note that we are supporting virtual memory space up to 1GB memory. Thus, two L3 page table is enough to cover the space and L2 page table should have entries no more than 2. Whenever you make a page table entry, please make sure you set the proper bits value.

3. **Implement** `IntoIterator` for `&PageTable`.

   The returned iterator should iterates from the first entry of the first L3 page table and moves on to the second L3 page table.

   Hint

   The chain() method will be useful.

4. **Implement `KernPageTable::new()` method.**

   `KernPageTable` is a struct that internally has smart pointer of `PageTable`. First, create a `PageTable` with proper permission for the kernel page table.

   Then, load all the memory space residing in SDRAM starting from 0x0000_0000. You can get ending address by calling `allocator::memory_map()` function implemented in the lab3. Note that this space is normal memory that is internally shared. Set proper settings for L3 entries and set those entries in the page table.

   Then, set L3 entries for I/O memory range from `IO_BASE` to `IO_BASE_END` as device memory. The range variables are defined in `pi/src/common.rs`. The entries should have same settings with the one for normal memory space, except that its attribute should be set as *device memory* and it is outer sharable. Also, make sure to set the proper bits value for each page table entry you create.

   Note that we are using 64KB page granularity for both cases.

5. **Implement `UserPageTable` struct.**

   For the user page table, you don’t need to set any pages into the pagetable in creation time.

   When `alloc()` function is called, allocates a 64KB page, creates an L3 entry with proper settings, set the physical address of the allocated page to L3 entry’s `ADDR` field, and set the entry in the page table. Before looking up the page table with virtual address, note that the virtual address for user process starts from `USER_IMG_BASE`, which is `0xffff_ffff_c000_0000`. You have to subtract this base address from the virtual address to lookup the page table and set entries. Make sure to set the proper bits value for each page table entry you create.

6. **Implement `Drop` traits for `UserPageTable`.**

   To implement `Drop` traits, iterate the internal pagetable and `dealloc()` each entry that exists.

7. **Let a `process` struct have a `UserPageTable`.**

   You can simple do this by uncommenting `pub vmap: Box<UserPageTable>,` line within the `Process` struct in `process.rs`. Modify `Process::new()` method to return the `Process` struct holding `vmap`.

8. **Finally, implement `VMManager` in `kern/src/vm.rs`.**

The virtual memory manager is a thread-safe wrapper around a kernel page table. Before its first use, it must be initialized by calling `initialize()` which internally calls `setup()` method. Setting up the virtual machine manager includes configuring some proper values to the several relevant registers.

Here is the registers used by the `setup()` method :

- `ID_AA64MMFR0_EL1` : AArch64 Memory Model Feature Register 0 (ref: D7.2.43)

  `Tgran64` field is used to check whether the current system support 64KB memory translation granule size or not. `startup()` panics if it's not supported. `PARange` field is used to check the range supported for physical address.

- `MAIR_EL1` : Memory Attribute Indirection Register (EL1) (ref: D7.2.70)

  Provides the memory attribute encoding corresponding to the possible AttrIndx.

- `TCR_EL1` : Translation Control Register (ref: D7.2.91)

  It controls other memory management features at EL1 and EL0. (guide: 12.2)

- `TTBR0_EL1` and `TTBR1_EL1`

  Specifies the base address of the translation table.

- `SCTLR_EL1` : System Control Register (ref: D7.2.88)

  Provides top level control of the system including memory system.

Now complete the implementation for `VMManager` by writing codes for unimplemented methods.

Hint

Instead of using raw bit and bit operation, use wrapper functions and variables from `aarch64` library to prevent potential typos and mistakes.

Hint

We also have interface for registers in `aarch64` library.

Why do we need `Deref` and `DerefMut` traits? (Deref)

In `pagetable.rs`, we have `Deref` and `DerefMut` traits for `KernPageTable` and `UserPageTable`. What are the roles of these traits and why do we need them?

! We do not support page access rights permissions

Even though in the implementation of `UserPageTable::alloc` you might see `_perm` for memory page access rights permission (i.e. RW pages or RWX), we do not actually use it. It is there just as a stub for future support of this feature. Note, however, that we do support MMU page permissions isolating the kernel and user space pages. We do this by setting the proper values in the page table entries, `AP`.

Testing Virtual Memory

First, we'll add one line of code in `kern/src/main.rs`. Call `initialize()` method for `VMM` before calling scheduler initialization. Thus, your `kmain()` will now have six lines like below:

```
unsafe fn kmain() -> ! {
    ALLOCATOR.initialize();
    FILESYSTEM.initialize();
    IRQ.initialize();
    VMM.initialize();
    SCHEDULER.initialize();
    SCHEDULER.start();
}
```

As your kernel now enables virtual memory, your process should have a user page table. We're going to implement several user programs as well as loading those programs as a process in the next phase. Before moving to the next phase, let's simply test whether your user process can be run with user page table.

In `initialize()` function in `kern/src/process/scheduler.rs`. We can start using the virtual address instead of the physical address.

Since we do not have any process in the virtual space, yet. We have provided a test function `test_phase_3`. This function, given a process, will allocate a new page at address `USER_IMG_BASE` in the process's virtual space, and will copy the function `test_user_process` to that page. So that now if we set `elr` to `USER_IMG_BASE`, it should run `test_user_process`.

Set `ttbr0` and `ttbr1` appropriately for each process. `ttbr0` should have the base address of the kernel page table while `ttbr1` should have the base address for the user page table. Then, call `test_phase_3()` function for each process, so that they have something to execute at `USER_IMG_BASE`, and set `elr` to `USER_IMG_BASE`. `test_user_process` is a simple function that invokes a sleep system call. Finally, add the processes to the scheduler queue.

If everything works well, you can see your each process sleeps for 10 seconds falling to the `Waiting` status right after it is scheduled. You may want to print timer interrupts and scheduling queue to check it.

Phase 4: Programs In The Disk

In this phase, you will further modify `Process` structure implementation to load programs binaries from the filesystem image to create a process.

Subphase A: Load A Program

In the previous phase, we generated user-level processes using `extern` function and started them by passing the address of the `extern` function to the `ELR` register of trap frame. However, implementing all the user programs in the kernel code is not practical nor safe. Instead, we'll implement features to load programs written by the user outside of the kernel, compiled and stored in the disk as binary.

In this subphase, you will be working in `kern/src/process/process.rs`.

To convert a program binary from the disk to a `Process` struct, you need the following steps:

1. Open the binary file, and create a process object.
2. Allocate a 64KB page within the process's user page table for user process's *stack*. This page should have read and write permission.
3. Allocate a 64KB page within the process's user page table with virtual address starting from `USER_IMG_BASE`. This page should have read, write and execute permissions. Start reading the binary file and store it in the allocated page. Keep allocating additional pages until you are done reading the whole binary file.
4. Set the trap frame for the process with the proper values.

Implementation

First, complete the helper functions. There are four functions returning `VirtualAddr` in `kern/src/process/process.rs`. You can find relevant constant variables defined in `kern/src/param.rs`

- `get_max_va()` : Returns the highest virtual address for user process.
- `get_image_base()` : Returns the base address for the user virtual memory space.
- `get_stack_base()` : Returns the base address for the user process's stack in the virtual space. Since stacks grow from higher memory address to lower memory addresses, a good stack base would be the last page in the user's virtual space. Note that the address should be aligned by PAGE_SIZE.
- `get_stack_top()` : Returns the top of the user process's stack. This will be set to the stack pointer once the process is run. It should be the maximum stack pointer possible. Remember, in ARM, stack pointers are 16 byte aligned.

Then, complete `load()` and `do_load()` methods.

`do_load()` method gets a path to the file as a parameter and returns a wrapped `Process` struct. `do_load` needs to create a new process struct, allocate the stack in process virtual space, opens a file at the given path and read its content into the process virtual space starting at address `USER_IMG_BASE`.

`load()` method internally call `do_load()` method. Then, it should sets the trap frame for the process with the proper virtual addresses in order to make the process run with user page table. Finally, it returns the process object ready to be run.

Compiling User Programs

Now that your operating systems support loading and running user programs for the file system, we should give it a run. But first, we need to compile the user programs to our OS needs.

In commodity OSes, like Linux, executables are usually compiled into an executable file format such as `elf`. The process of loading user programs into memory and getting them to run is usually more complicated than what we do in our operating system. You may read about how it is done in the Linux kernel How programs get run: ELF binaries.

For our operating system, however, we do not have an elf interpreter, and so we will need to compile our programs into a plain binary file that expect to be loaded at address `USER_IMG_BASE`.

We have provided two user programs ready for you to compile at `user` directory. `fib` actually needs the system call `write` which we have not implemented yet. So, we will base our example on `sleep`, which have the same behavior as the `test_user_process` function you have seen before.

To compile `sleep`, run `make` at `user/sleep`. This will generate two files at `user/sleep/build`: `sleep.bin` and `sleep.elf`.

`sleep.elf` is the `elf` version of the sleep program. Since we do not have an `elf` interpreter, we cannot use it in our operating system. Instead, we will use `sleep.bin`, which is literally just a stream of the executable bytes.

Once you have compiled `sleep`, copy `sleep.bin` into the SD card, and move on to the testing phase.

Test your implementation

We provides two user programs `fib` and `sleep` under `user` directory. As the name suggests, `sleep` is a program that calls the `sleep` system call. It has the exact same codes as the previous `test_user_process()` function. The behaviour of `fib` is also very simple. It just calculates 40th fibonacci number with recursive `fib()` function. You can find the source code in `user/fib/src/main.rs`.

Then, revisit the `kern/src/process/scheduler.rs`. Now, we don't have to manually set the registers and add the processes created from an extern function, but we can load multiple programs from our disk using `Process::load` function. Load four `sleep` programs and add them in the scheduling queue in `initialize()` method.

Finally, modify inline assembly in `GlobalScheduler::start()` function. Instead of the address of `_start` function, calculate the address of the next page and store it to the `sp` register. This way, we will have a clean page for the stack when we return to the kernel level. Before calling `eret`, don't forget to clean any register that may leak information or addresses from the kernel.

After boot up, if the four `sleep` programs starts, you can continue to the next subphase. Note that you can't test `fib` program yet because we haven't implemented `write` system call. We will implement it and some more system calls in the next subphase.

If you wish to use `qemu`, and need a file system image that have the user programs inside of it, you can use `user/build.sh` script. This script will generates `fs.img` that contains compiled binary of `sleep` and `fib` program.

Subphase B: User Processes

In this subphase, we will extend our system call library to make it more useful to user programs. We will also write and compile an example user program in Rust, which should be run on top of our operating system. Finally, we will apply changes in our system call handler to add the new system calls. You will be working primarily in `lib/kernel_api/src/syscall.rs` and `kern/src/traps/syscall.rs`.

User Programs

First, we will build the kernel api library for our operating system. This library will be imported by user level programs to initiate a system call to our operating system.

Start by implementing functions in `lib/kernel_api/src/syscall.rs`. Each function should be implemented to invoke system call corresponding to the name of the function.

To give you an example, lets walk through how does system call are handled in our system. The user program import the library `kernel_api`, and use its functions. The functions in `kernel_api` will implement the required assembly to issue the system call to the operating system, with the appropriate system call number and arguments. This will trap into the operating system `handle_exception`, which will forward the request to `handle_syscall`.

Similarly, `handle_syscall` will categorize the request based on the syscall number, and forward it to the responsible handler. The handler will execute the required functionality of the system call, and return back through the chain of calls to the user level.

Start implementing the library functions at `lib/kernel_api/src/syscall.rs`. All you have to do in these function is to prepare the argument for the required system call, and issue the `svc` instruction. We have provided `sleep` as an example. The system call numbers are defined in `lib/kernel_api/src/lib.rs`. Please refer to `lib/kernel_api/src/lib.rs` to find pre-defined variables for system call number.

Then, for each system call you implemented, make sure you complete the full chain of handling it just as we described above, i.e. add it to `handle_syscall` and write the respective `sys_{SYSCALL}` function in `kern/src/traps/syscall.rs`.

We recommend you to start from implementing `write` system call. When you are done implementing it, you can now run multiple `fib` programs with the same way you did for the `sleep` program.

If all is well, you can see `fib` processes running and their output is printed.

Note that you have to recompile the user programs whenever you are making changes to `kernel_api`.

When you are done with implementing every functions in the `kernel_api` user library as well as its system call handler in the kernel, try to call and test every system call in the `fib` program. Also, if you feel adventurous, write your own programs. You can copy the same build system from `fib` and use it in your program to generate a working binary.

! Check your code before submit!

Before submitting your code, please change your `TICK` variable in `kern/src/param.rs` to 10 ms. In addition, please check you are not leaving any system calls nor exceptions you used for debugging. Your `main` should have all 6 lines: 5 `initialize()` calls and 1 `start()` call.

Submission

Once you've completed the tasks above, you're done and ready to submit! Congratulations!

You can call `make check` in `tut/4-spawn` directory to check if you've answered every question.

Once you've completed the tasks above, you're done and ready to submit! Ensure you've committed your changes. Any uncommitted changes *will not* be visible to us, thus unconsidered for grading.

When you're ready, push a commit to your GitHub repository with a tag named `lab4-done`.

```
# submit lab4
$ git tag lab4-done
$ git push --tags
```