Systems II
Project 1, Parts A & B & C
The beginnings of a kernel

1 Overview

Now that we have a BIOS that can load a kernel, we need to start the kernel itself. It must "take control" of the processor, load a *process* (a running program), and jump to it, prepared to handle the possible interrupts that will occur. More specifically, our kernel must establish a *trap table* full of pointers to *interrupt handlers*. One of the interrupt handlers, the *system call handler*, must provide the capability for a process to intentionally vector to the kernel for a particular task.

2 Getting started

2.1 Getting the project source

You can download and then unpack the source files from the command-line, and then look at the files:

```
(sys2) $ cd ~
(sys2) $ curl -L https://bit.ly/cosc-275-2526s-p1 -o project-1.tar.xz
(sys2) $ tar -xJvpf project-1.tar.xz
(sys2) $ cd project-1
(sys2) $ ls
prof-bios.asm  kernel.asm
```

The `prof-bios.asm` file is complete. You are welcome to use it, but I recommend building on your own work by using your `bios.asm` from your `project-0/` directory:[1]

```
(sys2) $ cp ../project-0/bios.asm .
```

The `kernel.asm` file is incomplete. It creates an initial stack for the kernel and calls `main()`, which currently prints some messages to the console and then exits. Your work, described below in Section 3, will be to add to this kernel code.

[1] Always keep an eye out for the trailing *period* on commands like this one. Notice that there is a *space* before the dot. For the shell, a single dot (`.`) refers to the current directory, while a double-dot (`..`) specifies the parent directory.

2.2 Assembling and running

As before, each of the source code files needs to be assembled independently, and then they can be run together in the simulator:

```
(sys2) $ f-assemble bios.asm
(sys2) $ f-assemble kernel.asm
(sys2) $ f-assemble do-nothing.asm
(sys2) $ f-simulate bios.vmx kernel.vmx do-nothing.vmx
```

In the simulator, set the simulator to run until the code halts (with the `step -1` command). You will see lots of instructions fly by, and you will see messages printed on the console.[2] The BIOS will load the kernel and then jump to it. Ultimately, it will end when its `main()` returns and the kernel then halts. You can then check the console (if needed), and you can see, in register `a0`, the exit code. That exit code should be `a0 = 0xffff0000`,[3] and the console should look like this:

```
Fivish BIOS v.2026-01-24
(c) Scott F. Kaplan / sfkaplan@amherst.edu
  Searching device table for kernel...done.
  Copying kernel into RAM...done.
  Vectoring into kernel.
Fivish kernel v.2026-02-07
COSC-275 : Systems-II
Initializing trap table...done.
Halting kernel..._
```

3 Your assignments

This project is **divided into multiple parts, each with its own work to submit**. Each part adds and tests new capabilities; each part also builds on the previous one, so be sure to complete each fully before moving onto the next.

3.1 Part A: Handle interrupts, run one process

Begin by modifying the kernel to handle interrupts; then have it load one user process and execute it. Have that process terminate itself via a *system call*.

[2] The simulator tries to figure out your environment when it starts. If it is able to open up windows to show the console and the registers, it will. If it cannot do that, it will try to show you a *smart text* interface that splits the terminal window to show those same things. If it can't do either of those, it you will need to use the `show registers text` and `show console text` commands to see them printed.

[3] If you look at the `.Numeric` section of `kernel.asm`, you will see that `0xffff0000` is labeled `kernel_normal_exit`, implying that it is the exit code for when no errors occurred.

Notice that the kernel already contains a procedure, `default_handler()`, that sets an error code and halts the processor. This handler should be used for every interrupt for which we do not yet have a more specific handler function.

1. **Create and initialize a trap table:** When the kernel begins execution, it should create and set the entries of a *trap table*. (See Section 4 of the Fivish documentation for details on each interrupt.) You may, initially, have all of the entries point to `default_handler()`, but with one exception: you shold write a `syscall_handler()` function that is called in the event of a `SYSTEM_CALL` interrupt.

2. **Set the CPU to handle interrupts:** Set the processor's trap base register (the `tb` CSR) with the base address of the trap table.

3. **Load a process:** The second half of RAM (starting at `kernel_limit`, ending at `RAM_limit`) is set aside for loading a user-level program. Adapt code from the BIOS to find the 3^{rd} ROM (which can be the included `do-nothing` program), copy it into that space, and then execute that program (which requires use of the `eret` instruction). It will exit by using the `ecall` instruction to generate a `SYSTEM_CALL` interrupt.

3.2 Part B: Expand system calls

Previously, the system call handler needed only to terminate the user process (and halt the kernel). Here, we expand the set of operations that can be requested via *syscall*. The user process is expected to place a *syscall code* into register `a0` immediately before an `ecall` instruction to trigger the interrupt into the kernel. This code needs to be interpreted by `syscall_handler()` to determine what service to perform. The operations and codes are:

1. `0xca110000` — `NOOP`: This is the system call that *does nothing*. It simply resumes the calling process at its next instruction.

2. `0xca110001` — `EXIT`: End the user process, never returning to it. As before, this operation should halt the kernel.

3. `0xca110002` — `PRINT`: The user process should provide, in register `a1`, the address of a *null-terminated string* to be printed to the console. The kernel should use its own `print()` function to print that string, and then the kernel should resume the calling process at its next instruction.

Update your `syscall_handler()` function to determine which operation is being requested, and then perform that operation. Continue to break down tasks in the kernel by writing

separate functions. Have `syscall_handler()` call a separate function for each of the three *syscall* operations.[4]

3.3 Part C: Run multiple processes in sequence

Rather than load a single user process, modify the kernel to run a sequence of user processes. If the simulator is run with 5 ROMs, then ROMs 3, then 4, then 5 should each be loaded into RAM and executed. When each process performs an `EXIT` *syscall*, the kernel should then advance to the next ROM. When no more ROMs remain, the kernel should halt.

Additionally, each process should be run in a *single-segment virtual address space* by setting the processor into virtual addressing mode, and by setting the *base* (`bs`) and *limit* (`lm`) registers for the *memory management unit (MMU)*. This change will require alterations to how the kernel uses any user-process addresses (e.g., the string address passed in `a1` for the `PRINT` *syscall*).

4 Submitting your work

Copy your `kernel.asm`, as well as any new or modified test programs, into the `project-1a`, `project-1b`, and `project-1c` directories in your shared Google Drive folder as you complete each part of the project.

Part A: Due at **11:59 pm** on **Fri, Feb-13**
Part B: Due at **11:59 pm** on **Wed, Feb-18**
Part C: Due at **11:59 pm** on **Mon, Feb-23**

[4]There will be additional system call operations that we will add as the semester goes on, so make this code nicely organized so that adding more codes and operations will be simple.