Operating Systems — Project 1
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 processor, load a *process*, 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 Updating ***Fivish***

The assembler and simulator have been updated with a handful of bug-fixes. You should begin by updating that code and then compiling it. Open a shell on whatever system you have been working on the assignments, and change into your `fivish/` directory, and then update and compile like so:

```
$ cd fivish
$ git pull
$ cd assembler
$ make
$ cd ../simulator
$ make
$ cd ..
```

This sequence of steps should leave you back at your `fivish/` directory, with the newly updated assembler and compiler ready to run. Note that you can check the file `version.txt` in the base level directory to be sure that you've pulled the latest update; it should read `v.2024-02-17`.

2.2 Getting the project source

Download the source assembly files:

`bit.ly/cosc-277-p1`

You can download and then unpack the source files from the command-line like so, and then take a look at the files.

```
$ wget -i https://bit.ly/cosc-277-p1
$ tar -xJvpf project-1.tar.xz
$ cd project-1
$ ls
bios.asm  kernel.asm
```

The `bios.asm` file is complete. You are welcome to use it, or to copy your own `bios.asm` from your `project-0/` directory and use that. You may wish too look at how I implemented `copy_kernel()`, using a previously undescribed feature of the bus controller known as *direct memory access (DMA)*. With this feature, the hardware will copy any range of (valid) addresses to any other, eliminating the need for a word-by-word load-and-store loop. Can you see how this capability is used?

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.

2.3 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.

```
$ cd assembler
$ ./assemble ../project-1/bios.asm
$ ./assemble ../project-1/kernel.asm
$ cd ../simulator
$ ./simulate ../project-1/bios.vmx ../project-1/kernel.vmx
```

Once in the simulator, you can open the console in a separate window with the `showconsole graphic` command. Note that this command may fail if you are connected to one of the Linux servers (e.g., `remus.amherst.edu`) and have not set up *X11 tunneling*. If you don't know how to do that, send a question on Slack, and I will try to provide some directions (depending on your type of computer). As an alternative, you can always use the command `showconsole text` to have the current console printed to the terminal.

In the simulator, open the console window (if you can), and then we will set the simulator to run until the code halts. You will see lots of instructions fly by, and if you are able to watch the console change, you will see messages printed. 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:

```
[pc = 0x00015000]: until -1
...
[pc = 0x000030a8]: showconsole text
Console @0x00019000
----- CONSOLE BEGINS -----
```

```
Fivish BIOS r1 2023-12-17
(c) Scott F. Kaplan / sfkaplan@amherst.edu
  Searching device table for kernel...done.
  Copying kernel into RAM...done.
  Vectoring into kernel.
Fivish kernel r1 2024-02-07
COSC-277 : Operating Systems
Initializing trap table...done.
_
----- CONSOLE ENDS   -----
[pc = 0x000030a8]: showregister a0
a0 = 0xffff0000
```

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

An aside about naming: Notice that, in `kernel.asm`, many of the labels use follow a particular prefix-based naming convention. Specifically:

- `_procedure_`: The entry point label to every procedure begins with this prefix. What we think of as `print()` begins with the label `_procedure_print:`.
- `_static_`: Every word-sized value in the `.Numeric` segment is prefixed this way. The console width is labeled `_static_console_width`.
- `_string_`: Each string in the `.Text` segment is labeled with a name using this prefix.

I strongly urge you to use this naming convention. We soon will introduce a compiler for a simple, higher-level language that wiill generate assembly for us. The compiler's output will use this naming convention. Therefore, we want to use these same naming conventions so that procedures and statically allocated numbers and strings can easily be used both in our assembly code **and** in our higher-level code that will be compiled.

3 Your assignment

3.1 Handle interrupts

Begin by modifying the kernel to handle interrupts. Notice that the kernel already contains a procedure, `default_handler()`, that sets an error code and halts the processor.

1. **Create and initialize a trap table:** When the kernel begins execution, it should create and set the entries of a *trap table*. (See Chapter 4 of the Fivish documentation for details on each interrupt.) You may, initially, have all of the entries point to `default_handler()`.

2. **Set the CPU to handle interrupts:** Set the processor's TBR with the base address of the trap table, and its IBR with the base address of the interrupt buffer. Then test your code to be sure that interrupts trigger a vector to a handler function.

3. **Change the system call handler:** Write a new procedure meant to handle system calls. The `SYSTEM_CALL` interrupt should be redirected to this procedure, which should print a message to the console before halting. We will expand this procedure later.

3.2 Create a process

Once interrupt-handling is established, assume that the simulator will now be run with a **third** executable file (`.vmx`), thus creating a third ROM. This third executable should be a *regular, user-level program* that does something simple (e.g., adding a few numbers).

1. **Load the first program:** The kernel should find the third ROM—that is, the first user program—and load its contents into a free portion of main memory.

2. **Execute the program:** Jump from the kernel to the beginning of the loaded user program. This program must execute in *user mode* (in contrast to the *supervisor mode* in which the kernel executes).

3. *Implement the `EXIT` system call:* The `SYSTEM_CALL` interrupt should be handled by code that examines a chosen register that contains a *system call code.* Assign some constant to be the code for the `EXIT` syscall. The syscall handler should examine this location, determine if an `EXIT` was requested, and then do it. That is, end the program's execution by printing a message and then…

4. *Execute a sequence of user programs:* After one user program properly exits, find the *next* ROM that holds a user program, and then load and execute that one. When there are no more ROMs, the kernel should print a message and then halt with a success code.

4 How to submit your work

Copy your `kernel.asm` into the `project-1/` directory in your shared Google Drive folder for this class. Also copy any user-level programs you have created to test your kernel.

This assignment is due at **11:59 pm** on **Friday, Mar-01**.