

# SYSTEMS II — PROJECT 2

## Revision 0 — 2015-Apr-04

### Processes

## 1 Overview and motivation

We assume, at the start of this project, that you have a working BIOS as well as a fledgling kernel that can establish a trap table and interrupt buffer, thus taking control of the hardware. In Project-1, this kernel loaded and ran a **single** *user-level process*, which then ran until an interrupt occurred, triggering a vector into the kernel, which then, in turn halted the machine.

For this project, you update your kernel to **load and execute multiple processes**. That is, your kernel should be able to create a number of processes and then switch periodically between them, thus scheduling a bit of each at a time.

## 2 Your assignment

You must write a second version of your kernel in assembly code (`kernel.asm`) such that, once loaded and executed by the BIOS, it does the following:

1. All of the things done by your current kernel from Project-1: setting up the trap table and interrupt buffer such that the kernel has control of the hardware.
2. Create, as the first process, the special user-level program `init` (written as `init.asm` and assembled into `init.vmx`), assumed to be the 3<sup>rd</sup> ROM as given at the command line for the simulator. This user-level process must use the `CREATE` system call (see below) to run one process each for the 4<sup>th</sup> through  $n^{\text{th}}$  process (assuming  $n$  total ROM images (a.k.a., ".vmx" files) listed at the command line).
3. Each process, including the `init` process, should use a simple *single-segment allocation base/limit virtual memory model*. When a given process is *scheduled* on the CPU (that is, made to run until an interrupt occurs), the kernel should set its the *base* and *limit* registers, and turn on *virtual addressing* (but *not* paging) so that the MMU will enforce the *base/limit* addresses.
4. *Schedule* that first process on the CPU. That is, `JUMPM`D to the copied user-level program's first machine code instruction. Be sure to set all relevant registers properly prior to this jump into *user-space*.
5. Write all of the *interrupt handlers* needed to support these processes. For some interrupts (e.g., `DIVIDE_BY_ZERO`), the handler cannot address the problem, and so it should simply terminate the interrupted process and then schedule another. If the kernel itself is interrupted due to an error, it should *panic* and halt with some kind of error indication (e.g., a message on the console). If the process table ever becomes empty, the kernel should halt indicating an error-free shut-down.

6. Among the interrupt handlers, be sure to write a *system call handler* that can allow each process to intentionally interrupt itself with the SYSC psuedo-instruction, passing along an encoded request (e.g., via one of a set of specially selected constants in some register) for the kernel to perform some function. The system call handler should figure out which of its own functions to call based on this encoded request. Among these functions must be at least the following set of system calls:

- EXIT: Terminate the process.
- CREATE: Create another process.
- GET\_ROM\_COUNT: Return the total number of ROM's available in the system.
- **[Optional]** PRINT: Print a string of characters to the console device. (Note that a *null byte*—a byte whose value is 0—marks the string's end.)

**Regarding functions:** Your kernel will be sufficiently complex at this point that it must be divided into a set of functions, with an *initial entry point* (e.g., `main()`) that is the first function called. Thus, the kernel must give itself space sufficient to maintain an *activation stack*.

Keep in mind that each interrupt handler is a special type of function—one that is called via the trap table when an interrupt triggers a vector into the kernel—and is known as a *re-entry point*. Although it is not called in the same way normal functions are, it is often will preserve the interrupted process's registers and **then perform a normal function call** to some other part of the kernel to perform the necessary tasks.

### 3 Some helpful code

If you follow the link below,<sup>1</sup> you will get a copy of my `kernel-stub.asm` code. Like code begins with an initial entry point labeled `__start`, where it finds the base and limit addresses for RAM in the physical address space, sets some registers to store those boundaries, and then performs a *caller prologue* before CALLing the function `_procedure_main`. Of course, this function doesn't yet appear in this starter code; you must provide it.

Additionally, this assembly code has a number of pre-written functions, including one for finding devies in the device table, and a group for printing null-terminated text strings to the console.

The link is:

<https://www.amherst.edu/~sfkaplan/courses/2015/spring/COSC-261/projects/kernel-stub.asm>

### 4 How to submit your work

Use the CS submission systems to submit your work. Specifically, you will need to submit your `kernel.asm` and `init.asm` files, as well as **at least two user-level programs** as `.asm` files.

---

<sup>1</sup>Note that the link is itself clickable, even if your PDF viewer doesn't show it. If you don't just click it, and instead copy and paste the link, be sure to **retype the tilde** in the URL, since the copied one from the document may not match the normal one you type.

These must be constructed for your kernel by performing system calls exactly as your kernel expects them. You may optionally submit an updated `bios.asm` file if you have changed it since your Project-1 submission.

This assignment is due at **11:59 pm** on **Monday, April 20<sup>th</sup>**.