

SYSTEMS II

PROJECT 2

Multitasking processes

1 Overview

Given the kernel that you developed in Project 1, which is capable of running a number of processes in sequence, we transition now to a kernel that loads multiple processes at once, and then interleaves execution between them. In short, we make our kernel support *multi-programmed* execution.

2 Getting started

2.1 Updating *Fivish*

Fivish has received a few bug fixes and minor (but essential) changes. To update your copy of it, open a shell, activate your `sys2` environment, change into your `fivish/` directory. Then update and compile like so:

```
$ micromamba activate sys2
(sys2) $ cd fivish
(sys2) $ git pull
(sys2) $ make
```

2.2 Enabling C code

Fivish includes the ability to compile C code and include it in our kernel and programs. To make the use of these capabilities possible, use the following commands to make the new `f-compile` and `f-build` commands available:

```
(sys2) $ cd compiler
(sys2) $ ln -s $PWD/f-compile ~/.local/bin/
(sys2) $ ln -s $PWD/f-build ~/.local/bin/
(sys2) $ cd ..
```

2.3 Getting the project source

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

```
(sys2) $ cd ~
(sys2) $ curl -L https://bit.ly/cosc-275-2627F-proj2 -o project-2.tar.xz
(sys2) $ tar -xJvpf project-2.tar.xz
(sys2) $ cd project-2
(sys2) $ ls
bios/ kernel/ programs/
```

The bios/ directory: This directory contains a working BIOS that you can use. Feel free also to replace it by copying your own BIOS code from an earlier project into this directory.

The kernel/ directory: The kernel source is no longer a single assembly file; it is now a number of files. The kernel provided here fulfills the Project 1 requirements—it will batch-process each user-program ROM available, one at a time to completion, in sequence, until all of the ROMs have been used. The code for critical parts of the Project 1 code has been moved to C source. Additionally, there are additional code files—some complete and some incomplete—that aren’t used by this Project 1 kernel at all. They exist because you will need them as you develop your Project 2 kernel.

Here are the files, with some info on each:

- **kernel_stub.asm:**¹ This assembly source contains much of the initial code for Project 1, still containing a number of familiar functions (e.g., `main()`, `find_device()`, `print()`). They exist to get the kernel started, but now call on C functions to do most of the work. (For example, see `syscall_handler()`, which performs a few critical operations before calling `do_syscall()`, a C function in `kernel.c`, described below.) `main()` now initializes the trap table and then calls `run_init()` (another C function) to start the first process.

¹Observe that this file is **not** named `kernel-stub.asm`, with a *dash* separating `kernel` and `stub`. It is `kernel_stub.asm`, using an *underscore* separator. When our C code is compiled into assembly, the name of the file is incorporated into the assembly labels—and labels, in the *Fivish* assembler, cannot contain the *dash* (–) character. Never name one of your C source code files using the *dash*; instead, as we have here, use the *underscore* (–) character.

The primary purpose of any code you add to this file should be to perform operations that require specific RISC-V opcodes (e.g., the `csr*` instructions) that cannot be directly expressed in C. The rest of the work should be written in C.

- **kernel_stub.h**: This is a *C header file*—somewhat similar to a Java *interface*—that lists the functions and global/static variables defined in **kernel_stub.asm**. The C compiler needs these definitions for any C code that uses those variables or calls those functions.
- **kernel.c**: This is where most of our kernel code will go. Note that it contains, among other critical functions, `do_syscall()`, which is now the function that determines which system call was requested, and then calls an appropriate function to perform that system call. It also handles returning from system calls (where appropriate) to the calling process. It relies on functions defined in **kernel_stub.asm** that perform specialized RISC-V operations (e.g., `userspace_jump()` performs the actual `eret` to jump into userspace.)
- **mm_alloc.h** and **mm_alloc.c**: This module defines the functions for allocating and deallocating blocks of RAM for user-level process space. The initial kernel code does not use these functions at all, but you should use them for the Project 2 kernel. These functions are complete, but you should examine them to see how they work.
- **heap_alloc.h** and **heap_alloc.c**: This module provides the functions for allocating objects on the kernel's heap (between the statics and the stack). Again, these are not used by the Project 1 kernel, but they will be needed for Project 2. The `TO BE COMPLETED` comments mark the incomplete elements of this code. Notice that code already exists to initialize the data structures that organize the heap.
- **io.h** and **io.c**: A pre-made module for generating output. These functions are complete and can be used wherever needed.
- **types.h**: Another *C header file*, which defines some data types and constants that we will use in **kernel.c** and other C code that composes the kernel.
- **stub_end.asm**: A small bit of assembly to be placed at the end of the combined result, providing a marker for where the *statics* region ends, and thus where the *heap* can start.
- **Makefile**: Used by the `make` command to determine how to build the kernel.

User programs: The `programs/` directory contains a few user-level programs that you are invited to use, change, enhance, or augment in order to test your kernel. Some of the provided programs are from Project 1 (e.g., `do-nothing`), and can be assembled and used as usual. There are a few new ones written in C. The programs are:

- `do-nothing.asm`: The classic. Written all in assembly in this single source file.
- `syscall-test.asm`: Another oldie from Project 1, also a single assembly file (`syscall-test.asm`).
- `longloop.c`: A simple program with a loop that counts up and, every so often, prints an update message of its progress.
- `fib.c`: Another C program that calculates a Fibonacci number recursively.²
- `init.c`: This program is the test-case for your final implementation of this project. As described below in Section ??, the kernel will ultimately only start *one* process itself—the *initialization process*. It will then become the task of that process to use system calls to request the start of additional processes. The `init` program is written to be that initialization process.

Each of the C programs depend on some additional source files:

- `stub.asm` and `stub_end.asm`: These provide the initial and final assembly code that is “wrapped around” the assembly generated by the C compiler, `clang`. In `stub.asm`, you will find code that initializes the stack and calls `main()`, and then handles the end of the process when `main()` returns. The actual `main()` itself is assumed to be provided by the C code. The `stub_end.asm` assembly provides an end-of-statics markers in case the C code wants to create and manage a heap for itself.
- `syscall.asm`: Specialized assembly functions for performing system calls. These are written in assembly because they place values in specific registers and then use the `ecall` opcode—things that C code cannot do.

²Calculating Fibonacci numbers recursively is a typical example of double recursion, but it’s also a terrible use of recursion because it is so slow. Nobody who actually needs to calculate a Fibonacci number should do so this way.

- **syscall.h**: The C *header* file. When C code *includes* this header file (written as `#include "syscall.h"`), then that C code is allowed to call the functions defined in `syscall.asm`.
- **types.h**: Definitions of some common types that are used in our code, especially `word_t` and `address_t`, specifying the size and type of a typical value and a typical address in our RISC-V code.
- **utility.h** and **utility.c**: A collection of useful functions that are normally provided by standard C functions that aren't available in our nascent, custom-made system. Again, *including* the header file (`utility.h`) allows C code to call on functions defined in the code file (`utility.c`).

2.4 Building

BIOS: In addition to the BIOS assembly code (which you can replace with your own if you like), you will also find a **Makefile**, which is used by the `make` command to determine how to perform the build (i.e., the assembly and/or compilation). Specifically, to build this BIOS and have it ready to use, and starting from your `project-2` directory:

```
(sys2) $ cd bios
(sys2) $ make
```

Kernel: To build this starter kernel with Project 1 capabilities, do the following, starting from your `project-2/` directory:

```
(sys2) $ cd kernel
(sys2) $ make
```

`f-build`³ on the right combination of source assembly and C files to compile, combine, and assemble it all into a `kernel-rom.vmx` executable and a `kernel-rom.log` assembly information file. If you look at the files in your directory, you will also see other files generated from this build process. For example:

- **kernel.s**: This is the output of the `clang` C compiler on `kernel.c`. It is standard RISC-V assembly generated by the compiler.

³Internally, `f-build` relies on the familiar `f-assemble` to process assembly source code, and the new `f-compile` to process C source code.

- `kernel.asm`: The `kernel.s` file is transformed into *Fivish* assembly, which is a little non-standard, and this file is the result.
- `kernel-rom.asm`: The concatenation of all of the `.asm` files to make a single, large assembly file.

User programs: The programs provided can all be compiled in one fell swoop:

```
(sys2) $ cd programs
(sys2) $ make
```

It is also possible to build an individual user-program. For example, if you change something in `fib.c` and want only to recompile/rebuild that one program:

```
(sys2) $ make fib
```

2.5 Running

To run this kernel on these user programs, it should look much as it did in Project 1, except with a couple of subdirectories appearing in the commands. From your `project-2/` directory, where you may have copied your BIOS, it would look something like this:

```
(sys2) $ f-simulate bios/bios-rom.vmx kernel/kernel-rom.vmx
        programs/do-nothing.vmx programs/syscall-test.vmx
        programs/fib-rom.vmx
```

3 Creating a multitasking kernel

3.1 The setup

For this new, multitasking kernel, we must view the ROMs and processes a bit differently...

- The 3rd ROM must be an `init` program—the executable image of the one process that the kernel runs to create the first process in the system.
- Each subsequent ROMs (4th and beyond) contain additional user programs.

- Via a new system call, `RUN`, a process can request the creation of a new process by specifying the ROM number to load and execute.
- Processes terminate via an `EXIT` system call.
- When all of the processes have terminated, the kernel halts.

3.2 Kernel behavior

To manifest the arrangement described above, the kernel will need to create and manage processes in the following manner...

- Each process is given a *fixed-sized memory allocation* (e.g., $32 \frac{KB}{process}$).
- When running, each process is in *virtual addressing mode*, using single-segment, base/limit address translation and enforcement in the MMU.
- Each process is scheduled for a *scheduling quanta* whose length can easily be changed (e.g., by changing a static's value). The quanta should expire with an `ALARM` interrupt.
- The kernel's *CPU scheduler* should be invoked at the end of a scheduling quantum, choosing the next process to run for the next quantum.

3.3 Kernel functions (a.k.a., *What you need to do*)

You will need to implement the following capabilities in the kernel in order to implement this kernel (and system) behavior:

1. Implement the `allocate()` and `deallocate()` functions in `heap_alloc.c`, providing the kernel with a heap.
2. Implement the `RUN` system call by writing...

```
int do_run (int ROM_number)
```

This function should:

- Load the given `ROM_number` into a new userspace allocation (obtained via `mm_allocate()`).
 - Allocate a new *process information object* (via `allocate()`) that stores information about the process: where its userspace memory is located, what its preserved `epc` and `sp` are, and a *unique process ID* (`pid`). Set the fields in this object for this new process.
 - Add the process object into a list of processes.⁴
 - Return the `pid` (which must be non-negative) to indicate success; `-1` to indicate that the given `ROM_number` was out of range (less than 4, greater than the last ROM number); `-2` if allocation of the process object failed; `-3` if allocation of the main memory space failed.
3. Modify `do_syscall()` to call `do_run()` when the *code* is `0xca110003`. The return value from `do_run()` should be returned to the calling process.
4. Modify `do_exit()` to:
- Remove the calling process's object from the process list.
 - Free the process's userspace memory (via `mm_deallocate()`).
 - Free the process object's memory (via `deallocate()`).
5. Modify the `do_alarm()` function to trigger a *context switch* from the current process to the next one.
6. Modify `cpu_scheduler()` to *set the alarm* for the processor. That is, it must enable the *alarm bit* in the mode register.⁵ Then, it must set the `a1` CSR to the number of user-level processor cycles that corresponds to a scheduling quanta. That register will count down as the process runs; when `a1 == 0`, the processor will generate an `ALARM` interrupt.

With all of these capabilities in place, your kernel should, after initializing itself, automatically `RUN` the initialization program, making it the initial process in the system. The kernel should handle scheduling and system calls as processes run. Once the last process has performed an `EXIT`, the kernel should halt.

⁴This list could be a *circular linked list*, thus ensuring that the *next* process is always non-null.

⁵Bit 4, counting from bit 0 in the least significant position, is the mode bit that enables alarm interrupts to occur when in *user* mode.

4 How to submit your work

Copy into the `project-2/` directory in your shared Google Drive folder **all** of your source code: Every `.asm`, `.h`, and `.c` file. Include any test programs. It should be straightforward to build and run your kernel and test programs.

This assignment is due at **11:59 pm** on **Sunday, October 25th**.