Systems II
Project 3, Part A
File system (read-only)

1 Overview

Next up for our *Fivish* system, a kernel that uses a file system. This project is again a modification and extension of your previous kernel (from Project 2). It will include some existing code and tools for working with a file system implemented on a virtual block device. Your job will be to modify your kernel to read files from this file system.

2 Getting started

2.1 Updating *Fivish*

Fivish has been slightly modified to use file system. Open a shell, activate your `sys2` environment, change into your `fivish/` directory, and update as shown below. Notice that no recompilation is needed because the changes are all to the scripts (e.g., `f-simulate`) that we use to build/compiler/assemble and then run the *Fivish* system.

```
$ micromamba activate sys2
(sys2) $ cd fivish
(sys2) $ git pull
(sys2) $ cat version.txt
v.2026-03-29
```

2.2 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-275-2526s-p3 -o project-3.tar.xz
(sys2) $ tar -xJvpf project-3.tar.xz
(sys2) $ cd project-3
(sys2) $ ls
bios/ fstool/
```

The `bios/` directory: This is a new BIOS. Much like the kernel, it has been re-implemented with most of its functions written in C. It is fully functional, and it is the BIOS that you should use henceforth. It has been extended to exhibit a couple of new capabilities:

- **Displaying the current time:** The code in `date_time.c` reads the *wall clock device* to get the current time, given as a number of seconds and milliseconds since *the epoch*[1]. It then can convert those values into the exact date and time of day, using that information to generate and print it as a formatted string.

  This capability is just a fun exercise in making our system a bit of realistic by connecting it to something in the real, external world.

- **Reading the kernel from the file system:** This BIOS accesses the file system on the block device, finds the file named `kernel.vmx`, and reads that file into RAM.

  This capability is central to the restructuring of this assignment. Only the BIOS itself will appear as a ROM in the system; all other executable images (i.e., `.vmx` files) will be read from the file system. The code in the BIOS can serve as an example of how the file system can be accessed.

Much like our kernel, `bios_stub.asm` contains the initial code of the BIOS that finds RAM, creates a stack, and then calls `main()`. The source in `bios.c` provides this `main()` function and is a good starting point to understand how the BIOS is structured.

The `fstool/` directory: This program is to be used **outside** of *Fivish*. It allows us to manipulate a *virtual block device*—that is, a file that will be used by the simulator as the storage for its block device. It can be used to format the file system on the device, list the files stored on the file system, copy files into and out of the file system, and delete files. It will allow us to pre-load a block device with files that our BIOS and kernel can access.

The files in this directory also provide a body of code that we can adapt for the kernel itself. If we want to know how the superblock, free-space bitmap, directory, or inodes are formatted, this code is the definitive guide.

And the `kernel/` and `programs/` directories? These begin from wherever you left off with Project-2. So copy those directories from your Project-2 directory into your Project-3 one. The implication is that you will need to modify the kernel to add new capabilities, and that you will need to write your own test programs to test and debug your modified kernel.

[1] January 1, 1970, 00:00:00.000

2.3 Building

The file system tool: To compile this tool, start in your `project-3/` directory:

```
(sys2) $ cd fstool
(sys2) $ ./build.sh
```

You should see a series of uses of `clang`, our C compiler. The result should be a file named `fstool`, which is the executable that we will use in Section 2.4 to create a file system.

BIOS: Again from your `project-3/` directory, build the new and fancy `BIOS+`:

```
(sys2) $ cd bios
(sys2) $ ./build.sh
```

Like our other build-scripts, no news is good news. The result of the construction is the `full-bios.vmx` executable image. Along with it are the `full-bios.asm` that provides the complete assembly source, as well as the `full-bios.log` output from the assembler that may provide useful debugging information (if needed).

Your kernel: In your `kernel/` directory, build its `full-kernel.vmx` executable image as usual. While its functionality will initially be quite limited, we will be able to load and run it as-is, providing a starting point for your work.

2.4 Running

Preparing a file system: The BIOS now expects to read the kernel from a file system on the block device. Therefore, you must create a block device, format it with a file system, and then copy your kernel onto it before you can run the simulator. Here, from your `project-3/` directory, we will create a 128 KB block device with the conveniently short filename `bd`, and then populate it with the file system and kernel:

```
(sys2) $ ./fstool/make-block-device.sh
USAGE: ./fstool/make-block-device.sh <filename> <size (4 KB blocks)>
(sys2) $ ./fstool/make-block-device.sh bd 32
Creating the block device...done.
(sys2) $ ls -l bd
-rw-------. 1 sfkaplan sfkaplan 131072 Mar 29 11:17 bd
(sys2) $ ./fstool/fstool format bd
(sys2) $ ./fstool/fstool cpin bd kernel/full-kernel.vmx kernel.vmx
(sys2) $ ./fstool/fstool list bd
kernel.vmx    5352
```

Simulating: The BIOS will henceforth be the only executable image provided to the system as a ROM; the kernel and all user programs will be read from the file system. Simulating our system looks like this:

```
(sys2) $ f-simulate bd bios/full-bios.vmx
```

Proceed within the simulator as usual. You should see the BIOS read the `kernel.vmx` file stored on the block device, loading it into RAM and jumping to it. At this point, your Project-2 kernel will discover that there are no additional ROMs, and therefore fail to run the *init* process.

3 Modifying the kernel to read from the file system

There are two goals for this project:

1. **Modify your kernel to read its user programs from the file system.** The kernel should expect to load the *init* program by reading the executable image file `init.vmx`. Your *init* program should then, in turn, load some set of other user programs via the `RUN` system call, where each such program must be requested by its filename (e.g., `do-nothing.vmx`).

   This change will require not only code for finding a reading a file from the file system, but also alterations to the `RUN` system call's interface, as well as modifications to internal kernel functions to handle filenames instead of ROM numbers. Additionally, your *init* program should be modified since it can no longer loop through the available ROM numbers.

2. **Add system calls for user process access to files.** A user program should be able to call on the kernel to access files that are already on the file system. Specifically:

   - `GETLENGTH (0xca110004)`: Used to obtain the length of one file. A system call wrapper should look like this:

     ```
     int getinfo (char* filename);
     ```

     The `filename` argument should point to a null-terminated string with the name of the file requested. If `filename` is `NULL`, then the length of the directory, given as a number of entries, is returned. If `filename` specifies a file that does not exist, then `-1` is returned as an indication of an error.

- `GETDIR (0xca110005)`: Provide a list of the names of the files. Its wrapper prototype, which depends on the maximum filename length of 60 bytes, is:

  ```
  typedef filename_t char[60];
  void getdir (filename_t* file_list);
  ```

  `file_list` must point to a pre-allocated space that `getdir()` will use. If there are n entries in the directory, then this buffer must an array of n `filename_t` spaces (that is, 64 byte character arrays).

- `READ (0xca110006)`: Read bytes from a given file into a provided buffer space.

  ```
  int read (char* filename, int offset, int length, void* buffer);
  ```

  The `filename` specifies the file from which to read. From that file, `length` bytes, starting at position `offset`, should be read into the pre-allocated userspace `buffer`. It is the responsibility of the caller to ensure that `buffer` is a space that is at least `length` bytes.

  Upon success, this call should return *the number of bytes read*. If the call fails, its return value depends on the nature of the failure:

  - `-1` (*invalid filename*): `filename` is `NULL` or is too long (exceeds the 60-byte limit).
  - `-2` (*nonexistant file*): `filename` does not exist in the directory.
  - `-3` (*invalid range*): If any of `offset` through `offset + length` fall outside of the file's size.
  - `-4` (*invalid buffer*): If `buffer` is `NULL` or is an invalid address.
  - `-5` (*other failure*): Any other cause of failure.

As a final test of these capabilities, write an *init* program that reads the directory, and for each `.vmx` file that it finds, calls on the kernel to `RUN` that program.

4 How to submit your work

Copy into the `project-3/` directory in your shared Google Drive folder **all** of your source code: Every `.asm`, `.h`, and `.c` file. Include test programs and data files (if any) that those programs might use. It should be straightfoward to build and run your kernel and test programs.

This assignment is due at **11:59 pm** on **Sunday, April** 12^{th}**.**