Systems II
Project 3
Completing a file system

1 Overview

Next up for our *Fivish* system, a kernel that uses a file system. Our starting point is our multitasking kernel Project-2 that has been modified to *read* (to a limited extent) files from a file system. Your job will be to modify your kernel to provide a range of system calls that allow processes to create, delete, rename, read, and write files.

2 Getting started

2.1 Updating *Fivish*

Fivish has incurred a few updates to its block device to support the implementation of a file system. Open a shell, activate your `sys2` environment, change into your `fivish/` directory, and update as shown below. Note the need to recompile the simulator.

```
$ micromamba activate sys2
(sys2) $ cd fivish
(sys2) $ git pull
(sys2) $ cat version.txt
v.2026-10-28
(sys2) $ cd simulator
(sys2) $ make
```

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-275-2627F-proj3 -o project-3.tar.xz
(sys2) $ tar -xJvpf project-3.tar.xz
(sys2) $ cd project-3
(sys2) $ ls
bios/ fstool/ kernel/
```

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 more realistic by connecting it to something in the real, external world. It's not terribly important to your work on this project, but it is cool.

- **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.

The `kernel/` directory: This kernel is a Project-2 kernel that has been extended, much like the BIOS, to read whole files from the file system. It uses this capability as part of `do_run()`, allowing it to load an entire `.vmx` file into main memory as part of running a new

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

process. **This extension is written but not tested!** Your starting point is to test and debug this part of the code, enabling the kernel to load an `init.vmx` in order to get things going.

The `programs/` directory: This contains the same user programs from Project-2. However, the *init* program has been changed to use new system calls that you are going to need to add for it to work. Additionally, if you wrote any new or modified test programs for Project-2 that you want to use here, copy them from your `project-2/programs/` directory to here.

2.3 Building

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

```
(sys2) $ cd fstool
(sys2) $ make
```

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) $ make
```

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

Your kernel: In your `kernel/` directory, build its `kernel-image.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/kernel-image.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 -b bd bios/bios-rom.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, the kernel will discover that there is no `init.vmx` file on the file system to run as the *initial process*, and therefore fail and halt.

3 Modifying the kernel to read from the file system

There are two goals for this project:

Complete the kernel's use of the file system to load user programs. The initial kernel contains an untested implementation of reading a `.vmx` file from the file system as part of creating a process. Test and debug this capability.

Add system calls for user processes to access the file system. A user program should be able to call on the kernel to perform a number of operations on the file system, including accessing the directory and reading/writing files. Specifically, **you must implement the following system calls**:

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

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

The `filename` argument should point to a null-terminated string with the name of the file requested.

  - verb|filename != NULL| and correctly speecifies a file, then the length of that file is returned.

  - If `filename != NULL` and specifies a valid filename, but that filename does not exist in the directory, then `-1` is returned.

  - If `filename != NULL` and but provides an invalid filename (longer than 60 characters), then `-2` is returned.

  - If `filename == NULL`, then the *length of the directory*, given as a number of entries, is returned.

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

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

`file_list_buffer` must point to a pre-allocated space that `getdir()` will use. If there are n entries in the directory, then this buffer must be an array of n `filename_t` spaces (that is, 60 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` (*nonexistent 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.

- `WRITE (0xca110007)`: Write bytes from a buffer space into a given file.

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

  The `filename` specifies the file to which to write. `length` bytes are copied from `buffer` and into the file starting at position `offset` in the file. It is the responsibility of the caller to ensure that `buffer` is a space that is at least `length` bytes.

  Note that the `offset` may be any value from `0` to the *length* of the file. By setting this argument to the file's current length, the written bytes are appended to the end of the file. If `offset + length` exceeds the file's length, then the file is extended.

  Upon success, this call should return *the number of bytes written*. 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` (*nonexistent file*): `filename` does not exist in the directory.
  - `-3` (*invalid range*): If `offset` exceeds the *length* of the file.
  - `-4` (*invalid buffer*): If `buffer` is `NULL` or is an invalid address.
  - `-5` (*allocation failure*): The block device has no free blocks.
  - `-6` (*other failure*): Any other cause of failure.

- `CREATE (0xca110008)`: Create a new file and provide a name by which it can be accessed. The function prototype is:

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

The `filename` argument should point to a null-terminated string with the name that should lead to a newly created file object. Upon success, this call should return `0`. Failure should return a negative value as follows:

- `-1` (*invalid filename*): `filename` is `NULL` or is too long (exceeding the maximum filename length of 60 characters).
- `-2` (*file exists*): `filename` already appears in the directory.
- `-3` (*allocation failure*): The block device has no free blocks.
- `-4` (*other failure*): Any other cause of failure.

- `DELETE (0xca110009)`: Delete a file by removing both its directory entry and the file object itself.

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

  Return `0` upon success; return `-1` for an *invalid filename*, and return `-2` if the named file does not exist.

- `RENAME (0xca11000a)`: Update the directory by changing the name of a file.

  ```
  int rename (char* oldname, char* newname);
  ```

  If `oldname` exists in the directory, its value should be substituted in the directory entry with `newname`.

  Upon success, this call returns `0`. Upon an error, return:

  - `-1` (*invalid filename*): `oldname` or `newname` are either `NULL` or provide names that are too long.
  - `-2` (*nonexistent file*): `oldname` does not exist in the directory.

- `-3` (*file exists*): `newname` already appears in the directory.

- `-4` (*other error*): Any other cause of failure.

Rewrite the *init* program. Modify this initial program so that it:

1. Opens a file named `runlist.txt`, which should contain a list of programs to run. For example, it could look like:

   ```
   do-nothing.vmx
   fib.vmx
   looptest.vmx
   do-nothing.vmx
   ```

2. Read each line of the `runlist.txt` file.

3. For each line read, use the `RUN` system call to execute the named program.

4 How to submit your work

Be sure that your group's GitLab repo contains your completed kernel and test programs. Once again, include a `README` file that describes how to run and test your kernel, describing the test cases and how to invoke them.

This assignment is due at **11:59 pm** on **Tuesday, November** 17^{th}**.**