Operating Systems 202.1.3031

Spring 2022/2023 Assignment 4

File Systems

Responsible Teaching Assistants:
Roee Weiss Lipshitz and Hedi Zisling

Ben-Gurion University
of the Negev

Contents

1 Introduction

In this assignment, you will get to know the xv6 file system by implementing two new features that are not supported by xv6 by default. First, you will add the standard functionality of `seek()`, which allows to read and write from anywhere in a file, by changing the file descriptor offset. Second, you will implement a new device file which outputs pseudo-random characters when read. This mimics a behavior that exists on different UNIX-like systems, Linux and macOS among them, where the `/dev/random` file is used to generate random numbers.

2 Submission Instructions

- Make sure your code compiles without errors and warnings and runs properly!
- We recommend that you comment your code and explain your choices, if needed. This would also be helpful for the discussion with the graders.
- You should submit your code as a single `.tar.gz` or `.zip` file. **No `.rar` files will be accepted.**
- We advise you to work with git and commit your changes regularly. This will make it easier for you to track your progress, collaborate and maintain a working version of the code to compare to when things go wrong.
- Submission is allowed **only in pairs** and **only via Moodle**. Email submissions will not be accepted.
- Before submitting, run the following command in your xv6 directory:

  ```
  $ make clean
  ```

  This will remove all compiled files as well as the `obj` directory.
- Help with the assignment and git will be given during office hours. Please email us in advance.

3 Task 1: Seek

In this section you will add support for the `seek()` functionality in your file system, which changes the offset of an open file descriptor. This means that the next read or write operation will start from the new offset. To add this new functionality, you will need first to add the new function `fileseek()` in *file.c* which implements the seeking logic. The prototype of the `fileseek()` function is:

```
int fileseek(struct file *f, int offset, int whence);
```

Next, you will need to add the system call `seek()` which will call the `fileseek()` function. The prototype of the `seek()` function is:

```
int seek(int fd, int offset, int whence);
```

The `fd` argument is the file descriptor to change the offset of, the `offset` argument is the new offset to set, and the `whence` argument is the reference point from which the offset is calculated.

The `whence` argument can be one of the following:

- `SEEK_SET` - The offset is set to the `offset` argument.
- `SEEK_CUR` - The offset is set to its current location plus the `offset` argument.

Note: To get the basic idea of how to implement `fileseek()`, you can look at the `fileread()` and `filewrite()` functions in *file.c*. Make sure you know when to use the file lock.

3.1 Specifications

- If `whence` is `SEEK_SET`, and `offset` is negative, the offset should be set to 0.
- If `whence` is `SEEK_CUR`, and the result of `f->off + offset` is negative (which implies that the new file offset is negative) the offset should be set to 0.

- Similarly, if the resulting file offset for either `whence` case is larger than the file size, the offset should be set to the file size.
- The `seek()` function should return -1 if the file descriptor doesn't exist.
- Seek should not be supported for any file type that is not `FD_INODE`. In this case, the `seek()` function should return -1.
- The `seek()` function should return 0 on success.
- The SEEK_SET and SEEK_CUR cases should be defined in *fcntl.h* using the `#define` macro.

Task 1

1. Implement the `fileseek()` function in *file.c* which performs the logic for the `seek()` functionality.
2. Implement the `seek()` system call in *sysfile.c* which calls the `fileseek()` function.

4 Task 2: Random Device

In this section you will add a new device file type to the file system. The device file will be called `/random` and will output pseudo-random 8-bit numbers (`char`) when read, similarly to the `/dev/random` device file that exists in Linux and macOS. This device file will be implemented using type `FD_DEVICE`. This type indicates to xv6 that operations on this file should be mapped to operations on a device. To enable this, xv6 defines an interface with two operations – `read()` and `write()`. Read the code in *console.c* to see an example of using this interface. Make sure you understand how the `read()` and `write()` functions in *console.c* work and where the console device is exposed to userspace by creating an inode for it.

Your task is to implement the `/random` device, by writing two functions that conform to the device interface. This code should be implemented in *kernel/random.c*.

4.1 Specifications

- A call to the `read(int fd, void *dst, int n)` system call on this device file should read `n` pseudo-random bytes into the buffer `dst`. The function should return the number of bytes written to the buffer. On failure, the function should return the amount of bytes it managed to write before the failure. A failure example is when given `dst` is not a valid address.
- A call to the `write(int fd, const void *src, int n)` system call on this device file, when n is 1, should seed the random number generator with the byte pointed to by `src`. If n is not 1, the function should return -1. The function should return 1 on success.
- At the initialization of the device file, the random number generator should be seeded with the value `0x2A`. The rest of the initializtion should be done similarly to the `console` device.
- **After xv6 starts up**, the device file should be available at `/random`. Again, see how it's done for `/console`. **No points will be given to a solution for which the file is not in the correct location after boot up.**

- To generate pseudo-random numbers, you can use the `lfsr_char()` function provided later in this document. This function implements a linear feedback shift register (LFSR) which is a simple pseudo-random number generator. The function is seeded with a value, and each call to it returns the next pseudo-random number. That is, after every subsequent call to it, you should update the seed with the returned value (which is also the output random number).
- The implementation should be safe in terms of concurrency, that is – if multiple processes are using the device file concurrently, the random numbers should be generated correctly and no two processes may get the same values. Make sure that the file state is shared, and that there isn't a different state (seed) for each process.

4.2 The LFSR function

The code is based on the Fibonacci LSFR example in Wikipedia.

```
// Linear feedback shift register
// Returns the next pseudo-random number
// The seed is updated with the returned value
uint8 lfsr_char(uint8 lfsr)
{
  uint8 bit;
  bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 4)) & 0x01;
  lfsr = (lfsr >> 1) | (bit << 7);
  return lfsr;
}
```

Task 2

1. Implement the `read()` function for the `/random` device file in *kernel/random.c*.
2. Implement the `write()` function for the `/random` device file in *kernel/random.c*.
3. Add the `/random` device file to the file system (after the **system boot**, you should be able to see it when running `ls` in the shell).

5 Task 3: Testing

Write a user program to test both tasks. Both tests should be comprehensive enough to check that your code follows our instructions. The assignment tests expect the signature and behavior to be exactly as specified in the assignment. Deviating from the signature or specifications will result in failed tests and **substantial** loss of points, so make sure you follow the instructions carefully. Therefore, it is important to test your code thoroughly.

TIP

When testing the `/random` device file, make sure that after 255 calls to (read()) on it, it outputs the initial state (whether it's the default 0x2A or any state you set it to). This is the period of the 8-bit LFSR.

IMPORTANT

Some important failure cases to check:

- Implementing system calls with an incorrect signature will result in failed tests.
- Changing the signature of existing system calls (read/write) will result in failed tests.
- The `/random` file SHOULD BE IN THE SPECIFIED LOCATION! If it is not, you will not get points for this task.
- Failing to define SEEK_SET and SEEK_CUR in *fcntl.h* will result in failed tests, and most likely will cause compilation errors.

Failing in one or more of the above will result in a **substantial** loss of points, and we will not accept appeals due to these mistakes.