ICSI412 3 – Devices

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. Especially ".class" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

Introduction

In this assignment, we will be implementing the device API so that our operating system can work with devices. If you are not familiar with open(), close(), read(), write() and seek(), as used in C, review this reference:

https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/

There are a few different constraints which make the infrastructure for devices more complex than we might think.

1) Devices are very diverse. A device can be anything - a disk drive, video card, sound input, sound output, or even a virtual device, like a filesystem.
2) Memory in the kernel is statically, not dynamically allocated (most of the time), so we need to ensure that we use resources as lightly as possible.
3) Each process must track its open devices. If a process dies (or quits), we must ensure that all its devices are freed.
4) Remember that pointers/references can't (and shouldn't!) cross the kernelland-userland barrier.
5) The function calls for all devices must be standardized, since they are defined in the kernelland-userland transition point.

We start with an interface – the Device interface.

```
public interface Device {
    int Open(String s);
    void Close(int id);
    byte[] Read(int id,int size);
    void Seek(int id,int to);
    int Write(int id, byte[] data);
}
```

Every device that we make will implement that interface.

Notice that Open() takes a String. Why? Extensibility. We have no idea what parameters a generic device will require. String lets us pass (pretty much) anything.

The “id” is a device id. It is a generated number that the caller can use to access the particular device and configuration that we requested in Open(). Consider this code:

```
int id1 = open(“somefile”);
int id2 = open(“report.txt”);
```

Those two id values both reference the filesystem, but very different files.

From userland, which you are familiar with, we communicate with the kernel. The part of the kernel that handles devices is called the VFS or Virtual File System. The VFS is responsible for associating each id with a device and id pair. VFS also implements the Device interface. Let’s walk through a userland program:

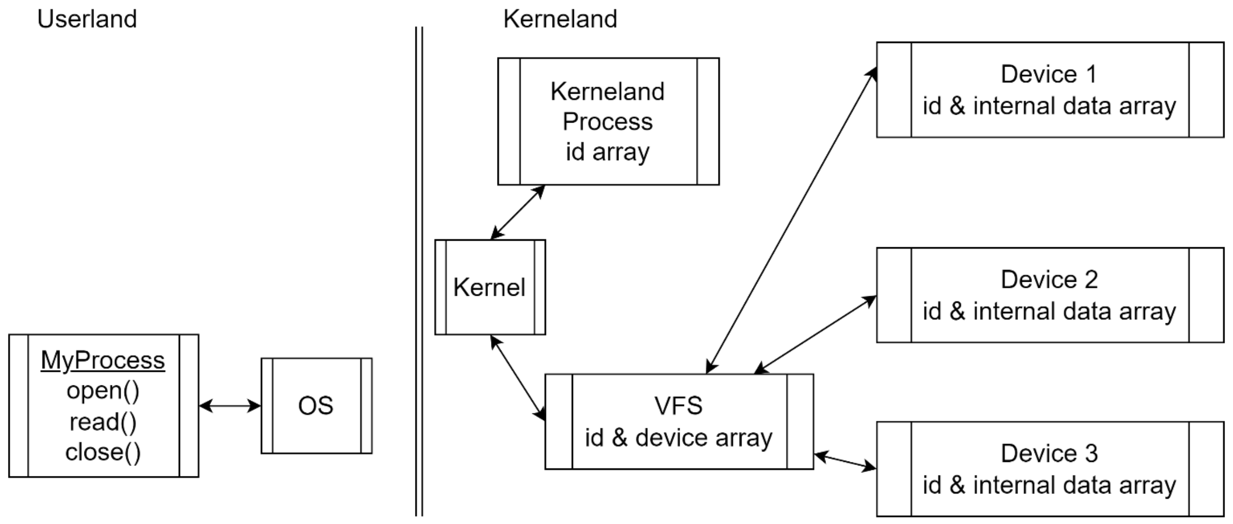
```
int id = open (“somedevice”);
read(id,10);
close(id);
```

Userland open() gets sent to kerneland open(). Kerneland open() calls VFS open(). VFS looks at the incoming string and decides what device should be used. In LINUX, for example, “/dev/random” is a device that returns the requested amount of random data. Once VFS knows which device to talk to, it calls that device’s open(). That device does whatever work it needs to do (which is device dependent) and creates an id. It returns the id to VFS. VFS associates that id and device with an id of its own which it returns to the kernel. The kernel then associates that stores that id in the process’ data (so that it can track what you have open). The kernel then returns the index of that id in the process data to userland.

When you call read(), the kernel uses the id index to get a VFS index. It calls read() on VFS with that index. VFS looks up that index to get a device and index. It calls read() on the device. The device looks up that id to get the device specific data that it needs and does the read. The data returned gets passed back to userland.

When you call close(), the kernel used the id to get the VFS index/id. It calls close() on VFS. VFS finds the device and id and calls close() on the device. The device frees the id for future use and does device specific work and returns. VFS then frees the VFS index and returns. Finally, the kernel frees the id/index in the process table and returns.

When a process shuts down, if any device ids are in use, it calls close() on each one.

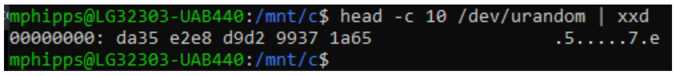

Userland
Kerneland
Kerneland Process id array
Kernel
MyProcess open() read() close()
OS
VFS id & device array
Device 1 id & internal data array
Device 2 id & internal data array
Device 3 id & internal data array

Details

Create the Device interface (using the methods from above).

Create the Random Device

The Linux operating system creates a device for random number:



```
mphipps@LG32303-UAB440:/mnt/c$ head -c 10 /dev/urandom | xxd
00000000: da35 e2e8 d9d2 9937 1a65                 .5.....7.e
mphipps@LG32303-UAB440:/mnt/c$
```

Make a new class, RandomDevice which implements Device. It should keep an array (10 items) of java.util.Random. Open() will create a new Random device and put it in an empty spot in the array. If the supplied string for Open is not null or empty, assume that it is the seed for the Random class (convert the string to an integer). Close will null the device entry. Read will create/fill an array with random values. Write will return 0 length and do nothing (since it doesn't make sense). Seek will read random bytes but not return them.

Create the Fake File System

For our fake file system, we will just expect a simple filename in open(); no need to worry about directories. Java has a class that will do most of the underlying work: RandomAccessFile; you will need an array (again, 10 is enough) of these.

https://docs.oracle.com/javase/10/docs/api/java/io/RandomAccessFile.html

Create another device - FakeFileSystem. You will be passed a filename; if the filename is empty or null, throw an exception. Open will create and record a RandomAccessFile in the array. Since this class implements read(), write() and seek(), this should be very straightforward. Make sure that you close the RandomAccessFile and clear out your internal array when close() is called for this device.

Create the VFS (Virtual File System)

VFS is just another device, much like Random or FakeFileSystem. But its job is mapping calls to the other devices and ids. You will need an array of Device and an array of int. These parallel arrays will map a VFS id to a (Device/Id) combination. You can make a class for this, if you choose, and have an array of that class.

Open

For our VFS, we will do a very simple scheme (simpler than UNIX) to name devices: the first word of your input string is the device; the rest of the open string is passed to the device.

Examples:

open("random 100") – opens the random device and uses 100 for the seed

open("file data.dat") – opens a file called data.dat

Your VFS should look at the first word to determine the device, then remove that from the string and pass the remainder to the open() call on the device (so, RandomDevice would get "100" in the first example).

Close will remove the Device and Id entries. Read/Write/Seek will just pass through to the appropriate device.

Modify OS, Kernel, KernelandProcess

Add an array of 10 integers to your kerneland process. Add "getCurrentlyRunning()" as an accessor to the scheduler. This will let the kernel access the process data to get to the array in the kerneland process.

Make Kernel implement Device. For Open(), use getCurrentlyRunning() and find an empty (-1) entry in the kerneland process' array. If there isn't one, return -1 (fail). Then call vfs.open. If the result is -1, fail. Otherwise, put the id from vfs into the kerneland process' array and return that array index. For the other methods (read, write, seek), we get an id from userland – use the array in kerneland to convert that to the id that vfs expects and then pass the call through to the vfs. Close also needs to use the array to convert to vfs, but it also needs to set the kerneland array entry to -1.

Finally, when a process ends (remember, that is in the process scheduler, we check for .isDone()) we need to close all of its open devices. To do this, I had to make the scheduler hold a reference to the kernel.

Test your code!

Test multiple devices/process, multiple processes connecting to the same device and more.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary,

				private members (10)
Random Device	Doesn't Exist(0)	Interface implemented (5)	Open/close implemented correctly (10)	All functions implemented correctly (15)
Fake File System	Doesn't Exist(0)	Interface implemented (5)	Open/close implemented correctly (10)	All functions implemented correctly (15)
VFS	Doesn't Exist(0)	Interface implemented (5)	Open/close implemented correctly (15)	All functions implemented correctly (15)
Kerneland changes	No changes (0)			array exists, is initialized and populated with -1 (5)
Kernel – open	No changes (0)	One of: Finds user space id, calls VFS, sets user space to map to VFS id (3)	Two of: Finds user space id, calls VFS, sets user space to map to VFS id (6)	Finds user space id, calls VFS, sets user space to map to VFS id (10)
Kernel – close/read/write/seek	No changes (0)			All functions translate user id->vfs id, then pass through to vfs (10)
deleting a process	No changes (0)			Closes all open devices (10)
Testing	None (0)	Few tests (3)	Mostly tested (6)	Test processes that show all functionality working (10)