# ICSI412 Operating Systems – Project 4 – Devices

## Overview

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 kernelland open(). Kernelland 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.

Diagram

Description automatically generated

**Task 1** – A little design change

Notice that OSInterface now extends both ProcessInterface and Device (which is new).

**Task 2** – Create the random device

The Linux operating system creates a device for random number:



Make a new class, RandomDevice which extends 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.

**Task 3** – Fake a 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.

**Task 4** – Create the pipe device

If you are not familiar with UNIX pipes, review this reference:

https://www.howtogeek.com/438882/how-to-use-pipes-on-linux/

Users of our OS will connect to it by passing a name. If the name exists, we will “attach” to it. if not, we will create a new pipe. Data written to the pipe will be held in a buffer (that you make) until someone reads it. Read will read up to the amount requested. Many operating systems will suspend your process if they cannot read enough to satisfy your read request. We will not be doing that. Seek discards (skips over) the specified number of bytes. Close removes the attachment to the pipe. If this is the last process/client attached to the pipe, the pipe (not the class, but the entry in the class) is destroyed.

In this case, the id has to map to a pipe data structure that you will build. This is a little harder than random.

**Task 5** – Create VFS

VFS is just another device, much like Random or Pipe. 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(“pipe joe”) – opens a pipe named Joe

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 remained 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.

**Task 6** – Modify OS & KernelandProcess/PCB

Finally, OS needs to implement the Device methods. Open and close (much like VFS) are a little special. When a device is opened, we record the VFS id in the kernel storage for the process. When we close the device, we remove that record. When a process is deleted, we have to call VFS’ close() on every open device. Depending on how you implemented your code, you may have to make some helper methods in PriorityScheduler (I did).

**Test your code!**

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

One thing that you might need to think about – how can I test my code when every time my run is called, it starts over? How can I have my program continue? I used a state machine. In my test object, I have an int for the state. In my run() method, I have an if block:

if (state==0)

// do the “first run”

else if (state == 1)

// do the “second run”

etc…

***You must submit buildable .java files for credit.***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rubric | Poor | OK | Good | Great |
| Random Device | Doesn’t Exist(0) | Interface implemented (5) | Open/close implemented correctly (10) | All functions implemented correctly (15) |
| Pipe device | Doesn’t Exist(0) | Interface implemented (5) | Open/close implemented correctly (10) | All functions implemented correctly (20) |
| 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 (20) |
| OS | No changes (0) | Interface implemented (5) | Open/close implemented correctly (15) | All functions implemented correctly (20) |
| DeleteProcess | No changes (0) |  |  | Closes all open devices (10) |