University of Windsor

School of Computer Science
Faculty of Science
COMP-2560: System Programming (Fall 2024)

Lab#	Date	Title	Due Date	Grade Release Date
Lab07	Week 07	**Devices as Files**	One-Week Lab Oct. 29 Tuesday Midnight EDT	Nov. 4

The main objective of this lab will be to learn how to UNIX handles devices as files. For the sake of this lab, whenever we refer to a file that is a device, we say *special* file. Otherwise, it is a *normal* file like an image file or program file.

Step 1. UNIX Devices as Special Files

UNIX's kernel at its high-level of File System (File Manager) defines *file* as an abstract entity to read from or write to. File could be part of a hard disk drive that stores your C program main.c, another part of the hard disk drive that stores your program executable main, or the whole hard disk drive. UNIX's kernel at its File System assumes all other hardware devices are also files. The differences between the devices are handled by another part of the kernel, Device Manager. So, like your files in your home directory, devices are also files at /dev directory:

```
hfani@alpha:~$ ls /dev
autofs           fb0         kvm                    ptmx     shm         tty14  tty26
block            fd          log                    ptp0     snapshot    tty15  tty27
bsg              full        loop-control           ptp1     snd         tty16  tty28
btrfs-control    fuse        mapper                 pts      stderr      tty17  tty29
bus              hidraw0     megaraid_sas_ioctl_node random  stdin       tty18  tty3
char             hidraw1     mem                    rfkill   stdout      tty19  tty30
console          hpet        mqueue                 rtc      tty         tty2   tty31
core             hugepages   net                    rtc0     tty0        tty20  tty32
cpu              hwrng       null                   sda      tty1        tty21  tty33
cpu_dma_latency  initctl     nvram                  sda1     tty10       tty22  tty34
cuse             input       port                   sda2     tty11       tty23  tty35
disk             ipmi0       ppp                    sda5     tty12       tty24  tty36
dri              kmsg        psaux                  sg0      tty13       tty25  tty37
```

As seen, we have cpu (processor) and mem (memory), which are the two essential parts of any computer system. School's computer system is a multiprocessor system; that is, it has multiple cpus. So, you can go inside the /dev/cpu and see how many cpus exist:

```
hfani@alpha:~$ ls /dev/cpu
0  1  10  11  12  13  14  15  2  3  4  5  6  7  8  9
```

Let's go into the first cpu:

```
hfani@alpha:~$ ls /dev/cpu/0
cpuid  msr
```

Amazingly, devices are managed like files and directories that are directly connected to the devices. You may be tempted to open them as normal files using vi:

```
hfani@alpha:~$ vi /dev/cpu/0/cpuid
"/dev/cpu/0/cpuid" is not a file
```

You see that vi is not able to read from cpu/0/cpuid like a normal file. Let's try it for memory device:

```
hfani@alpha:~$ vi /dev/mem
"/dev/mem" is not a file
```

Is this because devices are not files and what we say is incorrect, or vi program is not able to read from these *special* files? We'll answer this shortly.

What are other files? What is the file for mouse, keyboard, monitor, sound, mic, hard disk drives, etc?

Important parts of a computer system are storage devices. Depending on the type of storage, the file name for the device is either start with hd or sd followed by a letter that distinguishes multiple hard disk storages. For instance, sda is the first hard disk, sdb is the second, and so on. Also, each hard disk storage can have different partitions, which are identified by a number. For instance, sda1 is the first partition of the first hard disk storage. As you can see, the School's system has one hard disk storage with 3 partitions.

```
hfani@alpha:~$ ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda5
```

Who select these names for devices? Are there any standards? Can I rename the file of a device? For instance, can I change the /dev/mem to /dev/memory?

Step 2. Standard File Descriptors

There are three files that *look like* they are devices, but they are not. They are file descriptors (numbers) that are connected to either a normal file or a special file (device):

```
hfani@alpha:~$ ls /dev/fd
0  1  2  3
hfani@alpha:~$ ls /dev/std*
/dev/stderr  /dev/stdin  /dev/stdout
```

Traditionally, /dev/fd/0 or /dev/stdin was connected to keyboard device, meaning that reading from keyboard file (device) was the same as reading from /dev/fd/0 or /dev/stdin file descriptors. /dev/fd/1 or /dev/stdout was connected to printing device or monitor, meaning that writing to printer file (device) was the same as writing to /dev/fd/1 or /dev/stdout file descriptors; likewise, for /dev/fd/2 and /dev/stderr to print out errors. Why not use special files' names directly for keyboard or printer instead of working with these numbers? Very good question!

Working with special files (device) requires somebody to be next to the School's computer system. For instance, if you want to read from the keyboard from its assigned special file:

```
/dev/input/by-path/pci-0000:00:13.0-usb-0:3:1.1-event-kbd
```

somebody should be in the School's server room and press any keys in the keyboard device attached to the School's computer system. This is not convenient for us when we connect remotely to a computer system. Therefore, by default, UNIX-based/like operating system developers came up with the idea of standardizing the input and output for programs regardless of where or what the actual device is. So, they fixed these three file descriptors that can be connected to any special file (device) or normal file. For the program, it does not matter. The program reads from 0 and writes to 1. If error, the program writes to 2.

You may connect 0 and 1 and 2, all to a normal file (how?). Then your program reads input from the file and writes output and errors, if any, to the same file.

But how do we run our program, like the one in Lab03, at School's computer system that reads input and writes outputs? This is handled by either kernel or shell, who attach standard file descriptors to a virtual terminal device called tty (TeleTYpewriter) with a unique number for a user connected to the computer system. This virtual device is the window with the black background that shows you the shell command prompt:

```
hfani@alpha:~$
```

You can see your virtual terminal device number by using the tty command:

```
hfani@alpha:~$ tty
/dev/pts/9
```

This is the device your program reads from or writes to, not the keyboard or monitor at the School's server room or at your home!

When your program, like in Lab03, asks an input value, it reads from 0 which is connected to your tty. So, when you type numbers or characters by your keyboard attached to your computer at home, it goes to your tty, and then it forwards them to your program. Can you explain what happens when your program writes outputs and you see it in your monitor?

Step 3. Read/Write to Devices as Files via Shell

vi is a program to open normal files that are stored in storage devices. But it cannot open special files (devices) and simply raise an error. However, UNIX, which claims that devices are files, must provide the same system calls to work with devices the same way as files. In the next step, we'll see UNIX fulfills its promise. Not only that, now we see that its shell also provides a built-in command cat (concatenate)to do the operations on devices and files in the same way.

To read from a normal file or a special file (device), you can simply use cat followed by the name of the file/device:

```
hfani@alpha:~$ cat test.c
#include <fcntl.h>
#include <stdio.h>
void main(void){
        char filename[20]="test.txt";
        int fd = creat(filename, 0);
        if(fd==-1){
                printf("error happend!");
        }

}
hfani@alpha:~$ cat /dev/mem
cat: /dev/mem: Permission denied
hfani@alpha:~$ cat /dev/cpu/0/cpuid
cat: /dev/cpu/0/cpuid: Permission denied
hfani@alpha:~$ cat /dev/sda1
cat: /dev/sda1: Permission denied
hfani@alpha:~$
```

As seen, we do not have enough permission to read from some devices. But we have permission to access our assigned virtual terminal device tty. Let's find our unique tty again:

```
hfani@alpha:~$ tty
/dev/pts/9
```

Then, read from it:

```
hfani@alpha:~$ cat /dev/pts/9
```

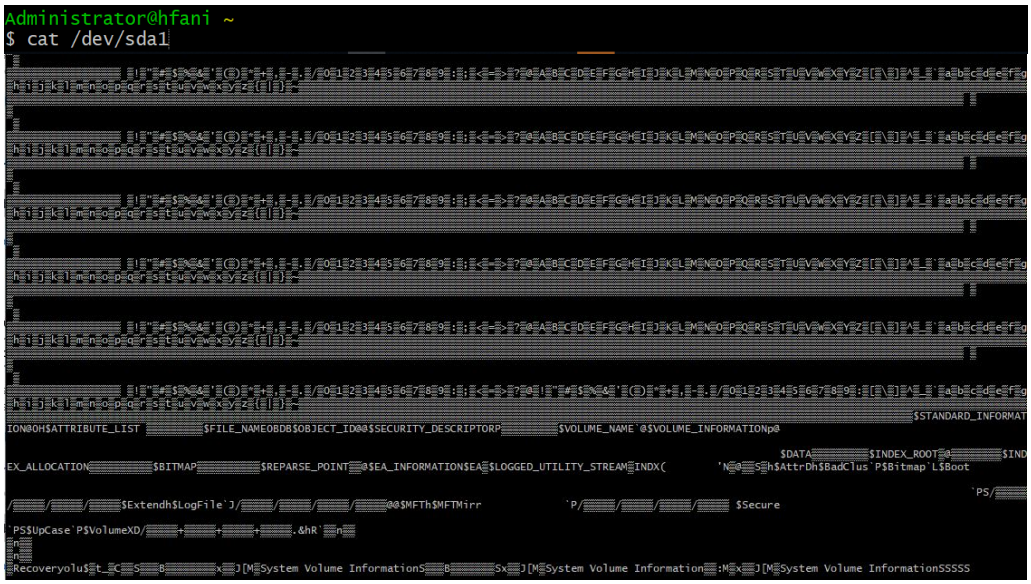
As seen, it is empty initially. You can put some data to it by typing and pressing enter:

```
hfani@alpha:~$ cat /dev/pts/9
Hello my virtual terminal device!
Hello my virtual terminal device!
How is it going?
How is it going?
```

You see that the cat reads from the tty and shows it to us. You can continue to send more info to the tty and cat reads them and show them to you. You can exit cat by CTRL+C.

I used cat on my first hard disk storage on *my own computer system,* and here is the output:

```
Administrator@hfani ~
$ cat /dev/sda1
```



Technically, `cat` opens the special file (device) `sda1`, reads the content and writes them into 1 which is connected to `tty`, and you can see the content of my hard disk drive.

Let's write to a device. You can use `cat` with `>` operator that connects `1` to a device instead of your `tty`. Then, you are able to input data and it writes them to that device:

```
hfani@alpha:~$ cat > /dev/mem
-bash: /dev/mem: Permission denied
```

As seen, you are not able to write to memory due to lack of enough permission.

However, you can write to your `tty`:

```
hfani@alpha:~$ cat > /dev/pts/9
Hey my tty again!
Hey my tty again!
I'm back!
I'm back!
```

You see the same effect as when you read from `tty`. Can you explain what is happening?

I don't dare to write on my hard disk storage by `cat`! But I can write to my sound device, whose file is `/dev/dsp`.

```
Administrator@hfani ~
$ cat > /dev/dsp
Hello my sound device. Would you please sing this for me: "some say love, it is a rive ..."
```

I have to exit `cat` to hear my sound device. If you do it on your sound card, you will hear nothing but a loud noise! This is because the sound device needs audio signals and cannot read my sentences in English. To send an audio signal to my sound device:

1) I record my voice in a file `test.wav`

2) I send the content of this file to the sound device

cat can connect two files/devices by reading from one file or device (source) and write it into the second file or device (target). For example,

```
hfani@alpha:~$ cat test.c > /dev/pts/9
#include <fcntl.h>
#include <stdio.h>
void main(void){
        char filename[20]="test.txt";
        int fd = creat(filename, 0);
        if(fd==-1){
                printf("error happend!");
        }

}
```

which reads from the file test.c and writes it to your tty device. Now, I want to send my test.wav to my sound device:

```
Administrator@hfani ~
$ cat /cygdrive/c/test.wav > /dev/dsp
```

Great! **Please note that we cannot do this on the School's computer system due to lack of permission.** You can try it yourself if you have UNIX-base/like OS or UNIX emulator software (e.g., Cygwin[1] on Windows.)

Step 4. Read/Write to Devices as Files via System Calls

Now that we know how devices and files are treated the same in UNIX, let's open a device and read from or write to it like a normal file in a C program. Due to our lack of enough permission on the School's computer system, we use the virtual terminal device tty in our example. First, we need to know the tty number that assigned to our terminal:

```
hfani@alpha:~$ tty
/dev/pts/9
```

Now, /dev/pts/9 is our read and write device. First, we need to open it and have a file descriptor to the device. The open system call is in the file control header (fcntl.h):

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
void main(){
        int fd_in = open("/dev/pts/9", O_RDONLY);
```

Then, we need to read from it using the file descriptor at hand. The read system call is in UNIX standard header (unistd.h):

```
        char buf[20];
        ssize_t bytes_read;

        while ((bytes_read = read(fd_in, buf, sizeof(buf))) > 0) {
                printf("block read: \n<%s>\n",buf);
        }
```

[1] https://www.cygwin.com/

Commented [A1]: TTYNAME(3) Linux Programmer's Manual TTYNAME(3)

NAME
ttyname, ttyname_r - return name of a terminal

SYNOPSIS
#include <unistd.h>

char *ttyname(int fd);

int ttyname_r(int fd, char *buf, size_t buflen);

DESCRIPTION
The function ttyname() returns a pointer to the null-terminated pathname of the terminal device that is open on the file descriptor fd, or NULL on error (for example, if fd is not connected to a terminal). The return value may point to static data, possibly overwritten by the next call. The function ttyname_r() stores this pathname in the buffer buf of length buflen.

RETURN VALUE
The function ttyname() returns a pointer to a pathname on success. On error, NULL is returned, and errno is set appropriately. The function ttyname_r() returns 0 on success, and an error number upon error.

****

moradib@charlie:~/Unix2022/lab06$ tty
/dev/pts/3

moradib@charlie:~/Unix2022/lab06$ ./getttyname
MY TTY name is: /dev/pts/3

moradib@charlie:~/Unix2022/lab06$ cat get_ttyname.c
#include<unistd.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
char *MyTTYName = ttyname(0); /*
Note: ttyname takes Int fd(STDIN_FILENO) */
printf("MY TTY name is: %s\n", MyTTYName);
return 0;
}

Step 5. Lab Assignment

There are two tasks for your submission:
Our example in step 4 never stops. Add lines of code such that when the user writes END, it exits the while loop, safely closes the device, and returns.

1) Instead of printing what we read from the device to output (the terminal device itself), write them into a normal file, named log.txt.

You are not allowed to use any library routines, no stdio.h! Only system calls in fcntl.h and unistd.h. The sample code for step 4 has been attached in a zip file named lab07_hfani.zip.

1.1. Deliverables

You will prepare and submit the program in one single zip file lab07_uwindid.zip containing the following items:

(90%) lab07_uwinid.zip
- (70%) tty.c => built with no error
- (10%) log.txt => the output of the program
- (10%) results.pdf/jpg/png => the image snapshot of the program run
- (Optional) readme.txt

(10%) Files Naming and Formats

Please follow the naming convention as you lose marks otherwise. Instead of uwinid, use your own account name, e.g., mine is hfani@uwindsor.ca, so, lab07_hfani.zip