O.S. Lab 6: File I/O Part I; Shell Part II

Overview to the File System

We have implemented almost all functionality needed for our operating system but have a problem: in order to access a file stored on the disk you have to know its location and size. For user convenience we prefer to have the o.s. track this information so the user can simply refer to files by name. This is the purpose of a file system; to match a file name with its location and size on the disk. It is this structure which we will now describe and implement.

What You Will Need

Download the linked files from the web page and add them to your existing pieces (**bootload**, **bdos.txt**, **kasm.o**, **kernel.c**, **config**, **kitty2**, **fib**, **msg**, **Stenv** and **compileOS.sh**). These files include

- The disk image file **map**, explained next.
- The utility file **loadFile.c**, which is compiled with gcc (**gcc –o loadFile loadFile.c**) and used to insert files into our disk image.
- The utility program **ddir.c** that prints a list of the files in your disk directory.
- The file **error.c** to be cut-and-pasted into your kernel as-is.

Initial Map and Directory

The additional file **map** contains a used space map for a file system consisting of only the boot loader and kernel. We also have the **config** file from the last lab to add to our disk image. You should edit your **compileOS.sh** file to include the following lines in the indicated place:

```
dd if=/dev/zero of=floppya.img bs=512 count=2880
dd if=bootload of=floppya.img bs=512 count=1 conv=notrunc
dd if=map of=floppya.img bs=512 count=1 seek=256 conv=notrunc
dd if=config of=floppya.img bs=512 count=1 seek=258 conv=notrunc
bcc –ansi –c –o kernel.o kernel.c
ld86 –o kernel –d kernel.o kasm.o
dd if=kernel of=floppya.img bs=512 conv=notrunc seek=259
```

(The non-boldfaced italicized lines should already be in your script.) This sets up your initial file system. You will add to the end of this file later.

Introduction to the File System

The primary purpose of a file system is to keep a record of the names and sectors of files on the disk. The file system in this operating system is managed by two sectors toward the beginning of the disk. The *disk map* sits at sector 256, and the *disk directory* sits at sector 257. (This is the reason your configuration file starts at sector 258 and the kernel at 259.)

Once you rebuild **floppya.img** and do a **hexdump** you should see something like this:

```
00000000  b8 00 10 8e d8 8e d0 8e  c0 b8 f0 ff 89 c4 89 c5  |................|
00000010  b5 07 b1 08 b6 00 b4 02  b0 0a b2 00 bb 00 00 cd  |................|
00000020  13 ea 00 00 00 10 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00020000  ff 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00020010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00020400  00 0f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00020410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00020600  55 89 e5 57 56 81 c4 fe  fd e8 dc 05 31 c0 89 86  |U..WV.......1...|
…
00020ea0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00168000
```

Map ↵

↵ Config

↵ Kernel ↵

The map (starting at 0x20000) tells which sectors are available and which sectors are currently used by files. This makes it easy to find a free sector when writing a file. Each sector on the disk is represented by one byte in the map. A byte entry of -1 (0xFF) means that the sector is used. A byte entry of 0 (0x00) means that the sector is free. (In this example sector 0 is in use by the boot loader, sectors 1 through 255 are free.)

Not pictured is the directory (starting at 0x20200) which lists the names and locations of the files. There are 32 file entries in the directory, each of which contains 16 bytes (32 × 16 = 512, which is the storage capacity of a sector). The first eight bytes of each directory entry is the file name. (This is an historic hold-over; MSDOS file names followed an 8-dot-3 pattern.) The next two bytes are the starting position and number of sectors, respectively, which tell where the file is on the disk. If the first byte of the entry is zero (0x0), then there is no file at that entry.

We've seen that adding test files by hand is a lot of trouble without help. As described above you are provided with the utility **loadFile.c**, which reads a file and writes it to **floppya.img**, modifying the disk map and directory appropriately. For example, to copy our existing test files to the file system, type:

./loadFile kitty2
./loadFile fib
./loadFile msg

This saves you the trouble of using **dd** and modifying the map and directory yourself.

(Note that **loadFile** will truncate long file names to 8 letters after loading. In other words the file name **myMessage.txt** will become "myMessag" in our file system.)

Continuing our example, after adding these files to **floppya.img** and re-running **hexdump**, we note the following changes:

```
                …
Map             *
↳               00020000  ff ff ff ff 00 00 00 00  00 00 00 00 00 00 00 00  |................|
                00020010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
                *
↱               00020200  6b 69 74 74 79 32 00 00  01 01 00 00 00 00 00 00  |kitty2..........|
Dir             00020210  66 69 62 30 33 00 00 00  02 01 00 00 00 00 00 00  |fib.............|
                00020220  6d 73 67 00 00 00 00 00  03 01 00 00 00 00 00 00  |msg.............|
                00020230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
                *
↱               00020400  00 0f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
Config          00020410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
                *
                …
```

Consider the boldfaced directory entries in our new disk image. These indicate that there are valid files (with legal names indicated by the visible hex characters) occupying one sector each at sectors 1, 2 and 3. (Zero is not a valid sector number but a filler since every entry must be 32 bytes). If a file name is less than eight bytes, the remainder of the first eight bytes should be padded out with zeros. Also note that the disk map changed to reflect the use of sectors 1 through 3 by the three files.

You should note, by the way, that this file system is very restrictive. Since one byte represents a sector, there can be no more than 256 sectors used on the disk (128K of storage). Additionally, since a file can have no more than 255 sectors, file sizes are limited to this 128K. We can expand the amount of useable storage by using multiple sectors for the map and directory, but for this project this is adequate storage. For a modern operating system, this would be grossly inadequate.

Updating Interrupt 33

As before you will now add functionality to the o.s. by writing or updating three new functions. This code should be inserted into **kernel.c** before the **error()** function (more later). Additionally, once these functions are complete, you will have to add new interrupt 33 calls to access them, as seen atop the next page. We will now consider the writing of these described functions.

Function	AX=	BX=	CX=	DX=	Action
		…			
Read a disk file.	**3**	Address of character array containing the file name.	Address of character array where file contents will be stored.	Total number of sectors to read.	Call **readFile(bx,cx,dx)**
Load and execute a program.	**4**	Address of character array containing the file name of program to run.	Segment in memory to place the program into.		Call **runProgram(bx,cx)**
Terminate a running program.	**5**				Call **stop()**
		…			
Issue error message	**15**	Error number			(See below)

Part I. Reading a File

The new *main()* function for the kernel appears on the next page. This is basically a repeat of Lab 3: load and print **msg**. The trick is to use the file system to find the file to print.

The first thing to do is read the display parameters from the configuration file and use them to set up the display. Since this is one sector at a predictable location, you only need concepts from prior labs to do this. As you can see, you simply read the correct sector, recover the foreground and background colors, then clear the screen.

```
void main()
{
  char buffer[512]; int size;
  makeInterrupt21();

  /* Step 0 – config file */
  interrupt(33,2,buffer,258,1);
  interrupt(33,12,buffer[0]+1,buffer[1]+1,0);
  printLogo();

  /* Step 1 – load and print msg file (Lab 3) */
  interrupt(33,3,"msg\0",buffer,&size);
  interrupt(33,0,buffer,0,0);
  while (1);
}

/* more stuff follows */
```

Step 1: Loading and Printing a Text File

Start by creating a new function

void readFile(char* fname, char* buffer, int* size)

that takes a character array containing a file name and reads the file into a provided buffer. It should work as follows:

1. Load the disk directory (sector 257) into a 512-byte character array using **readSectors**.
2. Go through the directory trying to match the file name. If you don't find it, return with a "file not found" error (see below).

Remember that you don't have a library function to rely on for this; you'll have to do a character-by-character comparison.

3. Using the starting sector number and sector count in the directory, load the file into *buffer* via **readSectors**. Return when done.

Tie your function to interrupt 33/3. To test edit *main()* in **kernel.c** to match the code above. Note that 130,560 bytes is the maximum file size. All that happens here is that *main()* reads the text file into the buffer then prints it out before hanging up. After doing a **./compileOS.sh** to rebuild the disk and compile, execute **./loadFile msg** to add your test file to the disk image before executing. As before, if you run **bochs** and the message prints out, your function works. Also remember to test the "file not found" error (see below).

Step 2. Error Messages (Interrupt 33/15)

Error reporting becomes complicated because (1) the version of C recognized by bcc doesn't permit easy initialization of character strings, and (2) output from the kernel becomes complicated one we get multiple address spaces in play. (We previously ran into this trying to clear the screen.) As a work-around download the file **error.c** from the web page, cut-and-paste it into **kernel.c** above the interrupt handler and rewrite the kernel's interrupt handler like so:

```
void handleInterrupt21(int ax, int bx, int cx, int dx)
{
  switch (ax) {
    …
    case 15: error(bx); break;
    default: error(3);
  }
}
```

As seen, based on the parameter, the kernel prints the proper error message and then stops. (For now it hangs; eventually it will return control to the shell.) We will add options/errors as we complete the operating system. For now make sure that **readFile()** calls interrupt 33/15.0 if it can't find the requested file. Test this functionality by rewriting the **main()** function of the kernel to try and load **msg2**.

Part II. Executing a Program

Next rewrite **main()** of the kernel to look like the code at right. This should look familiar, since we did this in Lab 4. The only difference is to require the o.s. to find the program we want to run by name instead of us providing the location and sector count.

```
void main()
{
  char buffer[512];
  makeInterrupt21();
  interrupt(33,2,buffer,258,1);
  interrupt(33,12,buffer[0]+1,buffer[1]+1,0);
  printLogo();
  interrupt(33,4,"kitty1\0",2,0);
  interrupt(33,0,"Error if this executes.\r\n\0",0,0);
  while (1) ;
}

/* more stuff follows */
```

Now edit the **runProgram** header to become

void runProgram(char* name, int segment)

What follows should be clear. Replace the call to **readSectors** with one to **readFile** to load the requested information to the local buffer based on the file name. Tie interrupt 33/4 to **runProgram**. Test with all of our existing sample files (**kitty1**, **kitty2**, **fib**, and **Stenv**) as well as "file not found" (use a made-up name.)

As a final test, download, compile and run the **ddir** program from **main()**. (Refer to Lab 4 for instructions regarding this.) This program just lists all files stored on disk, skipping those whose names begin with a capital letter. Use **loadFile** to add files to your disk image and see what results **ddir** delivers.

Part III. Updating the Shell

To start, edit **kernel.c** in three places as seen atop the next page. From now on the kernel simply sets up interrupt 33, then loads and executes the shell in the second segment. Also, instead of simply hanging up, programs terminating in BlackDOS resume execution of the shell. This includes programs terminating due to errors as seen earlier.

Next edit your stub code in the shell to *implement* these commands:

- **ddir** List disk directory contents; load and execute **Ddir** at segment 4.
- **exec** *filename* Call interrupt 33/4 to load and execute *filename* at segment 4.
- **help** Display the user manual, described next.
- **prnt** *filename*
  Load *filename* into memory and print its contents (to the printer) using interrupt 33 calls.
- **senv** Set environment variables; load and execute **Stenv** at segment 4.
- **show** *filename*
  Load *filename* into memory and display its contents onscreen using interrupt 33 calls.

```
void main() {
  char buffer[512];
  makeInterrupt21();
  interrupt(33,2,buffer,258,1);
  interrupt(33,12,buffer[0]+1,buffer[1]+1,0);
  printLogo();
  interrupt(33,4,"Shell\0",2,0);
  interrupt(33,0,"Bad or missing command interpreter.\r\n\0",0,0);
  while (1) ;
}

/* more stuff here */

void stop() { launchProgram(8192); }

/* still more stuff here */
```

Notes on the User Manual

When the user types **help** a simple manual describing how to use the o.s. and shell should be displayed on screen. The manual should contain enough detail for a UNIX beginner to use it. For an example of the sort of depth and type of description required, execute **man csh** or **man tcsh**. These shells have much more functionality than yours, so your manuals don't have to be quite so large. Keep in mind that this is an operator's manual and not a developer's manual.

There are two ways to do this. One is to create a text file to load from disk and display on demand. Alternately (and preferably) write a simple program to display the manual one page at a time. At the end of a page of test, prompt the user to hit [ENTER] to move on.

Conclusion

That's a lot of pieces for one lab but we are almost finished with the full operating system. When finished with this lab, submit a .tar or .zip file (no .rar files) containing all needed files (**bootload**, **bdos.txt**, **kasm.o**, **compileOS.sh**, **kernel.c**, **config**, **map**, **loadFile**, **shell.c**, **basm.o**, **Stenv**, **Help**, **Ddir** and **README**) to the drop box. Make sure that your disk image includes the **fib**, **cal** and **t3** executables. **README** should explain what you did and how the t.a. can verify it. Your .zip/.tar file name should be your name.

Last updated 7.18.2019 by T. O'Neil, based on material by M. Black and G. Braught. Previous revisions 1.8.2019, 1.26.2018, 2.2.2017, 1.25.2016, 1.27.2015, 2.21.2014, 2.22.2014, 11.3.2014.