O.S. Lab 4: File I/O

Part I. Overview to Reading from Disk

We are now ready to begin tackling I/O involving the floppy disk. We begin with an overview of our file system. Then we will learn the use of BIOS system calls to read an individual disk sector. Finally we will write a routine to read an entire file into memory.

What You Will Need

Download the linked files from the web page and add them to your existing pieces (**bootload.asm**, **osxterm.txt**, **kernel.asm**, **kernel.c**, **config** 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 simple **spc02** file that we will use for testing purposes in this lab.

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
…
```

(The non-boldfaced italicized lines should already be in your script.) This sets up your initial file system.

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 16 file entries in the directory, each of which contains 32 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 remaining 24 bytes are sector numbers, 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.

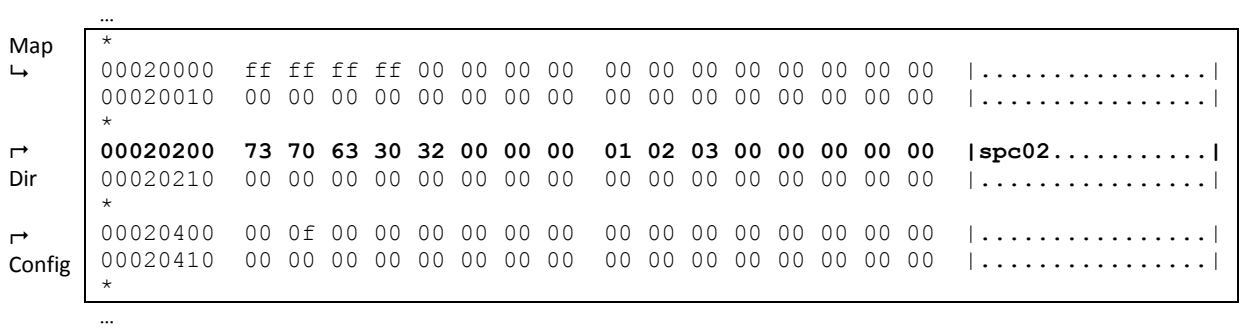
We've seen that adding test files by hand is a lot of trouble without help. As described above you are provided with the utility **loadFile.c**, which reads a file and writes it to **floppya.img**, modifying the disk map and directory appropriately. For example, to copy **spc02** to the file system, type:

./loadFile spc02

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 the **cal01** file to **floppy.img** and re-running **hexdump**, we note the following change:



Consider the boldfaced directory entry in our new disk image. It indicates that there is a valid file named "spc02" (visible hex characters **73 70 63 30 32**) occupying 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 **spc02**.

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 24 sectors, file sizes are limited to 12K. 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.

Error messages

Insert a function **void error(int bx)** into your program above the interrupt handler. This function accepts an error number as a parameter, prints out the corresponding error message (via interrupt 33/0) then hangs. Tie the function to interrupt 33/15 (see below). For now there are four relevant error messages:

Code (BX=)	Error message
0	File not found.
1	Bad file name.
2	Disk full.
Default	General error.

We will add to this as we add functionality to our operating system.

Updating Interrupt 33

As before you will now add functionality to the o.s. by writing four new functions. This code should be inserted into **kernel.c** before the *error()* function. Additionally, once these functions are complete, you will have to add new interrupt 33 calls to access them:

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)**
	…				
Delete a disk file.	**7**	Address of character array containing the file name.			Call **deleteFile(bx)**
Write a disk file.	**8**	Address of character array containing the file name.	Address of character array where file contents will be stored.	Total number of sectors to be written.	Call **writeFile(bx,cx,dx)**
		…			
Issue error message	**15**	Error number			Call **error(bx)**

We will now consider the writing of these described functions.

Step 0: Reading the Config File

The new *main()* function for the kernel appears below at right. This is the final version to be submitted. You should implement and test the required functionality a little at a time and not just try to do everything at once.

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.

Step 1: Loading and Printing a Text File

Next create 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 **readSector**.
2. Go through the directory trying to match the file name. If you don't find it, return with an error.

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 sector numbers in the directory, load the file, sector by sector, into *buffer*. You should add 512 to the buffer address every time you call **readSector**. Return when done.

To test edit *main()* in **kernel.c** to match the code above. Note that 12288 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 spc02** 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.

```
void main()
{
  char buffer[12288]; int size;
  makeInterrupt21();

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

  /* Step 1 – load/edit/print file */
  interrupt(33,3,"spc02\0",buffer,&size);
  buffer[7] = '2'; buffer[8] = '0';
  buffer[9] = '1'; buffer[10] = '8';
  interrupt(33,0,buffer,0,0);

  /* Step 2 – write revised file */
  interrupt(33,8,"spr18\0",buffer,size);

  /* Step 3 – delete original file */
  interrupt(33,7,"spc02\0",0,0);

  while (1) ;
}

/* more stuff follows */
```

Step 2: Writing a File

Next add the

void writeFile(char* name, char* buffer, int numberOfSectors)

function to the kernel that writes a file to the disk. The function should be called with a character array holding the file name, a character array holding the file contents, and the number of sectors to be written to the disk.

Writing a file means finding a free directory entry and setting it up, finding free space on the disk for the file, and setting the appropriate map bytes. Your function should do the following:

1. Load the disk directory (disk sector 257) and map (disk sector 256) to 512 byte character arrays (buffers).
2. Search through the directory, doing two things simultaneously:
   a. If you find the file name already exists, terminate with interrupt 33/15, function 1 (error one, "duplicate or invalid file name", see above).
   b. Otherwise find and note a free directory entry (one that begins with zero)
3. Copy the name to that directory entry. If the name is less than 6 bytes, fill in the remaining bytes with zeros.
4. For each sector making up the file:
   a. Find a free sector by searching through the map for a zero.
   b. Set that sector to 255 in the map.

c. Add that sector number to the file's directory entry.
d. Write 512 bytes from the buffer holding the file to that sector.

5. Fill in the remaining bytes in the directory entry with zeros.
6. Write the map and directory sectors back to the disk.

If there are no free directory entries or no free sectors left, your **writeFile** function should terminate with interrupt 33/15, function 2 (error two, "insufficient disk space", see above). Don't forget to add **writeFile** to interrupt 33 as described above.

To test edit the file you read slightly and save the revised copy under a new name. Running a **hexdump** on the disk image should verify that this worked.

Step 3: Delete File

Finally, now that you can write to the disk, you can delete files. Deleting a file takes two steps. First, you need to change all the sectors reserved for the file in the disk map to free. Second, you need to set the first byte in the file's directory entry to zero.

Add a **void deleteFile(char* name)** function to the kernel. Your function should be called with a character array holding the name of the file. It should find the file in the directory and delete it if it exists. Your function should do the following:

1. Load the disk directory and map to 512 byte character arrays as before.
2. Search through the directory and try to find the file name. If you can't find it terminate with interrupt 33/15, function 0 (error zero, "file not found").
3. Set the first byte of the file name to zero.
4. Step through the sectors numbers listed as belonging to the file. For each sector, set the corresponding map byte to zero. For example, if sector 7 belongs to the file, set the *eighth* map byte to zero (index 7, since the map starts at sector 0).
5. Write the character arrays holding the directory and map back to their appropriate sectors.

Notice that this does not actually delete the file from the disk. It just makes it available to be overwritten by another file. This is typically done in operating systems; it makes deletion fast and un-deletion possible.

Conclusion

When finished, execute the command **hexdump –C floppy.img > dump** at the Linux command line. Submit a .tar or .zip file (no .rar files) containing all needed files (**bootload**, **osxterm.txt**, **kernel.asm**, **compileOS.sh**, **kernel.c**, **map**, **config**, **loadFile**, **spc02**, **dump** and **README**) to the drop box. **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 1.26.2018 by T. O'Neil, based on material by M. Black and G. Braught. Previous revisions 2.2.2017, 1.25.2016, 1.27.2015, 2.21.2014, 2.22.2014, 11.3.2014.