O.S. Project 2: Loading and Executing Programs

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**, **lib.asm**, **kernel.c** and **compileOS.sh**). These files include

- Two disk image files **map.img** and **dir.img**, 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.
- A revised **kernel.asm** file with added functionality for manipulating registers. (The old **kernel.asm** file is no longer needed.)
- The simple text file **msg** and executable programs **test1** and **test2** that we will use for testing purposes.

Initial Map and Directory

The two additional files **map.img** and **dir.img** contain a map and directory for a file system consisting of only the kernel. You should edit your **compileOS.sh** file to include the following two 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.img of=floppya.img bs=512 count=1 seek=1 conv=notrunc
dd if=dir.img of=floppya.img bs=512 count=1 seek=2 conv=notrunc
bcc –ansi –c –o kernel.o kernel.c
…
```

(The two 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 at the beginning of the disk. The *disk map* sits at sector 1, and the *disk directory* sits at sector 2. (This is the reason your kernel starts at sector 3.)

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  b1 04 b6 00 b5 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  ff ff ff ff ff ff ff ff  ff ff ff ff ff 00 00 00  |................|
00000210  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  4b 45 52 4e 45 4c 03 04  05 06 07 08 09 0a 0b 0c  |KERNEL..........|
00000410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000600  55 89 e5 57 56 83 c4 b0  bb 87 01 53 e8 38 00 44  |U..WV......S.8.D|
…
000007a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00168000
```

Map ↳

↳ Dir

The map (starting at 0x200) 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 sectors 0 through 12 are in use, the rest are free.) You will not need to read or modify the map in this project since you are only reading files in this project; you will in the future.

The directory (starting at 0x400) 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 six bytes of each directory entry is the file name. The remaining 26 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.

For example, consider the boldfaced directory entry in our sample disk image. It indicates that there is a valid file named "KERNEL" (visible hex characters **4b 45 52 4e 45 4c**) occupying sectors 3, 4, … 12 (0x0c). (Zero is not a valid sector number but a filler since every entry must be 32 bytes). If a file name is less than six bytes, the remainder of the first six bytes should be padded out with zeros.

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 26 sectors, file sizes are limited to 13K. For this project, this is adequate storage, but 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 four new functions. This code should be inserted into **kernel.c** before the *handleInterrupt21()* function. Additionally, once these functions are complete, you will have to add new interrupt calls to access them:

Function	AX=	BX=	CX=	DX=	Action
		…			
Read a disk sector.	**2**	Address of character array where sector will be stored.	The number of the sector being read.		Call **readSector(bx,cx)**
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			Call **error(bx)**

We will now consider the writing of these described functions.

Reading a Disk Sector – Interrupt 19 (0x13)

BIOS interrupt 19 can be used to read or write sectors from the floppy disk. Reading sectors takes the following parameters:

- AH = 2 (this number tells the BIOS to read a sector as opposed to 3 for write)
- AL = number of sectors to read (use 1)
- **BX = address where the data should be stored to (pass your *char** array here)**
- CH = track number (*trackNo*)
- CL = relative sector number (*relSecNo*)
- DH = head number (*headNo*)
- DL = device number (for the floppy disk, use 0)

Recall that AX, CX and DX can be derived by applying the formulas:

- **AX = AH * 256 + AL = 513 in this case;**
- **CX = CH * 256 + CL = *trackNo* * 256 + *relSecNo*;** and
- **DX = DH * 256 + DL = *headNo* * 256.**

So this interrupt requires you to know the *relative* sector, head, and track numbers of the sector you want to read. In this project we are dealing with *absolute* sector numbers. Fortunately, there is a conversion for floppy disks:

- *relSecNo* = (*absSecNo* MOD 18) + 1
- *headNo* = (*absSecNo* DIV 18) MOD 2
- *trackNo* = (*absSecNo* DIV 36)

Recall from the last lab that bcc does not support MOD and DIV so we will have to use the functions we wrote there for these calculations.

You should now have everything you need to write the function **void readSector(char* buffer, int sector)** which takes two parameters: a predefined character array of 512 bytes or bigger, and an absolute sector number to read. Your function should compute the relative sector, head, and track, and call interrupt 19 to read the sector into *buffer*.

To test edit *main()* in **kernel.c** as follows:

```
void handleInterrupt21(int,int,int,int);

void main()
{
  char buffer[512];
  makeInterrupt21();
  /* BlackDOS header and clear screen */
  interrupt(33,2,buffer,30,0);
  interrupt(33,0,buffer,0,0);
  while (1) ;
}

/* more stuff follows */
```

All we need now is something at sector 30 to read. After compiling **kernel.c** and rebuilding **floppy.img**, execute the command

```
dd if=msg of=floppya.img bs=512 count=1 seek=30 conv=notrunc
```

to place the text file **msg** from the lab web page at disk sector 30. If you run **bochs** and the message prints out, your function works.

The loadFile Utility

You can see that adding test files by hand is going to be 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 **msg** to the file system, type:

./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 six letters after loading. In other words the file name **message.txt** will become "messag" in our file system.

Error messages

Write a function **void error(int bx)** that accepts an error number as a parameter and prints out the corresponding error message. Tie the function to interrupt 33/15. For now there are two error messages: error 0 is "File not found", while the default error is "General error". We will add to this as we add functionality to our operating system.

Loading and Printing a Text File

You can now 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 directory sector 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**.
4. Write the sector count back and return.

To test edit *main()* in **kernel.c** to match the code at right. Note that 13312 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.

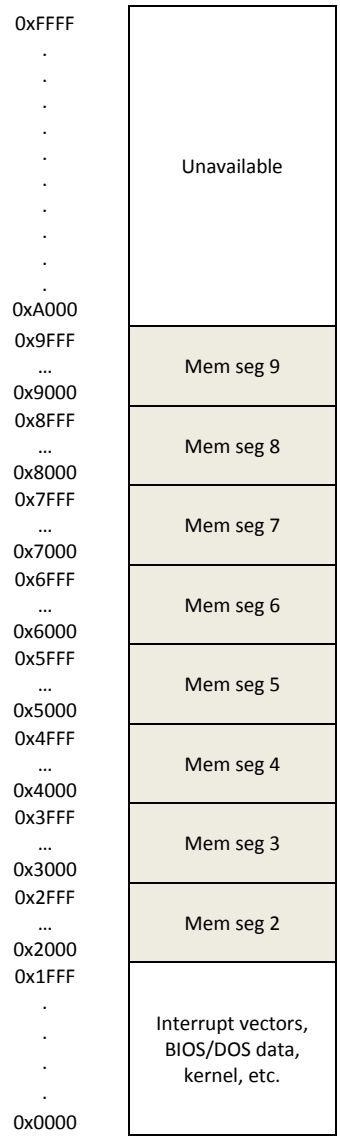
```
void handleInterrupt21(int,int,int,int);

void main()
{
  char buffer[13312];
  int size;
  makeInterrupt21();
  /* BlackDOS header and clear screen */
  interrupt(33,3,"msg\0",buffer,&size);
  interrupt(33,0,buffer,0,0);
  while (1) ;
}

/* more stuff follows */
```

Part II. Overview to Working with Executable Programs

Now that we can load entire files into memory, we should be able to load executable files into memory and run them. To simplify this we will have to begin with a consideration of the memory layout in the original IBM-PC architecture.

Background: Bochs RAM Layout

The original IBM-PC was limited to 640K of useable memory, which required 20-bit registers to access it. Since the engineers had only 16-bit registers they developed a *segment-offset memory scheme*. To derive a real 20-bit address the CPU would left-shift the 16-bit segment address by one byte, then add the 16-bit offset. There were numerous problems with such real-mode operations (e.g. redundant representations of the same address corrupting multiple segments), so eventually the IBM chip set had to be upgraded to the modern 32- (now 64-) bit address *protected mode*.



Now, if you're limited to 640K of memory, the highest segment number possible is 0x9C40, limiting the range of possible addresses. Furthermore some memory is used by the system (for interrupt vectors, BIOS, DOS, device drivers, …) and unavailable to user programs. To make most efficient and flexible use of the memory left the IBM architecture assigns *segment registers* to keep track of the starting addresses of code (CS), data (DS) and stack (SS) segments, plus index registers to hold data and instruction offsets relative to the segment bases (notably the stack pointer SP for the top of the stack/offset from SS). Thus segments vary in number and size over the computer's execution cycle.

So why the history lesson? *BIOS interrupts are only available in real mode*. This means that all IBM-compatible operating systems boot in 16-bit real mode so that they can use BIOS interrupts, then transition to protected mode for most user functions. If the o.s. needs BIOS functionality it traps to the kernel, enters real mode, does its business and returns.

Based on this discussion it should be clear that our simulation has been running in real mode. Therefore, as we now consider loading programs to memory and running them, we will implement our own primitive segment-offset addressing, depicted at right above. We will have fixed-sized segments with base addresses that are multiples of 4096 (0x1000) for a total of nine useable 4K memory segments. Note that if a program needs more than 4K it may occupy multiple adjacent segments.

What You Will Need

There are two new functions that you will use contained in the new **kernel.asm** file:

- **putInMemory(int baseLocation, int offset, char c)**
  Add *baseLocation* and *offset*, then write *c* to this computed address.
- **launchProgram(int baseLocation)**
  Set up registers, jump to *baseLocation* and commence execution.

Loading and Executing a Program

The process of loading a program into memory and executing it really consists of four steps:

1. Loading the program into a buffer (a big character array)
2. Transferring the program into the bottom of the segment where you want it to run
3. Setting the segment registers to that segment and setting the stack pointer to the program's stack
4. Jumping to the program

You should write a new function **void runProgram(char* name, int segment)** that takes as a parameter the name of the program you want to run (as a character array) and the segment where you want it to run (2 through 9 inclusive).

Your function should do the following:

1. Call **readFile** to load the file into a buffer.
2. Multiply *segment* by 0x1000 (4096) to derive the base location of the indicated segment.
3. In a loop, transfer the loaded file from the buffer into the memory based at the computed segment location, starting from offset 0. You should use **putInMemory** to do this.
4. Call **launchProgram** which takes the base segment address from (2) as a parameter.

After adding this function to interrupt 33/4 rewrite **kernel.c** to become this:

```
void handleInterrupt21(int,int,int,int);

void main()
{
  makeInterrupt21();
  /* BlackDOS header and clear screen */
  interrupt(33,4,"test1\0",2,0);
  interrupt(33,0,"Error if this executes.\r\n\0",0,0);
  while (1) ;
}

/* more stuff follows */
```

To test recompile your program, rebuild the floppy disk, **./loadFile test1** and execute. Run **bochs** and see

which message prints out.

Terminate Program System Call

This step is simple but essential. When a user program finishes, it should make an interupt 33 call to return to the operating system. This call terminates the program. For now, you should just have a terminate call hang up the computer, though you will soon change it to make the system reload the shell.

Two steps:

1. Add the code **void stop() { while (1) ; }** to your kernel prior to the interrupt handler.
2. Tie interrupt 33/5 to this function.

You can verify this all works with the **test2** program which does not hang up at the end (as **test1** does) but calls the terminate program interrupt.

Homemade Test Cases

Part of our justification for tying everything to interrupt 33 was so that programs other than our kernel could use o.s. functions. Let us now study a simple example.

From the lab page download the files **blackdos.h** and **fib.c**. A quick look at the listing of **fib.c** (seen at right) reveals what's happening. The program itself simply prints out a requested number of terms in the Fibonacci sequence. What is different are the invented commands for I/O. In the local header file we have created C-like macros for inputting and outputting strings (SCANS/PRINTS) and integers (SCANN/PRINTN), as well as for program termination (END). The preprocessor will substitute the correct interrupt calls for us, permitting us to hack code at a high level and ignore the underlying details. (It's still a little clumsy but to do better will require much more work.)

We compile the program in the same manner as the kernel but compile and link to **lib.asm** instead of **kernel.asm**, since we only need the single assembly function *interrupt()*. Any other low-level functions will be provided by our o.s. kernel.

```
#include "blackdos.h"

void main()
{
  int i, a = 1, b = 1, c, n;
  PRINTS("How many terms? \0");
  SCANN(n);
  if (n < 3) n = 3;
  PRINTN(n);
  PRINTS(" terms: \0")
  PRINTN(a);
  PRINTS(" \0");
  PRINTN(b);
  PRINTS(" \0");
  for (i = 0; i < n - 2; i++)
  {
    c = a + b;
    PRINTN(c);
    PRINTS(" \0");
    a = b;
    b = c;
  }
  PRINTS("\r\n\0");
  END;
}
```

As review, we now compile our program via the instruction sequence

```
bcc -ansi -c -o fib.o fib.c
as86 lib.asm -o lib_asm.o
ld86 -o fib -d fib.o lib_asm.o
```

then use **loadFile** to add the fib executable to our BlackDOS disk. Running the kernel to execute **fib** will now produce the desired result. For practice you should verify that this does indeed work. You should also create test cases of your own to experiment with going forward.

Conclusion

When finished, submit a .tar or .zip file (no .rar files) containing all needed files (**bootload**, **osxterm.txt**, **kernel.asm**, **compileOS.sh**, **kernel.c**, **dir.img**, **map.img**, **loadFile** 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 1.25.2016 by T. O'Neil, based on material by M. Black and G. Braught. Previous revisions 1.27.2015, 2.21.2014, 2.22.2014, 11.3.2014.