Project 2-C   Loading Files and Executing Programs

Build an Operating System from Scratch

Objective

In this project you will write routines to read files into memory and execute programs. You will then write a basic shell program that will execute other programs and print out ASCII text files.

What you will need

You will need the same utilities you used in the last project, and you will also need to have completed the previous projects successfully. Additionally, you will need to download the new *kernel.asm, map.img, dir.img, lib.asm, message.txt, tstprg,* and *tstpr2* for this project.

The File System

The main 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 Directory sits at sector 2. This is the reason your kernel starts at sector 3.

The Map 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 0xFF means that the sector is used. A byte entry of 0x00 means that the sector is 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 next.

The Directory lists the names and locations of the files. There are 16 file entries in the Directory and each entry contains 32 bytes (32 times 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 0x0, then there is no file at that entry.

For example, a file entry of:

```
4B 45 52 4E 45 4C 03 04 05 06 00 00 00 00 00 00 00 00 00 00...
K  E  R  N  E  L
```

means that there is a valid file, with name “KERNEL”, located at sectors 3, 4, 5, 6. (00 is not a valid sector number but a filler since every entry must be 32 bytes).

If a file name is less than 6 bytes, the remainder of the 6 bytes should be padded out with 00s.

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 (128kB of storage). Additionally, since a file can have no more than 26 sectors, file sizes are limited to 13kB. For this project, this is adequate storage, but for a modern operating system, this would be grossly inadequate.

Initial Map and Directory

You are provided, on Blackboard, with two additional files: *map.img* and *dir.img*. These contain a Map and Directory for a file system consisting of only the kernel. You should add to your *compileOS.sh* file two lines, just below the first line that creates the empty floppya.img:

```
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
```

This sets up your initial file system.

LoadFile

You are provided with a utility loadFile.c, which can be compiled with gcc (gcc -o loadFile loadFile.c). LoadFile reads a file and writes it to floppya.img, modifying the Map and Directory appropriately. For example, to copy *message.txt* from the previous project to the file system, type:

```
./loadFile message.txt
```

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

Note that "message.txt" is more than six letters. LoadFile will truncate the name to six letters on loading it. The file name will become "messag".

Step 1 - Load a file and print it

You should create a new function *readFile* that takes a character array containing a file name and reads the file into a buffer. When you complete your function, you should make it an interrupt 0x21 call:

Read File:
AX = 3
BX = address of character array containing the file name
CX = address of a buffer to hold the file

Your readFile function 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 do not find it, return.
3. Using the sector numbers in the directory, load the file, sector by sector, into the buffer array. You should add 512 to the buffer address every time you call readSector
4. Return

Testing

Use your readFile function to read in a text file and print it out. You can use your message.txt file from the previous project.

In main:
```
        char buffer[13312]              /*this is the maximum size of a file*/
        makeInterrupt21();
        interrupt(0x21, 3, "messag\0", buffer, 0);          /*read the file into buffer*/

        interrupt(0x21, 0, buffer, 0, 0);                   /*print out the file*/

        while(1);                     /*hang up*/
```

Then, after you compile, type:

```
        ./loadFile message.txt
```

Step 2 - Load a Program and Execute it

The next step is to load a program into memory and execute it. This 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

To try this out, you are provided on Blackboard with a test program *tstprg*. You should write your function to load tstprg into memory and start it running. Then, after compiling, you should use loadFile to load tstprg into floppya.img.

You should write a new function *void executeProgram(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.

The segment should be a multiple of 0x1000 (remember that a segment of 0x1000 means a base memory location of 0x10000). 0x0000 should not be used because it is reserved for interrupt vectors. 0x1000 also should not be used because your kernel lives there and you do not want to overwrite it. Segments above 0xA000 are unavailable because the original IBM-PC was limited to 640k of memory. (Memory, incidentally, begins again at address 0x100000, but you cannot address this in 16-bit real mode. This is why all modern operating systems run in 32 or 64-bit protected mode).

Your function should do the following:

1. Call readFile to load the file into a buffer.
2. In a loop, transfer the file from the buffer into the bottom (0000) of memory at the segment in the parameter. You should use putInMemory to do this.
3. Call the assembly function *void launchProgram(int segment)*, which takes the segment number as a parameter. This is because setting the registers cannot be done in C. The assembly function will set up the registers and jump to the program. The computer will never return from this function.

Finally, make your function a new interrupt 0x21 call, using the following:

Load program and execute it:
AX = 4
BX = address of character array holding the name of the program
CX = segment in memory to put the program

Testing

In main(), write the following:

```
makeInterrupt21();
interrupt(0x21, 4, "tstprg\0", 0x2000, 0);
while(1);
```

If your interrupt works and tstprg runs, your kernel will never make it to the while(1); Instead, the tstprg will print out a message and hang up.

Step 3 - Terminate program system call

This step is simple but essential. When a user program finishes, it should make an interupt 0x21 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.

You should first make a function *void terminate().* terminate for now should contain an infinite while loop to hang up the computer.

You should then make an interrupt 0x21 to terminate a program. It should be defined as:

Terminate program:
AX = 5

You can verify this with the program provided on Blackboard *tstpr2*. Unlike tstprg, tstpr2 does not hang up at the end but calls the terminate program interrupt.

Step 4 - The Shell - making your own user program

You now are ready to make the shell! You should download the file *lib.asm* from Blackboard. It contains a single assembly language function: *interrupt*. Your shell should not need any more assembly functions since all low level functions are provided by the kernel.

Your shell should be called *shell.c* and should be compiled the same way that the kernel is compiled. However, in this case, you will need to assemble *lib.asm* intead of *kernel.asm* and link *lib.o* instead of *kernel_asm.o*. Your final file should be called "shell".

After you compile the shell, you should use loadFile to load the shell onto floppya.img. You should add all of these commands to your compileOS.sh script.

Your initial shell should run in an infinite loop. On each iteration, it should print a prompt ("SHELL> " or "A:> " or something like that). It should then read in a line and try to match that line to a command. If it is not a valid command, it should print an error message ("Bad Command!" or something similar) and prompt again. Since you do not have any shell commands yet, anything typed in should cause Bad Command to be printed.

All input/output in your shell should be implemented using interrupt 0x21 calls. You should not rewrite or reuse any of the kernel functions. This makes the OS modular: if you want to change the way things are printed to the screen, you only need to change the kernel, not the shell or any other user program.

Kernel adjustments

In your kernel, *main()* should now simply set up the interrupt 0x21 using makeInterrupt21, and call an interrupt 0x21 to load and execute "shell" at segment 0x2000.

Also in your kernel, you should change *terminate* to no longer hang up. Instead it should use interrupt 0x21 to reload and execute "shell" at segment 0x2000.

Important note:
In terminate, do not write executeProgram("shell" *because strings are stored in data memory, which is still associated with the program that just ended (you'll learn about this in project E). Instead make a character array*

```
char shell[6];
```

copy the letters in one at a time

```
shell[0]='s';
shell[1]='h';
...
shell[5]='\0';
```

and then pass that array to executeProgram

```
executeProgram(shell,0x2000);
```

Step 5 - Shell Command "type"

You should now modify your shell to recognize the command "type filename". If the user types, at the shell prompt, *type messag*, the shell should load *messag* into memory and print it out the contents. You should implement this using interrupt 0x21 calls.

Step 6 - Shell command "execute"

Now modify the shell to recognize the command "execute filename". If the user types, at the shell prompt, *execute tstpr2*, the shell should call the interrupt 0x21 to load and execute *tstpr2* at segment 0x2000 (overwriting the shell).

You should test this by typing *execute tstpr2* at the shell prompt. If you are successful, tstpr2 should run, print out its message, and then you should get a shell prompt again.

Submission

You should submit a .zip (no .rar files please) containing all your files and a shell script for compiling on Blackboard Digital Dropbox. Be sure that all files have your name in comments at the top. Your .zip file name should be your name. You must include a README file that explains 1) what you did, and 2) how to verify it.