German University in Cairo
Faculty of Media Engineering and Technology
Dr. Amr Desouky
Eng. Nourhan Ehab
Eng. Rana Helal

CSEN 602-Operating Systems, Spring 2015
Course Project - Milestone 3: The Shell
Due on Thursday 31/3/2016 by 11:59 pm

Milestone Objective

In this milestone you will write routines to read, delete, and write files into memory as well as executing programs. You will then write a shell program that will execute other programs, print out ASCII text files along with other several commands. At the end of this milestone , you will have a fully functional single-process operating system that is about as powerful as CP/M.

Before you start

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. All of the new files are in `M3.zip`.

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.

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.

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Amr Desouky
Eng. Nourhan Ehab
Eng. Rana Helal

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 inadequate.

Initial Map and Directory

You are provided 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.

You are provided with a utility `loadFile.c`, which can be compiled with gcc by typing `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 just like you did with `printString`, `readString` and `readSector` in the previous milestone. If interrupt `0x21` is called with `AX=3`, `readFile` should be called. `BX` will contain address of character array containing the file name and `CX` will contain the 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`

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Amr Desouky
Eng. Nourhan Ehab
Eng. Rana Helal

To test your implementation, type the following in your `main` function:

```
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 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 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 IBMPC was limited to 640k of memory. (Memory, incidentally, begins again at address 0x100000, but you cannot address this in 16bit real mode. This is why all modern operating systems run in 32 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. When interrupt `0x21` is called with `AX=4`, `executeProgram` should be called. `BX` will contain the address of character array holding the name of the program, and `CX` will contain segment in memory to put the program.
To test your implementation, type the following in your `main` function:

```
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 a 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. If interrupt `0x21` is called with `AX=5`, `terminateProgram` should be called. You can verify this with the provided program `tstpr2`. Unlike `tstprg`, `tstpr2` does not hang up at the end but calls the terminate program interrupt.

Step 4: Write Sector
The first step is to create a `writeSector` function in `kernel.c`. Writing sectors is provided by the same `BIOS` call as reading sectors, and is almost identical. The only difference is that `AH` should equal 3 instead of 2 when calling interrupt `0x13`[1]. Your `writeSector` function should be added to interrupt `0x21`. When interrupt `0x21` is called with `AX=6`, `writeSector` should be called. `BX` will contain the address of the buffer that will be written to the specified sector, and `CX` will contain the sector number where the buffer will be written. If you implemented `readSector` correctly, this step will be very simple.

[1]Recall Step 3 in Milestone 2

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Amr Desouky
Eng. Nourhan Ehab
Eng. Rana Helal

Step 5: Delete File

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 `0x0`.

You should 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 specifically the following:

1. Load the Directory and Map to 512 byte character arrays.
2. Search through the directory and try to find the file name.
3. Set the first byte of the file name to `0x00`.
4. Step through the sectors numbers listed as belonging to the file. For each sector, set the corresponding Map byte to `0x00`. For example, if sector `0x07` belongs to the file, set the 8^{th} Map byte to `0x00` (you should set the 8^{th} byte not the 7^{th} since the Map starts with 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.

You should add `deleteFile` as an interrupt `0x21` call. When interrupt `0x21` is called with `AX=7`, `deleteFile` should be called. `BX` will contain the name of the file to be deleted.

You should test your function by adding the following lines in your main function:

```
char buffer[13312];
makeInterrupt21();
interrupt(0x21, 7, "messag\0", 0, 0); //delete messag
interrupt(0x21, 3, "messag\0", buffer, 0); // try to read messag
interrupt(0x21, 0, buffer, 0, 0); //print out the contents of buffer
```

then run the script file to compile `kernel.c` and load `message.txt` to `floppya.img`. If your delete function works, the message inside `message.txt` will not be printed out anymore. You can also verify that `deleteFile` works by opening `floppya.img` in `hexedit` and examining the directory entry for `messag`. The first byte should

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Amr Desouky
Eng. Nourhan Ehab
Eng. Rana Helal

be set to `0x00`. Additionally, The corresponding bytes representing the sectors making up `messag` should be set to `0x00` in the map.

Step 6: Writing a file

You should now add one last function to the kernel
`void writeFile(char* name, char* buffer, int secNum)` 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. You should then add `writeFile` as an interrupt `0x21` call. When interrupt `0x21` is called with `AX=8`, `writeFile` should be called. `BX` will contain the name of the file to be written, `CX` will contain the address to the array holding the file contents and `DX` will contain the number of sectors to be written.

In order to write a file successfully, you should find a free directory entry and set it up, find free space on the disk for the file, and set the appropriate Map bytes. Your function should do the following:

1. Load the Map and Directory sectors into buffers.
2. Find a free directory entry (one that begins with `0x00`).
3. Copy the name to that directory entry. If the name is less than 6 bytes, fill in the remaining bytes with `0x00`.
4. For each sector making up the file:
   - Find a free sector by searching through the Map for a 0x00.
   - Set that sector to 0xFF in the Map.
   - Add that sector number to the file's directory entry.
   - Write 512 bytes from the buffer holding the file to that sector.
5. Fill in the remaining bytes in the directory entry to `0x00`.
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 print an error message and return. You can test your function by writing the following in your main function:

```
int i=0;
char buffer1[13312];
char buffer2[13312];
buffer2[0]='h'; buffer2[1]='e'; buffer2[2]='l';  buffer2[3]='l';
buffer2[4]='o';
for(i=5; i<13312; i++) buffer2[i]=0x0;
makeInterrupt21();
```

```
interrupt(0x21,8, "testW\0", buffer2, 1); //write file testW
interrupt(0x21,3, "testW\0", buffer1, 0); //read file testW
interrupt(0x21,0, buffer1, 0, 0); // print out  contents of testW
```

If your function works, then "hello" will be printed out. You can also verify that `writeFile` works by opening `floppya.img` in `hexedit` and examining the directory sector. You should see an entry for `testW`.

Step 7: The Shell

You now are ready to make the shell! You are provided with a file `lib.asm` which 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.

To start the shell once your OS is booted, you need to adjust your kernel file. 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`.

Step 7-1: Shell Command - view

You should now modify your shell to recognize the command `view filename`. If the user types, at the shell prompt: `view messag`, the shell should load `messag` into memory and print out the contents of the file. You should implement this

using interrupt `0x21` calls.

Step 7-2: 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.

Step 7-3: Shell command - delete
You should add a `delete filename` command to the shell. Try loading `message.txt` onto `floppya.img` and set your kernel to load and execute the shell program just like you did above. When you type `delete messag`, the interrupt should be called and `messag` should be deleted. When you type `view messag`, nothing should be printed out. You should open up `floppya.img` with `hexedit` before and after you call `delete messag`. You should see the appropriate Map entries changed to 0 and the file marked as deleted in the Directory. Note that your shell should be user friendly. For example, if the user tries to delete or view a file that does not exist, he should be prompted that the file does not exist.

Step 7-4: Shell command: copy
In this step, you will write a copy command for the shell. The copy command should have the syntax `copy filename1 filename2`. Without deleting `filename1`, the copy command should create a file with name `filename2` and copy all the bytes of `filename1` to `filename2`. Your copy command should use only interrupt `0x21` calls for reading and writing files. If the user tries to copy a file that does not exist, he should be prompted. You can test this by loading `message.txt` onto `floppya.img`. At the shell prompt, type `copy messag m2`. Then type `view m2`. If the contents of `message.txt` print out, your copy function works. You should check the directory and map in `floppya.img` using `hexedit` after copying to verify that your writing function works correctly.

Step 7-5: Shell command: dir
In this step, you will write a `dir` command to list the contents of the directory. This command should print out the files in the directory. Only existent (not deleted) files should be listed. You should also print out the sizes of the files in sectors.

Step 7-6: Shell command: create
In this step, you will write a create command to create a text file. The create command should have the syntax `create filename`. The create command should

repeatedly prompt you for a line of text until you enter an empty line. It should put each line in a buffer. It should then write this buffer to a file. Test this step by calling `view filename`, and see if what you typed is printed back to you.

Submission

For this milestone you are required to submit a zip containing all of your files. You should use this webform `https://goo.gl/86CyRI` to submit your project. Late submissions will not be accepted.
Have fun :)