O.S. Project 3: BlackDOS Shell and Writing to Disk

Part I. Overview to Completing the Kernel

In this project you will implement functions for deleting and writing files, and add several new commands to your shell. At the end of the project, you will have a fully functional single-process operating system about as powerful as CP/M.

Updating Interrupt 33

One more time add options to the interrupt-33 handler in **kernel.c**:

Function	**AX=**	**BX=**	**CX=**	**DX=**	**Action**
		…			
Write a disk sector.	**6**	Address of character array storing sector to write.	The number of the sector being written.		Call **writeSector(bx,cx)**
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)**
		…			
Reboot DOS.	**11**				Call **interrupt(25,0,0,0,0)**
		…			

The last of these commands should be straight-forward. As for the others…

Step 1: Write Sector

The first step is to create a **writeSector** function **in kernel.c** to go with the **readSector** function. Writing sectors is provided by the same BIOS call as reading sectors, and is almost identical. The only difference is that **AX** should equal **769** instead of 513. Your write sector function should be added to interrupt 33, and should be handled as indicated above. If you implemented **readSector** correctly, this step will be very simple.

Step 2: 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 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 (disk sector 2) and map (disk sector 1) to 512 byte character arrays.
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(0), "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.

Step 3: Writing a File

Now add one last **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 map and directory sectors into 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(1), "duplicate or invalid file name", see below).
   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(2), “insufficient disk space”, see below). Don’t forget to add **writeFile** to interrupt 33 as described above.

Step 4. Error messaging and file name policies

Finally we extend the **error** function (interrupt 33/15) in the kernel to recognize and report these error messages:

Code (BX=)	Error message	Notes
0	File not found.	Carry-over from previous project
1	Bad file name.	Discussed above
2	Disk full.	Discussed above
Default	General error.	Carry-over from previous project
No code	Bad command or file name.	Exists only in shell
No code	Bad or missing command interpreter.	Related only to shell

For purposes of this implementation, we will assume that all user-created files must begin with a lower-case letter. System files (such as **KERNEL**, **Shell** and **Help**) begin with capital letters and have restricted access, i.e. the user cannot overwrite or delete them. Because this is a policy choice on our part and not necessarily required, we will implement this as part of the shell and not the kernel, which should be kept as small and general as possible.

Part II. Overview to the Shell

Your purpose now is to write a simple C-shell that will act as a command-interpreter interface between BlackDOS and a user making system calls. It should function like a simplified version of any popular UNIX shell (e.g. *csh*). This means using roughly the same syntax as UNIX shells and employing case sensitivity.

Modifications to Existing Pieces

Call your new file **Shell.c**. Add commands to **compileOS.sh** to compile it, assemble **lib.asm** and link it to the shell, compile **loadFile.c** and use it to load the shell executable onto the floppy disk. (See previous labs for reference.) In short the shell should be created and put on the disk automatically every time you remake your project from now on.

Next edit **kernel.c** in two places as seen at right. 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.

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

void main() {
  makeInterrupt21();
  /* BlackDOS header and clear screen. */
  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 */
```

The Shell Itself

Your initial shell should clear the screen, print a welcome message and then run in an infinite loop. On each iteration, it should print the sword prompt

cxxxx][===blackdos===>

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 the "bad command or file name" error message and prompt again.

All input/output in your shell should be implemented using interrupt 33 calls (with one exception). You should not rewrite or reuse any of the kernel functions. (You can use the **blackdos.h** file from the previous lab if you think that will make things easier.) 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.

Commands you are to recognize and implement, plus errors to catch:

- **boot**
  Reboot the system by calling 33/11 to reload and restart the o.s..

- **cls**
  Clear the screen by issuing 25 carriage return/line feed combos then interrupt 33/12.

- **copy** *file1 file2*
  Create *file2* and copy all bytes of *file1* to *file2* without deleting *file1*. Use only interrupt 33 for reading and writing files.

- **del** *filename*
  Delete the named file by clearing the appropriate map entries and marking it as deleted in the directory.

- **dir** List disk directory contents, described next.

- **echo** *comment*
  Display *comment* on screen followed by a new line (multiple spaces/tabs may be reduced to a single space); if no argument simply issue a new prompt.

- **help** Display the user manual, described next.

- **run** *filename* Call interrupt 33 to load and execute *filename* at segment 4.

- **tweet** *filename*
  Create a text file. Prompt the user for a line of text (shorter than 140 characters). Store it in a buffer and write this buffer to a file called *filename*.

- **type** *filename* Load *filename* into memory and print out the contents using interrupt 33 calls.

For **copy**, **del** and **tweet** issue a “duplicate or invalid file name” error (interrupt 33/15, error 2) if the file to be written or deleted has a name beginning with a capital letter, as discussed above. (Other error tests should have been built into the kernel.) All BlackDOS functions have been tested except for the file deletion and creation routines from this assignment. Test these by creating a simple text file, copying it then deleting the copy.

Notes on Directory

The floppy used by the simulator is a 3½-inch disk with 1.44Mb of storage. As noted previously, by the way we designed the file system, a total of 256 sectors (and 256 × 512 = 131,072 bytes) is being tracked by the BlackDOS file system.

As in the last project load the directory sector (sector 2) into a 512-byte character array and parse it 32 bytes at a time. Print out names and sizes (number of sectors used) of each file. (Note that there is a maximum of 16 total files.) At the end, print out total space used by the files and total free space remaining, both in terms of number of sectors.

Recall that deleted files (those whose file names begin with the number zero) do not appear in your directory or add to your file and sector counts. While here also construct **dir** so that file names beginning with capital letters (like **Shell** and **Help**) are hidden (i.e. not listed but adding to file and sector counts).

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.

Conclusion

Congratulations! You have now created a fully functional command-line based single process operating system that is almost as powerful as Bill Gates' first MS-DOS version. 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.c**, **shell.c**, **lib.asm** 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 2.17.2016 by T. O'Neil, based on material by M. Black. Previous revisions 1.27.2015, 2.22.2014, 2.24.2014, 11.10.2014.