Operating Systems

Project #4: Writing Files + Improved Shell

Objective

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 (an early PC operating system. See: http://en.wikipedia.org/wiki/CP/M).

Background

Like reading files, writing files requires that you understand the file system and how it keeps track of the names of the files on the disk and their locations. As such, there is no new background for this project. However, you may want to review the material on the Disk Directory and the Disk Map given in the Background section of Project 3.

Getting Started

There are no new files to be downloaded for this project. Make a copy of your `project3` directory as `project4`.

Writing a Disk Sector

The functionality for reading files was based on being able to read a sector. Similarly, the functionality for writing files will be based on being able to write a sector. Create a new function named `writeSector` in your kernel with the prototype:

```
int writeSector(char *buffer, int sector);
```

Writing sectors can be accomplished using the same BIOS call as reading sectors. The only difference is that `AH` should equal 3 instead of 2. Thus, you should be able to copy your `readSector` function and modify it fairly easily to write the data in buffer to the indicated sector. You can refer to the Project 2 assignment for the details of the BIOS call used in the `readSector` function.

To test your `writeSector` method you can have your main method write some known values into the first sector (0) or the last sector (2879) on the disk. You can then use the `hexdump` program to read the `floppy.img` disk image. If `writeSector` is working correctly the values that were written should appear at the beginning or the end of the disk image.

Since we do not want user programs writing arbitrary sectors on the disk, we will not provide access to `writeSector` through the system call interface.

Deleting a File

Deleting a file requires two steps:

1. All of the sectors allocated to the file must be marked as free (i.e., set to `0x00`) in the Disk Map.
2. The first character of the filename in the Disk Directory must be set to `0x00`.

These two steps do not actually erase the file (or even its full filename) from the disk. Rather, it just makes the directory entry and sectors used by the file available to be used for new files. The advantages of deleting files in this way are that it is fast and it also becomes possible to undelete a file (at least until its directory entry or sectors are reused). Most modern operating systems use a deletion process similar to this one.

Add a `deleteFile` function to your kernel with the following signature:

```
int deleteFile(char *fname);
```

This function should delete the named file using the two-step process described above. If the file is found and deleted, this function should return 1. If the file to be deleted cannot be found on the

disk, this function should return -1. After the Disk Map and Disk Directory are modified, they will need to be written back to the disk to save the changes.

The best way to test your `deleteFile` function is to have your main method delete a file (or files). Then open the `floppy.img` disk image and examine the Disk Map and Disk Directory to ensure that they have been modified to correctly reflect the deleted files.

Add and test a system call for deleting a file by modifying your `handleInterrupt21` function so that it provides the following service:

`deleteFile`:	delete a file from the disk.
`AX:`	`0x07`
`BX:`	The name of the file to be deleted.
`CX:`	Unused
`DX:`	Unused
`Return:`	1 if the file is successfully deleted or -1 if the file cannot be found.

Writing a File

Writing a file requires finding an empty entry in the Disk Directory and finding a free sector in the Disk Map for each sector making up the file. The data for each sector is written into a free sector on the disk and the sector numbers that are used are entered into the Disk Directory entry for the file. The modified Disk Directory and Disk Map must then be written back to the disk to save the changes.

Add a `writeFile` function to your kernel with the signature:

```
int writeFile(char *fname, char *buffer, int sectors);
```

This function should write `sectors` * 512 bytes of data from the buffer into a file with the name indicated by `fname`.

- If the file indicated by `fname` already exists, the new file overwrites it.
- The maximum number of sectors that may be written is 26. If `sectors` is larger than 26, then only the first 26 sectors should be written.
- If the file is successfully written, this function returns the number of sectors that were written.
- If there is no Disk Directory entry available for the new file, this function should return -1 and the file is not written.
- If the Disk Map contains fewer than `sectors` free sectors, this function should write as many sectors as possible and return -2.

One way to test your `writeFile` function is to use your `readFile` function to read a file into a buffer then write it to a new file. Once you have written the new file you can then use `readFile` again to read it in and print it out. If the contents are the same as the original file, that is an indication (but not a guarantee) that your `writeFile` function is working correctly.

Add and test a system call for writing a file by modifying your `handleInterrupt21` function so that it provides the following service:

`writeFile`: write a file to the disk.

AX:	`0x08`
BX:	The name of the file to be written.
CX:	The address of a buffer containing the data for the file.
DX:	The number of sectors to be written.
Return:	The number of sectors written or -1 if there is no Disk Directory entry available for the file or -2 if there are insufficient free sectors to hold the file.

Improvements

1. Add functions to `userlib.c` for deleting a file and writing a file to disk.
2. Use your user library to add the following functionality to your shell:

Shell Command: delete <file>

Extend your shell so that it recognizes the command: `delete <file>`. This command should remove the specified file from the disk. If the file does not exist on the disk, the shell should print "File not found."

Shell Command: copy <src> <dest>

Extend your shell so that it recognizes the command: `copy <src> <dest>`. This command should make a copy of the file `<src>` as a new file `<dest>`. If the destination file already exists, it should be overwritten. If the source file cannot be found, then the shell should print "File not found." If the destination file cannot be created because the directory is full the shell should print "Disk directory is full." If the disk becomes full before the entire source file is copied, the shell should print "Disk is full."

Shell Command: dir

Extend your shell so that it recognizes the command: `dir`. This command should display a list of the files currently stored on the disk. Since we do not want to allow general direct access to sectors, add a specialized function in userlib.c to read sector 2. You will need to update kernel.c accordingly.

If you completed the `printInt` function as a bonus exercise in project 2, you might also have the `dir` command display the size, in sectors, of each file.

3. Create a simple line-based text editor that can be executed as a user program. When executed your text editor should prompt the user to enter a filename into which the entered text will be stored. The editor should then read lines of text from the user, storing them into a buffer. The editor should not allow the user to enter more text than will fit in a file on the system. When the user types CTRL-D (ASCII `0x04`) followed by the return key, the editor saves the file and terminates. If the file specified by the user already exists, it is overwritten with the new text. Have your `run.sh` script compile your editor and add it to your disk image using the `loadfile` program.

Bonus Features

1. Most modern operating systems make it possible to have multiple names for the same file. In Windows these are called shortcuts, in Mac OS they are called aliases, and in Unix they are called symbolic links. You can learn more about symbolic links in Unix by typing the command `man ln` in the terminal window.

Add a system call to your kernel that creates a symbolic link to a file. Your system call should be implemented by modifying your `handleInterrupt21` function so that it provides the following service:

`linkToFile`:	create a symbolic link to a file
`AX:`	`0xA0`
`BX`:	The name of the file to link to.
`CX`:	The name of the link.
`DX`:	Unused
Return:	1 if the link is successfully created or -1 if the file to link to does not exist or -2 if the link cannot be created because the Disk Directory is full.

The name of the symbolic link should appear in the Disk Directory. To indicate that the Disk Directory entry is a symbolic link and not an actual file, the first bytes of the entry that indicate the file's sectors should be set to `0xFF`. The next six bytes can then be used to indicate the filename of the file to which the symbolic link points.

Create a user library function for creating symbolic links and add a `symlnk <src> <dest>` command to your shell.

When completed, all of your OS functionality should work equally well on real files and symbolic links. For everything except deleting a file, commands operating on symbolic links should behave

exactly as if they were operating on the actual file (e.g., `type`, `copy`). When a symbolic link is deleted, only the link is deleted; the original file is unchanged.

Note that if the file pointed to by a symbolic link is deleted the link should remain. An attempt to `type` or `copy` such a symbolic link should report "File not found." This also implies that if a new file is created with the same name as pointed to by the link, then the link will be a link to the new file!

2. Implement command history in your shell. In most Unix shells, the up and down arrow keys will scroll back and forth through a list of some number of previously entered commands. The previous commands are displayed at the command prompt, and the user can press enter to execute that command.

3. Implement scripting in your shell. The execute command should be able to execute either a standard executable program or a shell script. Shell scripts should be standard text files that begin with the character sequence "#!" (called a shebang). Note: the shebang is essentially a magic number for identifying shell scripts. If the file that is executed is a shell script, the shell should read each line from the file and attempt to execute it as if it were typed at the command line. Note that with this functionality you can now use your text editor to create a program (a shell script) that you can then run!

Submission

Drew University

Create a zip archive of your entire project4 folder and submit it to google classroom. Your zip should contain everything that is required. Unlike previous projects, I will not be replacing or adding any files to what you submit in order to compile or run your OS.

Washington and Lee University

Copy your `project4` directory into your `turnin` directory.

Grading

Your project will be graded on its correctness as well as its style. Your source code files and your `run.sh` script should be nicely formatted and well documented.

Acknowledgement

This assignment as well as the accompanying files and source code have been adopted with minor adaptations from those developed by Michael Black at American University. His original assignments can be found at: http://nw08.american.edu/~mblack/teaching.html. His paper "Build an operating system from scratch: a project for an introductory operating systems course" can be found in the ACM Digital Library at: http://portal.acm.org/citation.cfm?id=1509022.