CS355 Hw 7

Due by the end of day **Friday, 5/10** (seniors) or **Friday, 5/17**. Design document due on **Monday, 4/15** in class. Supplementary questions due on **Monday, 4/22** in class. Live demos are to be scheduled for **Thursday, 5/9**.

Programming Assignment: File System

In this project, you will implement a file system from scratch. A "disk" in this project is a single Unix file that is "formatted" so that it is understandable to your "OS". You can think of such a file as a disk image. You must provide a shell that will be able to navigate this file system and execute commands that allow a user to perform normal file system related activities. See below for details.

Your file system should have a hierarchical tree structure, support file permissions and mounting (EC). Provide a separate "format" utility that creates and initializes your disk files. The format program should run separately from your shell. This project should be completed in groups of max 4 students. You may base your implementation on previous work from any team member.

1 Implementation Details

The file system is the largest program you will write in this class. Implementing it cleanly, efficiently and extensibly is a challenge. A file system should have two components: the physical structure and the virtual structure. The basic concept of any file system is to take a large area of permanent storage and break it up into small chunks. The design of the file system internals is entirely upto you. Either a MS-DOS compatible FAT system or a Unix-like inode-based system are logical choices. As before, you should implement reasonable emulations of true FAT or inodes. In other words, if you do FAT, then it needs to be at least FAT-12 and if you do inodes, it needs to be at least as sophisticated as early Unix V7. Whichever type you choose it must support a hierchical directory system. When making decisions, you should go with conventions and not make unreasonable choices. For example, you should stick with the standard block size of 512 bytes (even if you choose FAT, you can assume that your disk isn't going to be large enough to require a bigger block size), and not assume too small a maximum file size. That is, design your data structures so that it is possible for a single file to take up many GBs (or the max allowable by FAT-12), even if your current disk isn't even remotely large enough.

Tannenbaum discusses physical implementations of file systems in Chapter 4, and in Sections 10.6 and 11.8, which are good references. As usual, you are encouraged to research widely and use any resources you find useful, but are cautioned against taking large amounts of code that you do not understand.

2 File System Functions

2.1 The Virtual File System

A Virtual File System (VFS) is a robust interface to multiple physical disks of varying types (you may only have one, of course). The basic idea behind VFS is to provide a generic representation of the hierarchical file system in memory, as well as a set of functions to access parts of the file system.

The basic structure of the VFS is the `vnode`. A `vnode` can represent a file, a directory or a mount point. The `vnode` contains generic information about what it represents (name, permissions, etc). It also contains specific information depending on the file system (physical inode, FAT pointer), and a pointer to a set of functions specific to that file system type. In general, you want to use the `vnode` to store the structure of the file system (i.e. location in hierarchy, location on disk, etc) and the physical layer to store individual file data.

```
typedef struct {
  fd_t (*f_open)(vnode_t *vn, const char *filename, int flags);
  size_t (*f_read)(vnode_t *vn, void *data, size_t size, int num, fd_t fd);
  ...  /* and the rest of the file system library functions */
} fs_driver_t;

typedef struct vnode {
  int vnode number;
  char name[255];
  struct vnode *parent;
  int permissions;
  int type;
  fs_driver_t *driver;
  union {
    struct unix_fs {   /* these structs need to be fleshed out by you */
      int inode;
    } unixfs;
    struct dos_fs {
      int fat_ptr;
    } dosfs;
    struct ram_fs {
      ulong offset;
    } ramfs;
  } fs_info;           /* file system specific info */
} vnode_t;
```

You are not required to implement the full VFS. As long as you fully implement the required file system library functions (see subsection below). However, in that case, your file system is hard-

wired into one design only and there is no easy way to accept another differently formatted and designed one without replacing the entire library.

2.2 FS Library Functions

As part of the VFS you need to create programmer-level library functions to provide access to your file system operations. A list of library functions that you should implement has been outlined for you. The listed calls are very similar to the equivalent standard ANSI C library functions. Consult K&R or Unix man pages for specific semantics that you should emulate. These functions should be in a separate file, which you can then choose to build into a shared library if you prefer.

Your shell should call the appropriate functions from your library when implementing the list of related built-in commands - see Section 3.

1. `f_open`: open the specified file with the specified access (read, write, read/write, append). If the file does not exist, handle accordingly. (rule of thumb: create file if writing/appending, return error if reading is involved). Returns a file handle if successful.
2. `f_read`: read the specified number of bytes from a file handle at the current position. Returns the number of bytes read, or an error.
3. `f_write`: write some bytes to a file handle at the current position. Returns the number of bytes written, or an error.
4. `f_close`: close a file handle
5. `f_seek`: move to a specified position in a file
6. `f_rewind`: move to the start of the file
7. `f_stat`: retrieve information about a file
8. `f_remove`: delete a file
9. `f_opendir`: recall that directories are handled as special cases of files. open a "directory file" for reading, and return a directory handle.
10. `f_readdir`: returns a pointer to a "directory entry" structure representing the next directory entry in the directory file specified.
11. `f_closedir`: close an open directory file
12. `f_mkdir`: make a new directory at the specified location
13. `f_rmdir`: delete a specified directory. Be sure to remove entire contents and the contents of all subdirectorys from the filesystem. Do NOT simply remove pointer to directory.
14. `f_mount`: mount a specified file system into your directory tree at a specified location. (Extra Credit)
15. `f_umount`: unmount a specified file system (Extra Credit)

3 The Shell

You should extend your shell to work with the new file system. If however you really hate your shell, or that it was too unstable, you can write a new one without providing support for job control. However, basic functionalities of executing non-file-system-related Unix binaries should still be possible, i.e. commands like `echo`, `ps`, etc, that is, start from the basic shell of hw2. File-system related commands should be replaced as decribed below in 5:

1. On start-up, the shell should look (in the same directory) for a file called "DISK" where your file system is located. If it is not found, please print a message asking the user to run the `format` command to make one. For more details on `format`, see Section 4.

2. Once your shell has found the file system, it should mount it at the root directory and display a log-in prompt.

3. Create at least a regular user and a super user, and start the user at the appropriate home directory after logging in.

4. Redirection
   Support redirections. You need to add parsing for `<`, `>` and `>>`. Again, just like the parsing for `&` and `;`, white spaces are not necessary to separate these tokens. If you are not familiar with redirections, please play with them first on a true Linux shell to understand how they work. Redirection is the only way to test file writing/creation with your file system (unless you want to write an editor too), so please make sure it works robustly. What this means is that you can use any of the following commands to create files (in fact, anything that writes to stdout - note that below, `ls` and `cat` are your commands while the rest are from Unix binaries, but it makes no difference):

   (a) `echo ‘‘roses are red’’ > f1.txt`

   (b) `echo ‘‘violets are blue’’ >>f1.txt`

   (c) `echo ‘‘honey is sweet’’ > f2.txt`

   (d) `cat f1.txt f2.txt > f3.txt`

   (e) `ls > f4.txt`

   (f) `ps > f5.txt`

   (g) `yes > yes.txt` - note that this creates a VERY large file very quickly. I recommend breaking with `ctrl-c` after counting to 3, maximum.

5. The following file-system related commands must now be implemented as built-in commands in your shell. You are not allowed to use the Unix-provided ones, not that they will work with your file system anyways.

   (a) `ls`
   ls lists all the files in the current or specified directory. Support flags -F and -l.

(b) `chmod`
chmod changes the permissions mode of a file. Support absolute mode and symbolic mode.

(c) `mkdir`
mkdir creates a directory

(d) `rmdir`
rmdir removes a directory

(e) `cd`
cd changes the current working directory according to the specified path, or to the home directory if no argument is given. Support `.` and `..`

(f) `pwd`
pwd prints the current working directory

(g) `cat`
cat displays the content of one or more files to the output.

(h) `more`
more lists a file a screen at a time

(i) `rm`
rm deletes a file

(j) `mount` (Extra Credit)
mount mounts a file system at a specified location. Note that it should be possible to format multiple disks and mount them at arbitrary locations in your file system

(k) `umount` (Extra Credit)
umount unmounts a file system

4 Misc

- `format`
  The `format` utility program is a separate program that you should make available.

  `format <name of file>` will format a file with your file system design and data structures. If the file doesn't exist, create it. The default size for such a disk image should start at 1MB. Add a "`-s #`" flag to allow the creation of different-sized disks, where `#` is a number in megabytes.

- Design document

  A design document is required from each group by Monday 4/15, where you sketch an outline of how your program is going to work. I want to see detailed explanations of your file system design in each of these areas:

  – Physical structure of your file system: FAT or inodes?

  - Virtual structure of your file system: full VFS or just a library? If not full VFS, where does the rest of the stuff go? Please describe in detail the steps your OS will perform when an open call is made, for example.
  - The file table - data structure to keep track of open files
  - Free-space management
  - Block-access within a file
  - Directory file structures
  - superblock and FAT entry/inode layout (as actual structs)

  As usual, try the best you can to lay out a skeleton program with calls to functions and declarations of important variables and data structures. Functions should be in prototypes only, with short comments on what the function takes, what it returns and what it is supposed to do.

- EXTRA CREDIT: We can discuss possible extensions if you finish early and really want to do more - there are many options. However, my experience is that this project is large enough that students are generally best served by spending the time making the outlined system more stable.

- Hints:

  - Plan ahead. Or just plan. In the "real world", more time is often spent on the design of projects than the actual implementation. Obviously this is not the real world, but every little bit of planning helps.
  - Use a debugger!
  - Write less code and test more. Things will be infinitely easier for you if you spend time to make sure that every component is FULLY functional before you move on.
  - As the previous projects, any crashing will be heavily frowned upon. You grade will also be infinitely better if you have a fully functional system that is missing a lot of the features than a non-functional system with all the features!
  - Memory leaks and memory errors are also not tolerated and will incur heavy penalties.
  - There will always be one nagging item in a particular code segment that you think if you had five more minutes you can fix it. At some point, you have to let it go. Your time will be better spent on other tasks that need to get done instead of making a single part of your code "perfect".
  - When in doubt, do what Linux does.
  - This is not a trivial assignment, start EARLY!

Supplementary Questions: Do the following questions from Tanenbaum.

- pages 429-434, problem numbers 12, 22, 31, 37 and 40.
- pages 589-592, problem numbers 4, 10, 15, 16, and 21.
- ~~pages 515-516, problem numbers 3, 12, 18, 20, and 34.~~

Deliverables:

1. Harcopy: Design document (Monday 4/15 in class - bring printouts so that I can read them and we can discuss during lab on Wed 4/17).
2. Hardcopy: Answers to the supplementary questions (Monday 4/22).
3. Electronic submissions of the programming assignment. Please include the `format` utility and one formatted 1MB disk (don't submit more than one, or one that's too large, or together you will blow my quota). Other things I expect to see in your archive:
   - **Multiple** source code files and header files that implement your file system library, shell, and `format`.
   - Makefile
   - Extended README (see below)
4. Hardcopy: An extended README document that discusses the following:
   (a) design of your file system (enough so that I can understand what you are actually doing)
   (b) whether your final implementation changed from your design document. If so, why.
   (c) a list of what's working, partially working and not working
   (d) how you tested.
5. Demo: to be scheduled for Thurs 5/9. Bring a hardcopy of your README to your demo.