CS 560 Assignment 1

Duwal Shrestha, Mahendra
`mduwalsh@utk.edu`

Nash, Drew
`anash4@utk.edu`

7 March 2015

Introduction

The program `filesys.c` implements a Unix-like shell and file system, according to the specifications outlined in the Programming Assignment 1 instructions. Since this program was written in C, to run this file system, it must be compiled using the following command:

```
host$ gcc -o sh filesys.c
```

Note that the `sh` can be replaced with any name that the user wishes to call the shell executable file. Then run the shell by using the following command:

```
host$ ./sh
virtual$
```

Note that once the shell is initialized, commands can be entered after the `$` character that is automatically printed by the shell. To exit the shell, send either the `exit` or `ctrl+d` command, since the shell will continuously accept commands until the `EOF` signal is reached.

Section 1 outlines the commands that were implemented in Part 1 of this assignment. Section 2 outlines how to initialize and use the remote shell that performs remote procedure calls. To run a script of commands, use the following syntax:

```
host$ ./sh < input.txt > output.txt
```

Everything that `sh` sends to `stdout` will be redirected to `output.txt`. Section 3 outlines how to run an executable file within the file system's shell.

This program is was implemented using the concept of *inodes*, which are the foundation of the UNIX file system. The `inode struct` contains the following components: `c_date` for modify date, `flags` to indicate to what type of data the inode points, `fSize` for file size, `l_count` to keep track of the number of links pointing to this inode, and `pointer[IPOINTS]` that is an array of pointers to the actual data blocks. For this program `IPOINTS` is set equal to thirteen. With 25 inode blocks implemented in `filesys.c`, and the pointers in each inode to 13 data blocks, this filesystem is primitive compared to today's typical operating system. However, considering the constant size is 100 MB, it does use its storage space effectively. Note that block 0 is the `struct superblock`, which keeps track of the states of all files in this system. Furthermore, a free list is used to keep track of available and occupied blocks. For more information, see the comments in `filesys.c`.

1 Shell Commands

The following commands have been implemented into this file system and can be called while the shell `sh` is running. The syntax and information regarding the appropriate use of each function is listed below. Note that `filesys.c` can handle commands up to 200 characters in length.

1.1 mkfs

Run this command using the following syntax:

```
virtual$ mkfs
```

This command initializes the file system. If no file system exists yet, then this should be the first command run once the shell is executed. Note that running this command again will erase the file system and reinitialize a new one.

1.2 open

Run this command using the following syntax:

```
virtual$ open filename flag
```

This command opens the file to which the `filename` link points. To open the file for reading, use the `r` flag. To open for writing, use the `w` flag. To open for reading and writing, use the `rw` flag. This function returns a file descriptor integer that points to the open file. If the returned file descriptor is equal to -1, then an error occurred while opening the file. Furthermore, if a file that does not exist is opened with the `w` flag, then a new file of size 0 is created.

1.3 read

Run this command using the following syntax:

```
virtual$ read filedescriptor size
```

This command reads `size` bytes from the file to which the `filedescriptor` points and prints the contents in the shell. If the file was just opened, then the reading starts at the beginning. Otherwise, it continues reading from where it most recently stopped. This location is referred to as the *file offset.*

1.4 write

Run this command using the following syntax:

```
virtual$ write filedescriptor string
```

This command writes `string` to the file to which the `filedescriptor` points, starting at the current file offset. If the offset is before the end of the file, then the contents of the file at that location will be overwritten. If the offset is at the end of the file, then the size of the file will be increased.

1.5 seek

Run this command using the following syntax:

```
virtual$ seek filedescriptor offset
```

This command moves the current file offset location in the file to which the `filedescriptor` points to the location `offset`, which is the number of bytes from the beginning of the file.

1.6 close

Run this command using the following syntax:

```
virtual$ close filedescriptor
```

This command closes the file to which the `filedescriptor` points. It should only be called for files that are currently open. Furthermore, it is recommended that the `close` function be called at some point after each `open` call prior to closing the shell.

1.7 mkdir

Run this command using the following syntax:

```
virtual$ mkdir directoryname
```

This command creates a directory called `directoryname` inside of the current directory of the shell. Users should not call this function with a `directoryname` that already exists within the current directory.

1.8 rmdir

Run this command using the following syntax:

```
virtual$ rmdir directoryname
```

This command removes the directory called `directoryname`, if and only if `directoryname` refers to an empty directory that exists inside of the current directory of the shell.

1.9 cd

Run this command using the following syntax:

```
virtual$ cd directoryname
```

This command changes the current directory of the shell to the `directoryname` location. Users should only call this function if `directoryname` refers to a valid directory.

1.10 link

Run this command using the following syntax:

```
virtual$ link filename linkname
```

This command creates a new link called `linkname` that points to an existing file called `filename`. Note that `filename` is technically an existing link to that file. Therefore, users should not call this function with `filename` and `linkname` being in the same directory. Furthermore, `filename` must be the name of a file and not of a directory.

1.11 unlink

Run this command using the following syntax:

```
virtual$ unlink filename
```

This command deletes the link `filename` that currently exists and points to a valid file. Once a file has zero links, the file itself is deleted. Therefore, this is the command that users must use to delete a file.

1.12 stat

Run this command using the following syntax:

```
virtual$ stat filename
```

This command will cause the shell to print the status of the file to which the `filename` link points. The status is the information that is contained in the file's *inode*, including type, link count, size, and number of allocated blocks.

1.13 ls

Run this command using the following syntax:

```
virtual$ ls
```

This command will cause the shell to print the contents of the current directory.

1.14 cat

Run this command using the following syntax:

```
virtual$ cat filename
```

This command will cause the shell to print the contents of the file to which the link `filename` points.

1.15 cp

Run this command using the following syntax:

```
virtual$ cp sourcename destinationname
```

This command will open the file to which the `sourcename` links points with read-only capability, open the file to which the `destinationname` link points with write-only capability, and copy the contents of `sourcename` to `destinationname`. Users should only call this function if `sourcename` refers to an existing, valid file.

1.16 tree

Run this command using the following syntax:

```
virtual$ tree
```

This command will cause the shell to print the contents of the current directory in the format of a tree, including the date and file size for files within the directory. However, size of sub-directories is not included.

1.17 import

Run this command using the following syntax:

```
virtual$ import sourcename destinationname
```

This command imports the `sourcename` file, which is located on the host machine, into the file system by copying its contents into the `destinationname` file, located in the current directory of the file system. Note that users should only call this function for `sourcename` files that exist, and for which the user has permission to read.

1.18 export

Run this command using the following syntax:

```
virtual$ export sourcename destinationname
```

This command exports the `sourcename` file, which is located in the current directory of the file system, into the host machine by copying its contents into the `destinationname` file. Note that users should only call this function with a `destinationname` for which the user has permission to write.

2 Remote Shell

This file system supports a remote shell, according to the specifications listed in Part 2 of the instructions. In this document, the host machine on which the `sh` executable is running is called the `server` and the remote machine is called the `client`. Since this file system can only support one user at a time, the host machine must be initialized into server mode. Once in server mode, all input on the server will be ignored, except the `ctrl+c` signal to end the entire shell process. Furthermore, all output, including `stdout` and `stderr`, will be redirected to the client. The following command, given while the `sh` executable is running, will call for the host computer to be initialized to server mode:

```
virtual$ startserv // command to start server entered in shell
Starting server mode...
Enter port number.  Ports 5555 and 5500 are recommended.
5555 // enter port number of choice here
```

Ports 5555 and 5500 are recommended, but any available port number on the `server` machine will work. If opening the port is successful, the following message will be displayed on the `server`:

```
Now redirecting input/output to Client.
```

Once this message has been displayed, then on the `client` machine, run the following `telnet` command:

```
host$ telnet machine.eecs.utk.edu port
```

Remember to replace `machine` with the name of a Hydra machine, such as `hydra7`, and `port` with the port number that was used on the `server` to initialize it. Also, note that this feature was only tested in the Hydra lab of the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. It is confirmed to be operational between two machines in this lab. If using this feature on other machines, ensure that firewall systems are not blocking traffic.

Once the `telnet` command has been successfully run on the `client`, remote procedure control is operational. A user can now interact with the shell on the `client` in the same manner as done on the `server`. Once the user is finished, the user can exit the shell to disconnect from the `server`.

3 Processes

This file system supports starting processes on the host machine by running executable files. This is done by calling the `execute` command. The syntax below shows an example of how to do this with an executable file located within the same directory as the file system:

```
host$ gcc -o hello hello.c
host$ ./hello
Hello world!
host$ ./sh
virtual$ execute ./hello
Hello, world!
virtual$ exit
host$
```

Final Notes

Functions written by Drew are documented as such in the comments of `filesys.c`. Mahendra wrote all other functions. Note that the authors worked on the following elements of this assignment:

Mahendra - File system infrastructure, C structures, `mkfs`, `open`, `read`, `write`, `seek`, `close`, `mkdir`, `rmdir`, `ls`, `cd`, `link`, `unlink`, `stat`, `ls`, `cp`, `tree`.

Drew - Documentation in LaTeX, remote shell infrastructure, processes infrastructure, `telnet` compatibility, `cat`, `import`, `export`, `startserv`, `endserv`, `execute`, `exit`.