Programming Assignment 1 Report and Usage Guide

Ellias Palcu and Caleb Williamson

Abstract

This document outlines the implementation and usage of a virtual file system. The virtual file system was designed to be compatible with a set number of commands, and utilizes a combination of RAM and disk space to improve performance while still maintaining a master file containing the entire virtual disk system.

I. Introduction

Programming Assignment 1 provides a list of commands that must be implemented to create a very simple virtual file system. This file system performs the basic functionality of a normal shell, including commands like ls and cat, and also allows for opening, writing, reading, and closing files from this pseudo-shell. A complete list of implemented commands and notes on this specific implementation is provided with this document.

II. Background and Usage

The following table describes the commands that needed to be implemented and notes on this project's specific implementation. Some assumptions were made when implementing these commands. These assumptions include:

- The virtual hard drive does not need to be persistent in this implementation, however it will output a master file that contains all information necessary to recreate the file system with another program. By storing the file information in internal caches, open accesses are more efficient.
- Calling mkfs twice will automatically delete the first instance of the virtual hard drive and create a new, blank file system.
- The only way to remove a file from this virtual file system is to call rmdir on a directory above the file in the tree.
- The shell will not provide a prompt, making it easier to pipe input into the shell and view the output. However, a prompt containing the current path can be shown by uncommenting the cout statements in the print_path function in the utils.cpp utilities file.
- Once a file is read from or written to, the resulting offset value will the move over to next character in contents once the desired amount was read from or written to.
- Once files are closed, their offset it reset to 0 corresponding to the beginning of the file.
- A file cannot be read from if opened with write flag; likewise, a file cannot be written to if open with read flag. User must close file and reopen with correctly specified flag.
- File descriptors will start from zero and continue incrementing; however, if a file closes and its file descriptor value is less than what the global incremented file descriptor value is, it will then receive the smaller file descriptor value. Furthermore, it is dependent on

the order in which the files are closed. If the file descriptor counter is at 3, and file 2 was closed before file 1, the newly opened file will receive file descriptor 2.

Shell Command	**Arguments**	**Description and Notes**
mkfs		Make a new file system, i.e., format the disk so that it is ready for other file system operations.
open	<filename> <flag>	Open a file with the given <flag>, return a file descriptor <fd> associated with this file. <flag>: 1: “r”; 2: “w” The current file offset will be 0 when the file is opened. If a file does not exist, and it is opened for “w”, then it will be created with a size of 0. This command should print an integer as the fd of the file. Example: open foo w shell returns SUCCESS, fd=5
read	<fd> <size>	Read <size> bytes from the file associated with <fd>, from current file offset. The current file offset will move forward <size> bytes after read. Example: read 5 10 shell returns the contents of the file (assuming it has been written)
write	<fd> <string>	Write <string> into file associated with <fd>, from current file offset. The current file offset will move forward the size of the string after write. Here <string> must be formatted as a string. If the end of the file is reached, the size of the file will be increased. Example: write 5 “hello, world”
seek	<fd> <offset>	Move the current file offset associated with <fd> to a new file offset at <offset>. The <offset> means the number of bytes from the beginning of the file. Example: seek 5 10 Notes: Seeking into the middle of a file and then writing to it will overwrite the contents of that file starting at the new offset.

close	<fd>	Close the file associated with <fd>. Example: close 5 Notes: Once a file is closed it is written to the master file containing the contents of the entire virtual hard disk.
mkdir	<dirname>	Create a sub-directory <dirname> under the current directory. Example: mkdir foo Notes: This commands supports making nested directories. For example, mkdir foo/bar will create both foo and bar if they do not already exist.
rmdir	<dirname>	Remove the sub-directory <dirname>. Example: rmdir foo Notes: This command automatically recursively removes the contents of the specified subdirectory.
cd	<dirname>	Change the current directory to <dirname>. Example: cd ../../foo/bar Notes: This command is implemented in such a way that if given a path that partially exists, it will cd into the bottom-most existing directory. For example, given the command cd foo/bar/test where foo/bar exists, but foo/bar/test does not, the command will effectively perform the command cd foo/bar.
ls		Show the content of the current directory. No parameters need to be supported.
cat	<filename>	Show the content of the file. Example: cat foo
tree		List the contents of the current directory in a tree-format. For each file listed, its date and file size should be included. Notes: This command uses the current directory as the root of the tree, meaning it will only print the tree from the current directory down.

import	<srcname> <destname>	Import a file from the host machine file system to the current directory. Example: import /d/foo.txt foo.txt
export	<srcname> <destname>	Export a file from the current directory to the host machine. Example: export foo.txt /d/foo.txt

III. Results and Conclusions

This implementation of Programming Assignment 1 took an approach that would combine the speed of RAM to provide fast performance with the reliability and persistence of the hard drive to allow for outputting and reconstruction of the virtual file system contents. This code is designed to mimic the fundamentals of UNIX file management, and does so by maintaining a tree of directories and files that can easily be sifted through, written to, and read from.

IV. Usage

The file system suite was developed in C++. Further included within the code is commenting to help direct a viewer in understanding the performed operations and functional purposes. The following list, below, explicitly outlines how to compile and run the suite:

1) make
2) ./main (Manual user input)
3) ./main < file.sh (Passing in file of commands)