CS431 Operating Systems
Programming Assignment 7 - File Operations

1 Overview

This programming assignment is intended to give you familiarity and experience with system calls related to file system operations. In this assignment you will complete the implementation of a simple shell, i.e., the interface between the user and the operating system, that reads user-typed commands from the command line and either executes them. Do not worry about running background processes, environment variables, control structures, return values, pipes, redirection, or any advanced features of modern shells. Simply write a reasonable approximation of each command specified in the assignment with appropriate error handling.

2 Requirements

I am giving you the source code to a program called *myshell* that prompts the user for the next command. If the command name entered is valid, i.e., corresponds to an existing command, the program executes the appropriate code internally to the shell then returns to the prompt. That's not to say the argument passed to a valid command is itself valid, so appropriate error checking and handling for each is required. A few of the commands are implemented for you already. Your job is to complete the rest.

NOTE: For this assignment do **NOT** use any of the `exec()` functions or the `system()` system call or similar to execute system programs. That is not the point of this exercise.

The commands you need to implement are:

`ls`	List the contents of a directory
`cat`	Print the contents of a file
`rm`	Remove a file
`pwd`	Print the current working directory
`mkdir`	Create a new directory
`rmdir`	Remove a directory
`stat`	Display file status

3 Commands

The next several pages includes additional pointers and instructions on implementing the requirements of this programming assignment.

ls. This command can either take a directory name as an argument, or if invoked without an argument, the current working directory is implied. The good news is, the code that checks for the latter case is already written. Your code simply needs to open a directory and loop through its contents, printing the name of every entry contained within it. You are not required to include any additional details about each entry, although you are welcome to do so if you wish.

To read a directory, you will need two data structures. The first is a directory stream, i.e., type `DIR*`, that represents the directory structure. This data type is returned by the library function `opendir()`, which takes the directory name as an argument.

Once you have the directory stream opened, your program will need a pointer to a `dirent` struct, declared as such:

```
struct dirent* d;
```

For each entry in the directory you are reading, there is a `dirent`, which can be accessed with the library function `readdir()`. The `readdir()` routine takes as an argument an instance of a `DIR*` and returns a pointer to a `dirent`, i.e., a `dirent*`. See the man page for `readdir()` for more information. The `readdir()` call will need to be in a loop; each call will assign a value to the pointer variable `d`. Then the name of the entry can be accessed through the struct's `d_name` field for printing, e.g., `d->d_name` is a character string that represents the name of a file or directory contained within the directory you are listing.

After all directory entries have been accessed, i.e., you are at the end of the list, the `readdir()` routine will return NULL. That is the condition to check for to terminate the loop. Then simply close the directory stream using `closedir()`.

Note that to use these functions, your program will need to include the header files `sys/types.h` and `dirent.h`. These have already been added to the code provided.

cat. Much of this code you already have written from Programming Assignment 1. The `cat` command takes names of files as input and prints the entire contents of each, in sequential order. For this exercise, your program must output the contents of exactly one file, so if the command accepts a single filename as an argument and handles only that, the requirement will be met. If it handles more than one file, that is optional (see the end of this assignment handout for more details on extra credit).

One thing to consider with this command is that if you get a chunk of a file using `read()`, outputting that chunk using `printf()` will not work correctly. The system call and library function are at different levels of abstraction, and your output will be inconsistent. Instead, pair the call to `read()` with a call to `write()`. Note that Unix convention suggests that standard input has a file descriptor value of 0, standard output has a file descriptor value of 1, and standard error has a file descriptor value of 2.

mkdir and **rmdir**. These commands work exactly as you would think, and there are system calls with the same names that should be used. See `man 2 mkdir` and `man 2 rmdir` for more details. For these system calls, your program will need to include the header files `sys/stat.h`, `sys/types.h` and `unistd.h`. Please note there are certain error conditions that can arise when using these system

calls, so you will want to check for return values and use `errno` appropriately.

One thing to consider with `mkdir` is that a new directory must have execute permissions in order to be able to traverse into it, so be sure to set permissions on the new directory appropriately.

rm. This simply removes a file. You can use the `unlink()` system call, but be sure to check for errors and use `errno` appropriately.

pwd. This outputs the current working directory. This information can be obtained using the system call `getcwd`. See `man 2 getcwd` for more information. Hint: it's already implemented somewhere in the code.

stat. The `stat` command provides basic information about a file, such as type of file, size, number of blocks, modification and access time, etc. Your code should provide some reasonable subset of the information, i.e., at least four items including size, number of links, inode, and name. For this command, use the `stat()` system call, which takes two arguments: first is the filename, and the second a `struct stat` that is passed to the function by reference.

After the call to `stat()`, the file information can be accessed through through the fields of the `stat` struct, e.g., assuming a struct named `stat_buf`, the file size would be `stat_buf.st_size`. I find it helpful to typecast the numeric fields to unsigned integers when using them with `printf()`. Again, be sure to check for errors, e.g., what happens if I `stat` a file that doesn't exist?

Extra Credit. You can earn up to 10 points extra credit on this assignment by either implementing additional commands or adding optional features to the required commands. Extra credit points will be assigned based on a subjective assessment of the amount of work that went into writing optional code. However, please keep in mind it is always best to make sure the required code fully works before attempting extra credit.

4 Requirements, Turn-In, and Due Date

Deliverables. Turn in the C code through WorldClass no later than the date specified in the Weekly Assignments page. **Be sure to include your name in the header block comments in the program code**.

Academic Integrity. I recognize that there are only so many ways to write code that meets the requirements of this assignment, and you may consult with other students in this class on the general approach to a solution. Having said that, sharing code, pseudocode, or other implementation details is not permitted, and I expect all work to be your own as evidenced by:

- Program structure, control flow, variable and function names (aside from C library functions, of course), etc. should be manifestly distinct from others' work.
- Your program should be commented in a manner sufficient for me to ascertain that you understand what the code you submit does.