CIS 415 Operating Systems

Assignment 1 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Ryan Gurnick

Report

Introduction

This project is meant to develop a terminal emulator that runs with the same system calls as a normal terminal interface (bash, sh, zsh, ...) In this project, we use the standard Linux system calls and standard library c calls. The project includes two runtime modes that can be used, the default will run an interactive interface that the user can enter commands right into and see the corresponding output. Additionally, there is a file interface in which a user can provide a text file that contains the same commands and that will be ingested the corresponding output will reside in the output.txt file. In both of these runtime modes the user can issue the same commands and will be able to interact with the operating system.

This project includes the commands as required by the requirements and there will be more detail following this paragraph. These commands are essentially just emulators for the standard commands provided by almost every Linux distribution. These commands allow for basic file manipulation and generation for a user, the code will compile on the Mint Linux VM that was set up for the University of Oregon CIS 415 class during Spring term 2020.

In this project, there are the following commands that the user can issue:

1. `ls` - This command lists the directory that the user is currently in, this command utilizes Linux system calls to get the current working directory `[getcwd()]`, the open directory system call `[opendir()]`, then will read the directory `[readdir()]`, finally we will close the directory when complete `[closedir()]`. This command does not take any parameters.
2. `pwd` - This command will print the current directory that the user is in (otherwise called the current working directory/CWD). It utilizes just the Linux system call to get that CWD `[getcwd()]`. This command does not take any parameters.
3. `mkdir` - This command will generate a new directory given a directory name that does not exist within the CWD. This function will utilize `[getcwd(), opendir(), readdir(), closedir()]` much like the `ls` command. Additionally, we utilize the mkdir `[mkdir()]` system call as well to request the OS to generate the folder.
4. `cd` - Naturally we need some way to change the directory that we are currently in, this pretty much only utilizes one system call to change the directory (how convenient) `[chdir()]`. This function requires a folder to change directory into.
5. `cp` - This command provides a way for users to copy one file to another file given the src and dist locations. This command utilizes a bunch of system calls so listing them will be best: `[getcwd(), opendir(), readdir(), closedir(), open(), read(), write()]`.
6. `mv` - For your file transportation needs, if a user desires to move a file this command can do it, given the src and dist locations. It utilizes `[getcwd(), opendir(), readdir(), closedir(), open(), read(), write()]` and then additionally calls the `deleteFile()` function from within `command.c`
7. `rm` - The command will aid in the removal of a file. This function utilizes `[getcwd(), opendir(), readdir(), closedir(), open(), read(), write(), unlink()]` to provide the desired functionality.
8. `cat` - This command mirror the cat command on a normal Linux machine, this aims to show the content of files to the user. This function utilizes `[open(), read(), write()]` to provide the desired functionality.

Background

In this project, there are some design decisions made in the main.c file to allow for ease of readability. Anyone looking at the code will notice that there are additional functions in the main.c file, one to handle interactive mode and another to handle the file mode. In order to ensure that the code in the main file is as DRY as possible, there is also an execution [execute()] function to handle the execution of commands after the interactive() or file() functions parse the input from the user. Then the execute function will process the parsed data and attempt to execute the commands based on the user input method (interactive/file).

When looking at command.c we will notice that much of the DRY mentality cannot be adjusted since we are not allowed to add additional functions. Keeping that in mind there are similar code blocks utilized for various functions and the code will be virtually identical to its counterpart in other functions. One example of this is the routine for getting the current working directory, this is used within the following functions: [`listDir()`, `showCurrentDir()`, `makeDir()`, `copyFile()`, `moveFile()`, `deleteFile()`]. I will discuss this more in the implementation area as an example, just understand that the code is intentionally similar between function blocks.

Implementation

For grading:

Please do not run the application in debug mode for grading purposes. There is a huge amount of output that is there to help with debugging and will generally confuse.

Please ensure that you change the preprocessor config in the main.c file if you would like to enable the debug mode.

```
#define true (1==1)
#define false (1==0)
#define debug false
#define interactive_input stdin
#define interactive_output stdout
#define file_output "output.txt"
```

Now onto actual algorithms...

From the following main function in the main.c file you can see how we switch between the interative and file mode. This is just handled by processing the inputs (ie: argv and argc) and then changing the functionality based on these inputs. Then it will hand off the control over to the specific function that handles the input mode selected and inputted.

```
// main()
// Args: * argc - argument count as provided by standard library
//       * argv - string arguments that have been passed
// Purpose: bootstrap the application
// Design Rationale: this function is completely required by c standard library
int main(int argc, char *argv[])
{
  if(argc == 1)
  {
    // no flags thrown
    // while loop the data
    interactive();
    return 1;
  }
  else if (argc >= 1)
  {
    // flags thrown
    if (strcmp(argv[1], "-f") == 0)
    {
      return file(argv[2]);
    }
  }
}
```

Next, we will take a brief look at the file and interactive function which are both pretty similar, the following code snippets and a combination of the C programming language and pseudo-code. (Please do not mistake it for submission code)

```
// interactive()
// Purpose: this function will run the interactive mode for the application
// Design Rationale: ensure that all functionality for interactive mode is tightly coupled
int interactive()
{
  declare the input string and length

  set the buffer for stdin to null

  loop:
     prompt the user for input and store it to the input string declared previously

     check for special "exit" command

     execute(input)

  return
}

// file()
// Args: * file - a string reference to the file that can be used and bassed.
// Purpose: this function will run the file mode for the application
// Design Rationale: ensure that all functionality for file mode is tightly coupled
int file(char *file)
{
  open the file and check for errors

  using freopen() open and stream all output to the output file

  check for errors in opening the output file

  loop through each line:
     execute(line)

  free the allocated line var
  close the output file
  close the input file
  return
}
```

From here you can see that the two specific functions handle the various facets of parsing the inputs for the two different input methods. Then they continue the execution by forwarding the command inputs to the execute function for continuing processing. The file and interactive functions also set up the proper output for the given input method (ie: file mode outputs to the output file, interactive mode outputs to stdout). Moving on to the star of the show the execute function.

```
int execute(char *line)
{
  declare some counters
  determine the number of tokens needed.
  declare the arrays that we will use to tokenize.
  get an array of tokens for the given line

  loop
    if the slot in array is nonempty
    {
      tokenize the line further and then store to the final array
    }
  }

  loop through the rows of the final array
    if the command token is missing
    {
      warn the user about unrecognized command
    }
    loop through the columns of the final array
      remove any empty records
      {
        ensure that we are dealing with the cmd slot in the final array
        {
          if "ls"
          {
            check for unsupported parameters
            listDir();
          }
          else if "pwd"
          {
            check for unsupported parameters
            showCurrentDir();
          }
          else if "mkdir"
          {
            if directoryName provided
            {
              makeDir(directoryName);
            } else {
              missing directoryName
            }
          }
          else if "cd"
          {
            if directoryName provided
            {
              changeDir(directoryName);
            } else {
              missing directoryName
             }
          }
          else if "cp"
          {
              if src && dist provided
              {
                copyFile(src, dist);
              } else {
                missing src and or dist
              }
          }
          else if "mv"
          {
            if src && dist provided
            {
                moveFile(final[a][1], final[a][2]);
            } else {
                missing src and or dist
            }
          }
          else if "rm"
          {
            if fileName
            {
              deleteFile(fileName);
            } else {
              missing fileName
            }
          }
          else if "cat"
          {
            if fileName
            {
              displayFile(fileName);
            } else {
              missing fileName
            }
          }
          else
          {
            warn the user about an unrecognized command
          }
        }
      }
    }
  }
  return 0;
}
```

From here all of the code ends up being executed by the standard library and passed along to the relatively simple code in the command.c file. The command.c file is modeled after the command.h headers provided, otherwise, the code would be broken into more functions to make the code more DRY. I have provided comments in the code for those functions and do not think it's completely needed to explain those algorithms more in-depth as I assume that you understand the way it should be already. There might be some minor bugs in the execution order of some of the commands however in testing I was unable to find any while giving many edge cases. Mainly there are lots of edge cases on the execute function that we previously discussed and it can sometimes break in a fun way. In the future this if statement block would be switched out for a table-driven design, however, I did not have time to do this.

Performance Results and Discussion

```
==5283== Memcheck, a memory error detector
==5283== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5283== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5283== Command: ./a.out -f input.txt
==5283==
==5283==
==5283== HEAP SUMMARY:
==5283==     in use at exit: 0 bytes in 0 blocks
==5283==   total heap usage: 11 allocs, 11 frees, 267,296 bytes allocated
==5283==
==5283== All heap blocks were freed -- no leaks are possible
==5283==
==5283== For counts of detected and suppressed errors, rerun with: -v
==5283== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Overall with his project, I was not aiming to be the most efficient or optimized with my algorithms, mainly was aiming to provide a "prototype" of this system based on the requirements. The application is nominally responsive and provides errors as requested by the requirements. Additionally, when running the test input.txt for the project it consistently provides the following output:

```
ryan@clonedrive-cis415:~$ time ./a.out -f input.txt
real    0m0.001s
user    0m0.000s
sys 0m0.001s
```

In general, there were some minor issues I had with over-allocation of space if the user decides to type in "ls ls ls ls ls ls ; asdlkjasd", the interpreter will over-allocate memory (not too much though) and will provide additional errors about unsupported commands. Now, this seems to work fine since the commands entered would need to be highly misformed but you may see the same error multiple times if you create a hugely malformed command.

The application is written to have positive input validation, so any incorrect input will generate an error of some kind, which should help aid the user in correcting the commands. Sometimes this goes a bit far since I mainly work in cybersecurity however all the functionality requested exists.

Conclusion

To conclude there may be some minor hiccups in this terminal interface however for the most part it executes all the desired commands without leaking any memory or generating compiler or runtime errors (such as segfaults). There are places where you may get extra errors from the parser saying command not recognized even when one might not have been provided and this is partly because I was unsure when you wanted that to show. In the examples provided it appears that if there is a semi-colon after a valid command you want to interpret the space after it as another token. This causes the terminal in my case to think there are multiple unrecognized commands. This should be the only glitch that you might run into. After running Valgrind (as shown once above) there are no memory leaks or errors and this has been tested against many edge cases.