415 Project 1 Report
Christopher Wycoff (951078512)

Introduction:

Project 1 requires the creation of a program that acts similarly to the BASH shell environment. This program is called **pseudo-shell**. The commands the pseudo-shell will support are modified forms of existing bash commands:

ls, mkdir, pwd, cd, cp, mv, rm, cat

The program source code can be divided into the following pieces pieces:

main.c:

- Main logic:

The area of the source that reads stdin from user and acts accordingly

- File I/O logic:

Also located in main.c is logic similar to the previous section handles when the user provides the -f flag and a file to be read from for commands, e.g. ‘input.txt’

command.c

- This file contain the command logic to be run when a user executes a specific command/s
- **command.c functions must be implemented using system calls**

Background:

System calls are the interface that a user program has to call for the operating system kernel to do work for it. In linux (and UNIX) based OSs system calls are often called with a wrapper function from the library glibc and from other libraries. These wrapper functions are what will be used in the command.c functions that contain much of the logic powering pseudo-shell program.

The standard c libraries such as stdlib.h stdio.h and string.h enable the main.c portion of the pseudo-shell program to tokenize user commands and feed the commands and parameters into the command.c functions. For further details on this interaction see Implementation section.

Implementation:

When the pseudo-shell program is run the program checks for the -f flag and for an input file parameter. If file mode is enabled then a file ‘output.txt’ is opened and STDOUT is rerouted to ‘output.txt’ using the dup2(2) function.

```
dup2(fd_1, 1);
```

Following this the majority of the logic is the same for either file mode or user mode.
The user input is tokenized by the space character and parsed into an array.
The program uses a switch statement and a command identifier helper function to determine what command (if any) has been entered by the user.

```c
typedef struct{
    char *key;
    int value;
}legal_commands;


legal_commands command_lookup[] = {
    {";", SEMICOLON}, { "ls", LS }, { "pwd", PWD }, { "rm", RM },
    { "cat", CAT }, { "mkdir", MKDIR }, { "cd", CD },
    { "cp", CP }, { "mv", MV }
};

int number_of_keys = 9;

int get_value_from_string_key(char *a_key)
{
    if (a_key == NULL){
        return A_NULL;
    }
    for (int i=0; i < number_of_keys; i++) {
        legal_commands *sym = &command_lookup[i];
        if (strcmp(sym->key, a_key) == 0){
            return sym->value;
        }
    }
    // if gets to here there was no return thus return -1

    return INVALID;
}
```

The program then calls one of the command.c functions using the parameters parsed in main.c For example the cat command:

```c
/*for the cat command*/
void displayFile(char *filename){
    char the_buffer;

    int char_int;

    int source_file = open(filename, O_RDONLY);
    if (source_file == -1){
        printf("file not found\n");
        return;
        }

    int stop_counter = 0;
    while((char_int = read(source_file, &the_buffer, 1) != 0)){
        write(1,&the_buffer,1);
        stop_counter += 1;
    }
    close(source_file);

}
```

Upon the user input of the sting ‘exit’ the program terminates

Performance Results:
It is my belief that pseudo-shell performs all the tasks as expected.
For example the commands from ‘input.txt’, given with the project description, work as expected.

I had some issues making the mv and cp commands behave appropriately when given either explicit destination path or implicit one. E.g “cp ../file_to_be_copied.txt ./copy_of_file.txt” Vs. “cp file_to_be_copied .”

When no file name is given in the destination path cp and mv use the previous name of the file in source path to name the copied/moved file.

There may exist some permutation of commands that may cause unexpected behavior, but as of now I have not found them.

If a command is typed incorrectly or certain parameters are not able to be used:

- Either the program will terminate
- Or the program will issue a warning and continue execution

Conclusions:

I certainly gained perspective on what system calls are. I feel more comfortable envisioning the interface between applications and the operating system. I have a better understanding of the file and directory structure from the perspective of the OS. Additionally I have a greater appreciation for how difficult it may design applications that are flexible to multiple inputs to accomplish the same goal.