Programming Assignment 2 Report

Names: Andrew Ortiz and Mitchell Curtis

Problem Description

For this programming assignment we are creating a simple UNIX shell utilizing UNIX system calls such as, fork, exec, and wait. The program will take in commands from the user and executes each command in its own process.

Program Design

This program didn’t have the need for any data structures as it is just a simple shell and didn’t call for the need to organize or retrieve data in any fashion. Our program does have some simple algorithms that were utilized in dealing with the inputted commands from the user. The shell runs in a simple while loop that doesn’t exit and terminates the program until the user enters the exit command, in which upon doing so will set the should_run variable to 0 which will terminate the loop and return main. This is a viable option, the program is supposed simply as a shell and shouldn’t close at any point during runtime unless specified by the user, so the program will continuously run and wait for the user to input a command to then determine if the command is valid and proceed based on that.

System Implementation

As stated before, the shell runs in a simple while loop that doesn’t exit and terminates the program until the user enters the exit command. Until then, the while loop will prompt users for commands and determine whether or not the command is valid, if valid the command is sent to the parse_command function. The parse_command function is one of the more important features, as it is how the user's commands will be sent to the args array to be executed in an orderly fashion. The parse_command function utilizes our own parsing process instead of using strtok because we wanted to set white spaces, tabs, and \n to null instead of just the white spaces. The function will still split the user input into tokens based on the white space imputed and then we dropped the \n on the last argument imputed from the user pressing enter, as this was the first problem we ran into. For the history function we simply created another char array to store the most recent command into and then put an if statement to check if the user

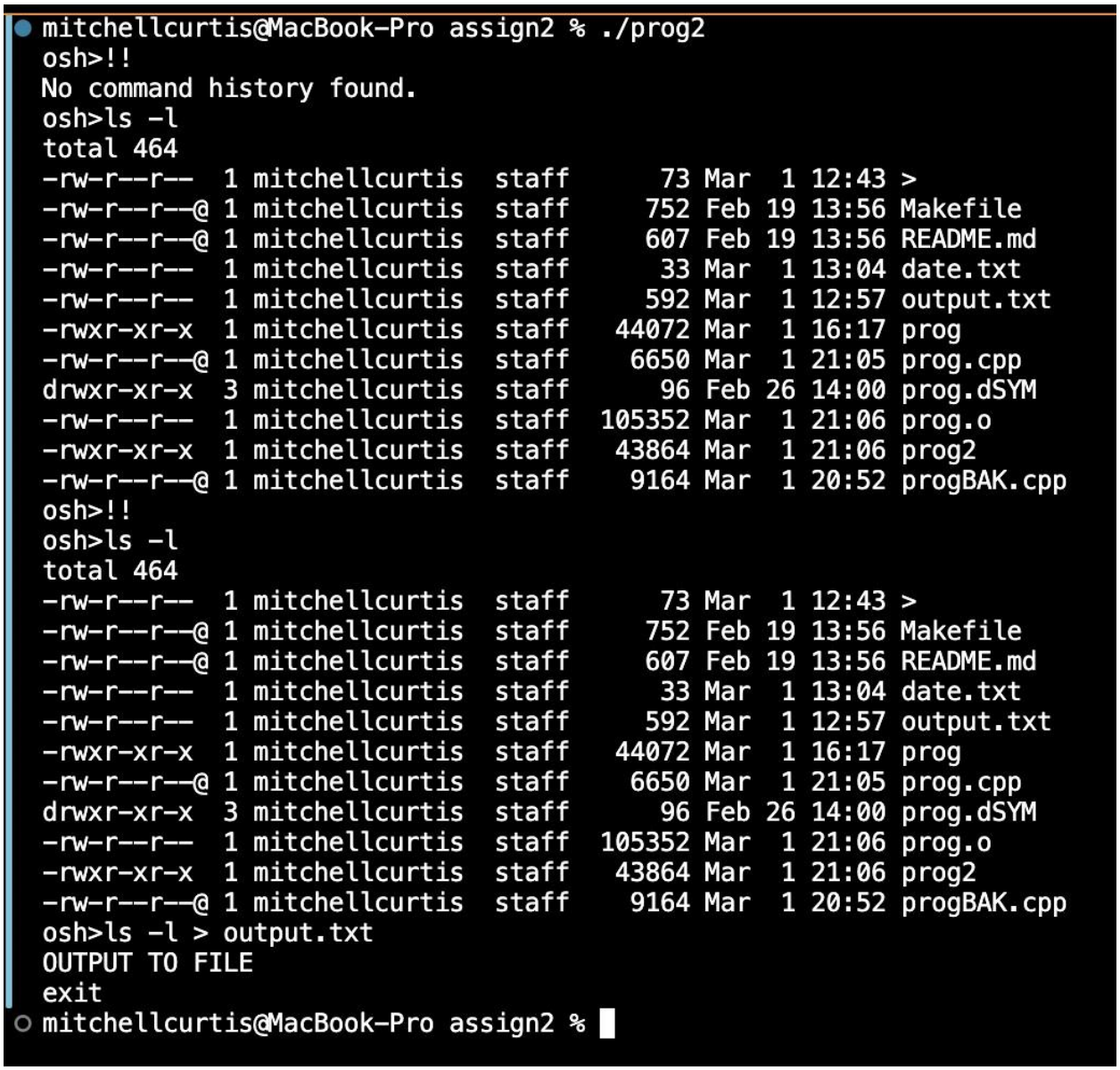
imputed !!. If the user imputed !! and there was no recent command the program would let the user know that there is no command to run and prompt for another input, if there was a recent command the char array would already have the last command stored be passed to the parse_command function and then be stored into args and executed again. We handled input output redirects the same way just checking the tokens for ‘<’ or ‘>’ and then handled them with dup2 accordingly. We couldn't properly handle the user entering an & at the end of the command without doing so, as now that the \n was removed from the last argument we could manually get the & and store it into a boolean and then check the boolean to see if the parent process needed to wait or not. Another issue we ran into was forgetting to implement the exit command for the user. We forgot that this was needed to terminate the program properly instead of just killing the shell program using control + c and we couldn’t properly test the program on gradescope without doing so, this halted the progress drastically as we couldn’t test on gradescope properly.

```c
/**
 * @brief parse out the command and arguments from the input command separated by spaces
 *
 * @param command
 * @param args
 * @return int
 */
int parse_command(char command[], char *args[])
{
    int numCommands = -1;   //account for initial loop

    //remove newline character at the end of the command (ls -l fix)
    size_t len = strlen(command);
    if (len > 0 && command[len - 1] == '\n') {
        command[len - 1] = '\0';
    }

    while (*command != '\0')    //loop unitl end of command
    {
        while (*command == ' ' || *command == '\t' || *command == '\n') //replace space/tab/nl with null
        {
            *command++ = '\0'; //make white space null
        }
        *args++ = command; //save the argument position
        numCommands++;
        while (*command != '\0' && *command != ' ' && *command != '\t' && *command != '\n') //loop until end of argument
        {
            command++; // skip the argument until
        }
    }
    return numCommands; //return the number of arguments
}
```

```
int main(int argc, char *argv[])
{
    char command[MAX_LINE];       // the command that was entered
    char *args[MAX_LINE / 2 + 1]; // hold parsed out command line arguments
    int should_run = 1;           /* flag to determine when to exit program */
    char commandHistory[MAX_LINE];  //save previous command for !! utilization
    int num_args;   //number of arguments in command
    bool ampersand, pipeFound = false; //check for ampersand symbol
    char *argsPipeOne[MAX_LINE / 2 + 1]; // hold parsed out command line arguments
    char *argsPipeTwo[MAX_LINE / 2 + 1]; // hold parsed out command line arguments

    while (should_run)
    {
        printf("osh>");
        fflush(stdout);
        // Read the input command
        fgets(command, MAX_LINE, stdin);

        //***DEBUG***
        //strcpy(command, "ls -l | grep \.txt\n");
        //cout << command << endl;

        //check for history command
        if (strcmp(command, "!!\n") == 0) {
            if (commandHistory[0] == '\0') {   //no history
                cout << "No command history found." << endl;
            }
            else { //print history
                //parse history
                num_args = parse_command(commandHistory, args);

                cout << "osh>";
                for (int i = 0; i <= num_args; i++) {
                    cout << args[i] << " ";
                }
                cout << endl;
            }
        }
        else if (strcmp(command, "exit\n") == 0)
            return 0;
        else { //normal command
            strcpy(commandHistory, command);    //save command for history
            num_args = parse_command(command, args);
```

Above is the algorithm implemented to parse the commands the way we wanted.

Results



Conclusion

In conclusion our implementation of the Unix shell successfully implemented all of the required features listed by the project description. The lessons we learned from the program was honestly just growth in understanding of how the fork system call works. We also got to practice utilizing and manipulating c strings.