Bryan Lee ID: 922649673
Github: BryanL43 CSC415 Operating Systems

Assignment 3 – Simple Shell with Pipes

Description:
The program is a simple shell that runs on top of the regular command-line interpreter for Linux and supports piping. It takes the user's input, parses it into commands and arguments, and executes them within the shell.

Approach:
My program design starts with handling the custom prompt. If there is no first argument in the argv vector, I will use the default prompt " >." The next step is instantiating a buffer with 163 bytes to store the input. The next crucial step is a large loop that will do the following:

- It repeatedly prompts the user until the exit command is input.
- During each loop iteration, the program will:
    - Receives input
    - Parses the input
    - Executes the command with these two cases:
        - Case 1: no pipes
        - Case 2: with pipes

The first step of the loop is to receive the input with the “fgets” function, which offers flexibility in checking for the “end of file” error with the “feof” function and truncation of the string to fit inside the buffer. I must also check for an empty input and handle the “exit” command.

The second step of the loop is to create two separate vectors, one for storing the entire command, i.e., {“ls -l -a”, “ps”}, and the other for storing the broken down arguments, i.e., {“ls”, “-l”, “-a”, NULL}. To do so, I must parse the input using strtok with the space delimiter and “|” to separate and store the arguments. Do note that I will add a NULL pointer at the end of the argument vector for the NULL termination of execvp.

The second step worked initially but could have been more efficient at handling commands without pipes (more details in the issue section). My new design choice was to use a 2D array. I will still use the same parsing logic with the delimiter “|” and space. Instead of using the function strtok, I have to use strtok_r to handle a nested loop. To clarify the logic better, I will use the following pseudo-code:

```
char* 2darray;
char* token = strtok_r with delimiter "|"
char* subtoken;

while (token is not empty) {
        subtoken = strtok_r with space delimiter
        while (subtoken is not empty) {
                add subtoken (the arguments) to the array
```

```
        }
        Add a null terminator to the end of the vector.
}
```

The string parsing should yield the necessary broken-down arguments for the execvp parameters. An example of the arguments in the 2d array:

```
char* cmds[][] = {
        {“cat”, “Makefile”, NULL},
        {“wc”, “-l”, “-w”, NULL}
}
```

or

```
char* cmds[][] = {
        {“ls”, “-l”, “-a”, NULL}
}
```

The third step is to handle the execution of commands without pipes. Due to the exec command's self-destructive nature (execvp is a variant of it), it is necessary to use the fork function to create a child process that will execute the command safely. However, using fork requires me to handle the following instances:

- Error (fork returns -1)
    - display error message and exit with failure
- In the child process (fork returns 0)
    - Call the execvp function with its appropriate parameters:
        - Pass the specific command name, such as “ls,” into the first parameter of execvp
            - The command name will come from the first array index inside the 2d array, i.e., cmds[0][0] or cmds[1][0].
- In the parent process (fork returns the child’s PID)
    - the parent will wait for the child process to die by using the “waitpid” function
    - display the child’s PID and exit status

The fourth step is to handle the second case of commands where there are pipes. I must create multiple pipes (number of commands - 1 amount of pipe(s)) and a pipe file descriptor. An essential aspect of the piping is having a variable that saves the read end of the file descriptor. Saving the read end of the file descriptor allows me to pass the previous child process’s output into the following child process. Within the fork function, I will handle the three instances as follows:

Note: The fork will also duplicate the pipe file descriptor; therefore, it is necessary to close the file descriptor in both the child and the parent processes.

- Error (fork returns -1)
    - display error message and exit with failure

- In the child process (fork returns 0)
    - In the 2nd pipe (if there is one):
        - Copy the previous child's output to the current pipe's read end, which will merge two commands dependent on each other
    - If there is another command in the pipeline after the current one:
        - Redirect the pipe's write end to the child process's output
    - Close both ends of the child process's pipe
    - Call execvp as described in the case without pipes
- In the parent process (fork returns the child's PID)
    - Close the ends of the pipe that are no longer necessary
    - Update the saved read end to the current pipe's read end, ensuring that the following child process will receive the correct information
    - the parent will wait for the child process to die by using the "waitpid" function
    - display the child's PID and exit status

The last step is simply to free the buffer.

Issues and Resolutions:
My initial issue involved setting up pipes. I needed clarification on the number of pipes required when handling multiple commands in a pipeline, for example, "cat Makefile | grep bryan | wc -l -w." I initially created a new pipe for every argument (shown in the image), which created an unnecessary pipe.

```
//Iterate through each command to parse it into individual commands and arguments.
//After parsing, fork a child process for each command and execute them.
for (int i = 0; i < cmdsCount; i++) {

    //Creates a pipe with pipe file descriptor and checks for errors
    if (pipe(pipefd) < 0) {
        perror("Failed to create a pipe!\n");
        exit(EXIT_FAILURE);
    }
```

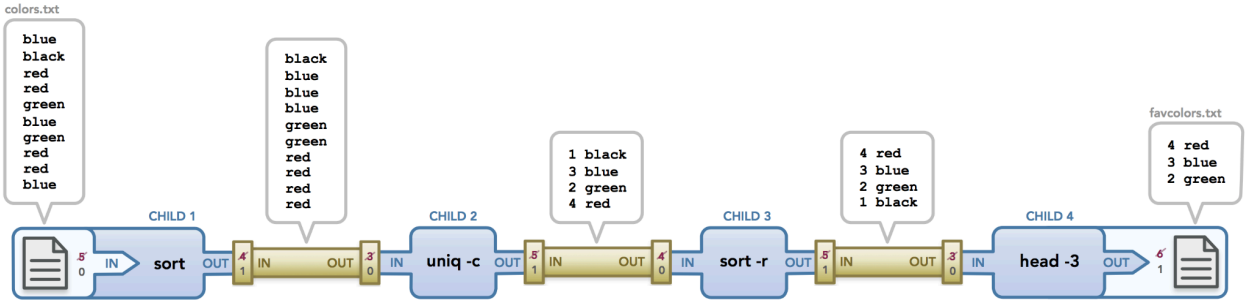
I resolved this issue after office hours by limiting the number of instantiated pipes to the number of commands minus one. The program ensures that only the necessary amount of pipes correspond to the number of "|" in the inputted command.

The revised version of the pipe code:

```
for (int i = 0; i < cmdsCount; i++) {
    //Creates a pipe with pipe file descriptor and checks for errors.
    //Do not create a pipe for last command
    if (i < cmdsCount - 1) {
        if (pipe(pipefd) < 0) {
            perror("Failed to create a pipe!\n");
            exit(EXIT_FAILURE);
        }
    }
```

The next issue involves closing the pipes and correctly managing the file descriptors. Initially, I believed my logic was correct and the shell properly handled piping. However, I assumed that there is only one pipe per iteration and did not consider the duplication event that occurs when forking. With that misconception, I closed or write to the pipe more coherently.

I resolved this by using #define to organize my file descriptors properly and visualized the process using the following image.



Source: https://www.rozmichelle.com/pipes-forks-dups/

Another critical step in resolving this issue was saving the read-end file descriptor for the following child process.

```
oldFileDesc = pipefd[READ_END];
```

I was only able to figure this out after watching an informative video about dup2 to visualize and understand its functionality:
https://www.youtube.com/watch?v=EqndHT606Tw&ab_channel=holidaylvr

The last issue was how I stored the parsed commands, especially with piping. I initially created two separated arrays to store the entire commands, i.e.:

Pseudocode to demonstrate my issue:
char* cmds[(BUFFER_SIZE / 2) + 1] = {"cat Makefile", "wc -l -w"};

and

char* args[(BUFFER_SIZE / 2) + 1] = {“ls”, “-l”, “-a”};

The bracketed red in the image shows the conflicted issue I had:

```
//Iterate through each command to parse it into individual commands and arguments.
//After parsing, fork a child process for each command and execute them.
for (int i = 0; i < cmdsCount; i++) {

    if (pipe(pipefd) == -1) {
        printf("Failed to create pipe\n");
        break;
    //Creates a pipe with pipe file descriptor and checks for errors
    if (pipe(pipefd) < 0) {
        printf("Failed to create a pipe!\n");
        exit(EXIT_FAILURE);
    }

    //Note: The first index of the vector will be the command,
    //the rest will be arguments.
    char* argArray[82];
    int argArrayIndex = 0;

    //Split and tokenize the individual parts of the commands
    char* token2 = strtok(cmds[i], " ");

    //Iterate through the token and store the seperated components in a vector
    while (token2 != NULL) {
        argArray[argArrayIndex++] = token2;
        token2 = strtok(NULL, " ");
    }
```

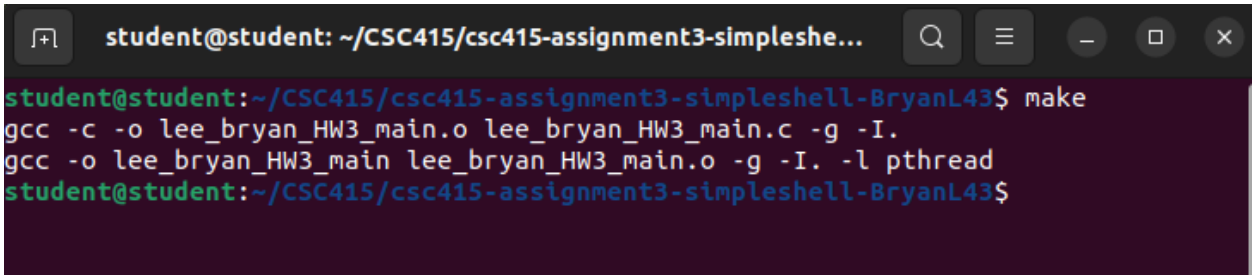
The separated arrays created an issue where I handled the commands with and without pipes under a single operation. The poor design choice created unnecessary steps that could generate a silent logic error when parsing the commands. I also couldn’t move the tokenization of arguments before the for loop due to its dependency on the number of commands in the pipe.

I resolved this by switching to a 2D array, which was more straightforward to handle and allowed me to create separate operations for both command cases.

```
char* cmds[(BUFFER_SIZE / 2) + 1][(BUFFER_SIZE / 2) + 1];
```

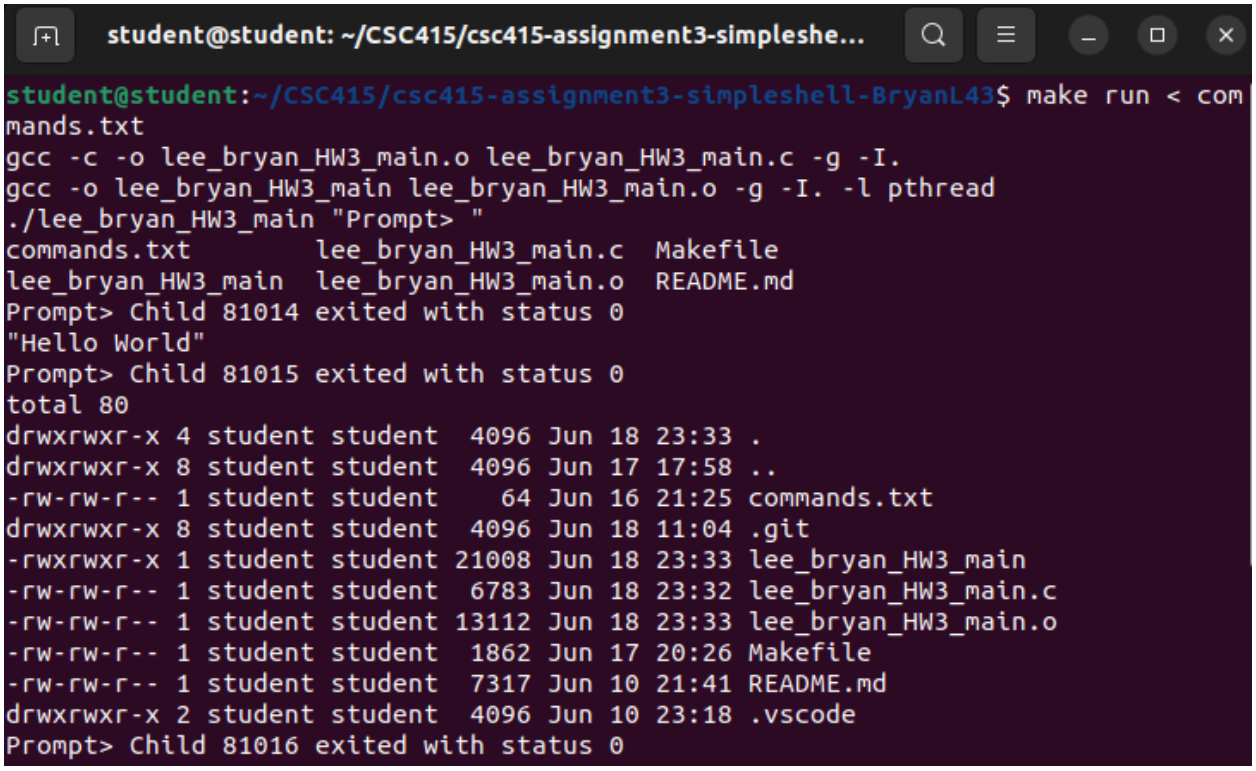
Analysis: n/a

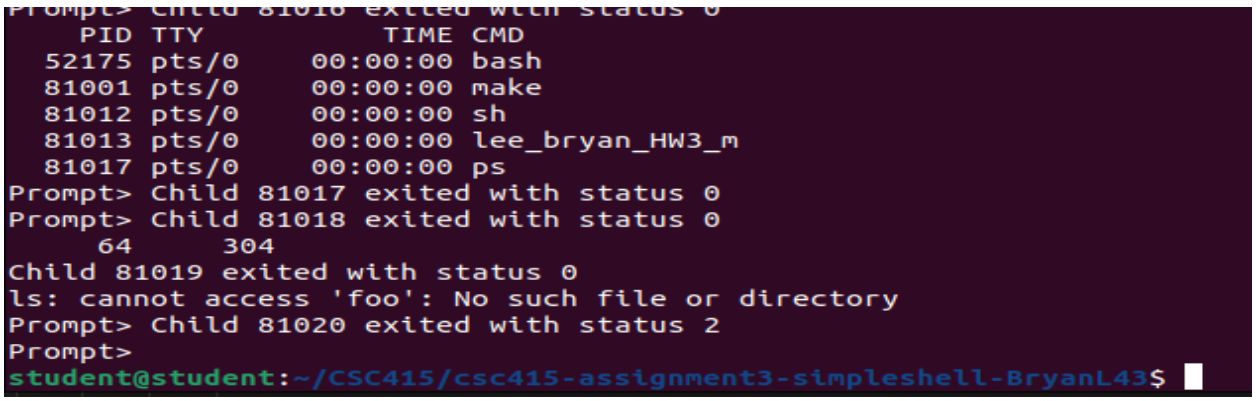
Screenshot of compilation:



Screenshot(s) of the execution of the program:

Compilation with commands.txt:





Compilation with manually inputted commands: