

**Name:** Danny Pham

Course Number: CSC3350

Assignment Title: wash Shell Extra Credit Project

**Professor:** Andy Cameron

**Date:** June 9<sup>th</sup> , 2024

**ATTESTATION:** The code submitted to this assignment is all my original code and a result of my individual work. I did not get any code from any website, book, source, AI tool, or person. If I did get any of the submitted code from somewhere or someone else, I have cited it in a “References” page (or a block comment), and I understand that using it will invoke a penalty.

Signed – *Danny Pham*

**Abstract:** The wash Shell – “Washington Shell”- project presents a customized command-line shell. The shell was designed for use on Unix-like operating systems. This covers basic functionalities of traditional shells, especially bash and dash, with C to be the constructive language. The shell allows users to execute commands, manage directories, and manipulate the environment. The project includes implementing built-in commands (exit, echo, pwd, cd, setpath, help), external command execution with path management, and output redirection capabilities. Furthermore, a custom utility command “head\_new” is developed to display the first N lines of a file or standard input.

In general, the project aims to give a deeper understanding of operating systems and command-line interfaces by utilizing low-level system calls and process control mechanisms. With the emphasis on error handling, efficient memory management, and adherence to coding standards and documentation practices.

## Introduction

The “Washington Shell” project is to create a custom commandline shell, *wash*, specialized to run on Unix-like operating systems. This includes Linux-based distributions, macOS, and Windows WSL2 (Windows Subsystem for Linux). The ultimate goal of this project is to reproduce fundamental functionalities of traditional shells like *bash* and *dash* in order to provide users with a functional command-line interface.

### *Core Features and Functionalities:*

#### Built-in Commands:

- *exit*: Terminates the *wash* shell session.
- *echo*: Prints the given arguments to the console.
- *pwd*: Displays the current working directory.
- *cd*: Changes the current working directory to the assigned path or user’s home directory if no path is provided.
- *setpath*: Sets the *PATH* environment variable to the directories specified by users.
- *help*: Lists all built-in commands with functional descriptions.

External Commands Execution: The shell executes external commands by searching the specified directories in the *PATH*. *wash* can handle at least four external commands such as: *cat*, *ls*, *date*, and *man*.

Redirection: Using the syntax ‘command > filename’, *wash* implements the standard output to redirect the command to ‘<filename>.output’ and standard error to ‘<filename>.error’.

Custom Utility Command – *head\_new*: is a newly written command which emulates the functionality of the standard ‘*head*’ command but in the simplified options. The command displays the first *N* lines of a file or user’s standard input with the set default display of 7 lines if ‘-n’ option is not specified.

## File Submissions:

*wash.c* – The shell program.

*head\_new.c* – The utility command program.

*file.txt* – test text file for using *head\_new* command

*list\_of\_test\_commands.txt* – list of commands for test run

*wash* – executable file for *wash* shell

*head\_new* – executable for *head\_new* command

*README* – Usage instructions and help content.

Project Document – Detail report of the project structure.

## **How to Compile, Build, and Run:**

To compile and build the wash shell:

```
gcc -o wash wash.c
```

```
gcc -o head_new head_new.c
```

To run the wash shell and more information:

```
./wash
```

```
./wash help
```

Or inside *wash*, you could run:

```
wash:3 gcc -o head_new head_new.c
```

```
wash:3 ./head_new -h
```

Alert: To enable external commands, you must set the PATH environment to either */bin* or/and */usr/bin*. To run *./head\_new* you must place *head\_new.c* within the set PATH and compile *head\_new* within that path.

For example, WSL2 locks user from manually add customized command or executable files to *bin* or */usr/bin*, and *wash* cannot automatically set the PATH environment once *pwd* is changed. The default PATH environment is */bin*.

```

phamd4@DESKTOP-V7D64E0:/mnt/d/shell/wash$ gcc -o wash wash.c
phamd4@DESKTOP-V7D64E0:/mnt/d/shell/wash$ ./wash
wash:3 setpath /mnt/d/shell/wash
New PATH: /mnt/d/shell/wash
wash:3 gcc -o head_new head_new.c
gcc: command not found
wash:3 setpath /bin
New PATH: /bin
wash:3 gcc -o head_new head_new.c
wash:3 setpath /mnt/d/shell/wash
New PATH: /mnt/d/shell/wash
wash:3 ./head_new -h
    -h          Print this help message.
    -n N        Print the first N lines (default is 7).
    file.txt    The file to read from. If not specified, reads from standard input.
wash:3

```

We can see that, if the PATH environment wasn't in `/bin` we cannot use `gcc` command to compile the `head_new.c`, also we must be in the directory where `head_new` executable file is output to.

Built-in Commands:

- `exit`: Ends the shell process.

```

wash:3 exit

```

```

phamd4@DESKTOP-V7D64E0:/mnt/d/shell/wash$

```

- `echo`: Prints the arguments given to the console.

```

phamd4@DESKTOP-V7D64E0:/mnt/d/shell/wash$ ./wash

```

```

wash:3 echo Hi User, I'm wash

```

```

Hi User, I'm wash

```

```

wash:3

```

- `pwd`: Prints the current working directory.

```

wash:3 pwd

```

```

/mnt/d/shell/wash

```

```

wash:3

```

- `cd`: Changes the current working directory.

```
phamd4@DESKTOP-V7D64E0:/mnt/d/shell/wash$ ./wash
wash:3 cd ..
wash:3 pwd
/mnt/d/shell
wash:3 cd wash
wash:3 pwd
/mnt/d/shell/wash
wash:3 cd
wash:3 pwd
/home/phamd4
wash:3 cd /
wash:3 pwd
/
wash:3
```

- setpath: Sets the path for executable programs.

```
wash:3 setpath /bin
New PATH: /bin
wash:3 setpath /bin /usr/bin
New PATH: /bin /usr/bin
wash:3 setpath /bin /usr/bin /mnt/d/shell/wash
New PATH: /bin /usr/bin /mnt/d/shell/wash
wash:3 ./head_new
Enter lines of text. Press CTRL+D to end input and display the first 7 lines:
1
2
3
4
5
6
7
Your input was:
1
2
3
4
5
6
7
wash:3 ls
bin  dev  home  lib  lib64  lost+found  mnt  proc  run  snap  sys  usr
```

- help: Lists all built-in commands with descriptions

```

wash:3 -h
wash - My own shell
Built-in command lines:
  exit - Exit the shell process.
  echo - Print arguments from console.
  pwd - Print the current directory.
  cd [dir]- Change the current directory.
  setpath <dir> [dir] ... [dir] - Set the path for executable programs.
  help - List all built-in commands.
wash:3

```

External Commands:

- man: allows users to access detailed information about various commands, utilities, and system calls. Enter q to quit.

```

wash:3 man man
wash:3

```

```

MAN(1)                                Manual pager utils                                MAN(1)

NAME
  man - an interface to the system reference manuals

SYNOPSIS
  man [man options] [[section] page ...] ...
  man -k [apropos options] regexp ...
  man -K [man options] [section] term ...
  man -f [whatis options] page ...
  man -l [man options] file ...
  man -w|-W [man options] page ...

DESCRIPTION
  man is the system's manual pager. Each page argument given to man is normally the name of a program, utility or function. The manual page associated with each of these arguments is then found and displayed. A section, if provided, will direct man to look only in that section of the manual. The default action is to search in all of the available sections following a pre-defined order (see DEFAULTS), and to show only the first page found, even if page exists in several sections.

  The table below shows the section numbers of the manual followed by the types of pages they contain.

  1 Executable programs or shell commands
  2 System calls (functions provided by the kernel)
  3 Library calls (functions within program libraries)

Manual page man(1) line 1 (press h for help or q to quit)

```

- cat: concatenate and display the content of files.

```
wash:3 cat file.txt
HI My name is Danny
How are you
I will see you soon
Is there a place for me
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8wash:3 █
```

- date: list the current date and time.

```
wash:3 date
Tue Jun 11 15:08:34 PDT 2024
wash:3 █
```

- ls: list the files existing in the current directory.

```
wash:3 ls
HelloWorld.c
a
a.out
add.c
cmd
cmd_line
cmd_line.c
cmd_line_args.c
datafile.txt.error
datafile.txt.output
execve.c
file.txt
fork
fork.c
getline.c
getln
head_new
head_new.c
```

- df: (disk-free) display the amount of disk space used and available on the file system.

```
wash:3 df -h
Filesystem      Size  Used Avail Use% Mounted on
none            3.9G  4.0K  3.9G   1% /mnt/ws
l
none           222G  147G   75G  67% /usr/li
b/wsl/drivers
/dev/sdc       1007G   2.6G  954G   1% /
none           3.9G   84K  3.9G   1% /mnt/ws
lg
none           3.9G     0  3.9G   0% /usr/li
b/wsl/lib
rootfs         3.9G   2.1M  3.9G   1% /init
none           3.9G  868K  3.9G   1% /run
none           3.9G     0  3.9G   0% /run/lo
ck
none           3.9G     0  3.9G   0% /run/sh
m
tmpfs          4.0M     0  4.0M   0% /sys/fs
```

head\_new Command:

Usage: ./head\_new [-h] [-n N] [file.txt]

The command is used to mimic the functionality of the standard 'head' command in Unix OS. It is used to display the beginning of a file or input stream.

- Display First N Lines: The primary functionality of head\_new would be to display the first N lines of one or more files. If no specific line count is provided, it defaults to displaying the first 7 lines.

```
wash:3 ./head_new file.txt
HI My name is Danny
How are you
I will see you soon
Is there a place for me
line 1
line 2
line 3
wash:3 █
```



```
wash:3 ./head_new -n 3 file.txt
HI My name is Danny
How are you
I will see you soon
wash:3
```

- Help Option: displays usage information.

```
wash:3 ./head_new -h
Usage: ./head_new [-h] [-n N] [file.txt]
Options:
    -h          Print this help message.
    -n N        Print the first N lines (default is 7).
    file.txt    The file to read from. If not specified, reads from standard input.
wash:3
```

- Error handling: Proper error handling for cases as missing files, permissions issues, and invalid arguments.

```
wash:3 ./head_new file.txt 4
Error opening file: No such file or directory
wash:3 ./head_new -n file.txt 4
Error opening file: No such file or directory
wash:3
```

- Source code:

```
- #include <stdio.h>
- #include <stdlib.h>
- #include <string.h>
-
- #define DEFAULT_LINES 7
- #define MAX_LINE_LENGTH 1024
- #define MAX_LINES 1000
-
- void print_usage() {
-     printf("Usage: ./head_new [-h] [-n N] [file.txt]\n");
-     printf("Options:\n");
-     printf("  -h          Print this help message.\n");
```

```

-     printf(" -n N      Print the first N lines (default is 7).\n");
-     printf(" file.txt  The file to read from. If not specified, reads from standard input.\n");
- }
-
- void print_lines(char *lines[], int num_lines, int total_lines) {
-     for (int i = 0; i < num_lines && i < total_lines; i++) {
-         printf("%s", lines[i]);
-         free(lines[i]); // Free the allocated memory for each line
-     }
- }
-
- int main(int argc, char *argv[]) {
-     int num_lines = DEFAULT_LINES;
-     char *filename = NULL;
-     int i;
-
-     for (i = 1; i < argc; i++) {
-         if (strcmp(argv[i], "-h") == 0) {
-             print_usage();
-             return 0;
-         } else if (strcmp(argv[i], "-t") == 0) {
-             printf("args reached\n");
-         }
-         else if (strcmp(argv[i], "-n") == 0) {
-             if (i + 1 < argc) {
-                 num_lines = atoi(argv[++i]);
-             } else {
-                 fprintf(stderr, "Error: -n option requires an argument.\n");
-                 return 1;
-             }
-         }
-         else {
-             filename = argv[i];
-         }
-     }
- }

```

```

-
- FILE *file;
- int from_stdin = 0;
-
- if (filename) {
-     file = fopen(filename, "r");
-     if (file == NULL) {
-         perror("Error opening file");
-         return 1;
-     }
- } else {
-     file = stdin;
-     from_stdin = 1;
- }
-
- char *lines[MAX_LINES];
- int count = 0;
- char line[MAX_LINE_LENGTH];
-
- if (from_stdin) {
-     printf("Enter lines of text. Press CTRL+D to end input and display the first %d lines:\n",
num_lines);
- }
-
- // Read lines from the input
- while (fgets(line, sizeof(line), file) != NULL && count < MAX_LINES) {
-     lines[count] = strdup(line);
-     count++;
- }
-
- if (from_stdin) {
-     printf("Your input was: \n");
-     print_lines(lines, num_lines, count);
- } else {
-     print_lines(lines, num_lines, count);
- }

```

```
-     fclose(file);  
- }  
-  
-     return 0;  
- }
```

Redirection:

The functionality implements output redirection for commands using the syntax ‘command > filename’, where standard output is directed to ‘<filename>.output’ and standard error to ‘<filename>.error’.

- Usage: command > filename

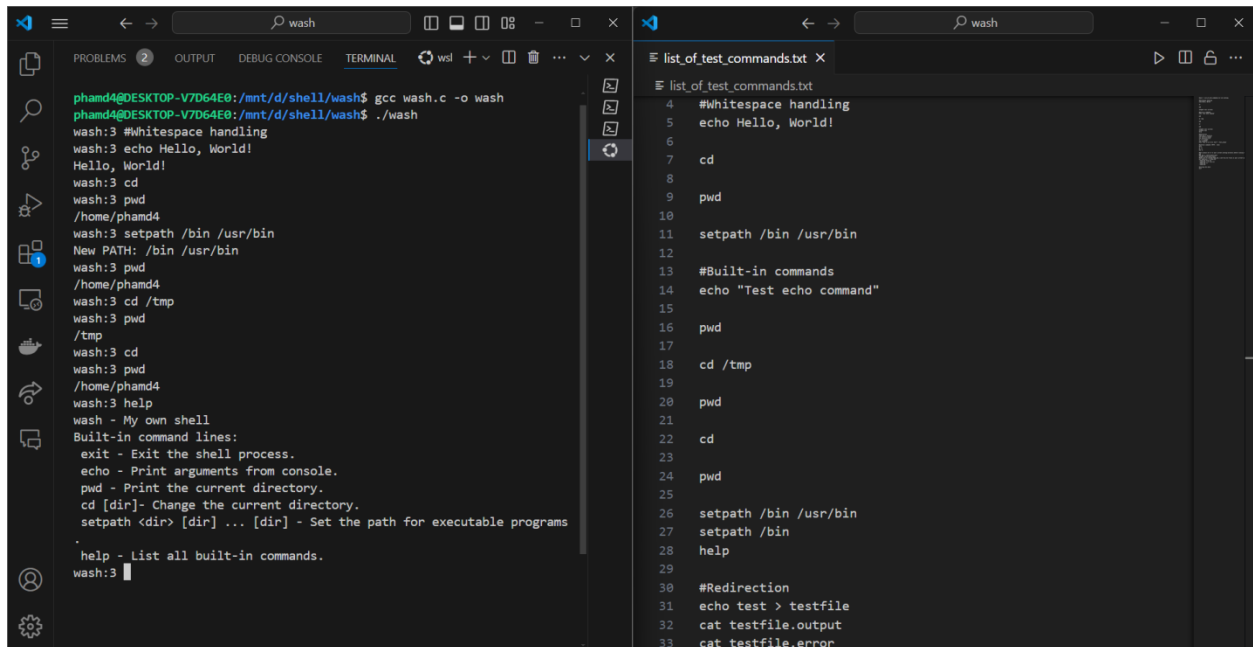
```
wash:3 echo Hi there! > testfile  
wash:3 cat testfile.output  
Hi there!  
wash:3 cat testfile.error  
wash:3
```

## Sample Input and Out (Test run):

The commands for the test run is included in the ‘list\_of\_test\_commands.txt’ which provides an overview of the commands and the commands themselves. Unfortunately, the *wash* cannot perform automatic command inputting from file:

```
phamd4@DESKTOP-V7D64E0:/mnt/d/shell/wash$ ./w  
ash < list_of_test_commands.txt  
(null): command not found  
#wash: command not found  
(null): command not found  
#Whitespace: command not found  
wash:3 wash:3 wash:3 wash:3 wash:3 Hello, World!  
(null): command not found  
chdir: No such file or directory
```

Thus the test run is execute as follows:



The screenshot shows a VS Code interface with two panels. The left panel is a terminal window titled 'wash' showing the execution of the 'wash' shell. The right panel is an editor window titled 'list\_of\_test\_commands.txt' showing a list of test commands.

```
phamd4@DESKTOP-V7D64E8:/mnt/d/shell/wash$ gcc wash.c -o wash
phamd4@DESKTOP-V7D64E8:/mnt/d/shell/wash$ ./wash
wash:3 #Whitespace handling
wash:3 echo Hello, World!
Hello, World!
wash:3 cd
wash:3 pwd
/home/phamd4
wash:3 setpath /bin /usr/bin
New PATH: /bin /usr/bin
wash:3 pwd
/home/phamd4
wash:3 cd /tmp
wash:3 pwd
/tmp
wash:3 cd
wash:3 pwd
/home/phamd4
wash:3 help
wash - My own shell
Built-in command lines:
exit - Exit the shell process.
echo - Print arguments from console.
pwd - Print the current directory.
cd [dir]- Change the current directory.
setpath <dir> [dir] ... [dir] - Set the path for executable programs
help - List all built-in commands.
wash:3
```

```
list_of_test_commands.txt
4 #Whitespace handling
5 echo Hello, World!
6
7 cd
8
9 pwd
10
11 setpath /bin /usr/bin
12
13 #Built-in commands
14 echo "Test echo command"
15
16 pwd
17
18 cd /tmp
19
20 pwd
21
22 cd
23
24 pwd
25
26 setpath /bin /usr/bin
27 setpath /bin
28 help
29
30 #Redirection
31 echo test > testfile
32 cat testfile.output
33 cat testfile.error
```

More test commands are included. Please open 'list\_of\_test\_commands' to explore more.

## Implementation Details:

- Command Parsing and Execution: The wash shell continuously reads user input, parses it, and determines whether it is a built-in command or an external program. For built-in commands, it handles the functionality directly. For external programs, it forks a child process to execute the program and waits for the child process to complete.

```
- int main(int argc, char *argv[]) {
-     //Initialize paths with default /bin
-     paths[0] = "/bin";
-
-     //implement help()
-     if (argc > 1 && strcmp(argv[1], "help") == 0) {
-         help();
-     }
-     return 0;
- }
```

```

-     }
-
-     char input[MAX_INPUT_SIZE];
-     while(1) {
-         printf("wash:3 ");
-         if(!fgets(input, sizeof(input), stdin)) {
-             break;
-         } //get user input
-
-         //Remove trailing newline
-         input[strcspn(input, "\n")] = 0;
-         char *trim_input = trim_leading_space(input);
-
-         //check for redirection
-         char *redirect_pos = strstr(trim_input, ">");
-         if (redirect_pos != NULL) { //if there are args
-             //redirect output
-             *redirect_pos = '\0'; // split cmd and filename
-             char *filename = trim_leading_space(redirect_pos + 1);
-             redirect_output(trim_input, filename);
-         } else {
-             if (strcmp(trim_input, "exit") == 0) {
-                 break;
-             } else if (strncmp(trim_input, "echo ", 5) == 0) {
-                 echo(trim_input);
-             } else if (strcmp(trim_input, "pwd") == 0) {
-                 pwd();
-             } else if (strncmp(trim_input, "cd", 2) == 0) {
-                 cd(trim_input);
-             } else if (strcmp(trim_input, "help") == 0) {
-                 help();
-             } else if (strncmp(trim_input, "setpath", 7) == 0) {
-                 setpath(trim_input);
-             } else if (strncmp(trim_input, "#", 1) == 0) {
-                 continue;
-             }
-         }
-     }

```

```

-     }
-     else {
-         external_command(trim_input);
-     }
- }
- }
-
- if (path_count > 1) {
-     for (int i = 0; i < path_count; i++) {
-         free(paths[i]);
-         printf("Freed %s\n", paths[i]);
-     }
- }
-
- return 0;
-
- }

```

- Error handling: each function implements robust error handling to manage incorrect inputs, missing files, and command execution failures gracefully, providing informative error messages to the user.

```

void pwd() {
    char curr_dir[MAX_PATH_LENGTH];
    //use getcwd, takes 2 args, buffer path and size
    //returns pointer to the buffer, NULL if error occurred
    if (getcwd(curr_dir, sizeof(curr_dir)) != NULL) {
        // getcwd succeeded, print curr dir
        printf("%s\n", curr_dir);
    } else { //if fail, print the descriptive error message
        perror("pwd - Can't get the current directory");
    }
}

```

- Path management: The initial PATH includes /bin. Users can modify the PATH using the setpath command to include additional directories, allowing wash to locate and execute programs from those directories.

```
phamd4@DESKTOP-V7D64E0:/mnt/d/shell/wash$ ./wash
wash:3 echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/lib/wsl/lib:/mnt/c/Windows/system32:/mnt/c/Windows:/mnt/c/Windows/System32/Wbem:/mnt/c/Windows/System32/WindowsPowerShell/v1.0:/mnt/c/Windows/System32/OpenSSH:/mnt/c/Program Files/dotnet:/mnt/c/WINDOWS/system32:/mnt/c/WINDOWS:/mnt/c/WINDOWS/System32/Wbem:/mnt/c/WINDOWS/System32/WindowsPowerShell/v1.0:/mnt/c/Program Files/Tailscale:/mnt/c/Program Files/Docker/Docker/resources/bin:/mnt/d/Git/Git/cmd:/mnt/d/PuTTY:/mnt/c/Program Files/CMake/bin:/mnt/c/Users/pdung/AppData/Local/Programs/Python/Python312/Scripts:/mnt/c/Users/pdung/AppData/Local/Programs/Python/Python312:/mnt/c/VSARM/armcc/10 2021.10/bin:/mnt/c/Users/pdung/AppData/Local/Programs/Python/Launcher:/mnt/c/Users/pdung/AppData/Local/Microsoft/WindowsApps:/mnt/c/Program Files/JetBrains/CLion 2023.2.2/bin:/mnt/c/Users/pdung/AppData/Local/Microsoft/WindowsApps:/mnt/c/Users/pdung/AppData/Local/Programs/Microsoft VS Code/bin:/mnt/c/Users/pdung/AppData/Local/GitHubDesktop/bin:/mnt/c/VSARM/mingw/mingw32/bin:/snap/bin
wash:3 setpath /bin
New PATH: /bin
```

## Reflections:

Three things that I learned are:

- I learned how to write the source code for the operation of each command in the shell by test it with the bash shell and formulate the source code.
- I learned the foundation of file system and file management where working directory and working PATH determine the structure of operation of each command.
- Finally, I learned to handle input strings and errors when user input at the standard input.

The most challenging part of this assignment is definitely string manipulation with handling external command. I had to review the concept of running another process within the parent process in OS and I had to process user inputs with strtok, strncmp which I have never used before.

A task that I did well was to be able to comprehend the essential tasks for making a shell, including implementing OS process control, user input string manipulation, and file system.

I would add more instructions detail on the hint of Thread Assignment because I did not use the hint to fulfil the assignment's requirements.

## Appendix:

### CSC3350 Extra Credit Coding Project

#### wash Shell

**[Up to 20% of your final grade]**

**[100 Total Points]**

In this lab, you will implement your own command line shell, **wash**, written in C (preferred) or C++. The “wa” is for Washington (as in Washington State), and the “sh” is for shell, i.e., “Washington shell”. Your shell program can target Windows WSL2 (Linux), MAC OS (terminal),



or a Linux-based distribution. Indicate which you are targeting in your documentation document (see below).

You will **work individually for this lab**. You may discuss high-level concepts with other students, but you should never share solutions or show each other code.

There are four requirements for this assignment that contribute to the assignment's grade:

- The shell program itself: `wash.c`
- A utility command: `head_new.c`
- Comments and coding style in both
- A documentation document that describes what you did and reflects on what you learned, amongst other things

## What is a shell?

A shell interpreter, often called a shell or command line interface (CLI), is a program that continually loops, accepting user input. The shell interprets the user's input to execute programs, run built-in functions, etc. Here is a high-level pseudocode of the behavior of a shell:

```
while true
    read user_input                // The Parser
    if user_input is a built-in command // The Executor
        handle it
    else if user_input is a program on the path
        fork child process to run the program
        wait for the child process to complete
    else
        print error
```

The default shell on Linux systems is `bash` (sometimes `zsh`), and the `dash` shell is also installed. You should start by skimming the `man` pages for `bash` and `dash` to get an idea of all the wonderful things shells can do for us. You will implement your own shell, `wash`. Your `wash` shell will have some “basic” functionality similar to `bash`, `dash`, and other shells.

## (60 Points) The Code: Shell Program and One User Command Program

Note on [Command Line Syntax](#): parameters in `[]` are optional; parameters in `<>` are required. The `|` (pipe) means “or.”

- **Invoking `wash`, usage, and errors (8 points)**
  - Usage: `wash [-h]`
    - The optional `-h` flag prints the help message (see below) and immediately exits.
  - After invoking `wash`, it runs until the user types the `exit` command.
  - Errors should **not** end a `wash` session. Instead, print an appropriate error message – for example, “XYZ command not found”.

- **Handling whitespace (2 points)**

- **Note on whitespace:** all commands typed into `wash` should ignore leading and trailing whitespace. For example, “`cd`” should work the same as “`cd`”.  
Tip: Start with an implementation that *does not* account for extra whitespace, and add this feature last (since it’s only 2 points).

- **Built-in Commands (15 points)**

Per the pseudocode above, `wash` should handle built-in commands as special cases – do *not* fork a child process; handle these built-ins “directly” in the shell program.

- **exit** – should end your shell process.
  - Usage: `exit`
- **echo** – print the arguments given to the console.
  - Usage: `echo [some text]`
- **pwd** – print the current working directory. Tip: you should use the `getcwd()` function.
  - Usage: `pwd`
- **cd** – should change the current working directory.
  - Usage: `cd [dir]`
  - If the optional `dir` is provided, change to the specified directory.
  - If no arguments are provided, change to the user’s home directory. Tip: use `getenv("HOME")` to retrieve the user’s home directory.
  - Tip: you should use the `chdir()` function.
- **setpath** – sets the path, the user must provide at least one argument (directory). The path is where `wash` will look for executable programs to run.
  - Usage: `setpath <dir> [dir] ... [dir]`
  - `setpath` **overwrites** the path with whatever arguments the user enters.
  - The path when `wash` launches should contain only `/bin`.  
Example:
    - User launches `wash`  
Path contains only `/bin`
    - User invokes: `setpath /bin /usr/bin`  
Path now contains `/bin` and `/usr/bin`
    - User invokes: `setpath /usr/share/bin`  
Path now contains only `/usr/share/bin`
- **help** – list all the built-in commands with short, user-friendly descriptions.
  - Usage: `help`

- **Redirection (15 points)**

- Shells typically allow for redirection between programs. For example, in `bash`, try the following: `echo blah > tmp_file_lab3.txt`  
In this example, nothing is printed to the console, and the text “`blah`” is instead directed to the file `tmp_file_lab3.txt`. Note: this file is overwritten if it exists!
- You will implement *simplified* redirection in `wash`: when the user invokes `command > filename`, redirect the `command`’s standard output to

<filename>.output and standard error to <filename>.error

A missing filename argument or multiple arguments should not be allowed: print a useful error message and do **not** run the command.

- **Other Commands (24 points)**

Show that `wash` handles five external commands, including 1 program you write, viz. “`head_new`” (see below). Note: These five *Other Commands* are in addition to the built-ins listed above.

- Show examples of `wash` handling 4 non-built-in external commands of your choice.
  - The command should only be executed if, and only if, it is in one of the directories specified in the user’s **PATH**. Search the paths in the order they appeared in the `setpath` built-in. Fork a child process, pass the appropriate arguments entered by the user to `exec`, and wait for the child to return before continuing to accept more user shell commands.
  - You might choose from the following list of Linux commands (or other favorites): `cat`, `date`, `diff`, `df`, `find`, `grep`, `man`, `ls`, `sum`, `tail`, `wc`.
  - Be sure to use the appropriate arguments for the 4 commands you choose.
  - Document which commands you chose in your write-up.
- Write the code for a new 5<sup>th</sup> command that can be invoked in `wash`: **`head_new`** and show the sample output of `wash` running it.
  - Usage: `./head_new [-h] [-n N] [file.txt]`
  - If `[file.txt]` is specified, print the first `N` lines of that file (default is 7 lines if `-n` is not used). Otherwise, read from `stdin` until the user hits CTRL+D, and print the first `N` lines typed by the user.
  - If the `[-n N]` flag is used, print the first `N` lines.
  - The `[-h]` flag should print an argument usage summary (help message).
  - The `[file.txt]` and `[-n N]` argument order should not matter – both orderings should be allowed.
  - Make sure to compare your output to the real Linux `head` command. Note this is not exactly like `head`: only *one* file argument should be accepted, and we are not implementing the various command-line flags (except for `-n` and `-h`). In addition, the Linux standard `head` prints the first 10 lines by default, whereas `head_new` prints the first 7.
  - Make sure to fail gracefully if the file does not exist, if too many (or too few) arguments are passed, or if the argument order is wrong
    - E.g., `./head_new -n file.txt 10`
  - Hint: The code you developed for the “Threads Assignment” might be of some use here.mul

- **README (6 points)**

- Provide a README file for users to know how to run this program. It contains the same information as what the program generates when you run it with `-h` or run the `help` built-in.
- Both `cat README` and `man wash` should display the contents of your README file.

**Tips (and some rules):**

-Get simple commands working first – for example, commands with no arguments – and test them as you go.

-Test your shell *thoroughly* – for example, you might try these three variations of `setpath`:

```
setpath /bin
```

```
setpath /bin /usr/bin
```

```
setpath /home/username/blah
```

After each one, try running a variety of programs, some in the directories and some not in the directories, to verify you are running the appropriate command from the proper directory. Be sure to test “happy cases” (the command is in one of the `setpath` paths) and “unhappy cases” (the command is not in any of the `setpath` directories, and therefore it fails and prints an error message).

-Include sample input and output from your tests in your assignment documentation document.

-An easily repeatable way to test `wash` is to put a series of test shell commands into a file, each on a separate line, and then run them through your shell via:

```
wash < list_of_test_commands1.txt
```

**Note:** This is one of the methods I will use to test and grade your code after I compile your code in my environment.

-If you run into segmentation faults (`segfault`, `sigsegrv`), but don’t see any output, add `fflush(<buffer>)` (e.g., `fflush(stdout)`) after important output statements to ensure the output buffer is flushed. Or, better yet, use `gdb` (or some other debugger) to debug!

-You may assume the user’s input on the command line is always 256 characters (bytes) or less.

-Your programs should fail *gracefully* when given bad input. *Gracefully* means you should print nice error messages to standard error when appropriate. There are no specific requirements here, so think carefully about what errors could occur. Frequent and thorough testing will help expose potential errors, as will paying close attention to the return values for all functions you use.

-You may not use `system()` (or the `exec()` family of system calls) to invoke `bash`, `dash`, or any other shell. Your code must execute everything directly and not rely on another shell.

-You should not use or include any code or code snippets that you find online or solicit code from others to implement this program. This is to be entirely your own original code, as that is how you will learn the most from this assignment. When you have a question, just use the Linux `man` pages.

## **(25 Points) Comments and Code Style**

### **Coding Style and Other Requirements:**

- Always check the return value for all functions you use. Always make informed decisions about what arguments you pass to functions, especially syscalls. See below for rules on comments.
- Pick a coding style and stick with it. Good, consistent variable names, identifier style (e.g., snakecase), curly brace placement, tabs/indentation, etc., are all important. No specific style will be enforced, but you should follow best practices (generally avoid single-letter variable names, global variables, excessive copy'Npaste of code, etc.).

### **Comment Requirements:**

- Your code should include a significant number of comments.
- All system calls you use should be well-commented, with explanations of what the parameters and arguments mean, what the function returns, and why you dealt with the return value the way you did. Here is an example of good comments for `fopen`:

```
/*  
    This call to fopen takes the path to a file to open  
    (the argument is argv[1], the filename provided by  
    the user), followed by the access mode (the "r" argument  
    means read-only). fopen returns a FILE * (pointer to  
    a stream representing the open file), or NULL if the  
    operation failed, in which case the program ends.  
*/
```

This comment shows that you have thought about each parameter, passed appropriate arguments, and understand what the function is returning.

Here is an example of an insufficient comment for `fopen`:

```
//fopen takes a path and a mode and opens a file
```

This comment is too generic; it doesn't explain what *you* are doing or discuss how the return values were handled.

- Comment style is up to you, but make it look neat and consistent.

**Remember, your comments demonstrate you know what your program does, and that it is your own original work. It's also a place to document improvements or ideas you have.**

- Include a short README file that explains how to use `wash`, including a brief description of the built-in commands. This is your `man` page for `wash`.

## **(15 Points) Project Documentation**

- See the “Submission Instructions” below for additional documentation requirements. Some items should be included in your code comments, and some belong in your documentation document.

## **Submission Instructions**

- Upload your source code files and an assignment documentation file to this Canvas assignment. Using a Word document, describe the operation of the program. List (with a brief description) all the files you are submitting, describe anything of particular note, and include 3 (or more) things you learned in a “Reflections” section at the end.
- Name the assignment documentation file: “<LastName> <FirstName> - CSC3350 wash Shell Extra Credit Project Documentation.docx”.
- Appropriately comment on your code inline and include a block comment/header at the beginning of each source file containing:
  - Your name, Course number, Assignment title, Professor, and Date.
- Document how to compile, build, and run your code. Be sure to list any packages that need to be installed to compile, build, and run your code.
- Remember, for this assignment, you are NOT to be fishing around on the internet; therefore, there should be no need for a “References” section in the documentation document. All the code you submit is to be your own original creation. Be sure to include the following attestation at the beginning of your documentation document:

**ATTESTATION:** The code submitted to this assignment is all my original code and a result of my individual work. I did not get any code from any website, book, source, AI tool, or person. If I did get any of the submitted code from somewhere or someone else, I have cited it in a “References” page (or a block comment), and I understand that using it will invoke a penalty.

**Signed – [Your Name Printed Here]**

- Include a section in your Documentation Document titled “Reflections” and answer the following questions. As always, good spelling, grammar, and punctuation are essential in everything you submit.
  - **Reflections**
    - A. What were three things you learned in this assignment?
    - B. What was one thing that was challenging in this assignment? You may list more than one.

- C. What is one thing you feel you did well or are proud of? You may list more than one.
- D. Is there anything that you would change, add, or remove from this assignment? Describe and explain why. List at least one item from the list and more if you have them.

- Include this Assignment Specification Document, "*CSC3350 My Own Shell.docx*", as an **Appendix** in your documentation document. Use this to track your progress on the different parts of this assignment. Highlight (or strike-out) each sentence in this problem statement as you handle it in your solution. Highlight in a different color those things that are informational but don't require you to do anything. Once everything is dealt with to your satisfaction and highlighted, you are done and ready to do a final proofread and submit. This best practice will ensure you don't miss anything in the assignment instructions. And if you don't get any parts of the assignment done, note those in your write-up. Finally, don't stress about not getting everything done; remember, this is extra credit.