Yarra Abozaed

CSC 3350

Professor Cameron

June 5, 2022

Wash Extra Credit

**ATTESTATION: The code submitted to this assignment is all my original, individual work. I did not get any code from any web-site, book, source, or person. If I did get any of the submitted code from somewhere or someone, I have cited it, and I understand that using it will invoke a penalty.**

**Signed – *Yarra Abozaed***

* **List of files:**
  + **washAY.c -**
    - All the source code for the assignment is in wash.c. It includes all the needed code to run the commands via the shell.
  + new\_headAY.c –
    - Empty file
  + READMEAY.txt –
    - Includes a manual and description of the shell.
* **How to use:**
  + To use the shell, open a terminal window, navigate to the folder containing the wash.c file, and type ./script.sh into your shell.
  + To build the shell, run **gcc washAY.c -o wash**
  + To start the shell, execute ./wash
  + If you would like to print all the commands in the shell at execution, run ./wash -h
* **Learnings:**
  + How different command line inputs work by building a few myself and how to use C to mimic those commands.
  + Using different string.h functions to process user inputs, including strlen(), strcmp(), etc.
  + File processing and printing file information to user.
* **Appendix:**

**CSC3350 Coding Project**

**wash Shell**

**[100 Total Points]**

In this lab, you will implement your own 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.

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 three requirements for this assignment that contribute to the assignment’s grade:

* the shell program itself: wash.c
* a utility command: new\_head.c
* comments and coding style in both

**What is a shell?**

A shell interpreter, often just called a shell or command line interpreter, 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 sketch of the behavior of a shell:

**while true**

**read user input** // The Parser

**if input is built-in command** // The Executor

**handle it**

**else if user input is a program in path  
 fork child process  
 wait for child process to complete**

**else  
 print error**

The default shell on Linux systems is bash (and sometimes zsh), but 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 will have some “basic” functionality similar to bash, dash, and other shells.

**(70 points) The Code: Shell Program and One User Command Program**

Note: 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 – usually, this will be things like “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
  + **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, 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 screen 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 command’s standard output to <filename>.out and standard error to <filename>.err. 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 (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.
    - The command should only be executed 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.
  + Write the code for a new 5th command invoked in wash: **new\_head** and show the sample output of wash running it.
    - Usage: ./new\_head [-h] [-n N] [file.txt]
    - If [file.txt] is specified, print the first N lines of that file (default is 5 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 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, head by default prints the first 10 lines, new\_head prints the first 5.
    - 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., ./new\_head -n file.txt 10).
    - Hint: The code you developed for the “Threads Assignment” might be of some use here.
* **README (6 points)**
  + Provide a README file for users to know how to run this program. It may contain the same information as the generated by the program when you run it with -h and run the help builtin.
  + 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

and after each, try running a variety of programs.

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

-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 ways I will test and grade your code.

-If you run into segmentation faults (segfault, sigsegv), 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* for bad input. *Gracefully* means you should print nice error messages to standard error when appropriate. There are no specific requirements here, so think very 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 needs to execute everything directly itself.

-You should not use or reference any code online or solicit help for how to implement this program. This is to be entirely your own work, as that is how you will learn the most from this assignment. Only look at *references* for the C functions. When in you have a question, just use the Linux man pages. **(30 points) Comments and Code Style**

**Style and other requirements:**

* Always check the return value for any 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/indention, 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 should be well-commented, with explanations for 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 demonstrates that you have thought about each parameter, passed appropriate arguments, and understand what the function is returning.

Here is an example of a bad 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 show that you understand what you have done.

* Comment style is up to you, but make it look neat and consistent.
* Include a short README file that explains how to use wash, including a short description of the built-in commands. This is the man page for wash.
* 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.

**Remember, your comments are how you demonstrate you know what your program does, and that it is your own, original work.**

**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 “Learnings” section at the end.
* Name the assignment documentation file: “CSC3350 wash Shell Extra Credit Project - LastName FirstName.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, 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, individual work. I did not get any code from any web-site, book, source, or person. If I did get any of the submitted code from somewhere or someone, I have cited it, and I understand that using it will invoke a penalty.**

**Signed – *[Your Name Printed Here]***

* Include this document as an Appendix in your documentation document. Then highlight 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 that you do anything. Once everything is highlighted, you are done and ready to do a final proofread and submit. This will ensure you don’t miss anything in the assignment instructions.