CSE 2431 **LAB 3** **SP 23**

Due: Tuesday, 4/18/23 at 11:30 p.m. on Carmen

NOTE: These instructions are relatively long, but my intention in making the instructions longer is that it will make it easier for students to complete the lab successfully, without spending unreasonable amounts of time writing the necessary code. Virtually every student spends considerably more time on this lab than on the first two (students typically report spending 6 to 8 hours for this lab, but for some students, it may be even more). By reviewing and following the instructions carefully, and by asking questions ***early*** if you are unsure about something, you will reduce the time and increase the probability of writing a lab that works correctly.

1. **Goal:** Enhance the shell written for Lab 1 with two new features, ***a history feature***, and ***a command rerun feature.***

2. **Introduction**

This lab assignment is an extension to the Linux/UNIX Shell interface which was built in Lab 1. You will add 2 additional features to the shell for Lab 1: (1) a history feature; and (2) a rerun feature; both will be added to the Linux/UNIX Shell we wrote for Lab 1.

These 2 features will allow users to (1) view and (2) possibly rerun any one of up to the 5 most recently entered/run commands. When your program outputs these commands (when the user utilizes the h or history command feature – see below) these commands will be numbered starting at 1 (the *least recent/oldest* of the last 5 commands) and will be numbered up to 5 (the *most recent/newest* of the last 5 commands). If the user executes the same command twice consecutively, it will appear twice in the history. A working version of the shell for Lab 1, shellA.c, has been placed on Carmen in the Labs folder which you can download into a lab3 directory which you create. You should start with this version of the code, and then add code to it as described below to implement the two new features being added for this lab.

We will use a historyBuff array to store the user's command history (up to the last 5 commands). This historyBuff array is declared in main, and has a size equal to 5 times the maximum length of a command, which for this lab (the same as for lab1), we will set to be 40 (these 40 bytes include the space for a newline at the end of the command, and also space for a null character which we will need to place at the end of the command after it is read from the command line by the setup function). Therefore, the historyBuff array will be an array of char of size 200.

To store commands in the historyBuff, ***we need to create a copy of the command which the user has entered***, after they enter it. What is the best way to do this? Notice that the setup function in the shellA.c source code uses a read file system call to read the command which the user has entered on the command line into the inputBuff array which is passed to the setup function. You should notice, however, that after setup reads the command into inputBuff, the command in inputBuff is ***modified*** by the setup function, so ***we cannot use the inputBuff array to get a copy of the command after it has been modified by setup!*** *W*e can, however, pass another array to setup, which we will call commandCopy (also declared in main), to store a copy of the command ***BEFORE it is modified*** (you have to figure out how to store the copy of the command before inputBuff gets modified; be sure, though, to use strcpy from the C library to copy the string). After the commandCopy array has a copy of the command, and after the setup function terminates and returns to main, we can use the commandCopy array to store a copy of the command which the user entered in the historyBuff array. **Be sure to pay close attention to what is said below about the required null byte terminator for strings in C, which will need to be placed at the end of the command in commandCopy;** in particular, when the user enters/types the command on the command line, the user types enter/return at the end of the command. **The last character in the inputBuff sequence of characters is therefore a newline ('\n'), and NOT a null character!** Therefore, after the read system call which puts the user entered command in inputBuff, at this point, **what is in inputBuff is *NOT (repeat, NOT)* a C string! We therefore CANNOT call C string library functions such as strcpy or strlen and pass them this sequence of characters (see explanation below).**

EXPLANATION: String functions in the C library have UNDEFINED behavior if we pass them sequences of characters which are not terminated by a null byte ('\0') (THAT IS, ALL OF THESE FUNCTIONS REQUIRE A NULL BYTE, '\0', AT THE END OF ANY STRING WHICH WE PASS TO THEM, in order to be sure that the function does what it is supposed to do, and not some undefined behavior. We can know, however, how many characters are in the command typed by the user, up to and including the newline, because the read system call returns the number of characters read, up to and including the newline, and we have declared an int variable in setup (called length) to store the value. Once we know how many characters are there, up to and including the newline, we can use this to get an index into the inputBuff array of characters to place a null byte character *after the newline* (another possibility is to replace the newline by the null byte, but having the newline there will be useful when we print the command, and also if we need to pass the command back to setup in order to rerun the command (see below), and the newline will not interfere with execution of the command when we use the command rerun feature later, **so I *strongly advise* that you leave the newline there and place the null byte after the newline!!!).**

One modification to Lab 1: In lab 1, we assumed that the maximum line length for a command entered by the user would be 40 characters. Here, we assume the same length (including the newline at the end of the command entered on the command line by the user of the shell, as well as the space for the null byte at the end of the command to make it a C

string). Therefore, since you need to store the last 5 commands in the history, and each line can be up to 40 characters (including the null byte termination), you need a historyBuff of 200 characters. We can just declare (in main) an array of char of length 200 called historyBuff, so the name of this array will be a constant pointer which points to the array. Passing historyBuff as a parameter to a function will allow you to access the various sequences of up to 40 characters which correspond to each command in the history (You can determine where each command starts by using the maximum length of 1 command, which is 40 characters total; for example, the first command starts at index (0 * 40), the second at index (1 * 40), the third at index (2 * 40), etc. We also declare an array commandPtrs, which is an array of 6 char pointers (pointers to strings); commandPtrs[0] is NULL, and does not point to a command in the historyBuff, but commandPtrs[1] points to the command numbered 1, commandPtrs[2] points to the command numbered 2, and so on. These pointers can be used to copy a command from one position in the historyBuff to another (for example, if there are 5 commands in historyBuff, and the user enters another, in order to move all of the last 4 commands up by one, and to store the most recent command as the command in position 5 in the historyBuff.

IMPORTANT REMINDER: When storing commands, be very sure that there is a null byte at the end of the command (recall that the null byte is the char '\0')! Recall that C string operations ***cannot be depended on to work correctly (they have undefined behavior)*** if a string is not terminated by a null byte string terminator. You should put this null byte at the end of each command before you copy it into the commandCopy array in setup; **BE SURE TO DO THIS IN THE CODE IN THE setup FUNCTION (you need to add code to the setup function at the appropriate place to do this); DO NOT TRY TO DO IT after setup terminates and returns to main!** ***A C string, by definition, is null byte terminated;*** **C library functions that work on strings ASSUME that any string has a null byte terminator; if this assumption is not true for a given array of characters, ALL C functions which work on strings have UNDEFINED BEHAVIOR (we do not know what the function will do, and debugging becomes virtually impossible!). Overlooking this important fact will cause your code to exhibit bugs which are *extremely* difficult to find and correct, so make sure you spend the time necessary to get this right from the beginning, and the lab will be much less stressful!**

3. Assignment

A user will be able to list up to the 5 most recently entered commands when the user types the command **history** or its alias/shortcut **h (if fewer than 5 commands have been entered so far, only the number of commands actually entered will be displayed by executing the h/history command**. **When your shell program outputs the commands in the list, it should number them from 1) (the least recently executed or oldest command in the list) to 5) (the most recently executed command in the history list; and the numbers in the list of commands should be printed as stated above, that is, the number followed by ')'). Each command in the list should be printed on a separate line. The h or history command should not be placed in the history list, and an rr/rnum command (described below) should not be placed in the history list either.** With the list of previous commands

output with h or history, the user can run any of those previous 5 commands by entering **rnum** where '**num**' is the number of that command in the history list (**and num *will always be from 1 to 5*, inclusive, *not 0 to 4*). So, for example, after entering the command *history* or *h*, if the user then, after the history list is printed, runs the command r3, the 3rd command in the list should be printed out and run again. Also, the user should be able to run the most recent command again by entering rr for 'run most recent,' which should print out and execute the last command (most recent command) in the history list.** Notice that, if, for example, there are 5 commands in the history list, running rr or r5 would both print out and run the command numbered 5 in the list.

For *rnum*, you can assume that no spaces will separate the r and the number and that the number will be followed by '\n' when entered by the user on the command line. Also, when *rr* is used to execute the most recent command you may assume that it is immediately followed by '\n' on the command line. To simplify, we will assume that the user will only enter either *rnum* or *rr* immediately after entering/running *history* or *h*; after entering *h/history*, the user can execute any number of *rr/rnum* commands. You can assume that a user of your shell will comply with these restrictions.

As stated above, any command that is executed in this fashion (that is, using rr or rnum) should be echoed on the user's screen and the command ***should not be*** placed in the historyBuff again (it is already there). (**rnum** and **rr** do not go into the historyBuff; ***the actual command that they are used to execute does not go into the historyBuff again either; in other words, executing a previous command again using rr or rnum does not modify the historyBuff list at all; executing h or history does not modify the history list at all either)***.

For a better idea of how *a similar* (***BUT NOTE CAREFULLY, NOT IDENTICAL!!!!!!)*** history feature actually works in the Linux CSE Environment, execute a few commands in the terminal window then type **history.** To execute a command from the history list type **!num** (for example, !3, with no space between the ! and the num). To execute the most recent command type **!!.** Depending on whether it is set up in your aliases for commands, **h** may work for history as well. Also note that when you type **!num** or **!!**, the full command from the history list is output then executed. If you type **history** again you can see the commands that were repeated are now in the updated history list. Our implementation of history does not use ! with a number, or double ! (!!) to rerun commands, but rather r with a number, or rr. **Also, our implementation of the history feature is not identical to the Linux feature in all ways, so *please DO NOT ASSUME that it is; you should implement the commands as they are described in this document*.** Trying this in Linux just gives you a sense of how such a feature works, but in your shell, it should be implemented as described earlier, and not based on the way it is implemented in Linux.

Advice on Writing the Code

history/h implementation: You should write a function to store each command that the user has entered on the command line, up to the last 5, in the historyBuff array. The function is

called updateHistory. You will also write a separate function to print the commands in the historyBuff, called printHistory. The code provided shows when to call these functions, but having separate functions to do these things makes the code more modular, and much easier to debug, so this is why we put this code in separate functions and DO NOT put the code in main or in a function that does other work (this would be a really poor way to write C code).

First, implement the isHistory function, which returns a Boolean. This function is passed commandCopy, which has a copy of the last command entered by the user, and it checks the command to determine if it is an h or history command (see the note below on the use of strcmp to compare strings). The function returns 1 if the command passed is h or history, or 0 if the command is not.

Second, implement the updateHistory function, which adds the last command run to the historyBuff (unless it is a rerun or history command). In main, there is a numCommands variable, which is used to count the number of commands that are not either history or rerun, which the user has run. This numCommands variable needs to be passed to updateHistory. For the code for this function, we need to understand there are 2 cases. One case is where numCommands is less than or equal to 5. In this case, you just need to add the commandCopy passed to the function at the end of the historyBuff (use numCommands to determine where to add the command). In the other case, where numCommands is greater than 5, your code first needs to move the 5 commands in the historyBuff up by 1 position (move command 2 to position 1, 3 to 2, 4 to 3, and 5 to 4). Finally, add the command in commandCopy to position 5. NOTE: The position numbers here refer to the numbering of the commands (1 to 5).

Third, write the code for printHistory. There will also be 2 cases for this function, depending on whether numCommands is less than or equal to 4 (print numCommands from the buffer), or is greater than or equal to 5 (print 5 commands).

Now test the history feature thoroughly, before attempting to write code to implement the rerun feature as described below.

How to implement the rr/rnum feature: After the setup function is called, and you store a copy of the user's command in commandCopy, you can check to see if the command is rr or rnum (r1, for example) by calling the isRerun function (which returns a Boolean depending on whether the command is a rerun command or not, that is, whether it is r1, r2, r3, r4, r5, or rr, or not). If the command is a rerun command, another function will be called, rerunIndex, which will return the index of the command in the historyBuff to rerun (If you determine that it is rr or rnum, you have a copy of the command that the user wants to execute in the historyBuff array (and you can use this copy to echo the command, which is supposed to be done for rr/rnum; the code provided calls printf to echo the command before calling setup a 2nd time for an rr/rnum command); however, in order to have the forked child execute the command, you need the command to be parsed into substrings, and to have an args array of char pointers pointing to the substrings that you can use to invoke execvp. How can the command in the historyBuff which is to be rerun be used to get an args array in order to rerun

the command? Well, the setup function parses commands and produces an args array, but it does this after reading the command the user entered using a read file system call. We can use a Boolean variable (int in C), which we call rerun, which we will pass to setup. If rerun is 0, then setup will read the command the user entered on the command line using a read call, but if rerun is 1, then instead of doing a read system call, setup will use the command we pass it as the commandCopy parameter (which can be copied into inputBuff before setup parses the command, if rerun is 1). You will need to put code in the setup function to modify it so that it works this way.

As stated before, your source code for the lab should be in a source file named shellA.c, which should have the code you add to the version of shellA.c which you download from the Lab 3 folder on stdlinux. Also, you are not only permitted to use string operations in the C standard library, ***but are strongly encouraged to do so*** (It is always better not to reinvent the wheel! For example, if you need to copy a string, do not write code to do this; instead, use strcpy in the C library; REMEMBER, though, as we said above, that C string functions assume any existing string passed is null byte terminated). You will not be able to do everything with library string operations, but they will help with a lot of what you need to do; it will be easier to use a library function wherever possible. Always keep in mind, as mentioned earlier, that library functions for strings in C ASSUME that any string passed as a parameter is null byte ('\0') terminated; ***be very sure*** that any string you pass to a library function has a null byte termination (Actually, if it does not, it is NOT a "string" in C!).

Also notice that, if the enters more than 5 commands (say, a total of 6 commands) before running h/history and rr/rnum, only the last 5 commands should be stored in the history buffer; after the first 5 commands are entered which are not h/history or rr/rnum, any additional command should move the oldest command (the one numbered 1) out of the history buffer, and move the commands numbered 2 to 5 up one spot; the most recently entered command should then be placed in spot 5. You will need to write code to do this if more than 5 commands (not h/history or rr/rnum) have been entered by the user.

NOTE on the use of strcmp: Use strcmp to compare strings, but remember that it returns 0 if the two strings passed to it are the same. Also remember that we left a new line character at the end of the command when we stored it in commandCopy, so be sure to include the new line when you pass parameters to strcmp. For example if we want to call strcmp on a command in commandCopy and the rr command, we would use:

if (!strcmp (commandCopy, "rr/n") .....

4. Submission

Please put all code for Part A in "**shellA.c**" (modify the code which you download from Carmen); at the beginning of your source file you need to tell the grader your full name (It should be in a comment; see the shellA.c source file; we have put a comment there, and you just need to fill in your name).

Submit this file on Carmen by starting a firefox web browser in the stdlinux environment:

$ firefox https://carmen.osu.edu/#

Submit only the source file specified above, and not an executable, or you will not get credit. Please ***DO NOT ZIP*** your source code file before submitting to Carmen.

All necessary code, and #include directives for necessary .h files from the C standard library, should be in a single C source code file for the lab. To make things easier for the grader, you should compile your code as follows (your Linux prompt may be different from $, but that does not matter):

```
$ gcc shellA.c -o shellA
```

To run the code, you should use:

```
$ shellA
```

Make very sure that your code compiles with no errors or warnings when using these compilation commands. If your code does not compile when the grader grades your lab, you will get no credit for the lab. Also make sure that your executable for each part runs without errors, and that it performs as described above.

To test your code, you can use the following Linux commands [NOTE: I suggest that you test only the first 6 commands and h/history first, to determine whether your h/history feature works correctly. Once that is debugged and working correctly, then work on the rr/rnum feature. (Remember also that the shell from lab 1 could run commands in the background or in the foreground) :

```
$ ls
$ ls  -al
$ pwd
$ date
$ ls &
$ pwd &
$ history        [history list with the last 5 commands above, numbered, should be
                  printed; the first ls command should not be in the history any
                  longer.]
$ rr             [Command 5, that is, pwd &, should be echoed and executed again)
$ r2             [Command 2, that is, pwd, should be echoed to the screen and
                  executed; try other values besides 2 to make sure they work too]
```

The code and documentation you write and submit must be your own independent, individual work. If the files do not compile or have runtime errors (such as a segfault), no credit will be given for the lab.

You should test your lab (of course!), so that you know when the grader runs it, it will perform as described. Test cases which you use h/history should show the last 5 commands that have been entered – seen by the command numbers displayed when the user types **history**. The test cases should also demonstrate the **rr** and **rnum** commands.

A late penalty of 25% will be assessed for any lab not submitted by the due date/time shown at the top of this document, but submitted within 24 hours of the due date and time.

You are welcome to use Piazza to discuss questions/problems you are having with the lab with the instructor or with other students. ***Please be sure, however, that you do not post code, unless you leave your post private (as it will be by default).***