Ryan Yurkanin

Intro to Systems

Lab 2 Shell Documentation

BRIEF OVERVIEW OF PROBLEMS

Problem A: Create a version of a Linux shell that is able to do the following, take in a command that is parsed into an array (argv [0], argv[n] … argv [n+1], and then executes the command, if found in the Linux System Call bin folder.

Problem B: Allow the user to enter the "&" character to declare that the processes should run concurrently.

Problem C: Allow the user to enter a "<" or a ">" if they wish to link two commands using I/O Redirection. Allow the user to enter a "|" if they wish to pipe two commands. Piping and I/O modifiers do not need to be handled in the same command line input.

FUNCTIONS USED

getcwd() - This function is designed to return an absolute pathname, as a string, to the current working directory.

gethostname() – This function is designed to return the host name of the computer that is currently running the shell. I use getcwd and gethostname to print out the current location of the shell, to help the user.

system() – I use this to clear the command line interface each time the shell is run. Keeps everything looking nice and tidy.

fflush() – I use this to clean out the output stream right at the beginning of each loop. Basic housekeeping function.

fgets() – Taking in commands from the user. We will later store this in the history array, and parse it into multiple commands and symbols.

strdup() – This function creates a c-string that is an exact replica of the argument, but assigned to a new memory address. I use this to store the input into the historical array, as well as getting the output/input redirect c-strings.

strlen() – This function travels through a c-string, and returns the length of it. I use this for loop control, as well as basic logic in a multitude of functions.

fork() – Linux system call that creates a new process, using the parent process, and waits for new code to execute, or executes the parent's code. I do error handling, by checking to see if it returns a negative number. I also use its return in logic statements to separate the code the child should run from the code the parent should run.

execvp() – Linux system call that replaces a processes code with different executable code. If the code can't be executed then it returns a negative number, otherwise it will just not return, because it is a new process. I use the negative number to error handle.

wait() – Another Linux system call, that stops a process from continuing until a certain condition is met. I often use this to make the parent wait until all of its children are done. That way the result is printed before the introduction of the shell in 100% of the cases.

dup2() – Safer than dup(), I use this to atomically close and open a specific file descriptor. That way nothing funky can happen with interrupts. I believe that this functions atomicity would be referred to as "coarse-grained".

close() – I use this function to close file streams that no longer need to be open for the current time.

printf() – I use this function to communicate with the user. I also use this function a lot in my testing when I am testing the code piece by piece.

exit() – I use this to exit the program if an error is encountered, or if the user sets the shell_running variable to FALSE.

malloc() – I use this to reserve space on the heap for a majority of the variables that I declare, as well as the struct that I create.

checkConcurrency() – Function I created, its purpose is to parse through argv [n] in an attempt to locate the "*&*". If it finds this symbol, then it marks down where it found it, replaces it with the NULL character, and sets the concurrencyactive flag to TRUE.

modifyHistory() – Function I created, its purpose is to save the input into the history array so that the user can reference it at a later time. If there are more commands saved in the array then MAX_HISTORY would allow, then it just starts rewriting from cliHistory[0].

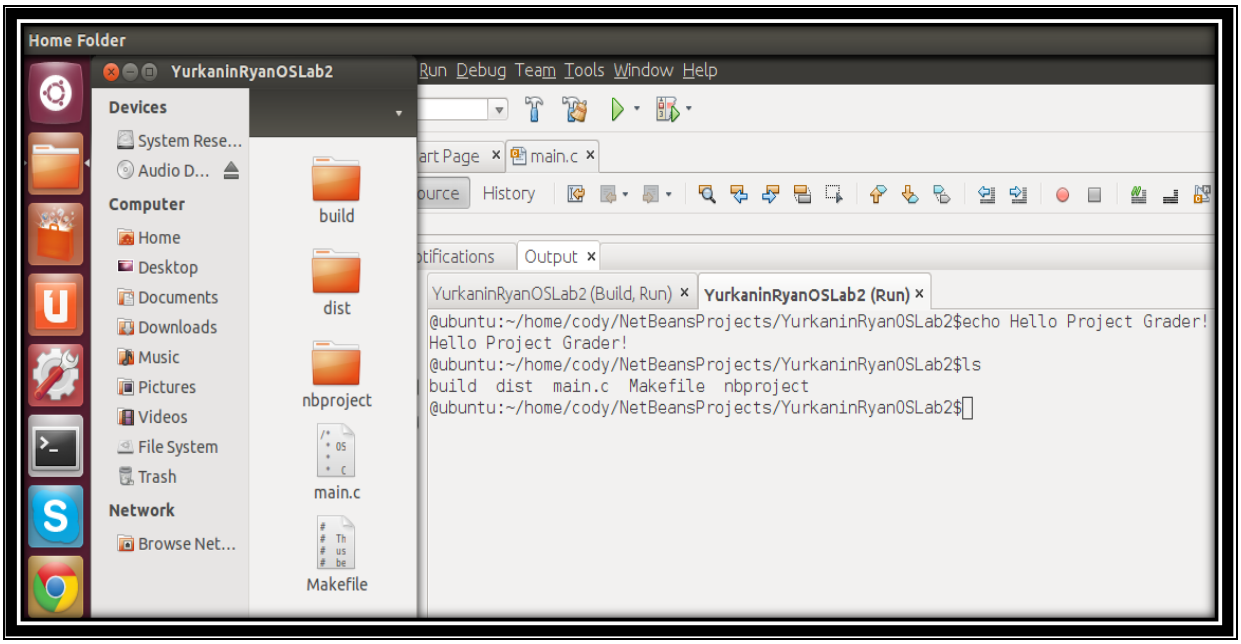
checkSpecialCommands() – Function I created, its purpose is to always be checking to see if the first command entered to the shell is either "exit" or "history". If it is exit, then shell_running is set to FALSE and the program exits. If it is history, then it iterates through the history array printing each input ever entered in chronological order.

checkOutputRedirection() – Function I created, it's purpose is to parse argv[n] and look for the ">" character. If found it sets the exclusivity flag to TRUE, dupes argv[locationof>+1], replaces the > with the null character, and returns TRUE.

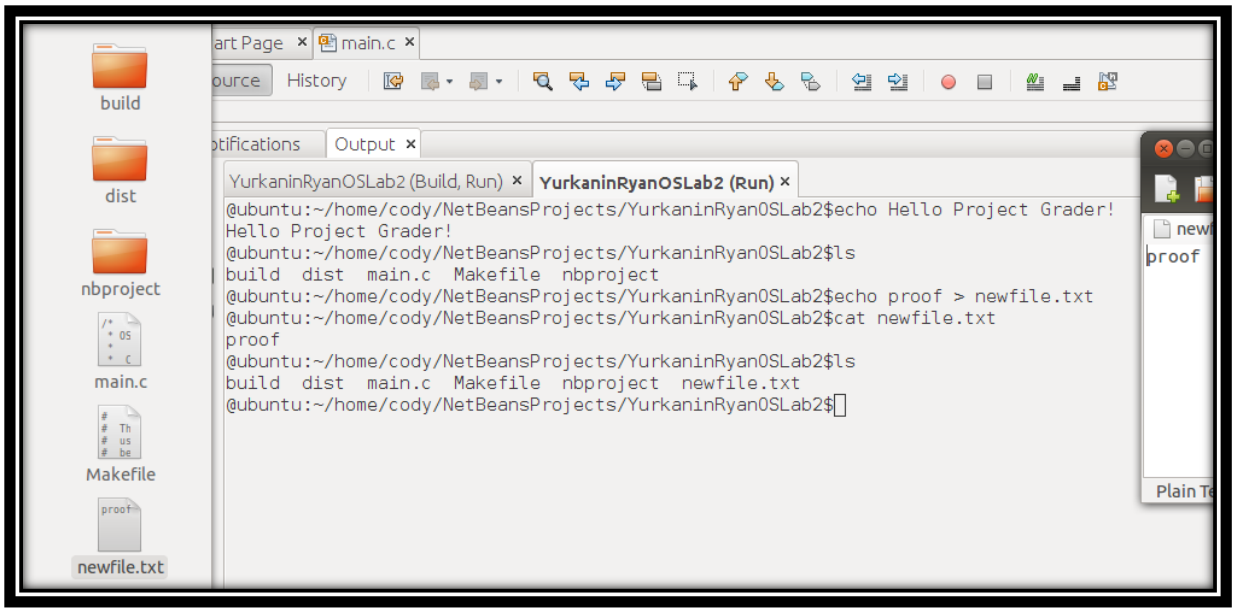
checkInputRedirection() – Function I created, that is practically an exact copy of checkOutputRedirection() however, instead of looking for the ">" char it looks for a "<"

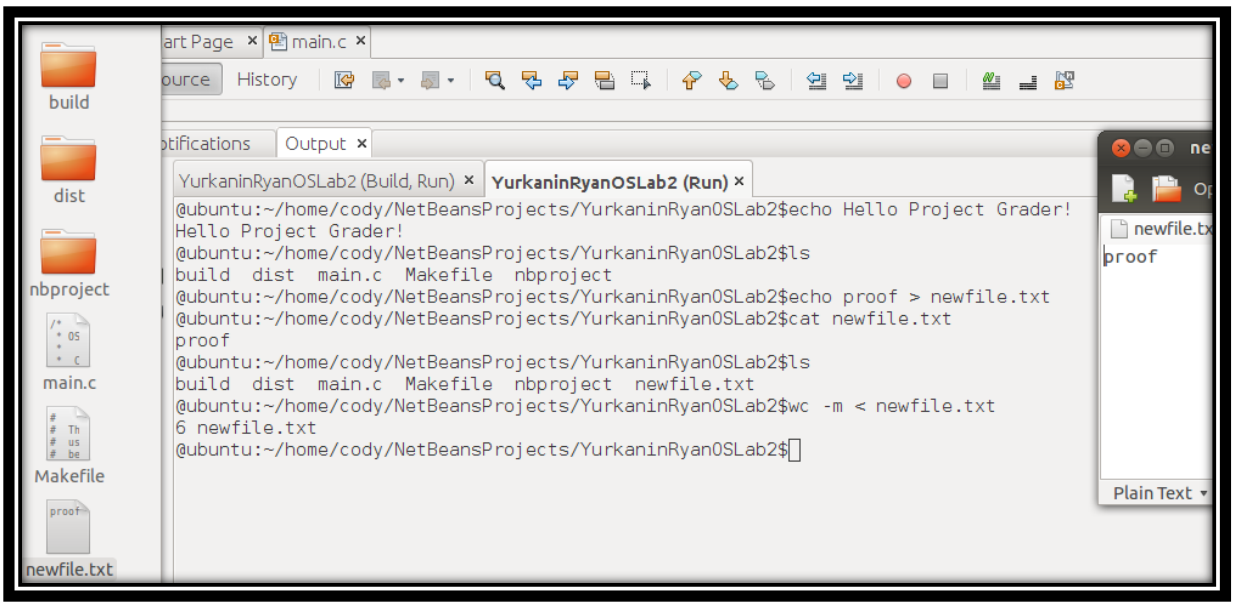
checkPipe() – Function I created, it's purpose is to parse through argv[n] and look for a "|" character. If found, it does the same stuff as checkOutputRedirection() and checkInputRedirection(), as well as store the location of the pipe to later be replaced by the NULL character.

TESTING LOG, BUG NOTES, AND PROOF OF SOLUTIONS

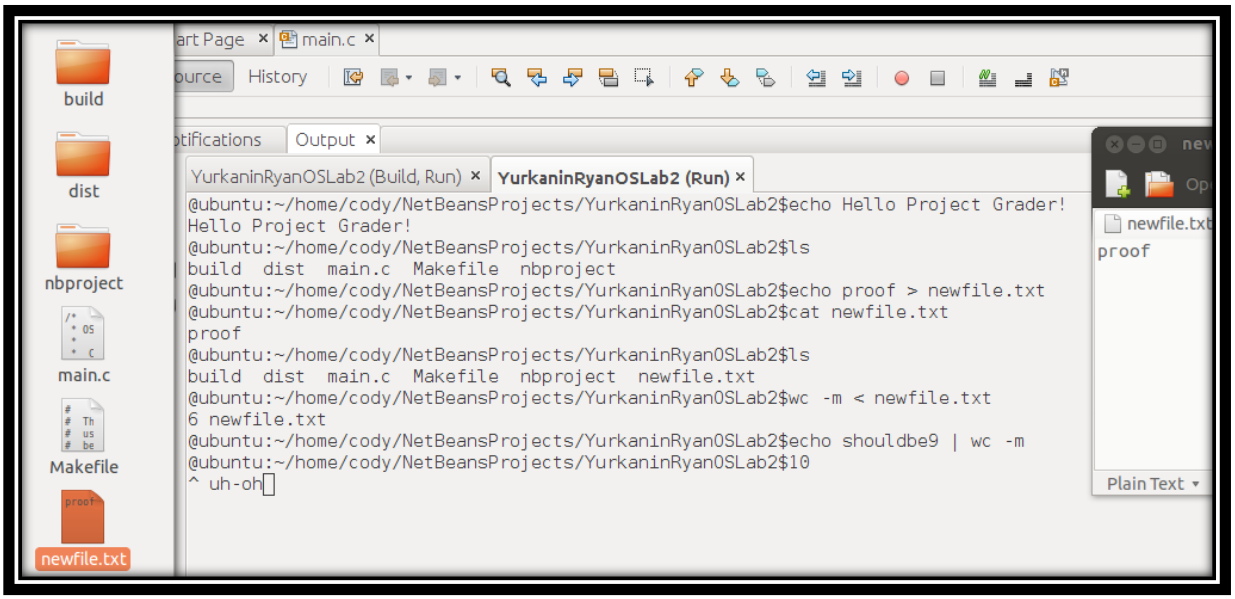


Solution A: The first part of my code that I tested was to see if it could even solve Problem A. It was able to meet all the requirements of Problem A and I didn't notice any bugs. Above in the screen shot, you can see that it proper executes "echo" as well as adding the arguments "Project, Grader!". The "ls" command also properly functions, and gives the directory that my shell resides in as evidenced by the File Manger on the left.

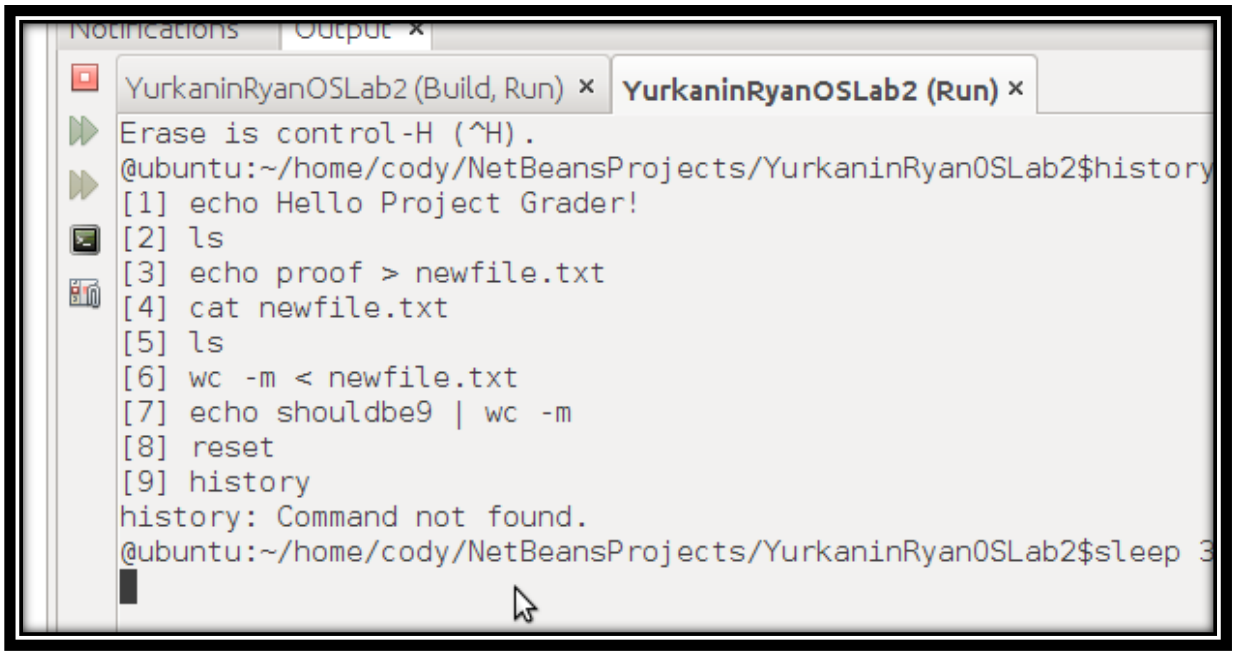




Solution C (I/O Redirection): My shell also properly solves Problem B as seen in the two screenshots above. In the first screenshot, I use output redirection to "echo" "proof" into a new text file aptly named newtextfile.txt. I then use input redirection to get a word count of the contents of newtextfile.txt. As a note, my shell can do redirection with concurrency, but it is difficult to prove so I just left it out until the end. Also, my program can't handle both a pipe and a redirection symbol in the same input. It simply ignores the pipe, and handles any I/O Redirection.



Solution C (Piping): My shell can do piping, however, towards the end of its development I noticed a bug. While it always gets the proper result that a pipe should get, it does not wait for the child before continuing, giving a weird looking line as an answer. What I should of done is fork() twice instead of having a parent be one command and one side of the pipe, that way I would have been able to wait for both children to return before continuing. This mistake has shown me the entire architecture of my shell could be vastly improved to eloquently handle many pipes and I/O redirection symbols.



Solution B: I saved this one for last because I knew that it would be the hardest to prove in the documentation. My program is able to switch into a concurrency mode simply by adding an ampersand anywhere at the end of the arguments list (attached to the final argument, or its own argument). I tried to take a screenshot as fast as humanly possible after entering both commands above to prove this. Also, this showcases the extra history function that I added to the shell. I found it very useful throughout all of my testing.