Introduction to Unix Systems – Project Paper

- **Login:** My login function is one of the first functions called when the shell starts up for the first time inside the main method. My login function returns a username if the login details are correct and a NULL value if incorrect. I therefore set up a while loop inside the main method that repeatedly calls the login function until a username is returned to allow for multiple login attempts. Inside the login function, the username and password are collected through the scanf function and are stored inside character arrays. The password file is then opened and read using the getline function. To get the usernames and passwords stored within the file, the strtok function was used. This was followed by a comparison of the two strings to check the validity of the login details. In terms of adding users, the new users' username and password were first collected from the command line. Then following the same process mentioned previously, the password file was read and the usernames were compared to see if the user already existed. If the user did not exist, then the password file was appended to using fprintf. Subsequently, a directory was created for the user within the home directory and a. tsh_history file was placed inside.

- **Command Evaluation:** My eval function first called the parseline function in order to pass the command line string and create an argument array. Subsequently, if there was only a single argument inside the argument array and the argument matched one of the built-in commands then control was passed off to the built-in command function. If the command line did not contain built-in arguments, then the eval function began the process of forking a child to run the command. Before forking a child, sigprocmask was called in order to block any interfering signals (unblocked later before execve). Once the child was forked, it was placed within its own process group and a proc directory and status file with its relevant details were created. Following this, the execve function was called to run the first argument within the command line along with any flags indicated. Outside this structure, the parent of the child process used sigprocmask again to add the job that was being executed to the job list. The parent was then made to wait before returning, if the child process being executed was a foreground job. This was achieved through the waitfg function that checked to see if the pid of the foreground job had been reaped by the SIGCHLD handler. If the process was a background job, the parent returned immediately and prompted for the next command.

- **Built-in Commands:** The quit built-in command deleted the proc entry of the shell that was created upon initialization (using remove and rmdir) and then exited the code. The logout command iterated through the jobs list to see whether there were any suspended jobs. If not, the proc entry of the shell was similarly deleted and the code was exited. The history command was implemented using a history array of size 10 that stored command line entries. A global variable was maintained to keep track of the index of the history array that was being pointed to at any point in the program. Each time a command line was evaluated using the eval function, a function was called named update_tsh_history. This function placed the command line content inside the history array at the index indicated by the global variable and subsequently incremented the global index variable by one followed by modulo ten. This way, the global index variable never exceeded ten and also overwrote the oldest commands when more than ten commands had been run. Furthermore, each time the history array was updated through update_tsh_history, its contents were immediately written to the .tsh_history file (overwritten each time). Additionally, if the contents of. tsh_history were non-empty after logging in, the file was read and its contents placed inside the empty history array so that previous session commands could be viewable. In terms of the history command itself, it relied upon the global index variable. If the global index variable pointed to an empty entry in the history array, it meant that less than ten commands had been executed, which meant that all commands until the index could be printed out (oldest to newest). If the entry was non-empty (more than 10 commands had been executed), this meant that the history index variable was pointing to the oldest command that had been stored. Therefore, all commands were printed from that index onwards until all ten commands had been printed. For the !N commands, a similar logic was used. The number N that was entered was first obtained. Subsequently, if the global history index variable pointed to an empty entry, it meant that the number aligned with the indices of the history array and that the appropriate entry could be found directly. If the variable pointed to a non-empty entry, then the actual position of the command inside the history array could be found by (N + (history_global_index -1)) % 10. Once the command was found in each case, it was evaluated once again using the eval function. The bg and fg commands were executed by passing control to the do_bgfg function (described later). The adduser command was executed using the add_user function (logic and workings described in the first login section above).

- **Proc:** A prof directory and status file was created for the shell upon login. Since the shell is the session leader, a global session_leader_pid variable was set to the pid of the shell. The values for the process id, parent process id, and the process group id were obtained from the getpid(), getppid(), and getpgid() functions respectively. These values were then written to the status file within the shell's proc directory along with the appropriate username and 'Ss' status to indicate a session leader. Subsequently, inside the eval function, every time a child process was forked, a proc directory and status file were created. The details about the process were obtained using the same functions mentioned. If the process was a foreground process then the status was determined to be 'R+' and 'R' if it was a background process. Proc directory and status file deletion was handled by the SIGCHLD handler which deleted the two items after reaping a child process. Deletion of the shell proc directory was handled by the built-in logout and quit commands. Changing of the status of a process to stopped was meant to be handled by the SIGSTP handler which upon stopping a process would edit its status file to indicate the changed status. Likewise, the bg and fg built-in commands would also go into the needed proc directory and edit the status file to indicate that a process was running again either in the background/foreground.

- **Job Control:** The SIGCHLD handler was implemented using the wait function inside a while loop that ensured that there were no more child processes and that all had been reaped. The handler handled proc entry deletion as well. Each time a child was reaped, it changed a global foreground pid variable to match the pid of the reaped child. This variable was used by waitfg which when called began a while loop that executed endlessly by sleeping until the global foreground pid variable was equal to the argument pid that it was given. Waitfg was used to ensure that a specific foreground child process had terminated before issuing the command line prompt. The SIGINT handler first obtained the pid of the foreground jobs by searching through the job list using the fgpid function. If there was a job in the foreground, it then proceeded to send its entire process group a SIGINT signal using the kill function. It then deleted the job from the job list as well as its proc entries. Likewise, the SIGSTP handler obtained the foreground job if any. It then sent the entire process group a SIGSTP signal using the kill function. The handler was then meant to change the status of the job to stopped inside the job list as well as edit the proc status file to indicate the change. The do_bgfg function were meant to obtain the job indicated by the job pid/jid from the job list by searching using the appropriate getjobjid/getjobpid function. After obtaining the job, it was meant to change its status inside the job list, update its proc status file to running, and actually continue the job in the background/foreground by sending it a SIGCONT signal using the kill function.

1. The most challenging aspect of the project was tracking segmentation faults. Given the number of functions/lines of code involved in the project it was very difficult to tell where exactly the code had broken down. These segmentation faults therefore led to several hours of manual de-bugging (printing statements at different points) and tracing lines of code to find a solution. The other challenging aspect of the project was allocating work between waitfg and the SIGCHLD handler function. It took me a good amount of time to figure out how they would both interact with the shared global variable and whether to use wait/waitpid in the handler with the appropriate arguments.

2. To enable the tsh shell to only be able to access and modify files that have been physically mounted we need to first create a file hierarchy for the tsh shell that mimics the tree file hierarchy of a UNIX system. The local file hierarchy would be inside the project folder and would be rooted at a particular point (similar to "./" in a UNIX system). This file hierarchy would be created upon the initialization of the shell.

   We could then implement the built-in mount command that takes the first argument to be a file path in the broader operating system and the second argument to be a file path in the project file hierarchy. The mount command would copy over the file contents into the local project file system. The command would also ensure that contents are mounted in the correct location and that the arguments provided are correct. The unmount command would take in an argument of a file path that has already been mounted into the local project file system. There would be a check to see if the argument provided was correct. It would then unmount the file contents so that they are no longer accessible by the tsh shell.

   In terms of actually running commands, before the eval function executes a file found in a particular location, it would have to first run a check to ensure that the file is within the bounds of the local project file system. If the file is within the local project scope, only then would the command be executed. If the file was found to be within the file system of the default broader bash shell, the command would be rejected.

3. To implement pipes, we would have to change how command lines are interpreted within the program. The first assumption made is that piped command lines would provide the full file location (/bin/ls rather than simply ls) for each of the commands within the line. Taking this assumption into account, changes would then have to be made to the parseline function.

   The parseline function would have to change in a way such that the command line argument string that it receives is first split based on the pipe operator. The parseline function would then take each split part of the command line string and create an argument array (much like it does now). After doing this for each part of the command line, it would then return an array of argument arrays to the eval function.

The eval function would then start to loop through the array and create a child process that executes the first argument array. Once this child process terminates, the eval function would then have to take the stdout output that was formed by the child process and append this output to the next argument array that is about to be executed. This would likely involve a helper function that copies over array contents and dynamically allocates array space based on the stdout.

This process of executing the next argument array along with the previous stdout would continue until the entire command line has been evaluated. A global flag could be implemented that ensures that the stdout of the final command in the pipe is the only one that is printed to the console. Similarly, changes would have to be made to the signal handling code logic that ensures that when a SIGINT or SIGSTP signal is received, the entire processing of the eval function ends/stops rather than only the process associated with a single section of the piped command line. The shell must also ensure that each section of the piped command line is run in the foreground/background as specified.