Running Executables (with & without arguments):

We added the main functionality of a shell by using the execv() version of the build in execute commands. If the input command is not in absolute path format, we check if we have access to execute the input command in the current subdirectory. If this check failed, the command was rejected with a command not found error. If the input command is in absolute path format, we search for it in the path before executing. In either case, if we found an executable that matched, we executed the command along with the list of arguments that came with it. This function forks a child process in order to carry out the execution and then executes the command with execv.

To test, we spent some time ensuring our PATH searching function was working correctly by entering a command, and printing out the returned PATH/command string, if we found a valid executable at that location. Once this was working, we tested specific command such as find, ls, and a few others. We also tested non absolute path format executables. For this part of the testing, we wrote a dummy program that terminated after 5 seconds waiting.

Setting PATH and HOME Environment Variables:

We were able to do this by using the built in function setenv(..), which took the name of the variable we wished to modify, the value we wanted to set it to, and a flag to determine if we wanted to overwrite the current value or not. After setting one of these variables, quash then outputs the variable and its new value. Testing this was obviously straightforward, since quash outputs the new variable value we were able to confirm that the value of PATH or HOME had indeed been changed to the specified new value.

Echo PATH and HOME Environment Variables:

The echo function works as intended for this project, but isn’t as in depth as a normal echo function. Our echo function simply prints out everything that was inputted by the user, except for the cases where that input matched the strings “$HOME” or “$PATH”, in which case, quash would output the value of the environment variable by using the getenv(...) function.This function was also easy to test as we simply had to write “echo $HOME randomText $PATH”, and the output would be the value of the home variable, randomText, and the value of the path variable. This function also works for text ever, so edge cases were not a problem here.

Changing Directories (cd):

Changing the working directory with the command cd was fairly straightforward. Since the requirement for changing directories was to accept cd (with arguments) as well as cd (no arguments) we accounted for the two different cases with a simple if-check. If there were no arguments passed along with the command, the user was routed to their HOME directory using the chdir(path_to_home_directory) function call. On the other hands, if arguments were passed with the command, the chdir() function targeted the given directory. If the directory didn’t exist in either case, the user was shown an error message acknowledging that the directory doesn’t exist.Again, testing this function was straight forward. We obviously started out in the directory where quash was located. If we typed “cd” with no arguments, we arrived at the location specified in the HOME variable. To test this case further, we then changed the home variable to something else using the set command, and typed cd again. Assuming the new HOME variable was a valid location, we were taken to that location. If an argument was sent in with this command, quash tried to change directory to that location. If it was a valid location, the function succeeded. If not, an error message was printed out.

Printing Working Directory:

In order to print the working directory, we utilized the the getcwd function. The getcwd simply stores the current working directory as a string and from that string we could print the string to the console. This was tested multiple times by calling the pwd command, confirming the correct directory was returned, then moving around to different directories and calling the pwd command again. Every time, the correct working directory was returned.

Exit & Quit to exit Quash:

These two key works (“exit” and “quit”) exit the program. Since “quit” was already set up for us in the base version of the program, all we had to do to add “quit” was another string comparison check in the same if-block in our

command handler. If either condition was met, the program announced it was exiting, then terminated the instance.Testing Exit & Quit was straightforward, if our shell terminated when given ‘exit’ or ‘quit’ then the function was working as desired.

Foreground & Background Processes:

In order to allow quash to create processes in the background, a special method was needed which forked and didn’t wait for the process to complete. We accomplish this in our function “execute_in_background(...)”, which forks off a child process that then executes the command normally. This simple method allowed us to continue the parent process without waiting for the child process to continue. As always, foreground processes were built in as the main (normal) function of quash.Testing this function was simpler than we originally thought. If I create a process with the command ./h (a dummy program to wait 5 seconds before exiting), it prints out the newly created process jobid and pid, and upon completion, prints out a message saying “jobid processid cmd” has completed. We also tested this with a nested instance of quash. So, if we entered a parent process of quash, started quash in the background (it would still request input), upon exiting the nested child process of quash, the parent process would print out a “finished” statement with the child’s pid.

Printing Current Running Jobs (background processes):

As mentioned in earlier sections, we used our own struct “job” which we defined in our class. This struct saves the newly created process id, jobid, and command of a background process. We then store all the jobs in an array of job structs, and keep the number of jobs in a global integer called numJobs. In order to print out the list of jobs, or background processes, we loop through our array of jobs and print out the job number, the process id, and the command associated with the job, as specified in the assignment sheet.It’s worth mentioning that this is the only function we did not fully implement. In the submitted version of quash, the program does not remove terminated/killed background processes from this list even though they are actually terminated correctly. This means that all processes created during the lifetime of the quash shell are stored in the job list.To test this function I created several instances of the program “h” by typing in “./h &” several times, after words, I printed out the job list and could see each individual job by jobid and process id. Once each process finished, a finish command would be displayed as well.

Child Process Inheritance (environment variables):

Child processes in any of our commands that forked retained the environment variables from the parent because of the properties of fork(). Forking a child process creates an exact copy of the currently running process, so any environment variables stored in the parent process would be copied into the child process.We were able to test this by running a nested quash process. First, we opened quash with ./quash, then opened a nested quash shell with the same command from inside the parent quash. We then echoed the $PATH and $HOME variables, and they were the same as the parent’s variables. To test this further, we exited the child quash instance, changed the environment variables, and tested again. Lo and behold, the environment variables were the same in the child process as the newly changed variables in the parent process.

Quash I/O Redirection:

Quash implemented I/O redirection so that users of the shell had the ability to both write command output to files as well as read already existing file content and process this content with their desired command. For example, if a user was to type the following command: ls > abc.txt, the shell would store all of the files/directories located in the current working directory into the abc.txt file. If the abc.txt file didn’t exist before running the command, the shell would create the file for the user and store the data in the file. If the file already existed before execution, whatever content that was originally within the file would be replaced by whatever value ‘ls’ returned to stdout. This worked simply by opening the file in quash.c and designating the file to writable and assigning stdout to the file itself. On the other hand, the user could change the direction of i/o by typing in a command such as: sort -n < numbers.txt. This simply assigned stdin to the file desired to be read from and allows the function on the left hand side of the ‘<’ to run on the data from the file on the right hand side. The I/O redirection was tested by running commands, such as ls, that

would generate data to write to files and, on the other hand, running commands such as sort that handle incoming data. It’s worth noting that running commands that use files as input require that the target file actually exists before running the command.

Implementing the Pipe (|) command:

In order to implement the pipe command in quash, we had to utilize pipes very similarly to how we utilized the pipes in Lab 3 of this course. To parse the command, we parsed the command on the left side of the | and stored this in a string, then parsed the right hand side of the | and stored this in a string as well. We then set up our single pipe that was able to connect the two processes. We ran the command on the LHS of the ‘|’ and wrote the results to the write side of the pipe. We then read out of the read end of the pipe and executed the command on the RHS of the ‘|’. After completing the operation on the read end of the pipe, the implementation of the pipe command was complete (single pipe in one command). In order to test the implementation of the pipe command we implemented, we ran multiple tests with different commands piped together. For example, one of the tests we used to make sure the pipe function was working properly was: find quash.h quash.c | sort, which would return quash.c quash.h as opposed to just quash.h quash.c which would be returned by running find alone.

It’s worth mentioning that we overwrote this functionality in favor of the extra credit version, which supports anywhere from 0 to n pipes. For a further discussion on this, please see the multiple pipes entry below.

Support of Multiple Pipes in one Command:

As mentioned above, we were able to implement multiple pipe support by modifying our original pipe function which only supported one pipe. This new pipe command still tokenizes the string by breaking it into two halves, the command up until the first pipe character was stored in a char array, and the rest of the command after this pipe character was also stored in a new char array. Next the left side (before the pipe) was parsed into an array of char arrays, which was then passed into the execv(...) function which actually ran the program. The right half of the command (after the first pipe) was then sent to our normal command handler to run as normal. This allowed for a chain of pipes to be executed because the input from the first command was still send to the second command, even though the second command might start a pipe of its own. To test the multiple pipe functionality, we conducted several tests with several different commands. One of our first tests was to run the command “find quash.h quash.c | sort | sort --reverse”, which would first send the list “quash.h quash.c” to sort, which would be sorted to”quash.c quash.h”, and then this would be reversed. This showed promising results, so we tested multiple pipes with a few other commands. One of the other tests we carried out was type “cat numbers.txt | ./A | ./A | ./A | sort”. The program “A”, which we wrote, took in three numbers, and multiplied them all by two. So, if we ran this three times in a row, the numbers would effectively be multiplied by 8, and then sorted. After getting correct output from this, as well as other tests, we were satisfied with the working of the multiple pipes.In the end, we were surprised at how simple this function was to implement, and how little we had to change from our original function to get it working. In fact, this implementation even cleaned up our original pipe function because we didn’t have to store so many variables for the second execution since we were just passing the whole right side of the command.

Delivering ‘Kill’ Command to Background Processes:

To implement the ‘Kill’ command to kill background processes, we used a function that took in the ‘kill’ keyword followed by the specified background process to be killed. Since we already had the capability to run processes in the background and stored these background processes in a global variable called jobList, killing the specified process was easy. We searched the jobList variable to see if the specified process actually existed in the list, and if the job was found in the list, we used the kill() function provided in C. This effectively killed the specified background process and accomplished the task at hand. In order to test our kill function, we ran a few background processes, printed the list of background processes running, and then specified a background process to be killed. Since our print_jobs() function didn’t remove terminated/killed jobs from the list as it should have, we confirmed that the kill function was working because the confirmation that the background process had finished never arrived. Obviously, when the process had time to run and compete, the shell would alert the user that the process had

completed, however we knew that we successfully killed the background process because that alert never arrived after we ran the kill command.