Haonan Hu - 2863545
Chance Penner - 2870583

Quash Report

Before testing, please look at the README. It states:

Ensure that the testing files are made properly. DO NOT copy paste test files. You MUST create a file within vscode and then copy the contents over. If you use a pre-made text file, it will NOT work. Make sure there is a trailing newline at the end as well. If you do not us vscode to create the test file, the input will be weird. This is because many different editors use different styles of saving whitespace, newlines, etc., so please just use vscode to create a new file and then you may copy the text into that file and add a newline at the end.

To summarize, basically when using textEdit and Atom and vscode, we were getting mixed results with reading from files. We found the best solution was to stick with just vscode. For example, when using the premade 'command6.txt' and 'command7.txt' files, they parsed weirdly, either failing at the end or not reading the last command in the file.
Instead, just open vscode and create a new file there. Then, you can copy over the contents of a text file into the newly created one and save it. We found that having a newline at the end made everything work better. For example:

```
uname

```

Notice how there is a newline at the end! This works best. We have sample test files in the directory, labeled 'test1.txt' through 'test13.txt' (note, these are copies of the given tests 1,2,3,5, alongside modified command6.txt and command7.txt. And then some we made).
Some of the copies of tests 1-7 had to be modified to work on our machine. For example, the original command7.txt was just:

```
file7
set MYVAR="NEW_ENV_VAR"
file7
quit
```

But this only works if you first have file7 in your path, so we changed it to:

```
set PATH=/mnt/c/Users/Chance/Desktop/eecs-678-quash
file7
set MYVAR="NEW_ENV_VAR"
file7
quit
```

Again, note the newline at the end! This way, file7 was in the PATH and would run as intended. Another way to do this would be to just add '.' to the PATH and then cd to that directory, or cd to that directory and change it to use ./file7 instead. We just chose the way we thought was best.

Run executables without arguments

- Quash should be able to run commands without arguments. (ex. quash> command)
- Once we retrieve the input from the user, we pass it into our ***execute*** or ***runProgram*** (Note:***runProgram*** function runs executable by "./") function And run it by calling ***execle*** function in unistd library. The environment will be Inherited from the user's initial environment.

Run executables with arguments

- Quash should also be able to run commands with arguments. (ex. quash> command arg1 arg2 …)
- Once we retrieve the input from the user, we need to parse the whole string and separate them by calling function ***strtok*** and set the delimiter to a single space(because all arguments are separated by a single space). After tokenizing all parts of the input string, we save them into a string array and pass both input and tokens in to function ***execute*** or ***runProgram*** and run the command by calling ***execvpe*** function in unistd library. The environment will be inherited from the user's initial environment.

set for HOME and PATH work properly

- Quash should be able to allow user set HOME and PATH variables. (ex. quash> set HOME=/myhomeDir or quash> set PATH=/path1:/path2:.....)
- By achieving this, we try to identify if the user's input contains the command "set HOME=" or "set PATH=", if so, we parse the input to get the right part of '='.
    - If the user tries to set HOME, we check the user's input directory existence , if exists, we set the HOME directory to the user's input directory, otherwise, we report error.
    - If the user tries to set PATH, we parse the user's input path by calling function ***strtok*** and set the delimiter as ":". Then we clear the quash's path and save the tokenized string into the cleared path variable.
    - We do not check if the user's input path is valid or not before we overwrite the path, because the function ***execute*** and ***runprogram*** will check the path before executing the command.

exit and *quit* work properly

- Quash should be able to quix and exit successfully when the user's type in "quit" or "exit" in the quash command line.

cd (with and without arguments) works properly

- Quash should be able to handle changing directories. (ex. quash> cd ; quash> cd ./ ; quash> cd .. ; quash> cd /myDestinationDir)
- By achieving this functionality, we first matching if the user input "cd" in the command, if so, then based on what is followed by the ***cd*** command, we have several cases:
    - Command is "cd": in this case, we just need to set the current working directory to be the HOME directory.

    - Command is “cd ./”: in this case, we do not need to do anything because ./ references the current directory, and changes the directory to be the current directory will simply not change anything.
    - Command is “cd ..”: in this case, we will parse the current working directory string by calling the function ***strrchr*** and set the current working directory to become the directory without the last directory.
    - Command is “cd /userDestinationDir”: in this case, we first need to verify whether the directory that the user inputs is valid or not. If the directory is valid, we make the current working directory equal to the user input directory and call function ***chdir*** and switch to that directory. If not, we print the error message to the shell and tell the user that the directory does not exist.

<u>PATH works properly. Give error messages when the executable is not found</u>

- The initial PATH should inherit from the bash terminal’s PATH and save in the quash as an individual variable. Every command that the user wants to execute should check if the executable is listed in the PATH variable or not. An error message will be printed to the shell if the executable is not listed in the PATH.

<u>Child processes inherit the environment</u>

- The child process will inherit the environment that created it. This is done by using the execle and execvpe commands, and passing in our envVars (environment variables) variable. The envVars variable first copies the current environment from main’s envp variable. Then, whenever the *set* command creates a new variable, we add it to envVars and increment its size. That way, when we call execle or execvpe, we can pass in envVars as the environment! We tested this using the *test7* provided to us, which attempts to print an environment variable that does not exist, which of course doesn’t work, but then it creates that variable and then prints it out successfully. Now all child processes will inherit from the environment that spawned them, and be able to add new environment variables as they see fit.

<u>Allow background/foreground execution (&)</u>

- After parsing an ‘&’ at the end of an input, we know that we need to execute whatever is on the left of that ‘&’ as a background process. So, we call our backgroundExecute() function, passing in the input, workingDir, path, and numPaths variables. Then, we need to parse our input as usual to get all the input argos and whatnot. After doing so, we will add a new job to our jobsList, which is an array of job structs that hold the jobId, pid, and command. We add the command that was passed in by the user, and then we fork. In the child, we print the necessary information that I describe in the next section. In the parent, instead of waiting, we update the pid and jobId and the jobsTotal. By not waiting, we are allowing the process to execute in the background. Of course, we also later check for any processes that finish executing so that we can update our jobsList accordingly and avoid zombie processes by checking for the return status. That way, when someone runs the *jobs* command, we only print the active jobs.

<u>Printing/reporting of background processes, (including the *jobs* command)</u>

- When running a process in the background, we create a struct of the process which holds the jobId, pid, and command. That way, we can print the information regarding the process. In our backgroundExecute() function, after creating a background process, we

will print that the process is running in the background, and after it finishes executing, we print that it finished. Within these statements, we are able to print the jobId, pid, and command as necessary, by accessing the struct's variables.

- Also, we add them to our global jobsList so that when we run the *jobs* command, we can print all active jobs. In order to remove a no longer active job, our removeJob() function will get the return status of any recently executed jobs, and look those up by matching the processId and childPid. That way, we know which one in our jobsList to remove. This ensures we only print active jobs when executing *jobs*.

Allow file redirection (> and <)

- In order to do file redirection, we first parse the input checking for a '<' or '>'. Once one is detected, we call the redirect() function. This function finds the index of the '<' or '>'. Using that index, it can then parse the left side and the right side.
- If the redirection symbol was a '<', then we will first check if we are executing with "./" or not. If so, it will open the file that was declared on the right side of the symbol for reading, use dup2 to redirect the file to STDIN_FILENO, and call our runProgram() function to then run the desired program. Afterwards, we close the file. If we are instead executing a command (so there is no "./" before the executable or command), then we do the same thing, but instead of calling our runProgram() function, we call our execute() function.
- If the redirection symbol was a '>', then we will also still first check if we are executing with "./" or not. If so, we open the file for writing, call our runProgam() function, and redirect the output with dup2 to the file. Then we close the file. If there is no "./", then we do the same thing, but instead of calling our runProgram() function, we call our execute() function to execute a command or executable within our PATH.

Allow (1) pipe (|)

- If the input includes a '|' character, then we will execute the makePipe function. This function parses the left side of the pipe and the right side of the pipe. Then, it will fork so that in the child process we can either run a program if the left side includes a "./", or execute a command if there is no "./". We use dup2 to redirect the output of the left command to the write end of a pipe. This will then allow us in the parent process, after waiting for the child (aka the left side of the pipe), to redirect the read end of the pipe into STDIN_FILENO when executing the right side of the pipe command. Of course, we also are careful to close the ends of the pipes within the child and parent process.

Supports reading commands from prompt and from file

- Reading commands from prompt was just simply having a case at the very end where we parse the user input and assume it is a command. Then, we attempt to execute that command. If that command exists within one of the PATHS, then it is successful. Otherwise, we print an error saying that command doesn't exist.
- Running commands from a file is done so from bash. The user will type a command similar to this:
  bash>./quash < test1.txt
  This will redirect the input from the test1.txt file, so then instead of our "input" variable being used for user input, it will instead read from the file. That way, all of our input

variable parsing works just the same as before. We also make sure the user gets access to the terminal again, in case the test file does not execute a quit command at the end.

Testing

- We tested quash by running the program with many different inputs. We would try as best as we could to “break” anything. Of course, we used some “printf” statements to double check that any input was parsed correctly.
- Aside from that, the more organized methods we used to test came after we implemented the ability to read a command from a file. Once we could read from a file, we would create test files that would implement commands for us. That way, we could easily test stuff with just file redirection from bash into quash. For example:
  bash>./quash < test1.txt
  This made it quick and easy to create tests and always double check that we didn’t break anything when adding new features, since we could just run all of our tests again.