Owen Mellema
Zach Freund

Quash Implementation Quest

1. Run executables without arguments (10)
    - To run an executable without arguments we have to parse the environment variables that we received for PATH and HOME into global variables we defined as env_path & env_home. Then a function that handles searching those variables if the executable's path was not given in absolute format and then follows the search with creation of a process using fork() and execvpe() using the true path of the desired executable if found.
2. Run executables with arguments (10)
    - To run an executable with arguments we follow a similar procedure to running an executable without arguments. The difference is the parsing of the arguments that comes after the executables path in the call. These arguments are passed as a char ** to execvpe() which then creates a new process with those arguments passed in.
3. set for HOME and PATH work properly (5)
    - In order to allow the user to set the values of HOME and PATH environment variables we parsed the user's input and if in valid form modify the global variables env_path & env_home respectively. These variables serve as our environment for the execution of the program all of the time.
4. exit and quit work properly (5)
    - The exit and quit commands were caught as input from the user and then we end the program by causing our top level while loop to have its expression result false therefore breaking us out of the while statement and ending the program.
5. cd (with and without arguments) works properly (5)
    - For the change directory command without arguments the command chdir() was used with the global variable representing our HOME environment variable was passed in as it's only argument. This results in the chdir() system call changing our current directory to the path stored in env_home.
6. PATH works properly. Give error messages when the executable is not found (10)
    - During parsing of user input the global variables env_path & env_home are searched and used as possible prefixes to the relative path given this was done using many cstring functions just as all other parsing had been done previously. The combinations of prefix & relative path are validated to be existent using the access() function which checks if a file is accessible to the current linux user.
7. Child processes inherit the environment (5)
    - The main idea for passing the environment onto child processes involves execvpe() which allows for a 3rd argument that many other exec() functions do not have. This third parameter is used to pass an array of c-strings of the environment variables stored in the form of '<VAR>=<absolutePath>:<asbolutePath>...'. The process created then inherits these environment variables upon execution.
8. Allow background/foreground execution (&) (5)
    - Executing processes in the foreground is accomplished by having the parent wait on the child process to finish and then continuing execution after. For background execution

thought we must not wait on the child process to finish but instead store the PID returned by the fork() call in an array stored within the parent process and print a message that the returned process is running in the background. This array will be composed of all processes executing in the background at that current time.

9. Printing/reporting of background processes, (including the jobs command) (10)
    - The jobs command was implemented looping over the array of background PID's and printing its contents in the desired format listing the PID's and a message stating the entry is running in the background. For reporting that a background process has ended we had to catch the signal SIGCHLD when it was produced by a terminating background process using signal(<signal>, <handlerFunction>) which calls a designated function upon catching a given signal.
10. Allow file redirection (> and <) (5)
    - Output redirection was achieved by forking a child process when the file redirection symbol(>) followed an executables path. Then using the freopen() call to reopen the stdout as the given file path after the redirection symbol. This allows for the process to run without changing the current quash execution's file descriptors. Then the command preceding the redirection symbol is finally executed.
11. Allow (1) pipe (|) (10)
    - The piping of a single commands output to the input of another single command was accomplished by creating a pipe and passing this pipe to both of the child processes which is then used along with a flag variable designating whether it is supposed to write to the pipe or read from the pipe. Each process is then executed from the parent in the order it is input and the read block on the second command causes it to run only when the first process has produced output for it to receive.
12. Supports reading commands from prompt and from file (10)
    - The execution quash program using input redirection from linux bash will pretty much take care of this for you. Inputting each line of the file given into the quash execution's stdin.

Unsupported/Unfinished

- Input redirection was not achieved successfully.

Testing Quash

- All of our testing was done by directly executing quash and inputting various commands. The most difficult areas to test were the background execution which required us to write a function that looped in some way to allow the process to exist for long enough to validate its existence with the jobs command executed by hand within quash.