Sean Boeck, David Gier EECS 678

**Implementation**

**Project 1 Report**

Quash creates a clone of a shell by reading user input and executing commands. The program prompts the user upon running to enter in any normal command and then processes the string of input. Using the strtok() function Quash reads the inputted line word by word looking for special tokens to denote certain jobs using the parse() function. Once read the list of created jobs is then executed as commanded using the execute() function.

**Features**

- Run executables without arguments (10)

  Executables can be run by typing in the name of the executable. The user can give the absolute path (./program) or the executable name (program). If the executable name is given, all locations within PATH will be searched for the program. Once the executable is found, quash attempts to run it with execvpe(), passing in the arguments and environment. If this fails, an error is printed and that particular process exits. Quash continues to run, however, and other commands can be issued.

- Run executables with arguments (10)

  The first argument to miss the pre-built commands is taken to be an executable. Any others are taken to be arguments for the executable if they too are not pre-built commands.

- 'set' for HOME and PATH work properly (5)

  Set works by finding the current working directory and appending the desired path to that path to make all paths in HOME/PATH absolute. Once the absolute path is gathered into a string, quash uses the system call 'setenv' to set the environment variables. Only one HOME can exist, so setting it overrides the previous HOME. Multiple locations can exist in PATH, so setting PATH appends the new path to the end of the PATH environment variable.

- 'exit' and 'quit' work properly (5)

  'exit' and 'quit' are pre-built commands that set a variable called exitbit to 1. Having the exitbit set to 1 allows the program to exit the while loop that continually prompts the user for input and finish the execution of 'main.' Earlier we were having problems where 'exit' and 'quit' only left the current processes

rather than the entire program. Now we make sure that every process closes when it should. There will be no foreground processes running alongside the process implementing ‘exit’ or ‘quit,’ so this process exiting ends the program. Background processes can continue to run, however.

- ‘cd’ (with and without arguments) works properly (5)

  ’cd’ uses the system call chdir() to change the directory. With no arguments, it uses getenv() on the HOME environment variable and changes the directory to HOME. With arguments it changes the directory to the given path. If the directory does not exist or changing directories fails for another reason, quash prints an error.

- PATH works properly. Give error messages when the executable is not found (10)

  When a command is executed, it first checks to see if that executable is in the current directory using the access() function from unistd.h. If not, it uses getenv(PATH) and splits PATH on colons to search all directories within PATH. If it is not in any of these places, it produces an error message.

- Child processes inherit the environment (5)

  Every child process that is produced in order to run a particular executable in the foreground or background inherits the environment variables that exist in the main program. This is done using fork() to create a child process that the parent waits for it to execute (if it is run in the foreground). The command execvpe() ensures that the environment is passed and used with execution of different programs.

- Allow background/foreground execution (including the jobs command) (10)

  Background execution is done using ‘&’. For foreground execution, the parent process waits, but for background execution, the parent process does not. Instead it adds it to a list of running background processes so that it can be listed when ‘jobs’ is called. All jobs are child processes that are overseen by these parent forks.

- Printing/reporting of background processes, (including the jobs command) (10)

  The list of background jobs is kept in a Job array called bgJobs. Processes are added to this array when they are called to run in the background. When a process running in the background completes its execution, it turns the variable bgRun in its Job struct to false so that the program knows it is not executing. When ‘jobs’ is called, the program runs through this array to see if the background processes are still running and prints their information if they are.

- Allow file redirection (> and <) (5)

  This is done by using dup2() to change file input/output to STDIN_FILENO/STDOUT_FILENO. When this job is finished the job input and output field are reset back to null so as to appear empty and not interfere with the next job.

- Allow 1 pipe (10)

  Same as multiple piping

- Supports reading commands from prompt and from file (10)

  Done by redirecting the file given to STD_IN after seeing the 'quash' command. Then quash read each of these commands sequentially and executed them until it ran across 'quit' or 'exit' in the text file.

- (Bonus) Support multiple pipes in one command (10)

  When the '|' token is seen the number of jobs is incremented to show the current job will be piping into the next. In execute() if the number of jobs is greater than 1 then piping must have occurred. The number of pipes is equal to the number of jobs minus one. Each job is then linked in sequence so that each pipes into the next. Each pipe that is not last to writes to the next pipe id and each pipe that is not first reads from the current pipe id. After the final job finishes all pipes close if they have not already

- (Bonus) Kill command (5)
  The kill command was not executed. It could have been by running through all background processes, checking their process ids and forcing them to exit if their process ids matched the input.

**Testing**

Testing was done through using four files and a folder. HelloWorld.c outputted a simple phrase without taking arguments. Mimic.c took a single word as an argument and appended it to a string before returning the string. TestWorld.c was a clone of HelloWorld.c placed in a sub folder 'bar' in order to test 'cd''s functionality. Piping was tested by combining writer.c and reader.c. Input and output redirect was tested using the FirstChar.c file in combination with Input.txt and Output.txt. Reading commands from files was done with 'commands.txt' which contained a list of quash commands.