Aaron Chan
ECE399 Winter 2017
Assignment 3

Program Assignment:

Custom Shell Program that supports I/O redirection, bash-like history capabilities, and background processes. This includes custom functions, such as displaying the last five entries of commands and also displaying a specific recent command.

Design

Shell	
Capabilities	Notes
Interpret and execute commands	● Read in user input and parse command from arguments ● Typos or incorrect arguments will be reported as errors, determined by how successful the execute command runs the input. ● Input of' 'newline (hitting enter only) will not do anything, and the shell prompt will be outputted again. ● fork() will be used to keep shell active and run the inputted command
Exit	● When user inputs ‘exit’ as a command, shell will close/exit. If user wants to use the shell, it requires them to start it up again.
Background Processes (daemon)	● If a ‘&’ is in the commandline, then that command will run in the background and report that if it was either completed successfully or not. Any arguments after the ‘&’ will be ignored. ● Report format: *DONE: < user_input >*

History	
Capabilities	Notes
Store last five entries	● Store in order of entry ● When more than five entries have been made, get rid of oldest entry and save new entry. Accomplished by saving the last four entries into a new array, then saving the newest input along with it. ● Don’t save immediate command before using custom functions.

(Custom command) !x < x > is an integer	● Repeat the <x> command in the history buffer. ● X can't be less than 1 or greater than 5.
(Custom Command) history	● Display all commands saved in history array. ● If less than 5 commands have been inputted before this call, only display that many. Don't display 'null' or any placeholders for the empty spots in the array.

I/O Redirection	
Capabilities	Notes
Pipe ' | ' arg1 | arg2	● arg1 output will be piped to arg2 for use. ● arg1 writes, arg2 reads
Output Redirection ' > ' arg1 > arg2	● arg1 output will be piped to arg2 ● arg1 writes, arg2 reads
Input Redirection ' < ' arg1 < arg2	● arg2 output will be piped to arg1 ● arg2 writes, arg1 reads
For all of these functions, we will need to find the location of the symbol and split the arguments into two different arrays.	

Log

- About 20 minutes were spent to think and write up the tables for 'shell' and 'history' above. Unsure about how to program redirection, so research must be done first before writing the design for this.
- 1 hour spent to write a part of the shell, only getting a buffer of user input that will be easy to work with for parsing and interpretation. Originally saved a large buffer of 1024 characters straight from stdin, but there was difficulty working with the newline character. Switching over to reading one character at a time made it easier and didn't have to delete or replace the newline character.
- Switched to using getline() to decrease the amount of code written. This function saves the newline character, so I added a step to replace it with a null character.
- Compared input buffer with newline so that I can 'ignore' it and have it prompt/read input again without doing anything.
- When execvp fails, the child process is still running the custom shell program and doesn't exit properly. It will then require multiple 'exit' command entries to leave the child, then the parent custom shell program. Arbitrary entries seem to also fail as well.

- Found misplaced “exit” line in code and moved to child process if branch so that it exits on failure properly.
- One hour spent to write code for custom history commands. Compares to output error messages also coded in. When user only inputs the exclamation mark, or when <x> is greater 5 or less than 1, we output error. We also output error when they request <x> old entry, but that many commands haven’t been entered yet. Current problem with history storing is that it stores the immediate command and doesn’t maintain the previous 5 entries. We only then have the previous 4 entries and the one immediate new command entry.
- Background processes have been implemented, but is only designed for cases of when the ‘&’ is the last argument in the entire command line input. Intending to cut off at ‘&’ when parsing so that it doesn’t use the other arguments and only executes commands before the ‘&’.
- After researching pipe and redirection, found out that they all use pipe except what writes and what reads would differ. All that is left is to find a way to compare for the different characters and figure out which direction to pipe what. Multiple redirections and pipes will be difficult, but for now we will work on only one pipe or one redirection argument.
- Will need a separate function to compare for the different redirection and piping arguments and then update a flag. Need to rewrite code for the ‘&’ so that we can use call the daemon execution function based off the flag as well.
- Another function will be needed to find a delimiter (the redirection or piping symbols) and use the returned integer to decide where to split the arguments
- Redirection works but occasionally breaks in a few cases.
- Pipe only works with commands on either side of pipe symbol. Upon successful runs, it exits the shell (which we don’t want).
- Approximately 3 hours were spent writing the pipes and redirection.
- Overall, the entire program (with missing parts) took about 15 hours to code and test to see if certain inputs and cases would break it.

Files turned in for assignment:
header.h
pipe_direct.c
read_parse.c
custom_exe.c
shell.c
Makefile