Andrew Park (ap1614), Elysia Heah (eyh24)
Operating Systems CS416

Project 04: C Shell

Instructions:

This program behaves like an actual shell. Simply run “make” with the Makefile and input csh commands to interact with the program as you would a normal C shell. Take note that using ctrl+C will only end the execution of any program running within the user-level shell, so to exit the program entirely type the custom command “exit” instead.

Functions:

- *printprompt()*: A simple function that grabs the current directory using getcwd() and attaches “ >” at the end of the string. It prints the string to the console to simulate the look in addition to the effect of the c shell.
- *shellLoop()*: the program enters this from main and runs a while loop to continually read in commands and execute them, given that the user is not inputting an empty line. It handles this in the following function, *shellExecute()*.
- *shellExecute()*: this parses the command the user typed and interprets what to do. It first checks whether or not the line is typed correctly with *isProper()*, and then continues to decipher it. It delimits the line with “;>|>>” to catch and manipulate special commands, and stores each delimited argument individually. Next it parses the arguments, making sure to keep track of several variables that will later help denote whether or not the command will need to issue a redirect or pipe. It calls strcmp() to check for some certain commands: if the user intends to exit the cshell program with “exit” (in which case it will free everything from memory first before exiting), re-execute with “csh-reload”, or change to a new directory with “cd”. In the case of “cd” it will try to change to the specified directory if it exists with chdir(); if not, it will print the appropriate error message.
  Next, the function runs through some checks for piping. Since piping is chained, the following needs to be processed carefully; it checks whether or not the argument currently being processed is piped, and uses arrays of size two for reading and writing as well as two flag variables useIn and useOut to denote whether it is currently being piped in (read) or out (write), which is used by calling the pipe() command. (Note the use of the enumerator in the header to make it clearer which index of the array is for reading and which is for writing.) Now the cshell has to actually execute the line. It forks and

waits on a child process which uses dup2 to feed the argument into the pipe chain depending on whether it's going in or out, and then executes using execvp(). If the command is invalid, an appropriate error message is printed and the command is abandoned. Now it deals with file redirection if the flag for it was set due to the detection of ">" or ">>" earlier. It opens the file passed in from the command line argument; if the "truncate" flag is set, it will do so. A char buffer is used to read from the result of the pipe and write to the file. If the file is not valid, it will display the error message and restore the environment of the program with longjmp(). Once the command and all of its arguments have been successfully processed and executed, all variables are freed.

- *IntHandler(int signum)*: This is a handler to deal with SIGINT. In this case when we get a keyboard interrupt ctrl+C, we don't want to exit the cshell program itself but whatever the cshell program is currently running. Therefore if we detect that something is in the middle of executing (a flag we set with isExec) we kill the child process (the one running the execution) and restore back to the previous state of the environment using siglongjmp().

Results

All aspects of the specified commands in the project report were completed. The user-level cshell supports execution of csh commands and takes special characters into account.

Difficulty

There were some tougher aspects of this project, such as understanding the logic and implementation of pipe chaining as well as how to take care of the SIGINT handler. Luckily, the clue in the pdf to use dup2() demonstrated how to use copies of file descriptors to feed the arguments through the pipe. The SIGINT handler was covered using longjmp() to effectively restore the environment back successfully. Overall despite the challenges that came with this project, it was still doable and resulted in a functional program.