C Shell Report- CS 416

Authors:

Krishna Prajapati	kjp189
Rakshay Kanthadai	rsk146

Program:

./shell

Command to start the shell.

The given makefiles and source code files can be compiled using the make command. This creates a binary executable of the C Shell program, a program that is designed to emulate the same functionality as the Linux shell.

Functions:

The following is a brief description of each of the functions used by the program and how they all work together to emulate a shell environment.

removeSpace():

This function removes any white space from the beginning of a string. This is mostly useful for redirecting to a file. It simply returns a char* pointing to the first non-whitespace character in the string or NULL if the string is comprised of whitespace.

read_line():

This function reads the user's input from standard input. The input is read byte by byte and stored into a buffer. The buffer is returned so that the input can be processed.

parse():

This function takes in an input string and a delimiter character and returns an array of strings.

The function creates an array of string pointers where each pointer points to a location of the given string. This location is determined using **strtok()**. After allocating the array of pointers and filling in each slot, a NULL pointer is appended to the end of the array. The whole array is returned.

This function is used to determine each "section" of the given input since the function can delimit strings by spaces, semicolons, redirects, and pipes.

execute():

This function takes in an array of string arguments, a boolean for pipes, and the file descriptors for standard input and output.

If the pipe boolean is true, then the function uses **pipe()** and **dup2()** to redirect all **stdout** prints from the current **execvp()** call to go to **stdin** for the next **execvp()** call. If the boolean is false, then we simply print out to whatever the current **stdout** is referring to (since the output could have been redirected to a file).

In both cases, assuming the command is not “cd”, “pwd”, or “exit”, **execvp()** is eventually called using the passed in array of arguments. The **fork()** function is used to create a separate process. The child will then call **execvp()** to perform the command that the user requested. The first element of the **char**** array will always be the command that needs to be run, while the rest of the array is the arguments needed. Note that this utilizes a **char**** that was returned from **parse()**, so it is NULL terminated.

If the command is “cd”, then **execvp()** is not called. Instead, **chdir()** is used to change the present working directory.

If the command is “pwd”, then **execvp()** is not called. Instead, **pathconf()** and **getcwd()** are used to get and print the present working directory.

If the command is “exit”, then **execvp()** is not called. Instead, **exit()** is called to quit the shell.

sigHand():

This function is the signal handler that handles **SIGINT**, usually from the control-c sequence. This function will kill the current command by sending the **SIGKILL** signal to the forked child process. All file descriptors are closed except for the original **stdout** and **stdin**, and all memory is freed. The console will be ready to take the next user input afterwards.

loop():

This function is the heart of the program and puts all of the components together. First, the function gets all the information needed about the current directory and displays it. Then, the **read_line()** function is called to get the user input. Afterwards, the **parse()** function is used multiple times to separate the commands, pipes, redirects, and arguments. The input is first parsed by semicolons to separate each command.

After separating each command, we then loop through the commands and parse each command by redirects, since there can only be one redirect. We then parse the remaining portion of the command by pipes. These piped commands are then tokenized by spaces. We then call **execute()** on each tokenized array to perform all the pipe instructions needed. The instruction after the last pipe is done separately outside of the loop since there is a possibility that we need to use **dup2()** to redirect the output to a file instead of the console.

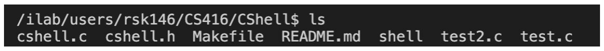
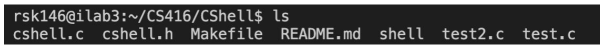
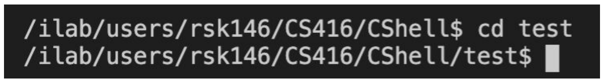
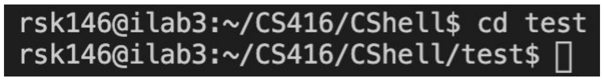
We perform the operations listed above for every command that was separated by semicolons.

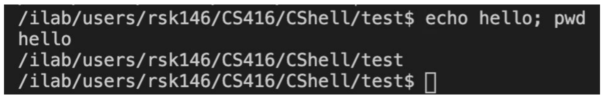
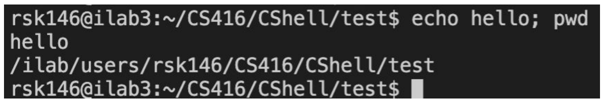
After finishing the operations, **loop()** function returns to the beginning of the do-while loop and awaits the user’s next input. The console prints the current working directory, and we repeat everything from the very beginning. The loop only stops when the “exit” command is given.

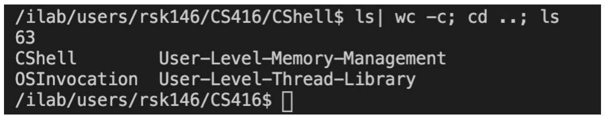
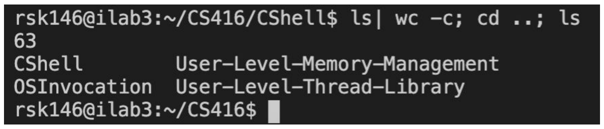
Executables:

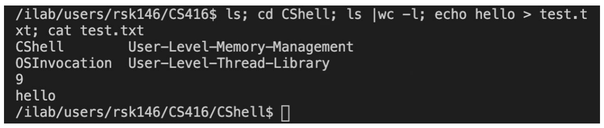
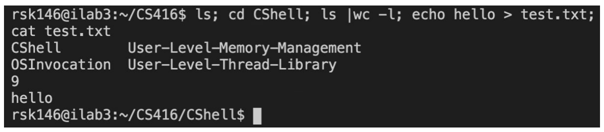
The shell is capable of running cd, pwd, as well as all binaries in the normal Unix/Linux suite. In particular, we primarily test with ls, cd, wc, cat, echo, and different combinations of them with pipes/redirection.

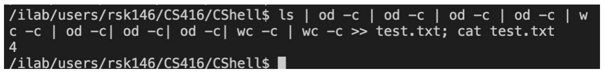
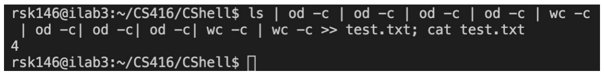
Benchmarks

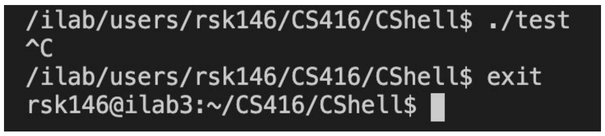
C Shell	**Unix Shell**
 `/ilab/users/rsk146/CS416/CShell$ ls` `cshell.c  cshell.h  Makefile  README.md  shell  test2.c  test.c`	 `rsk146@ilab3:~/CS416/CShell$ ls` `cshell.c  cshell.h  Makefile  README.md  shell  test2.c  test.c`
 `/ilab/users/rsk146/CS416/CShell$ cd test` `/ilab/users/rsk146/CS416/CShell/test$`	 `rsk146@ilab3:~/CS416/CShell$ cd test` `rsk146@ilab3:~/CS416/CShell/test$`

 `/ilab/users/rsk146/CS416/CShell/test$ echo hello; pwd`
`hello`
`/ilab/users/rsk146/CS416/CShell/test`
`/ilab/users/rsk146/CS416/CShell/test$`
	 `rsk146@ilab3:~/CS416/CShell/test$ echo hello; pwd`
	`hello`
	`/ilab/users/rsk146/CS416/CShell/test`
	`rsk146@ilab3:~/CS416/CShell/test$`

 `/ilab/users/rsk146/CS416/CShell$ ls| wc -c; cd ..; ls`
`63`
`CShell        User-Level-Memory-Management`
`OSInvocation  User-Level-Thread-Library`
`/ilab/users/rsk146/CS416$`
	 `rsk146@ilab3:~/CS416/CShell$ ls| wc -c; cd ..; ls`
	`63`
	`CShell        User-Level-Memory-Management`
	`OSInvocation  User-Level-Thread-Library`
	`rsk146@ilab3:~/CS416$`

 `/ilab/users/rsk146/CS416$ ls; cd CShell; ls |wc -l; echo hello > test.t`
`xt; cat test.txt`
`CShell        User-Level-Memory-Management`
`OSInvocation  User-Level-Thread-Library`
`9`
`hello`
`/ilab/users/rsk146/CS416/CShell$`
	 `rsk146@ilab3:~/CS416$ ls; cd CShell; ls |wc -l; echo hello > test.txt;`
	`cat test.txt`
	`CShell        User-Level-Memory-Management`
	`OSInvocation  User-Level-Thread-Library`
	`9`
	`hello`
	`rsk146@ilab3:~/CS416/CShell$`

 `/ilab/users/rsk146/CS416/CShell$ ls | od -c | od -c | od -c | od -c | w`
`c -c | od -c| od -c| od -c| wc -c | wc -c >> test.txt; cat test.txt`
`4`
`/ilab/users/rsk146/CS416/CShell$`
	 `rsk146@ilab3:~/CS416/CShell$ ls | od -c | od -c | od -c | od -c | wc -c`
	` | od -c| od -c| od -c| wc -c | wc -c >> test.txt; cat test.txt`
	`4`
	`rsk146@ilab3:~/CS416/CShell$`

*  `/ilab/users/rsk146/CS416/CShell$ ./test`
`^C`
`/ilab/users/rsk146/CS416/CShell$ exit`
`rsk146@ilab3:~/CS416/CShell$`
	terminal exits normally

***./test is an executable that runs an infinite loop**

Possible Improvements:

The standard input is read byte by byte, which can be slow for longer length inputs. This can be improved by reading multiple bytes at a time in order to avoid having to access the buffer many times. There is also a character limit per command in the Unix shell so it could be beneficial to implement one here and simply use a buffer of that length to decrease read times. Although this solution would tie into the next note on space usage.

Additionally, the space usage of the program is not entirely optimized since some parts of the buffers and char** arrays have too much space allocated to them. This could be fixed by looping through the input strings twice, once to determine buffer/char** array sizes/allocate them and again to fill them, but this would simply be taking a space complexity issue and offloading it into the programs time complexity, which we choose not to do here.

Finally, one extra feature that we could add to the shell is the storage of past commands. Similar to how the terminal already works, the user should be able to press the up or down arrows to cycle through commands that they have already entered. This would be a great quality of life improvement for our C Shell. It would also be nice to add an autocomplete feature based on commands (if it's the first thing being entered) or files/directories in the current working directory (if it's not the first thing being entered) similar to how tab autocomplete functions on the Unix shell.

Difficulties in Solving the Project:

One of the difficulties we had when doing the project was with the implementation of piping. We at first were unsure of the order that we should tokenize our command. After figuring out that we should do pipings before redirects, we ended up spending most of our time trying to understand and wrap our heads around what **dup()**, **dup2()**, and **pipe()** does.

We spent a lot of time trying to make **pipe()** work for 1 simple case, and then extend that for multiple pipe cases. However, we never seemed to be able to get the hang of extending the simple case, and since **pipe()** and **dup2()** were involved with stdout and stdin, it was very difficult to debug since print statements wouldn't show up in the console anymore. Additionally, gdb does not show what was being written to each file descriptor, and since we did not understand how the pipe's file descriptor was supposed to be used to pass information from the child's **execvp()** to the main process, we ended up wasting a lot of time playing around with **dup2()** and **pipe()**. We realized that our mistake was that we were not properly closing the file descriptors from pipe, so the output was never able to be read by the next command.

Another difficulty that we encountered was double frees. Since the user's input gets tokenized multiple times, we initially had a bit of trouble freeing the correct mallocs, especially with **strtok()**. This resulted in a lot of undefined behavior and was very frustrating and troublesome to debug since there was no way to pinpoint the problem. The only way to resolve this issue was to read through the code and look at every free statement to see if it could ever free anything that was not already alloc'd.