

Jackie Diep

Professor Bo Li

CSC 206 G001

26 October 2020

### Programming Assignment: Unix Shell and History Feature

**Abstract: The purpose of a Unix Shell is to provide users with an interface to interact and execute upon the Unix system. The shell functions similarly to many higher level languages and compilers, and it provides various personalizations for programmers to take advantage of. Typically, shells will use the first word of input and spaces delimit commands to comprehend commands and implement them.**

This assignment focuses on implementing various features into a given shell interface template written in C. The main priority was implementing all of the features using separate parent and child processes to run the commands with certain requirements. The first task was to modify the given shell to execute commands in child processes. Special attention should have been paid to whether or not the user included an ampersand to signal for the parent process to skip calling the wait function for the child. The second task specifies the creation of a “History” function that would allow the previous command to be repeated using the ‘!’ prompt. The command would then be output to the console again and added to the history buffer. If the ‘!’ prompt was the first command given, an error message “No commands in history.” was to be given. The third task was to implement the redirection operators, output: ‘>’ and input: ‘<’. Modification of the shell involved using the dup2() function to alter standard input and output using file descriptors. Students are allowed to assume that sequences of commands would not be used such that only one operator should be called per command line. The fourth and final task involved using the dup2() function once again to implement Unix pipes to allow Interprocess Communication. Students are once again allowed to assume that no command sequences would be used, and a suggestion was given to use

the parent process to call a child process to run the initial command. That child would then call another child process to receive the output and run the next command.

After researching the implementation and reinforcing my understanding of the theory behind parent and child processes, I was able to modify the given template into a usable Unix shell. I ran into a few issues with allowing the program to close properly, and I did struggle quite a bit with initially understanding how to use the Virtual Machine. I had an introductory period of learning how to share files using WinSCP and understanding how to navigate the VM's directories. One of the most confusing aspects of the first task for me was implementing the ampersand functionality. Out of all of the research I conducted, I did not manage to come across a single template that used any sort of code to read in the ampersand. This became a recurring problem for me throughout the assignment that I managed to get around for all tasks except for the last.

To implement the initial parent and child processes, I researched and used the strtok function along with a character array and the fgets() function. I then indexed through the tokens and set them equal to the given arg[]s array with an integer to check for and note the existence of an ampersand. Afterwards, I initialized a pid of pid\_t data type and forked it, returning an error if the fork failed. If the fork was successful, the child would run execvp(args[0], args), and the parent would choose to wait or run concurrently with the child process depending on the existence of an ampersand. My second biggest hurdle was figuring out a way to check for the ampersand, though I eventually came across the strcmp() function that simplified the task. My largest hurdle that was not resolved until the fourth task was crashing of my VM due to Segmentation errors. Seemingly without rhyme or reason, my VM would occasionally crash with a Segmentation error that I later solved by setting the array to NULL at the last indexed position. This was a problem that made it very difficult to debug because I could not tell what particular changes that I would make were causing problems in my implementation.

### **Pseudocode for task one: Executing Command in a Child Process**

```
{
```

```
  initialize a char* array to hold arguments to be read into
```

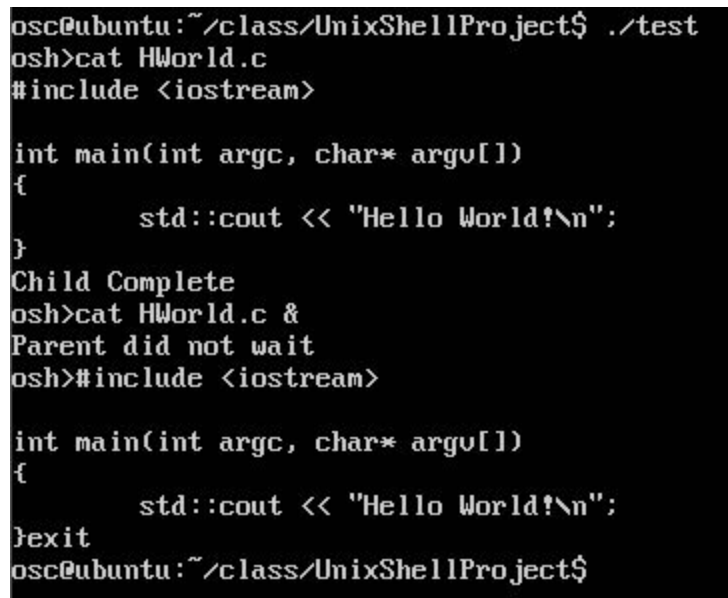
```

initialize an integer to handle the while loop
initialize an integer to note the existence of an ampersand
while (integer) {
    prompt for user input with a printf function
    clear the output buffer between each loop
        initialize a char array to hold the command line; 40 was used because the size of the given args[]
        call an fgets function to read the char array
        call strtok and initialize the results into a char pointer to parse through the tokens
        initialize an index integer
        set the ampersand checking integer to 0 between each loop
        while (parser is not null) {
            if (ampersand exists) {
                increment ampersand checking integer
                increment indexer
            }
            else {
                args[] array equal to parser at index
                increment indexer
            }
            move the parser forward using strtok
        }
    *** set args[] equal to NULL at the final indexer position
    create a pid_t type variable "pid" and fork it
    error check the fork
    if forking was successful,
    if child
        child will call execvp(args[0], args)
        *** return statement immediately after execvp to handle process failure
    if parent
        parent will check for asterisk
        if asterisk exists, do not wait
        if asterisk does not exist, call wait(NULL)
        if exit inputted, change while loop variable to exit condition
}

```

```
return 0;
```

Of particular note in my pseudocode for the first task are the NULL termination of the args array and the return statement after the `execvp` function. Two of my largest issues with the assignment were solved here, respectively being a random and extremely common series of Segmentation fault crashes and stopping an infinite amount of child processes being called, making me unable to exit my program. I did not fully solve these issues until the fourth task, but they are integral to all parts of my algorithm. I used the most simple “Hello World!” file provided by visual studio to test my implementation.



```
osc@ubuntu:~/class/UnixShellProject$ ./test
osh>cat HWorld.c
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello World!\n";
}
Child Complete
osh>cat HWorld.c &
Parent did not wait
osh>#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello World!\n";
}exit
osc@ubuntu:~/class/UnixShellProject$
```

This screen capture displays the result of calling a command, calling a command with an asterisk, and calling the `exit` command. When a command is called with an asterisk the `osh>` input prompt is displayed before the command runs because the parent does not wait. The `exit` command crashed my VM until I included the `return` to catch failed `execvp` calls noted in my pseudocode.

The implementation of the history feature was the easiest part of the assignment for me. First, I included an integer to note the existence of the ‘!’ command prompt and a character array to hold the previous command called by the user. Then, I added an else-if block to my `strtok` parsing loop that checks

for a token of '!!' that also nests an if-else loop to handle whether there were no commands called previously. In the else block, I included a strcpy() function to copy the previous command to the string and called the strtok function again. Finally, I included an if statement around the rest of my code that would restart the loop immediately if the history checking integer was marked, and at the end of the while loop, I call a strcpy() function that copies the finished command to the history buffer.

### **Pseudocode for task two: Creating a History Feature**

```
{
initialize a history buffer char array
while(exit command not called)
{
initialize a history checking integer
while(parsing command line)
{
    check if '!!' prompt was called
        if yes check if history buffer is empty
            if empty print("No commands in history.")
        if not empty
            strcpy history buffer to command line parsing array
            call the strtok function again to read the new array
    parse the changed tokens
}
if '!!' prompt was called and buffer was empty, skip the rest of the code and restart the loop
{
....
if a command was ran normally, or a command was called by the history function,
    set the command to the history buffer before restarting the loop using strcpy
}
}
```

Implementing the history feature was relatively easy because all relevant functions except for strcpy() were ones that I had already researched or implemented in the task prior. After discovering strcpy() the implementation was straightforward.

```

osc@ubuntu:~/class/UnixShellProject$ ./test
osh>!!
No commands in history
osh>cat HWorld.c
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello World!\n";
}
Child Complete
osh>!!
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello World!\n";
}
Child Complete
osh>exit
osc@ubuntu:~/class/UnixShellProject$ _

```

In this screen capture, the “No commands in history.” notification is displayed at the top when the history function is called before and commands. Afterwards, a command is called and the history function repeats the command.

```

osh>!!
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello World!\n";
}
Child Complete
osh>!!
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello World!\n";
}
Child Complete
osh>_

```

This screen capture displays that, after the history command, the command is added back into the buffer as specified.

The third task was an implementation of redirecting input and output operators. Though not as difficult as the fourth task, I did have to research quite a bit about the dup2 function, how the standard input and output of Unix shells behaved, and the way that the shell handles reading and writing with files. Once the behavior of STDIN and STDOUT became clear to me, I was able to implement the redirection operators very similarly to how I implemented the history and ampersand features. I implemented two more blocks into my if-else structure that parses the command line, searching for redirection operators. If the input operator '<' was found, I opened the specified file in read only mode and used dup2() to replace STDIN with the file descriptor. If the output operator '>' was found, I opened the specified file in write only mode and used dup2() to replace the file descriptor with STDOUT. After each action, I incremented the indexer and ran the program normally.

### **Pseudocode for task three: Redirecting Input and Output**

if token of '<' found

    increment the parser and read the specified file name

    open the specified file in read only mode with a file descriptor

    call the dup2() function and replace STDIN\_FILENO with the file descriptor

    increment the parser and run the program normally

if token of '>' found

    increment the parser and read the specified file name

    open the specified file in write only mode with a file descriptor

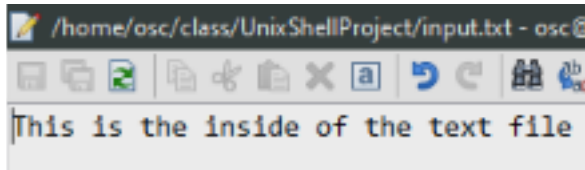
    call the dup2() function and replace the file descriptor with STDOUT\_FILENO

    increment the parser and run the program normally

There is not much to note about the pseudocode for this task. It was straightforward after understanding dup2 and the standard input and output of Unix shells.

**Screen captures for input:**

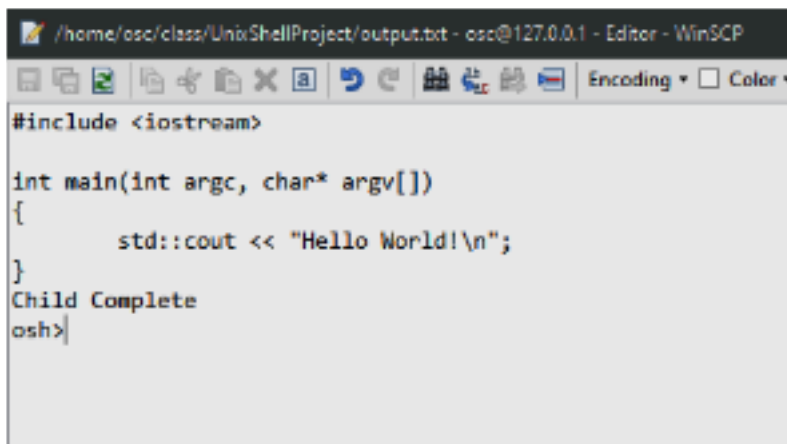
```
osc@ubuntu:~/class/UnixShellProject$ ./test
osh>cat < input.txt
This is the inside of the text file
Child Complete
osh>
```



The running screen capture above is somewhat redundant because it is the same output as “cat input.txt”, but I figured it was more consistent to use the same command (cat) as I have been using for all of my previous screen captures.

**Screen captures for output:**

```
osc@ubuntu:~/class/UnixShellProject$ ./test
osh>cat HWorld.c > output.txt
```



The screen captures above for output display the command line using the redirectional output operator and the text file afterwards.

The fourth and final task was the only task that I was not able to complete, and the task that I spent the most time on by far. I spent a lot of time researching the way that pipes work, and I developed a



general understanding, though it did not allow me to complete the task. I tried many things such as calling `execvp` in the parent function, creating a separate `args[]` array for the read side of the pipe, creating a buffer to read from the pipe directly, and having the first child create a second child to run the second command. I spent most of my time trying to store the input of the read side of the pipe into the second element of the pipe `args[]` array that I created with the first element being the command after the `'|'` from the command line. I created a few files to work on the fourth task after a few hours of undoing the code from my original project out of fear of breaking what I already had implemented. I will be submitting both the working original file and the file I was last attempting to implement on. I suspect that my problems primarily lie in attempting to read the pipe into an element of my array, but I was unable to find a proper way to use it in the last dozen or so hours of programming time.

An issue I mentioned early on in my report is that the references I found and research that I performed for this project never mentions any types of implementation required for the Unix operators to function. When I research how to use pipes, it does not talk about the command line comprehension for the shell. It was perplexing and more than a little frustrating that I could not find any type of mention of having to implement the `'|'` bar character to be read in and processed. Most of the implementations for it that I attempted ended up breaking other parts of my code, or flat out causing it to crash with memory errors. Including an integer to check for it like I did in other parts of my code made nothing be read into the `args[]` array at all, even though that segment of the if-else loop should have never been explored. Eventually I used whether or not the pipe `args[]` array was empty to signify the existence of it instead of an integer check. That did not break the rest of my code like the integer did, but it also did not solve my problems. From what I was able to tell, I could reach points where the pipe was written into, but I could not figure out how to use the read side of the pipe to add to the next command. This is why I theorize that my problems come from trying to read into the pipe `args[]` array. Currently, my code is in a still non-working state of trying to use the original `args[]` array to run it. I will comment and submit both the original file and the file that I was last using to attempt to implement the fourth task.

In this assignment, I believe that the main concept I learned was personalization. I learned a lot about the possible ways that the Unix system processes command line operators and what a lot of the backbone of them looks like from my references. I also had never considered what running multiple processes at once would be useful for, which has turned out to be the most interesting thing about the assignment. Even with that in mind, I still believe that personalization is what I have learned the most about Unix shells. When I have worked on programs for other classes, it always felt like building new features for certain tasks, like building the tools to design a swimming pool more efficiently. For the Unix shell, it felt more like building tools that have an overarching purpose to help build tools. The best visualization of what I am describing is like building a personalized work space that I can be most efficient within.

In terms of coding and algorithms, the `dup2()` function is the most interesting one to me, followed closely by `fork()`. I had not yet run into a situation where I considered replacing the standard input and output of the console, so as soon as it was introduced to me in my research, I took a break from my project to consider the uses of it. Though its uses are less apparent, `fork()` was also a new concept that I spent a lot of time researching. Using one program to run multiple processes is not something I had explored previously, but in hindsight, it answers a lot of questions that I have had in the back of my head about how things can be calculated concurrently in calculation heavy tasks.

In the Search program assignment, I mentioned that I felt I had learned the most from it out of any Computer Science assignment I had ever received. After writing this report, I want to amend that statement to be more about the most *information* I have ever learned. This assignment taught me more about having a different perspective on what lies underneath the tools we use in Computer Science. Rather than a focus on new information, this assignment felt more like a deeper understanding of the fundamentals of how true programs run.

## References

- Barnes, Ricky. *Process vs Parent Process vs Child Process*, TutorialsPoint, 11 Oct. 2018, [www.tutorialspoint.com/process-vs-parent-process-vs-child-process](http://www.tutorialspoint.com/process-vs-parent-process-vs-child-process). Accessed on October 8, 2020.
- “C Library Function - Strcmp().” *Tutorialspoint*, [https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_strcmp.htm](https://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm). Accessed on October 7, 2020.
- “C Library Function - Strcpy().” *Tutorialspoint*, [www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_strcpy.htm](http://www.tutorialspoint.com/c_standard_library/c_function_strcpy.htm). Accessed on October 10, 2020.
- “C Library Function - Strtok().” *Tutorialspoint*, [www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_strtok.htm](http://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm). Accessed on October 7, 2020.
- “Creating a Pipe (The GNU C Library).” *Gnu.org*, [www.gnu.org/software/libc/manual/html\\_node/Creating-a-Pipe.html](http://www.gnu.org/software/libc/manual/html_node/Creating-a-Pipe.html). Accessed October 21, 2020.
- Daniel Gilly & the Staff of O'Reilly & Associates. “UNIX in a Nutshell: System V Edition.” *Unix.org.ua*, O'Reilly & Associates, Inc., 6 Aug. 1998, [docstore.mik.ua/orelly/unix/unixnut/ch03\\_02.htm](http://docstore.mik.ua/orelly/unix/unixnut/ch03_02.htm). Accessed on October 7, 2020.
- “Dup.” *Opengroup*, The Open Group Base Specifications Issue 7, 2018 Edition, [pubs.opengroup.org/onlinepubs/9699919799/functions/dup.html](http://pubs.opengroup.org/onlinepubs/9699919799/functions/dup.html). Accessed on October 16, 2020.
- File I/O*. TutorialsPoint, [www.tutorialspoint.com/cprogramming/c\\_file\\_io.htm](http://www.tutorialspoint.com/cprogramming/c_file_io.htm). Accessed on October 16, 2020.

Kerrisk, Michael. *Dup(2) - Linux Manual Page*, man7, 4 Nov. 2020,

[man7.org/linux/man-pages/man2/dup.2.html](http://man7.org/linux/man-pages/man2/dup.2.html). Accessed on October 22, 2020.

Lakshman, Sarath. "Implementation of Redirection and Pipe Operators in Shell." *Sarath Lakshman*, 24

Sept. 2012,

[www.sarathlakshman.com/2012/09/24/implementation-overview-of-redirection-and-pipe-operators-in-shell](http://www.sarathlakshman.com/2012/09/24/implementation-overview-of-redirection-and-pipe-operators-in-shell). Accessed on October 20, 2020.

Modi, Archit. "An Introduction to Pipes and Named Pipes in Linux." *Opensource.com*, 23 Aug. 2018,

[opensource.com/article/18/8/introduction-pipes-linux](https://opensource.com/article/18/8/introduction-pipes-linux). Accessed on October 20, 2020.

"Strcmp." *Cplusplus.com*, [www.cplusplus.com/reference/cstring/strcmp/](http://www.cplusplus.com/reference/cstring/strcmp/). Accessed on October 7, 2020.

*Unix / Linux - What Is Shells?* [www.tutorialspoint.com/unix/unix-what-is-shell.htm](http://www.tutorialspoint.com/unix/unix-what-is-shell.htm). Accessed on October 7, 2020.

Yerraballi, Ramesh. "Pipe Dup Explained." *Youtube*, 8 September. 2018,

<https://www.youtube.com/watch?v=fOaK6oRqhEo>. Accessed on October 20, 2020.