**Shell Report**

**Zachary Essoh**

**Arielle Gilmore**

**COSC 4302 Operating Systems**

**Professor Bo Sun**

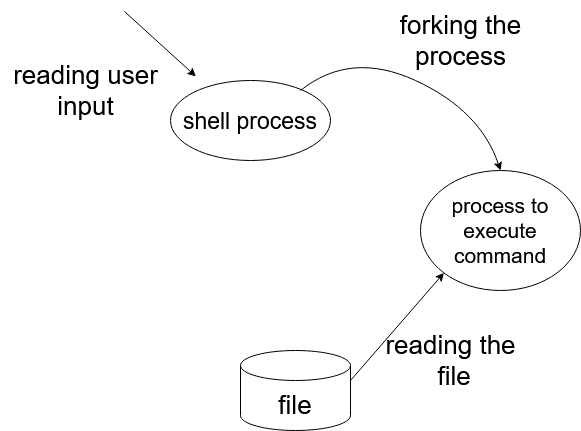
**November 18, 2022**

**Introduction**

This report is written to outline the approach, process, and implementation used throughout the duration of this project. In this project we were tasked with creating two variations of a shell program interface, one that uses fork and one that uses threads. Shell programs have many benefits such as allowing the user to more easily navigate their system and perform basic tasks.

**Approach**

As a group, we approached Part 1 of the project by first going through the instructions outlined in Chapter 2, Lab Exercise 2.1. Once we understood the project objective, we reviewed the idea of a shell and its states by referencing the following diagram:



This was an important step because understanding how the shell works would assist us in its implementation. Each section provided by the exercise we developed an understanding of in order to establish the knowledge needed to create a shell. Some examples of this in our project, is the use of sections **Printing a prompt** and **Getting the command line** for Part 1 in which we decided to send a prompt to the user that requested a command, take in the command as a buffer, and parse the command in a similar approach to the launch.c file but by instead using a resource that would allow us to wait on the status of the process for each command, which was our first primary source to understand the use of a shell. A good amount of research also went into this project. For example, in Part 2, it was important for us to understand the meaning of each argument taken in by pthread\_create() so that we could understand how to create a thread, feed a function into the thread, and send any information to the function for the thread to utilize. One challenge that we encountered was that of the use of pointers in Part 2, in which we needed to understand how information was communicated through thread parameters/arguments. In order to truly gain understanding throughout our research, we would often create or run small test files. An example of this is when we created a test C file in which we took in user input, passed the input to a function through pthread\_create, and returned the input back to the user:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

char x[50];

void \*myThreadFun(void \*arg)

{

char \*mystr = (char \*)arg;

printf("Here is your input: %s\n", \*mystr);

}

int main()

{

printf("Enter an integer: ");

scanf("%s", &x);

pthread\_t tid;

pthread\_create(&tid, NULL, myThreadFun, (void \*)&x);

pthread\_exit(NULL);

return 0;

}

From the above approaches taken, we were able to begin implementing Part 1 and Part 2 of the project.

**Implementation**

This outlines how we implemented the API’s in the respective programs. To start with the first part, we first used the fork() API to create separate child processes. These child processes will then read in the user entered command from a buffer variable USER\_BUFFER this variable needs a defined size of 1 kilobyte or 1024 bytes. A variable arg\_size is used to set the length of command the user is allowed to enter. Having a predefined argument size helps the program run more efficiently. Another important variable we used was user\_args, this variable is used to keep track of the number of command arguments the user has entered. A third variable that is crucial to the program is file\_path, this variable stores the location of the operating system commands so that the program can access them when needed. Before the entered command is executed, the child process uses the execv() API to replace the process with the path of the entered command in the operating system. Once the path is obtained the child process executes the given command. After the execution is complete the parent process continues to ask the user for new commands.

When implementing Part 2, through our research we found that it was not possible to create a process using only that of a thread, but that we were able to create threads that would create a process for each command. To refactor Part 1, we moved all of the initial code using child/parent processes in main(), and replaced it with the initial creation of a thread using pthread\_create(). This function took in as arguments, the thread id known as tid, a default attribute known as attr, and the function for creating each process– threading. This function would hold the while loop for Part 1 so that a thread could create a process. One obstacle that we came across was the inability to take in user input at the stage of main(), so our solution was to not take in any user input for the thread, but to use the thread to prompt the user for a command, verify the command through sets of conditionals that checked for the buffer limit, the length of the command, and whether the command had the return of a new line. We also created a counter for each thread that was created, thread\_count, which stored an integer value and updated itself each time a command (or process) was completed. The line printf("Thread %d complete\n", thread\_count); conveyed this information to the user as well so that they could view how the shell was multithreaded. Ultimately, one of our most significant solutions in Part 2 was the use of an if-statement in order to check whether a thread had ended, if so, to create the thread again, and otherwise, end the thread using pthread\_join() for the next command user input.

**Conclusion**

In conclusion, we faced many challenges during the course of this project but overall it was a wonderful learning experience. One of the challenges we faced was that we could not figure out how to integrate the cd command to enable the user to change directories. We also were unable to integrate the ls command that lists the contents of the present working directory. These challenges could have been alleviated had we used a different file path within the operating system. Lastly, through this group project we have learned a great deal about how the operating system parses and handles user commands and the different ways that a terminal-like application can be developed.