Overall:

In this project we implemented several functions in order to parse command based upon what a user enters. Using input redirection we were able to replicate a basic shell using C.

Learned and understood:

In this assignment I Learned a lot about how file descriptors and low level system io function. Prior to this project I didn't understand what the file descriptor table was or how it was used nor did I fully understand the fork command. With having to implement and debug this code I found it to be very informative at teaching me how those things worked.

Forking in this project allowed me to get a better understand of how C handles different processes. Several times in this project I would enter a wrong command and the program would just stall. After investigating and a few google searches I resolved the issue. Another thing I learned using fork was it's impact on input redirection. Prior to this assignment I don't think I would have expected that the child process input redirection would have an effect on a separate fork's input redirection but now after this project I understand how and why this occurs.

In this homework I also discovered that parts of my parsing function were not working as I intended, In assignment 2 I created a lot of unit tests to verify any change I made would not upset the return Code of the function. What I failed to do with my unit tests and to confirm that the results in the cmd1, cmd2, infile, and out file were what I expected. Had I done this I think it would have shaved a few hours off of this project.

Test cases:

Return codes 1-4

```
myshell-% ls
        a.out  log.txt  main.c
        return code is 1
myshell-% wc < log.txt
          7  44 268
        return code is 2
myshell-% ls >> outputfile
        return code is 3
myshell-% echo "End" >> outputfile
        return code is 3
myshell-% cat outputfile
        a.out
        log.txt
        main.c
```

```
        outputfile
        "End"
        return code is 1
myshell-% echo "Overwrite" > outputfile
        return code is 4
myshell-% cat outputfile
        "Overwrite"
        return code is 1
```

Return codes 5-8

```
myshell-% ls | grep c
        main.c
        return code is 5
myshell-% wc < outputfile
         1  1 12
        return code is 2
myshell-% wc < outputfile | grep 12
         1  1 12
        return code is 6
myshell-% ls | grep c >> outputfile
        return code is 7
myshell-% cat outputfile
        "Overwrite"
        main.c
        return code is 1
myshell-% ls -l | grep c > outputfile
        return code is 8
myshell-% cat outputfile
        -rwxr-x--- 1 cxm072 cxm072 18078 Oct 29 22:38 a.out
        -rw------- 1 cxm072 cxm072  2593 Oct 29 22:39 log.txt
        -rw-r----- 1 cxm072 cxm072 15279 Oct 29 22:38 main.c
        -rw------- 1 cxm072 cxm072     0 Oct 29 22:39 outputfile
        return code is 1
```

```
//Author: Cory McDonald
//Synposis: I made a shell. :)

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <limits.h>
```

```c
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <stdbool.h>

#define BUFSIZE 1024
#define CSTRSIZE 100
#define CMDSIZE 30

#define DEBUG 1


int parse_command(char *line, char **cmd1, char **cmd2, char *infile, char *outfile);
void exec_cmd(char** cmd1);
void exec_cmd_in(char** cmd1, char* infile);
void exec_cmd_opt_in_append(char** cmd1, char* infile, char* outfile);
void exec_cmd_opt_in_write(char** cmd1, char* infile, char* outfile);
void exec_pipe(char** cmd1, char** cmd2);
void exec_pipe_in(char** cmd1, char** cmd2, char* infile);
void exec_pipe_opt_in_append(char** cmd1,char** cmd2,char* infile,char* outfile);
void exec_pipe_opt_in_write(char** cmd1,char** cmd2,char* infile,char* outfile);

bool interface(char *line);
void logInfo(char *line);


int main ( int argc, char *argv[] )
{
   if (argc == 2)
   {
      interface(argv[1]);
   }
   else
   {
      bool active = true;
      char userCommand[BUFSIZE];
      while (active)
      {
         printf("myshell-%% ");
         //This makes sure we don't go past the limit of 256 characters. That would be very, very
bad
         fgets (userCommand, 256, stdin);
         //Removing newline from commandls
         if ((strlen(userCommand) > 0) && (userCommand[strlen (userCommand) - 1] == '\n'))
```

```
            userCommand[strlen (userCommand) - 1] = '\0';
            if (userCommand[0] != '\0' && userCommand != NULL)
            {
                if ( interface(userCommand) == true)
                {
                    active = false;
                    break;
                }
            }
        }
    }
    return 0;
}
bool interface(char *line)
{

    // Bad practices ftw
    char infile[CSTRSIZE];
    char outfile[CSTRSIZE];
    bool continueExecute = false;
    char *cmd1[CMDSIZE];
    char *cmd2[CMDSIZE];
    int i;
    int k;
    cmd1[0] = NULL;
    cmd2[0] = NULL;
    infile[0] = '\0';
    outfile[0] = '\0';

    i = parse_command(line, cmd1, cmd2, infile, outfile);
    if(i == 0)
    {
        continueExecute = true;
    }
    switch(i)
    {
        case 1:
            exec_cmd(cmd1);
            break;
        case 2:
            exec_cmd_in(cmd1, infile);
            break;
        case 3:
            exec_cmd_opt_in_append(cmd1, infile, outfile);
            break;
```

```
    case 4:
        exec_cmd_opt_in_write(cmd1, infile, outfile);
        break;
    case 5:
        exec_pipe(cmd1, cmd2);
        break;
    case 6:
        exec_pipe_in(cmd1, cmd2,infile);
        break;
    case 7:
        exec_pipe_opt_in_append(cmd1, cmd2,infile,outfile);
        break;
    case 8:
        exec_pipe_opt_in_write(cmd1, cmd2,infile,outfile);
        break;
    default:
        break;
}

if (i > 9)
{
    k = 0;
    while (cmd1[k] != NULL)
    {
        printf("cmd1[%d] = %s\n", k, cmd1[k]);
        k++;
    };
    k = 0;
    while (cmd2[k] != NULL)
    {
        printf("cmd2[%d] = %s\n", k, cmd2[k]);
        k++;
    };
    if (strlen(infile))
    {
        printf("input redirection file name: %s\n", infile);
    }
    if (strlen(outfile))
    {
        printf("output redirection file name: %s\n", outfile);
    }
}
printf("return code is %d\n", i);

return continueExecute;
```

```
}
int parse_command(char *line, char **cmd1, char **cmd2, char *infile, char *outfile)
{
    int returnCode = 9;
    int cmd1Index = 0;
    int cmd2Index = 0;

    char delimin[2] = " ";
    char outputRedirectedTo[3]; //Could be >>,>,<
    char *token;
    char *copyOfLine = (char *) malloc (strlen(line) + 1);

    bool isOutputRedirected = false;
    bool pipe = false;
    bool reset = true;
    copyOfLine = strcpy(copyOfLine, line);
    token = strtok(copyOfLine, delimin); //Tokenizing

    while (token != NULL)
    {
        if (strstr(token, "quit") || strstr(token, "exit")) //Quiting
        {
            printf("Program terminates successfully by the user\n");
            returnCode = 0;
            break;
        }
        else if (reset == true) //Taking in command, otherwise we will assume it is an argument
        {

            reset = false;
            //Return code stuff
            //This way we can make sure that we have all executables
            if(true)
            {
                if (pipe == true)
                {

                    cmd2[cmd2Index] = token + '\0';
                    cmd2Index++;
                    if(isOutputRedirected == true)
                    {
                      returnCode += 4;
                      isOutputRedirected = false;
                    }else
                    {
```

```
                returnCode = 5;
            }
        }
        else
        {
            //Simple command
            cmd1[cmd1Index] = token + '\0';
            cmd1Index++;
            returnCode = 1;
        }
    }else
    {
        returnCode = 9;
        break;
    }
}
else if (strstr(token, "|")) //Piping include a space
{
    reset = true;
    pipe = true;
    if (strlen(token) > 1) //more than 1 character, an argument has been attached to it
    {
        char substringToken[3];
        memcpy( substringToken, &token[0], 2 ); //Figure out what the hell i'm doing here.
        substringToken[2] = '\0';
    }
}
else if (strstr(token, ">>") || strstr(token, ">") || strstr(token, "<")) //Output redirected
{
    isOutputRedirected = true;
    strncpy(outputRedirectedTo, token, sizeof(outputRedirectedTo));
    outputRedirectedTo[sizeof(outputRedirectedTo) - 1] = '\0';
}
else if (isOutputRedirected == true)
{
    if (strstr(outputRedirectedTo, ">>"))
    {
        strcpy(outfile, token);
        if (pipe == true)
        {
            returnCode = 7;
        }
        else
        {
            returnCode = 3;
```

```
            }
        }
        else if (strstr(outputRedirectedTo, ">"))
        {
            strcpy(outfile, token);
            if (pipe == true)
            {
                returnCode = 8;
            }
            else
            {
                returnCode = 4;
            }
        }
        else if (strstr(outputRedirectedTo, "<"))
        {

            if (pipe == true)
            {
                returnCode = 6;
            }else
            {
                strcpy(infile, token);
                returnCode = 2;
            }
        }

    }
    else
    {
        if (pipe == true)
        {
            cmd2[cmd2Index] = token + '\0';
            cmd2Index++;
        }
        else
        {
            cmd1[cmd1Index] =  token + '\0';
            cmd1Index++;
        }
    }

    token = strtok(NULL, delimin);
}
cmd1[cmd1Index] = NULL;
```

```
    cmd2[cmd2Index] = NULL;
    return returnCode;
}


void exec_cmd(char** cmd1)
{
    pid_t pid;
    logInfo("Forking simple command");
    if ((pid = fork()) == -1)
    {
        perror("fork error");
    }
    else if (pid == 0)
    {
        logInfo("Executing simple command");
        execvp(cmd1[0], cmd1);
        printf("%s: command not found\n", cmd1[0]);
        logInfo("simple command failed");
    }else
    {
        waitpid(pid, NULL, 0);
    }
    logInfo("Done with simple command");

}
void exec_cmd_in(char** cmd1, char* infile)
{
    pid_t pid;
    logInfo("exec_cmd_in: Starting fork");
    if ((pid = fork()) == -1)
    {
      logInfo("Fork failed");
        perror("fork error");
    }
    else if (pid == 0)
    {
      logInfo("exec_cmd_in: Opening infile");
        int fd = open(infile, O_RDONLY);
        dup2(fd, 0);
        close(fd);
        logInfo("exec_cmd_in: Executing command");
        execvp(cmd1[0], cmd1);
        exit(1);
    }else
```

```
    {
        waitpid(pid, NULL, 0);
        logInfo("exec_cmd_in: Ending");
    }
}
void exec_cmd_opt_in_append(char** cmd1, char* infile, char* outfile)
{
    pid_t pid;
    logInfo("exec_cmd_opt_in: Starting fork");
    if ((pid = fork()) == -1)
    {
        logInfo("exec_cmd_opt_in: fork error");
        perror("fork error");
    }
    else if (pid == 0)
    {
        if(outfile[0] != '\0')
        {
            logInfo("exec_cmd_opt_in: Opening outfile");
            int outFD = open(outfile, O_APPEND | O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
            if(infile[0] != '\0')
            {
                logInfo("exec_cmd_opt_in: Opening infile");
                int inFD = open(infile, O_RDONLY);
                dup2(inFD, 0);
                close(inFD);
            }
            dup2(outFD, 1);
            close(outFD);
            logInfo("exec_cmd_opt_in: Executing command");
            execvp(cmd1[0], cmd1);
            exit(1);
        }
    }
    else
    {
        waitpid(pid, NULL, 0);
        logInfo("exec_cmd_opt_in: Done");
    }
}
void exec_cmd_opt_in_write(char** cmd1, char* infile, char* outfile)
{
    pid_t pid;
    logInfo("exec_cmd_opt_in_write: Forking");
    if ((pid = fork()) == -1)
```

```
    {
        logInfo("exec_cmd_opt_in_write: Error Forking");
        perror("fork error");
    }
    else if (pid == 0)
    {
        logInfo("exec_cmd_opt_in_write: Opening outfile");
        if(outfile[0] != '\0')
        {
            int outFD = open(outfile, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
            if(infile[0] != '\0')
            {
                int inFD = open(infile, O_RDONLY);
                dup2(inFD, 0);
                close(inFD);
            }
            dup2(outFD, 1);
            close(outFD);

            logInfo("exec_cmd_opt_in_write: Executing command");
            execvp(cmd1[0], cmd1);
            exit(1);
        }
    }
    else
    {
        waitpid(pid, NULL, 0);

    }
}
void exec_pipe(char** cmd1, char** cmd2)
{
    int pipefd[2];
    int pid2;
    int pid1;

    logInfo("exec_pipe: Creating pipe");
    pipe(pipefd);
    logInfo("exec_pipe: Forking");
    if ((pid1 = fork()) == -1)
    {
        perror("fork error");
    }
    else if (pid1 == 0)
    {
```

```
        logInfo("exec_pipe: Executing cmd1");

        close(pipefd[0]);
        dup2(pipefd[1], 1);
        close(pipefd[1]);

        execvp(cmd1[0], cmd1); //Ends child
        printf("%s: command not found\n", cmd1[0]);
        exit(1);
    }
    //In this situation I did not fork in the child because I could not wait for pid2 to finish inside the
parent

    pid2 = fork();

    if (pid2 == 0) //CHILD
    {
        waitpid(pid1, NULL, 0); //Waiting for child to finish
        logInfo("exec_pipe: Executing cmd2");
        close(pipefd[1]);
        dup2(pipefd[0], 0);
        close(pipefd[0]);

        execvp(cmd2[0], cmd2);
        printf("%s: command not found\n", cmd2[0]);
        exit(1);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid2, NULL, 0);
}
void exec_pipe_in(char** cmd1, char** cmd2, char* infile)
{
    int pipefd[2];
    int pid2;
    int pid1;
    int fd;
    logInfo("exec_pipe_in: Creating pipe");
    pipe(pipefd);
    logInfo("exec_pipe_in: Forking");
    if ((pid1 = fork()) == -1)
    {
        perror("fork error");
    }
```

```
    else if (pid1 == 0)
    {
        logInfo("exec_pipe_in: Executing cmd1");
        fd = open(infile, O_RDONLY);
        close(pipefd[0]);
        dup2(pipefd[1], 1);

        dup2(fd, 0);
        close(fd);
        close(pipefd[1]);

        execvp(cmd1[0], cmd1); //Ends child
        printf("%s: command not found\n", cmd1[0]);
        exit(1);
    }
    //In this situation I did not fork in the child because I could not wait for pid2 to finish inside the
parent

    pid2 = fork();

    if (pid2 == 0) //CHILD
    {
        waitpid(pid1, NULL, 0); //Waiting for child to finish
        logInfo("exec_pipe_in: Executing cmd2");
        close(pipefd[1]);
        dup2(pipefd[0], 0);
        close(pipefd[0]);

        execvp(cmd2[0], cmd2);
        printf("%s: command not found\n", cmd2[0]);
        exit(1);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid2, NULL, 0);
}
void exec_pipe_opt_in_append(char** cmd1,char** cmd2,char* infile,char* outfile)
{
    int pipefd[2];
    pid_t pid1;
    pid_t pid2;
    int fd;
    int outFD;
    pipe(pipefd);
```

```
    pid1 = fork();

    if (pid1 < 0)
    {
        perror("error with fork");
    }
    else if (pid1 == 0)
    {
        logInfo("exec_pipe_opt_in_append: Executing cmd1");
        fd = open(infile, O_RDONLY);

        close(pipefd[0]);
        dup2(pipefd[1], 1);
        dup2(fd, 0);
        close(fd);
        close(pipefd[1]);
        execvp(cmd1[0], cmd1);
        printf("%s: command not found\n", cmd1[0]);
        exit(1);
    }

    pid2 = fork();
    if (pid2 == 0)
    {
        close(pipefd[1]);
        dup2(pipefd[0], 0);
        close(pipefd[0]);

        logInfo("exec_pipe_opt_in_append: Executing cmd2");

        outFD = open(outfile, O_APPEND | O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
        dup2(outFD, 1);
        close(outFD);

        execvp(cmd2[0], cmd2);
        printf("%s: command not found\n", cmd2[0]);
        exit(1);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid2, NULL, 0);
}
void exec_pipe_opt_in_write(char** cmd1,char** cmd2,char* infile,char* outfile)
{
```

```
int pipefd[2];
pid_t pid1;
pid_t pid2;

pipe(pipefd);
pid1 = fork();

if (pid1 < 0)
{
    perror("error with fork");
}
else if (pid1 == 0)
{
    logInfo("exec_pipe_opt_in_write: Executing cmd1");
    int fd = open(infile, O_RDONLY);

    close(pipefd[0]);
    dup2(pipefd[1], 1);
    dup2(fd, 0);
    close(fd);
    close(pipefd[1]);
    execvp(cmd1[0], cmd1);
    printf("%s: command not found\n", cmd1[0]);
    exit(1);
}
if (pid1 > 0)
{
    pid2 = fork();
    if (pid2 == 0)
    {
        close(pipefd[1]);
        dup2(pipefd[0], 0);
        close(pipefd[0]);

        logInfo("exec_pipe_opt_in_write: Executing cmd2");
        int outFD = open(outfile, O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IWUSR);
        dup2(outFD, 1);
        close(outFD);

        execvp(cmd2[0], cmd2);
        printf("%s: command not found\n", cmd2[0]);
        exit(1);
    }
}
close(pipefd[0]);
```

```
    close(pipefd[1]);
    waitpid(pid2, NULL, 0);
}
void logInfo(char *line)
{
    int fd1;
    fd1 = dup(1);
    int fd = open("log.txt", O_APPEND | O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    dup2(fd, 1);
    close(fd);
    printf("PID: %d : %s\n", getpid(),line);
    dup2(fd1, 1);
}
```