Cory McDonald

Synopsis: This project operates both in “shell”-mode and argument mode as both situations call the function interface given a user input. From there the interface function can fork() and execute a command of return code 1 or otherwise simply parse the command.

For this assignment I learned a lot more about how processes are executed by the system and how the shell parses and handles commands. Prior to this assignment I had only a vague understanding of how the function fork() worked and even less about how child processes are handled. After completing the homework I was able to gain a much better understanding of parent and child processes along with how the operating system handles fork. Another key piece of knowledge that I’ve learned is when parsing user input the function fgets() takes in the return character from the user input. I figured this out when my simple command failed to function correctly and I wasn’t able to execute simple commands.

For the testing of my program I created a function that validates that the example command return codes properly match up from the parse_command function. This was a very effective way to test because if every different revision of my code I was able to quickly and successfully test that none of my changes broke any of the code that was previously working.

Initially when I implemented the test function only 23/26 of the tests successfully worked. Specifically the return code 2 and both return code 9. To solve this I logically followed my code structure to find where the errors were occurring and fixed it.

```
//Author: Cory McDonald

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <limits.h>
#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);
bool interface(char *line);
void unitTest();

int main ( int argc, char *argv[] )
{
   bool debug = true;
   if(debug == true)
   {
      unitTest();
   }
   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 command
```

```
            if ((strlen(userCommand) > 0) && (userCommand[strlen (userCommand) - 1] == '\n'))
                userCommand[strlen (userCommand) - 1] = '\0';

            if (userCommand != NULL)
            {
                if ( interface(userCommand) == true)
                {
                    active = false;
                    break;
                }
            }
        }
    }
    return 0;
}
bool interface(char *line)
{
    bool continueExecute = false;
    char infile[CSTRSIZE];
    char outfile[CSTRSIZE];
    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);

    printf("return code is %d\n", i);
    if(i == 0)
    {
        continueExecute = true;
    }
    if(i == 1)
    {
        pid_t pid;
        if ((pid = fork()) == -1)
        {
            perror("fork error");
        }
        else if (pid == 0)
        {
            execvp(cmd1[0], cmd1);
            printf("%s: command note found\n", cmd1[0]);
        }else
        {
             waitpid(pid, NULL, 0);
        }
    }
    else 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);
        }
    }

    return continueExecute;
}

void unitTest()
{
    char infile[CSTRSIZE];
    char outfile[CSTRSIZE];
    char *cmd1[CMDSIZE];
    char *cmd2[CMDSIZE];
    int i;
    int success =0;

    cmd1[0] = NULL;
    cmd2[0] = NULL;
    char returnCode1Test1[50] = "ls";
    char returnCode1Test2[50] = "ls –l";
    char returnCode1Test3[50] = "ls –l –a";

    char returnCode2Test1[50] = "wc < filename";

    char returnCode3Test1[50] = "ls >> outputfile";
    char returnCode3Test2[50] = "ls –l >> outputfile";
    char returnCode3Test3[50] = "ls –l –a >> outputfile";
    char returnCode3Test4[50] = "wc < filename >> outputfile";

    char returnCode4Test1[50] = "ls > outputfile";
    char returnCode4Test2[50] = "ls –l > outputfile";
    char returnCode4Test3[50] = "ls –l –a > outputfile";
    char returnCode4Test4[50] = "wc < filename > outputfile";


    char returnCode5Test1[50] = "ls | grep c";
    char returnCode5Test2[50] = "ls –l | grep c";
    char returnCode5Test3[50] = "ls –l –a | grep c";

    char returnCode6Test1[50] = "wc < filename | grep 3";

    char returnCode7Test1[50] = "ls | grep c >> outputfile";
    char returnCode7Test2[50] = "ls –l | grep c >> outputfile";
    char returnCode7Test3[50] = "ls –l –a | grep c >> outputfile";
    char returnCode7Test4[50] = "wc < filename | grep 3 >> outputfile";

    char returnCode8Test1[50] = "ls | grep c > outputfile";
    char returnCode8Test2[50] = "ls –l | grep c > outputfile";
    char returnCode8Test3[50] = "ls –l –a | grep c > outputfile";
    char returnCode8Test4[50] = "wc < filename | grep 3 > outputfile";

    char returnCode9Test1[50] = "cat file1 | grep c | wc";
    char returnCode9Test2[50] = "netbeans&";


    i = parse_command(returnCode1Test1, cmd1, cmd2, infile, outfile);
    if(i==1) { success++; } else { printf("%s test failed. 1 != %d\n", returnCode1Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode1Test2, cmd1, cmd2, infile, outfile);
    if(i==1) { success++; } else { printf("%s test failed. 1 != %d\n", returnCode1Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode1Test3, cmd1, cmd2, infile, outfile);
    if(i==1) { success++; } else { printf("%s test failed. 1 != %d\n", returnCode1Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    i = parse_command(returnCode2Test1, cmd1, cmd2, infile, outfile);
    if(i==2) { success++; } else { printf("%s test failed. 2 != %d\n", returnCode2Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    i = parse_command(returnCode3Test1, cmd1, cmd2, infile, outfile);
    if(i==3) { success++; } else { printf("%s test failed. 3 != %d\n", returnCode3Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode3Test2, cmd1, cmd2, infile, outfile);
    if(i==3) { success++; } else { printf("%s test failed. 3 != %d\n", returnCode3Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode3Test3, cmd1, cmd2, infile, outfile);
    if(i==3) { success++; } else { printf("%s test failed. 3 != %d\n", returnCode3Test3 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode3Test4, cmd1, cmd2, infile, outfile);
    if(i==3) { success++; } else { printf("%s test failed. 3 != %d\n", returnCode3Test4 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    i = parse_command(returnCode4Test1, cmd1, cmd2, infile, outfile);
    if(i==4) { success++; } else { printf("%s test failed. 1 != %d\n", returnCode4Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode4Test2, cmd1, cmd2, infile, outfile);
    if(i==4) { success++; } else { printf("%s test failed. 1 != %d\n", returnCode4Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode4Test3, cmd1, cmd2, infile, outfile);
    if(i==4) { success++; } else { printf("%s test failed. 1 != %d\n", returnCode4Test3 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode4Test4, cmd1, cmd2, infile, outfile);
    if(i==4) { success++; } else { printf("%s test failed. 4 != %d\n", returnCode4Test4 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    i = parse_command(returnCode5Test1, cmd1, cmd2, infile, outfile);
    if(i==5) { success++; } else { printf("%s test failed. 5 != %d\n", returnCode5Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode5Test2, cmd1, cmd2, infile, outfile);
    if(i==5) { success++; } else { printf("%s test failed. 5 != %d\n", returnCode5Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode5Test3, cmd1, cmd2, infile, outfile);
    if(i==5) { success++; } else { printf("%s test failed. 5 != %d\n", returnCode5Test3 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    i = parse_command(returnCode6Test1, cmd1, cmd2, infile, outfile);
    if(i==6) { success++; } else { printf("%s test failed. 6 != %d\n", returnCode6Test1 ,i); }
```

```
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    i = parse_command(returnCode7Test1, cmd1, cmd2, infile, outfile);
    if(i==7) { success++; } else { printf("%s test failed. 7 != %d\n", returnCode7Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode7Test2, cmd1, cmd2, infile, outfile);
    if(i==7) { success++; } else { printf("%s test failed. 7 != %d\n", returnCode7Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode7Test3, cmd1, cmd2, infile, outfile);
    if(i==7) { success++; } else { printf("%s test failed. 7 != %d\n", returnCode7Test3 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode7Test4, cmd1, cmd2, infile, outfile);
    if(i==7) { success++; } else { printf("%s test failed. 7 != %d\n", returnCode7Test4 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;


    i = parse_command(returnCode8Test1, cmd1, cmd2, infile, outfile);
    if(i==8) { success++; } else { printf("%s test failed. 8 != %d\n", returnCode8Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode8Test2, cmd1, cmd2, infile, outfile);
    if(i==8) { success++; } else { printf("%s test failed. 8 != %d\n", returnCode8Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode8Test3, cmd1, cmd2, infile, outfile);
    if(i==8) { success++; } else { printf("%s test failed. 8 != %d\n", returnCode8Test3 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode8Test4, cmd1, cmd2, infile, outfile);
    if(i==8) { success++; } else { printf("%s test failed. 8 != %d\n", returnCode8Test4 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    i = parse_command(returnCode9Test1, cmd1, cmd2, infile, outfile);
    if(i==9) { success++; } else { printf("%s test failed. 9 != %d\n", returnCode9Test1 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;
    i = parse_command(returnCode9Test2, cmd1, cmd2, infile, outfile);
    if(i==9) { success++; } else { printf("%s test failed. 9 != %d\n", returnCode9Test2 ,i); }
i=0;cmd1[0] = NULL; cmd2[0] = NULL;

    printf("%d/26 provided tests completed successfully\n", success);
}

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
      //TODO Get files located in system PATH variable.
      //This way we can make sure that we have all executables
      if(!strstr(token, "cat") && !strstr(token, "&") )
      {
         if (pipe == true)
         {

            cmd2[cmd2Index] = token + '\0';
            cmd2Index++;
            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';
            // printf("The command line argument to the user command and program is: [%s]\n",
substringToken );
        }
        // printf("Pipe: yes\n");
    }
    else if (strstr(token, ">>") || strstr(token, ">") || strstr(token, "<")) //Output redirected
    {
        isOutputRedirected = true;
        strncpy(outputRedirectedTo, token, sizeof(outputRedirectedTo));
        outputRedirectedTo[sizeof(outputRedirectedTo) - 1] = '\0';
        // printf("Output Direction: %s\n", token);
    }
    else if (isOutputRedirected == true)
    {
        if (strstr(outputRedirectedTo, ">>"))
        {
            if (pipe == true)
            {
                returnCode = 7;
            }
            else
            {
                returnCode = 3;
            }
        }
        else if (strstr(outputRedirectedTo, ">"))
        {
            if (pipe == true)
            {
                returnCode = 8;
            }
            else
            {
                returnCode = 4;
                }
            }
            else if (strstr(outputRedirectedTo, "<"))
            {
                if (pipe == true)
                {
                    returnCode = 6;
                }else
                {
                    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;
}
```