Andrew Ma

Lab 3

CPE 435

2/1/21

Theory

A shell or a command line interpreter is the program that takes in a user input and sends the commands to the OS. The OS can then respond back to the user. There can be an always displayed message, or a command prompt. The first word in the command line (arg[0]) is the name of a command that is in the PATH, or a path to an executable file. The other words separated by spaces are the other arguments (args). The shell can interpret pipe commands with '|' character, and interpret redirecting stdout with the '>' character. The shell executes in a loop until it exits.

The strtok() splits a string into a sequence of token. On the first call, the string to be split is passed in as the first argument. Each call after that passes in NULL as the first argument, but the return value will be the split string portion. The second argument is the delimiter character, and for the shell we split with '|' for pipe, '>' for redirect, and ' ' for args.

The dup() function creates a copy of a file descriptor. In the program I use it to create a copy of stdout and stdin, that will be restored after each command output. The dup2() function is used to replace a file descriptor with another one. We can create a file descriptor for an output file, and replace stdout with the output file descriptor using dup2().

The pipe() function creates a one direction pipe that can be used for communicating between processes. It takes in an array of pointers, and the second item in the array is the WRITE end, and the first item in the array is the READ end. Data written to the write end can be read from the read end of the pipe.

The execvp() function takes in the executable name as the first argument, and the command line arguments array as the second argument. It will execute the command.

Observations

Compile output

```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_03/cmake-build-debug-wsl$ gcc ../lab3.c -o lab3 && ./lab3
am0165 >
```

a. User commands

```
am0165 > ls
CMakeFiles  Lab_03.cbp  Makefile  Testing  cmake_install.cmake  lab3
am0165 > date
Mon Feb  8 23:44:58 CST 2021
am0165 > ls -l -a
total 40
drwxrwxrwx 1 u u  4096 Feb  8 23:44 .
drwxrwxrwx 1 u u  4096 Feb  8 23:39 ..
drwxrwxrwx 1 u u  4096 Feb  8 23:38 CMakeFiles
-rwxrwxrwx 1 u u  5563 Feb  8 22:10 Lab_03.cbp
-rwxrwxrwx 1 u u  4852 Feb  8 22:10 Makefile
drwxrwxrwx 1 u u  4096 Feb  8 22:10 Testing
-rwxrwxrwx 1 u u  1565 Feb  8 22:10 cmake_install.cmake
-rwxrwxrwx 1 u u 17552 Feb  8 23:44 lab3
am0165 >
```

b. Commands with I/O re-direction

```
am0165 > ls -l > a.txt
am0165 > cat a.txt
total 40
drwxrwxrwx 1 u u  4096 Feb  8 23:38 CMakeFiles
-rwxrwxrwx 1 u u  5563 Feb  8 22:10 Lab_03.cbp
-rwxrwxrwx 1 u u  4852 Feb  8 22:10 Makefile
drwxrwxrwx 1 u u  4096 Feb  8 22:10 Testing
-rwxrwxrwx 1 u u     0 Feb  8  2021 a.txt
-rwxrwxrwx 1 u u  1565 Feb  8 22:10 cmake_install.cmake
-rwxrwxrwx 1 u u 17552 Feb  8 23:44 lab3
```

c. Commands with single pipe

```
am0165 > who
am0165 > who | wc -l
0
```

d. Command with piping and redirection

```
am0165 > ls -l | sort > b.txt
am0165 > cat b.txt
-rwxrwxrwx 1 u u     0 Feb  8  2021 b.txt
-rwxrwxrwx 1 u u   331 Feb  8 23:45 a.txt
-rwxrwxrwx 1 u u  1565 Feb  8 22:10 cmake_install.cmake
-rwxrwxrwx 1 u u  4852 Feb  8 22:10 Makefile
-rwxrwxrwx 1 u u  5563 Feb  8 22:10 Lab_03.cbp
-rwxrwxrwx 1 u u 17552 Feb  8 23:44 lab3
drwxrwxrwx 1 u u  4096 Feb  8 22:10 Testing
drwxrwxrwx 1 u u  4096 Feb  8 23:38 CMakeFiles
total 40
```

Conclusion

The shell program did work as expected. I learned what shells programs are, how to split by specific characters in C (strtok()), how to copy and replace file descriptors (dup() and dup2()), how to create and use pipes (pipe() with a read end and a write end), and how to execute programs with execvp(). I also got more practice with creating processes because I created new processes and executed the programs inside the child processes.

Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <ctype.h>

void trim(char *string);
int runCommand(char * commandString);
```

```
int main(int argc, char *argv[]) {
   int userInputLength = 512;
   char userInput[userInputLength];

   char *pipeCommand;
   char *redirOutputFileName;
   int redirOutputFD;
   int stdIn;
   int stdOut;
   int pipeIO[2];


   while (1) {
       // show shell prompt
       printf("am0165 > ");

       // save original stdin and stdout and restore after running commands
       stdIn = dup(0);
       stdOut = dup(1);

       // clear userInput buffer
       bzero(userInput, userInputLength);
       // get userInput
       fgets(userInput, userInputLength, stdin);

       // remove newline character at end
       userInput[strlen(userInput) - 1] = '\0';

       // split by '>', and save second split part to redirOutputFileName
       strtok(userInput, ">");
       redirOutputFileName = strtok(NULL, ">");

       // if there is a right side of '>' (we want to redirect output to a
file)
       if (redirOutputFileName != NULL) {
           // trim the output file name, since there is a space after '>'
character
           trim(redirOutputFileName);

           // open redirect output file descriptor
           redirOutputFD = open(redirOutputFileName, O_CREAT | O_RDWR |
O_TRUNC, 0644);

           // replace stdout with redirOutputFD
           dup2(redirOutputFD, 1);
       }

       // split by '|', and save second split part to pipeCommand
       strtok(userInput, "|");
       pipeCommand = strtok(NULL, "|");

       // create a one way pipe, with two file descriptors stored in pipeIO
       // bytes written on PIPEDES[1] (WRITE end) can be read from
PIPEDES[0] (READ end)
       if (pipe(pipeIO) == -1) {
```

```
            perror("Error with pipe()");
            goto ERROR;
        }

        if (pipeCommand != NULL) {
            trim(pipeCommand);

            pid_t PID = fork();
            if (PID == -1) {
                perror("Error with fork() for running userInput command");
                goto ERROR;
            } else if (PID == 0) {
                // child Process

                // replace stdout with pipe WRITE end
                dup2(pipeIO[1], 1);
                // close READ end in child process
                close(pipeIO[0]);

                // run userInput command, and the stdout will be piped
through WRITE end
                runCommand((char*)userInput);

                exit(0);
            } else {
                // parent Process

                // replace stdin with pipe READ end
                dup2(pipeIO[0], 0);
                // close WRITE end in parent process
                close(pipeIO[1]);

                // wait for child process to run userInput command, and write
stdout to pipe WRITE end, and exit
                wait(0);

                // run Pipe command
                runCommand(pipeCommand);
            }
        } else {
            // if no Pipe Command
            runCommand((char*)userInput);
        }

        dup2(stdIn, 0);
        dup2(stdOut, 1);
    }

    return 0;

    ERROR:
    return -1;
}

int runCommand(char * commandString) {
    char *args[32];
```

```
    // split pipeCommand by spaces
    args[0] = strtok(commandString, " ");
    int i = 0;
    while (args[i] != NULL) {
        i += 1;
        args[i] = strtok(NULL, " ");
    }


    pid_t PID = fork();
    if (PID == -1) {
        perror("Error with fork() for running pipe command");
        return -1;
    } else if (PID == 0) {
        // child process
        execvp(args[0], args);
    } else {
        // parent process

        // wait for child process to exit
        wait(0);
    }

    return 0;
}

void trim(char *string) {
    /* First remove leading spaces */

    const char *firstNonSpace = string;

    while (*firstNonSpace != '\0' && isspace(*firstNonSpace)) {
        ++firstNonSpace;
    }

    size_t len = strlen(firstNonSpace) + 1;

    memmove(string, firstNonSpace, len);

    /* Now remove trailing spaces */

    char *endOfString = string + len;

    while (string < endOfString && isspace(*endOfString)) {
        --endOfString;
    }

    *endOfString = '\0';

}
```