Operating Systems Lab

CPE 435-01

Lab 03: Shells

By: David Thornton

Lab Date: 25 January 2021

Lab Due: 01 February 2021

Demonstration Due: 01 February 2021

Introduction

The purpose of this lab is to give students an introduction to shells, piping, and build on forking.

Theory

Topic 1: Shells

- “In computing, a shell is a computer program which exposes an operating system's services to a human user or other program. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation. It is named a shell because it is the outermost layer around the operating system.” - Source

Topic 2: strtok()

- “A sequence of calls to this function split str into tokens, which are sequences of contiguous characters separated by any of the characters that are part of delimiters. … Once the terminating null character of str is found in a call to strtok, all subsequent calls to this function (with a null pointer as the first argument) return a null pointer.” - Source

Topic 3: dup() and dup2()

- “In Unix-like operating systems, dup (short for "duplicate") and dup2 system calls create a copy of a given file descriptor. This new descriptor actually does not behave like a copy, but like an alias of the old one.” - Source

Topic 4: pipe()

- “Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).” - Source

Topic 5: execvp()

- “The execvp() function replaces the current process image with a new process image specified by file. The new image is constructed from a regular, executable

file called the new process image file. No return is made because the calling process image is replaced by the new process image." - Source

Lab Assignment

You will create your own linux shell by writing a c program. Please go through the demo codes provided and try to understand the functions mentioned in the hint section.

Your new shell should support following commands and similar commands:

user commands, such as ls, date, ls –l –a
commands with I/O re-direction, ex : ls –l > a.txt
commands with a single pipe, ex : who | wc –l
command with piping and redirection; ex: ls -l | sort > b.txt

Like all Linux shells, your shell executes a loop. It prints the shell prompt, reads the command line (terminated with NULL), parses the command line and creates its arguments, executes the command with its arguments, then waits until the command finishes. It should be a child process which runs a command.

Following image shows an example solution in action where ls command is operated first, and others subsequently until the entered command is exit.

```
Myshell>>ls
a.out  cat  fofo  kp  lab2.pdf  Lab_Lec1.pdf  Lab_Lec1.ppt  myshell  myshell.c  myshell.c~  p21.cpp

Myshell>>ls -l -a
total 1032
drwxr-xr-x  2 ama0017 student   4096 Aug 27 22:42 .
drwx------ 62 ama0017      185  12288 Aug 27 22:33 ..
-rwxr-xr-x  1 ama0017 student   9830 Aug 27 11:27 a.out
-rw-r--r--  1 ama0017 student     90 Aug 27 22:42 cat
-rw-r--r--  1 ama0017 student    101 Aug 27 17:29 fofo
-rw-r--r--  1 ama0017 student     86 Aug 27 17:31 kp
-rw-r--r--  1 ama0017 student  69427 Aug 24 11:04 lab2.pdf
-rw-r--r--  1 ama0017 student 540308 Aug 24 11:04 Lab_Lec1.pdf
-rw-r--r--  1 ama0017 student 367104 Aug 24 12:31 Lab_Lec1.ppt
-rwxr-xr-x  1 ama0017 student   8836 Aug 27 22:41 myshell
-rw-r--r--  1 ama0017 student   1546 Aug 27 18:15 myshell.c
-rw-r--r--  1 ama0017 student   1539 Aug 27 17:32 myshell.c~
-rw-r--r--  1 ama0017 student    483 Aug 27 11:27 p21.cpp

Myshell>>ls -l | wc -l
12

Myshell>>date > rr

Myshell>>cat rr
Thu Aug 27 22:43:15 CDT 2015

Myshell>>exit
-bash-4.1$
```

Repeat the loop until exit command is entered

Hint

```
char *strtok(char *str, const char *delim);
```

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL.The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string.

int dup2(int oldfd, int newfd);

dup2() makes newfd be the copy of oldfd, closing newfd first if necessary, but note the following: If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed. If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.

char *gets(char *s) :

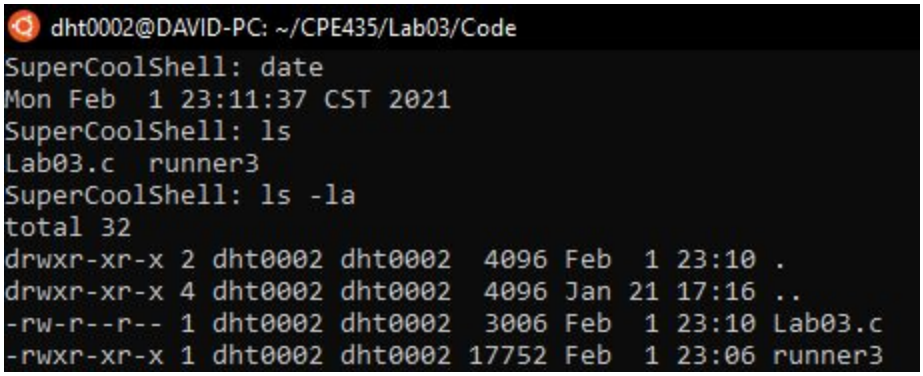
Reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte

int pipe(int pipefd[2]);

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1]refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

Observations

a. user commands
   i. ls, date, ls –l –a



```
dht0002@DAVID-PC: ~/CPE435/Lab03/Code
SuperCoolShell: date
Mon Feb  1 23:11:37 CST 2021
SuperCoolShell: ls
Lab03.c  runner3
SuperCoolShell: ls -la
total 32
drwxr-xr-x 2 dht0002 dht0002  4096 Feb  1 23:10 .
drwxr-xr-x 4 dht0002 dht0002  4096 Jan 21 17:16 ..
-rw-r--r-- 1 dht0002 dht0002  3006 Feb  1 23:10 Lab03.c
-rwxr-xr-x 1 dht0002 dht0002 17752 Feb  1 23:06 runner3
```

b. commands with I/O re-direction
   i. ls –l > a.txt

```
dht0002@DAVID-PC: ~/CPE435/Lab03/Code
SuperCoolShell: ls -l > a.txt
SuperCoolShell: ls
Lab03.c  a.txt  runner3
SuperCoolShell: cat a.txt
total 24
-rw-r--r-- 1 dht0002 dht0002  3006 Feb  1 23:14 Lab03.c
-rw-r--r-- 1 dht0002 dht0002     0 Feb  1 23:14 a.txt
-rwxr-xr-x 1 dht0002 dht0002 17752 Feb  1 23:14 runner3
```

c. commands with a single pipe
   i. ls | sort (for some reason `who` did not work on my system)

```
dht0002@DAVID-PC: ~/CPE435/Lab03/Code
SuperCoolShell: ls
1234asdf.txt  SuperCoolShell.c  runner  sample.txt  this_is_a_file.txt  zzz.txt
SuperCoolShell: ls | sort
1234asdf.txt
SuperCoolShell.c
runner
sample.txt
this_is_a_file.txt
zzz.txt
```

d. command with piping and redirection
   i. ls -l | sort > b.txt

```
dht0002@DAVID-PC: ~/CPE435/Lab03/Code
SuperCoolShell: ls
SuperCoolShell.c  runner
SuperCoolShell: ls -l | sort > b.txt
SuperCoolShell: cat b.txt
-rw-r--r-- 1 dht0002 dht0002     0 Feb  3 17:27 b.txt
-rw-r--r-- 1 dht0002 dht0002  4419 Feb  3 17:19 SuperCoolShell.c
-rwxr-xr-x 1 dht0002 dht0002 17248 Feb  3 17:19 runner
total 28
```

Conclusion

This lab was successful in introducing me to the concept of shells, piping, built on forking.
[Demo link]

Appendix

Appendix 1: lab03.c

```
// ****************************************
// Program Title: Lab 03
// Project File: SuperCoolShell.c
// Name: David Thornton
// Course Section: CPE-435, SP 2021
// Due Date: 02/01/2021
// ****************************************

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

#define LINE_MAX 100 // how long can a line be
#define ARGS_MAX 10 // how many different commands/symbols/options

int main(int argc, char* argv[])
{
      while(1) // infinite loop
      {
            int i = 0;
            int j = 0;
            int needsRedirect = 0;
            int needsPipe = 0;
            char* temp;
            char* myBrokenArgs[ARGS_MAX];
            char* myBrokenArgs2[ARGS_MAX];
            char myString[LINE_MAX];
            char myFileName[LINE_MAX];

            printf("SuperCoolShell: ");
            fgets(myString, LINE_MAX, stdin); // get input line
            myString[strlen(myString) - 1] = '\0'; // replace newline char with NULL
```

```
myBrokenArgs[0] = strtok(myString," "); // assumption in lab doc

while(myBrokenArgs[i] != NULL) // check for input
{
      if(myBrokenArgs[i][0] == '>') // check for output file
      {
            needsRedirect = 1;
            myBrokenArgs[i] = NULL;
            temp = strtok(NULL, ">");
            sprintf(myFileName, "%s", temp);
            break;
      }
      else if(myBrokenArgs[i][0] == '<') // check for i file
      {
            needsRedirect = 2;
            myBrokenArgs[i] = NULL;
            temp = strtok(NULL, "<");
            sprintf(myFileName, "%s", temp);
            break;
      }
      else if(myBrokenArgs[i][0] == '|') // check for piping
      {
            needsPipe = 1;
            myBrokenArgs[i] = NULL;
            myBrokenArgs2[0] = strtok(NULL, " ");
            while(myBrokenArgs2[j] != NULL) // for the 2nd command
            {
                  if(myBrokenArgs2[j][0] == '>') // check for output file in pipe
                  {
                        needsRedirect = 1;
                        myBrokenArgs2[j] = NULL;
                        temp = strtok(NULL, ">");
                        sprintf(myFileName, "%s", temp);
                  break;
                  }
                  else if(myBrokenArgs2[j][0] == '<') // check for input file in pipe
                  {
                        needsRedirect = 2;
                        myBrokenArgs2[j] = NULL;
                        temp = strtok(NULL, "<");
                        sprintf(myFileName, "%s", temp);
                        break;
```

```
                        }
                        myBrokenArgs2[++j] = strtok(NULL, " ");
                    }
                    break;
                }
                else
                {
                    myBrokenArgs[++i] = strtok(NULL, " ");
                }
            }
/*****************************************************************************************/
            if(needsPipe == 1) // if piping is needed
            {
                int myPipingDescriptors[2];
                if(pipe(myPipingDescriptors) == -1)
                {
                    exit(0);
                }
                int id = fork();
                if(id == 0) // child 1-a
                {
                    dup2(myPipingDescriptors[1],1);
                    close(myPipingDescriptors[0]);
                    close(myPipingDescriptors[1]);
                    execvp(myBrokenArgs[0], myBrokenArgs);
                    exit(0);
                }
                else if(id > 0) // parent process
                {
                    if(fork() ==  0) // child 1-b
                    {
                        dup2(myPipingDescriptors[0],0);
                        close(myPipingDescriptors[0]);
                        close(myPipingDescriptors[1]);
                        if(needsRedirect == 1 || needsRedirect == 2)
                        {
                            int pid = fork();
                            if(pid == 0) // child 2-b
                            {
                                if(needsRedirect == 1)
                                {
                                    int myDescriptor = open(myFileName, O_CREAT |
```

```
O_RDWR | O_TRUNC, 0644);
                                        dup2(myDescriptor, 1); // STDOUT
                                        close(myDescriptor);
                                    }
                                    else if(needsRedirect == 2)
                                    {
                                        int myDescriptor = open(myFileName, O_CREAT |
O_RDWR | O_TRUNC, 0644);
                                        dup2(myDescriptor, 2); // STDIN

                                        close(myDescriptor);
                                    }
                                }
                                else
                                {
                                    wait(0);
                                }
                            }
                            execvp(myBrokenArgs2[0], myBrokenArgs2);
                        }
                        else
                        {
                            close(myPipingDescriptors[0]);
                            close(myPipingDescriptors[1]);
                            wait(0);
                            wait(0);
                        }
                    }
                }
/*****************************************************************************************/
                if(needsPipe == 0) // if piping is not needed
                {
                    int pid = fork();
                    if(pid == 0) // make child process
                    {
                        if(needsRedirect == 1)
                        {
                            int myDescriptor = open(myFileName, O_CREAT | O_RDWR | O_TRUNC,
0644);
                            dup2(myDescriptor, 1); // STDOUT
                            close(myDescriptor);
                        }
```

```
                        else if(needsRedirect == 2)
                        {
                              int myDescriptor = open(myFileName, O_CREAT | O_RDWR | O_TRUNC,
0644);
                              dup2(myDescriptor, 2); // STDIN
                              close(myDescriptor);
                        }
                        execvp(myBrokenArgs[0], myBrokenArgs);
                  }
                  else
                  {
                        wait(0);
                  }
            }
/*****************************************************************************************/
      }
      return 0;
}
```