Project 1

Linux Shell, Commands and Help Feature

IMPORTANT DATES
Group Formation : May 24, 2017.
Assignment submission: June 8 and 9, 2017

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. This project can be completed on a Linux system. A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt myshell> and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the UNIX cat command.)

```
Myshell$ cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

```
Myshell$ cat prog.c &
```

The parent and child processes will run concurrently. The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() . Every shell is structured as the following loop:

1. Print out a prompt
2. Read a line of input from the user
3. Parse the line into the program name, and an array of parameters
4. Use the fork() system call to spawn a new child process, the child process then uses the execv() system call to launch the specified program and uses the wait() system call to wait for the child to terminate
5. When the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

```
#include <stdio.h>
#include <unistd.h>
#define MAX LINE 80 /* The maximum length command */

int main(){
   char * args[MAXLINE/2 + 1]; /* command line arguments */
   int shouldrun = 1; /* flag to determine when to exit program */
   while (shouldrun) {
      printf("myshell>");
      fflush(stdout);
      /**
      *   After reading user input, the steps are:
      *   (1) fork a child process using fork()
      *   (2) The child process will invoke execv()
      *   (3) If command included &, parent will invoke wait ()
      *   /
   }
   return 0 ;
}
```

Figure 1. Outline of a shell program

A C program that provides the general operations of a command-line shell is supplied in the code illustrated above in figure 1. The main() function presents the prompt myshell$ and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as should run equals 1; when the user enters exit at the prompt, your program will set should run to 0 and terminate.

This project is organized into three parts:

(1) Creating the child process and executing the command in the child.
(2) Implementing your own commands to perform the tasks of some common commands.
(3) Implementing a help command for your defined commands.

Part I— Creating a Child Process

The first task is to modify the main() function in Figure 1 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (args in Figure 1). For example, if the user enters the command ls –l /usr/include at the myshell> prompt, the values stored in the args array are:

```
args[0] = "ls"
args[1] = "-l"
args[2] = “/usr/include”
args[3] = NULL
```

This args array will be passed to the execv() function, which has the following prototype:

```
execv(char *command, char *params[]);
```

Here, command represents the command to be performed and params stores the parameters to this command. For this assignment, the execv() function should be invoked as execv(args[0], args). Be sure to check whether the user included an & to determine whether or not the parent process is to wait for the child to exit.

Part II— creating your own commands

The next task is to implement basic UNIX shell commands using your own programs. You will be using the system calls that we have seen in the class room to implement the commands. The commands that you are going to implement are:

1. File Management Commands:

i) **Creating a new file:** create a program that will create a new file in the current directory or any given directory. It should be name **cf** (create file). The command accepts one or two arguments representing a filename or file name and path.

```
myshell> cf test                will create the file test in the current directory
myshell> cf  test /home/user    will create the file test in the given path as a second argument.
```

ii) **Copying a file:** create a program that will copy the contents of one file into another file. The command should be named **cpf** (copy file). The command may accept different arguments depending on the copy type. The program should display error messages if the copy process cannot be performed. The following things should be considered to write this program.

a. If the source file (the file to be copied) does not exist, the program should display an error message.
b. If the destination file doesn't exist then the program should create a new file by using the given name.
c. If the destination file exists then it should be overwritten unless the –a command option is given.

Usage and arguments of the copy command is listed below.

```
Myshell$ cpf test backup
```

This will copy the file test to a file named backup in the current directory.

```
Myshell$ cpf  /home/student/test  /home/student/Desktop/backup
```

This will copy the file from the given path to the destination path.

Command line option:

-a: if the command option –a is specified, the program tries to copy the file to the destination file, if the destination file exists the file to be copied will be appended to the destination file.

```
Myshell$ cpf  -a  /home/student/test  /home/student/Desktop/backup
```

This will copy the file from the given path to the destination path. If the destination file exists then the two files will be appended.

iii) **Displaying contents of a file:** create a program that will display the contents of a given file. This program simulates the function of the **cat** command in UNIX shell. The program accepts one argument which is the file path and then displays the contents of the given file. You should name this command **display**. Usage of this command should look like:

```
Myshell$ display test
```

This will display the contents of the file **test** from the current directory. If the file is not found then the command displays an error message.

```
Myshell$ display /home/student/test
```

This will display the contents of the file **test** from the given path directory. If the file is not found then the command displays an error message.

2. Directory Management Commands:

i) **Creating a new directory:** create a program that will create a new directory in the current directory or any given directory. The program or command simulates the UNIX shell **mkdir** command. It should be name **ctdir** (create directory). The command can accept one argument or two arguments.

Myshell$ ctdir test	will create the directory named test in the current directory
Myshell$ ctdir test /home/user	will create the directory named test in the given path as a second argument.

ii) **Changing the working directory:** create a program that will be used to change the current working directory. It should be able to simulate the UNIX shell cd command. The command should be named **chgdir** (change directory). Usage and arguments of the copy command is listed below.

```
Myshell$ chgdir  /home/student/test
```

This will change the working directory to the test directory given in the path. If the directory is not find, then the command should display an error message.

3. **Listing current process information:** create a program that will be used to list processes that are running in the system. It should be able to simulate the UNIX shell **ps** command. The command should be named **pslist** (process list). Usage and arguments of the copy command is listed below.

```
Myshell$ chgdir  /home/student/test
```

Part III—Creating a Help command

The next task is to enable the shell to provide basic usage help for your defined commands. You should be able to make a help entry in your program about all commands and their usage in your help program. The user can use the help command to get basic usage information about the commands that are defined by you.
The name of the command should be **hlp.** It can accept no arguments or a single argument. If it is not given any argument it should display basic information about all of the commands that are defined by you. These commands are **cf, cpf, display, ctdir and chgdir.** One way to implement this command simply use a string array and provide the command name and the help texts that will be displayed when the help is called.

Basic usage of the help command should be as follows:

```
Myshell$ hlp
```

This will display the help contents for all commands defined by you.

```
Myshell$  hlp chgdir
```

This will display the help contents of the command chgdir only. If the user gives a command that is not defined, you should display an error message.

Project Submission and grading:

The final submission date will be June 8 and 9 2017 and you will present your project during submission. Any late submission will result in grade reduction and no late submission is allowed after three days of the submission date.

This project will be graded out of 15 points in which the 5 points will be based on the documentation and presentation. The remaining 10 points will be graded as part the program implementation. You are expected to provide the following as part of your project submission:

i. A project write up (documentation): This document should clearly describe you project specification, design and implementation and sample screen shots demonstrating the functionality of the programs.
ii. The project files: This includes all program codes and header files that are used in the program.
iii. A runnable executable file of the programs.