Com S 352 Spring 2018 Project 1:

UNIX Shell and History Feature

100 points

Due: Friday March 2, 2018, 11:59pm

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 any Linux, UNIX, or Mac OS X system.

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt **osh>** and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the UNIX cat command.)

osh> 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. This is similar in functionality to the new process creation illustrated in Figure 3.10 of textbook. 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

osh> 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() family.

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

This project is organized into two parts:

1. Creating the child process and executing the command in the child, and
2. Modifying the shell to allow a "history feature", "handling Multiple Commands", "handling Pipe(|)..MORE command" and "handling Change Directory (cd) command" (please see details below).

```
#include <stdio.h>
#include <unistd.h>
#define MAX LINE 80                    /* The maximum length command */
int main(void)
{
char *args[MAXLINE/2 + 1];             /* command line arguments */
 int shouldrun = 1;                    /* flag to determine when to exit program */
while (shouldrun) {
        printf("osh>");
        fflush(stdout);
/**
After reading user input, the steps are:
(1) fork a child process using fork()
(2) the child process will invoke execvp()
(3) if command included &, parent will invoke wait()
 */
}
 return 0;
}
```

Figure 1: Outline of simple shell.

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:

ps -ael at the osh> prompt, the values stored in the args array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

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

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

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(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

This part includes adding the following features to the Shell:

1. Creating a History Feature

The next task is to modify the shell interface program so that it provides a history feature that allows the user to access the most recently entered commands. The user will be able to access up to 10 commands by using the feature. The commands will be consecutively numbered starting at 1, and the numbering will continue past 10. For example, if the user has entered 35 commands, the 10 most recent commands will be numbered 26 to 35.

The user will be able to list the command history by entering the command history at the osh> prompt. As an example, assume that the history consists of the commands (from most to least recent):

ps, ls -l, top, cal, who, date

The command history will output:

6 ps

5 ls -l

4 top

3 cal

2 who

1 date

Your program should support two techniques for retrieving commands from the command history:

1. When the user enters **!!**, the most recent command in the history is executed.
2. When the user enters **!N** (a single ! followed by an integer N), the Nth command in the history is executed.

Continuing our example from above, if the user enters !!, the ps command will be performed; if the user enters !3, the command cal will be executed. Any command executed in this fashion should be echoed on the user's screen. The command should also be placed in the history buffer as the next command.

The program should also manage basic error handling. If there are no commands in the history, entering !! should result in a message "No commands in history". If there is no command corresponding to the number entered with/after the single !, the program should output "No such command in history."

2. Handling Multiple Commands

The simple command-line shell interface provided in Figure 1 can handle one command per line only, whereas, in actual Linux shell multiple commands can be entered in a line separated by ";", like

-bash-4.3$ date;cal;who

with the following **output** (outputs of the three commands):

```
Wed Sep 13 12:55:51 CDT 2017

   September 2017
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27
28 29 30

abi          pts/0        2018-01-28 13:24 (10.64.222.89)
abi          pts/1        2018-01-28 13:31 (10.64.222.89)
mabdulah     pts/2        2018-01-28 14:39 (10.25.71.69)
jcu          pts/8        2018-01-19 17:14 (10.24.46.63)
jcu          pts/9        2018-01-19 17:15 (10.24.46.63)
```

3. Handling pipe (|) and more command

The simple shell interface provided in Figure 1 does not handle the use of pipe (|) and “more" command, which is available in actual Linux shell,

-bash-4.3$ ls -l |more

with the following output:

```
-rwx--x--x.  1 mabdulah domain users   9160 Jan 25 16:28   cll
-rwx------.  1 mabdulah domain users    422 Sep 22 10:43   clone.c
-rwx------.  1 mabdulah domain users    896 Jan 25 15:50   abc.c
-rwx--x--x.  1 mabdulah domain users   8808 Jan 24 22:10   col
-rwx------.  1 mabdulah domain users    842 Jan 24 21:47   xyz.c
-rwx------.  1 mabdulah domain users    842 Jan 24 21:11   xyz.txt
--More—
```

4. Handling Change Directory (cd) command

The simple shell interface provided in Figure 1 does not handle the use of "Change Directory (cd)" command, which is available in actual Linux shell.

Examples:

- **Moving into a directory**:

  -bash-4.3$ **cd coms352**

- **Moving into multiple directories (or subdirectory):**

(aka moving to a directory by giving Absolute Path or Local Path)

  -bash-4.3$ **cd /home/isu/coms352/**

- **Moving back one directory:** **OR** **Multiple Directories**

  -bash-4.3$ **cd ..**

  -bash-4.3$ cd ../..
  -bash-4.3$ cd ../../..

- **Moving to Root directory:**

  -bash-4.3$ **cd /**

- **Moving to Home directory:**

  -bash-4.3$ **cd ~**

- **Switching to Previous Working directory:**

  -bash-4.3$ **cd -**

The above are a few examples (not comprehensive) of cd command, but for this project implementing the above examples should suffice for this part's full credit.

Modify the Figure 1 code or Write your own C program for a command-line shell that:

- **Forks a Child Process (this child process then executes the commands entered by user)**
  Note: (some commands like for example, the "exit" command to terminate the program may not need be sent to the Child process to execute.)
- **Can handle the features in Part-II above (1-4, as illustrated by the corresponding examples).**

Submission

- You should use C to develop the code.
- You need to turn in electronically by submitting a zip file named: Firstname_Lastname_Project1.zip.
- Source code must include proper documentation to receive full credit (you will lose 5% of your score, if the code is not documented).
- All projects require the use of a **make file** or a certain script file (accompanying with a **readme file** to specify how to use the script file to compile), such that the grader will be able to compile/build your executable by simply typing "make" or some simple command that you specify in your readme file.
- Source code must compile and run correctly on the department machine "pyrite", which will be used by the TA for grading.
- You are responsible for thoroughly testing and debugging your code. The TA may try to break your code by subjecting it to bizarre test cases.
- You can have multiple submissions, but the TA will grade only the last one.

Start as early as possible!