Operating System
CS 370
Fall 2016

Project: 2
A simple UNIX Shell

Intermediate Project Submission Date: Monday, September 19th 2016
Must be submitted before midnight

Due Date: Thursday, September 29th 2016
Must be submitted before midnight

Total Points: 210

Teaching Assistant: Pradip S Maharjan
Office Hours:
MW: 2:00 pm to 4:00 pm
TuTh:10:00 am to 12:00 pm

Course Web Page : `http://osserver.cs.unlv.edu/moodle/`

Requirements:

- Must be implemented in C (not C++).
- Must compile on cardiac.cs.unlv.edu.
- Must be submitted ONLY on the website.

Description:

In this assignment, you will write a Linux terminal shell. Your shell should act similar to bash, which is the default shell when you log into cardiac. At a high level, your shell should accomplish the following:

1. Print a prompt when waiting for input from the user.
2. Accept commands from the user and execute them until the user exits the shell.
    - If the command does not exist in the system, an error message should be displayed.
3. Accept the change directory command, and appropriately change the current working directory.
    - If the directory the user enters does not exist, an error message should be displayed.
4. Your shell must keep a history of the last 10 commands executed. The user should be able to cycle through the history using the up and down arrow keys.
5. Your shell must accept the pipe '|' operator. This will execute two commands, where the first process will output to the input of the second process.
6. Accept an exit command that will close your shell properly. Before executing the exit command you must print a prompt to confirm whether the use wants to exit or not.
7. Ignore characters such as the left arrow and right arrow.
8. Properly handle when the user types delete or backspace keys.
9. Create a custom command "merge" to merge the content of the two files into a third file. For Example: If someone types "merge file1.txt file2.txt > file3.txt" then contents of the files file1.txt and file2.txt should be merged to new file file3.txt. //Hint: use cat

Printing Prompt:

When your shell is waiting for input from the user, it should first print a prompt. The prompt should consist of the current working directory followed by the ">" character. Here is an example of the prompt:

```
/home/projects/cs370/spring13/shell->
```

After a user performs a command, the prompt should be printed again on a new line. If the user changes the current working directory, your prompt should show the change. We recommend using the printf() function from stdio.h for printing the prompt.

Accept Commands:

Your shell must accept commands from the user. The first step to implement this will be reading a line of input. This section will focus on what to do with the line of input after it is read. A command will consist of the name or path to the command, followed by zero or more arguments, separated by white spaces. For this assignment, we will define white space as one or more space characters. Here is an example of a command with arguments:

```
ls -al
```

In this argument, "ls" is the command, and "-al" is the first and only argument. To allow proper formatting the commands, we will put these commands into an array of char*. So, the first element of the array should be "ls", and the second element should be "-al". To parse the commands, we suggest using the *strtok* function from string.h. You can use just a space (" ") as the lone delimiter. Remember to consider the case where the user does not enter any command and hits the enter key. Your shell should simply print the prompt again on a new line.

Execute Commands:

Now that the user's command has been parsed into an array of char*, we can pass this to the OS to execute the command. To execute the command, use the *execvp()* function from unistd.h. You will pass your command array to this function which will execute the command. Remember that if this function executes properly, it will terminate the current process; so, it should be executed in a child process. The child process can be created with the fork() function which is also in unistd.h. While the child process is executing the command, the parent process should wait for the child to finish executing. This is done by using the *waitpid()* function which can be found in sys/wait.h. As you are waiting on the child process, you want to make this call blocking. You can do this by giving the *WUNTRACED*

option to the waitpid() call which is defined in sys/types.h. Remember that if the execvp() function does not complete successfully, the child process will not end. As this will only happen when an error occurs, this would be a good place to print the error message. This can be done easily by calling the *perror(NULL)* function. Make sure to end the child process by calling the *exit()* function. Once the command is done, you should print your prompt on a new line.

Change Directory

Your shell must accept a change directory command. This will be in the form:

```
cd path
```

where cd is the change directory command and the path is what you will change the directory to. As you will be parsing all commands from the user, this should be broken into two char* elements. You can simply call *chdir()* from unistd.h and pass the path string. Make sure to check if the command worked properly. This can be done by checking the return value of the function. If the function results in an error, you must print an error message. Once again this can be done by calling perror(). Finally, regardless of if the command was successful or not, the prompt should be printed on a new line.

Exit Command:

Your shell must accept the exit command. This command will be in the form:

```
exit
```

When you encounter this command, your shell should terminate. Please note that when your shell exits, you must restore the input parameters changed. This will be discussed later in this document.

Arrow Keys:

When the user is inputting commands into your shell, it should identify the left, right, up, and down arrow keys. To implement this, use the termios.h library to configure a custom input strategy. Remember that after changing the configuration, you should restore it to its original configuration before exiting the program. As shown in the slides in class, we will change the reads to be non-canonical and disable echoing. This will give us control on how to handle each character the user inputs. It is tricky to determine which character is left, right, up, or down . Most characters are only one byte long. On the other hand, the character that is output when an arrow is pressed,

is three bytes long. Here are the three byte sequences for left, right, up, and down arrows:

```
left  -> {27, 91, 68}
right -> {27, 91, 67}
up    -> {27, 91, 65}
down  -> {27, 91, 66}
```

Since we are not echoing characters, you will need to echo every character the user types. If the left or right key is pressed, nothing should happen. But, if the up or down arrow key is pressed, it should work as described in the History section below.

Delete / Backspace:

When the user is inputting commands into your shell, it should properly handle delete and backspace. When one of these characters is detected, you need to remove one character from the screen. For this, you can simply print a '\b' character to the screen. This will act as the backspace. Remember also to remove the character from the buffer you are using to store the input. Here are the byte values for the delete and backspace characters:

```
delete    -> 127
backspace -> 8
```

When the user presses delete or backspace, it should remove a character from the screen. But, it should not remove any of the prompts. For example, if the console looks like the following after the user typed the character 'a',

```
/home/projects/cs370/spring12/shell->a
```

then if the user presses the backspace twice, the console should still look like:

```
/home/projects/cs370/spring12/shell->
```

Notice that the prompt is still intact.

Break Keys

Like a normal read, your input should break on the new line character '\n' and also on the up and down arrow keys. You will process the input entered by the user when he/she enters the new line.

History

Your shell must keep a history of the last ten commands executed. When the user presses the up key, the current input should show the last command executed (as the user typed it). So, if the user executed the 'ls' command,

then pressed the up key, then pressed enter, the ls command should be executed. When the user presses the up key n times, you should replace the input with the n^{th} preceding command until you reach the end of the history list (maximum of 10). If the user presses the up key and reaches the end of the list, the next time up is pressed nothing should happen. Conversely, when the down arrow is pressed, you should cycle through the history in the other direction. So, if the user presses the down key without pressing up, nothing should happen because you will be at the bottom of the list. After a command is executed, it should be added to the list and the index of where the user is in the history should be reset. This functionality should be similar to how the Bash shell works.

Pipe

Your shell should accept and execute the pipe '|' operator. This will look like the following.

```
<command1> | <command2>
```

The functionality of this operator is to execute command1, send its output to the pipe, then execute command2, and use the pipe as input. You will need to use the close(), dup(), and pipe() functions discussed in class. Below is a sample of the execution.

```
/home/projects/cs370/spring12/shell>cat project2.c | grep include
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <string.h>
/home/projects/cs370/spring12/shell>
```

Overall Process

Below is a pseudo code for the overall process of a shell.

```
configure the input (termios)
while not exited
 print prompt
 read input (break on up, down and \n)
 if broken on up or down
  clear the current input
    execute appropriate command to generate corresponding message
  print message
 else
  parse input into a command
  if cd command
   change the current working directory
  else if exit command
   break
  else
   execute the command
restore the input configuration (termios)
```

Point Breakdown:

- Printing the prompt [10 pts]
- Executing commands from the user [40 pts]
- Implementing the exit command [15 pts]
- Implementing the cd command [15 pts]
- Properly ignoring the left and right keys [20 pts]
- Properly implemented the delete and backspace keys [30 pts]
- Implementing history [30 pts]
- Implementing the pipe operator [30 pts]
- Implementing custom command [20 pts]

Helpful Links:

Pointers
`http://home.netcom.com/~tjensen/ptr/pointers.htm`
Termios non canonical reads
`http://unixwiz.net/techtips/termios-vmin-vtime.html`