CSE 2431/5431
Instructor: Luan Duong, Ph.D.

LAB 2: UNIX Shell (Part 1)

Electronic Submission (Due): 11:59 pm, Wed Mar 05, 2025
Difficulty: ** (2 stars)
Point: 20 points (4% total grade)

1. Goal

1) This lab helps you learn the concept **of processes in real O.S. (linux)**, how to create, and how to terminate a process. In addition, you are expected to become familiar with system calls that are related to processes and file operations.
2) Main Goal: To **create a shell** using the coding template, **`shellA.c`**, which is provided on Carmen.

2. Introduction

This lab assignment asks you to build a simple shell interface using the C Programming Language that accepts user commands, creates a child process, and executes the user commands in the child process. The shell interface provides users a prompt after which the next command is entered. The example below illustrates the prompt sh% and the user's next command: `cat  prog.c`. This command displays the file `prog.c` content on the terminal using the UNIX cat command.

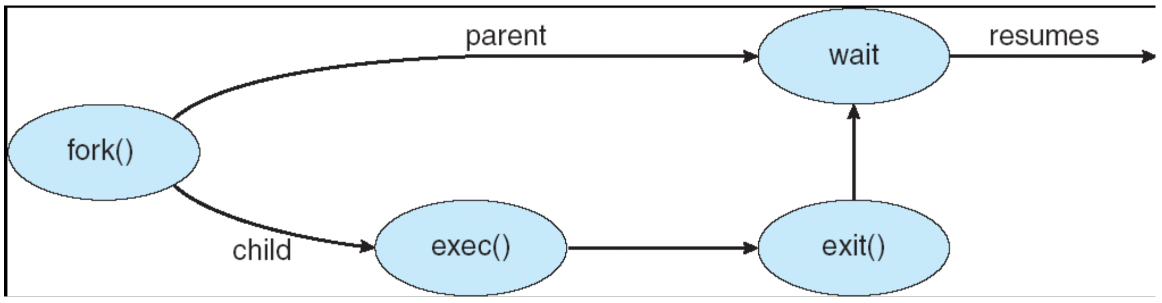
```
sh%  cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (i.e. 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 what is illustrated in Figure 1. However, UNIX shells typically also allow the child process to run in the background – or concurrently – as well by specifying the ampersand (&) at the end of the command. By rewriting the above command as

```
sh% cat prog.c &
```

the parent and child processes now run concurrently.

The separate child process is created using the ***`fork()`*** system call and the user's command is executed by using one of the system calls in the ***`exec()`*** family (for more details about the system call, you can use the man command for online documentation).



3. A Simple Shell

A C program that provides the basic operations of a command line shell is supplied in the file shell.c, which you can download from the instructor's web site. This program is composed of two functions: ***main()*** and ***setup()***. The ***setup()*** function reads in the user's next command (which can be up to 80 characters), and then parses it into separate tokens that are used to fill the argument vector for the command to be executed. (If the command is to be run in the background, it will end with '&', and *setup()* will update the parameter background so the *main()* function can act accordingly. This program is terminated when the user enters <Control><D> and *setup()* then invokes *exit()*.

The *main()* function presents the prompt Sys2Sh: and then invokes *setup()*, which waits for the user to enter a command. The contents of the command entered by the user are loaded into the *args* array. For example, if the user enters ls –l at the Sys2Sh: prompt, *args[0]* will be set to the string ls and *args[1]* will be set to the string –l. (By "string", we mean a null-terminated, C-style string variable.)

```
#include <stdio.h>
#include <unistd.h>

#define MAXLINE 80

/** The setup() routine reads in the next command line string storing it in input buffer.
The line is separated into distinct tokens using whitespace as delimiters. Setup also
modifies the args parameter so that it holds pointers to the null-terminated strings which
are the tokens in the most recent user command line as well as a NULL pointer, indicating
the end of the argument list, which comes after the string pointers that have been assigned
 to args. ***/

void setup(char iBuffer[], char *args[],int *bgrnd)
{
     /** full code available in the file shell.c */
}
int main(void)
{
char iBuffer[MAXLINE]; /* input buffer to hold command entered */
char *args[MAXLINE/2+1]; /* command line arguments */
int bgrnd; /* bgrnd is 1 if a command is followed by '&', else  0 */
   while (1){
      bgrnd = 0;
      printf("Sys2Sh: \n");
      setup(iBuffer,args,&bgrnd); /* get next command */
      /* Fill in the code for these steps:
      (1) Fork a child process using fork(),
      (2) The child process will invoke execvp(),
      (3) If bgrnd == 0, the parent will wait,
            else continue. */
   }`
}
```

This lab assignment asks you to create a child process and execute the command entered by a user. **To do this, you need to modify the *main()* function in shell.c so that upon returning from *setup()*, a child process is forked**. After that, the child process executes the command specified by a user. If an erroneous command is entered, an error statement should be printed then the user should be prompted for another command.

As noted above, the ***setup()*** function loads the contents of the ***args*** array with the command specified by the user. This ***args*** array will be passed to the ***execvp()*** function, which has the following interface:

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

where ***command*** represents the command to be performed and *params* stores the parameters to this command. You can find more information on ***execvp()*** by issuing the command “man execvp”. Note, you should check the value of ***bgrnd*** to determine if the parent process is to wait for the child to exit or not.

Hint:

1. Check the value of the background to determine whether the parent process should wait for the child to exit.
2. Use waitpid instead of wait
3. For all labs from now on, please check system calls’ return codes, which will tell you if errors occured. Recall that zero (0) return codes indicate that commands executed properly.
4. In general, you should log all information about command execution in your labs, including invalid commands, as this information is indispensable for debugging. (For example, suppose you type nonsense such as asdfg at the command line. How would your code tell you “this is nonsense,” and what would your program do in this situation?)

4. **Test**

A list of Unix commands is provided at the end of this assignment that you may use to test your shell.

Unix Commands
Here is a list of commands that you could use for testing. You may use other commands of your choosing as well.

```
pwd
ls
date
uptime
who
du
ps
ls –at
du –s
ps –A
cp file1 file2
mkdir dirname
rm filename
mv file1 file2
last -5
quota
w
more filename
cat filename
clear
df
hostname
man commandname
users
whoami
```

How to test background mode (with &)? Most of the previous commands return quickly, so you may not be able test background mode. To solve this problem, you can write another C program that performs sleep and printf in a while loop. Then you can run this program in background mode and see whether you can run other commands at the same time.

5. Submission Instructions:

a. Write your own Makefile (same as lab 1, but now pay attention to all the header files or c files applicable). The output file must be **lab2**
b. Again, we will run 2 commands as in your lab 01: >> `make` and >> `./lab2`
c. Pack all your file in a **zip file** and submit it on Carmen **on or before the due date.**
d. **Any program that does not compile will receive a zero.** Graders/I will not spend any time to fix your code due to simple errors you introduce at the last minute. It is your responsibility to leave yourself enough time to ensure that your code can be compiled, run, and tested well before the due date.