CSE 2431 **LAB 1** **AUTUMN 2015**

Electronic Submission Due: 11:59 pm Monday 10/5
Hardcopy Materials Due: Tues. 10/6 start of class

PART A:

1. Goal

Create a shell using the coding template, shellA.c, provided on Carmen.

2. Introduction

The first part of 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.

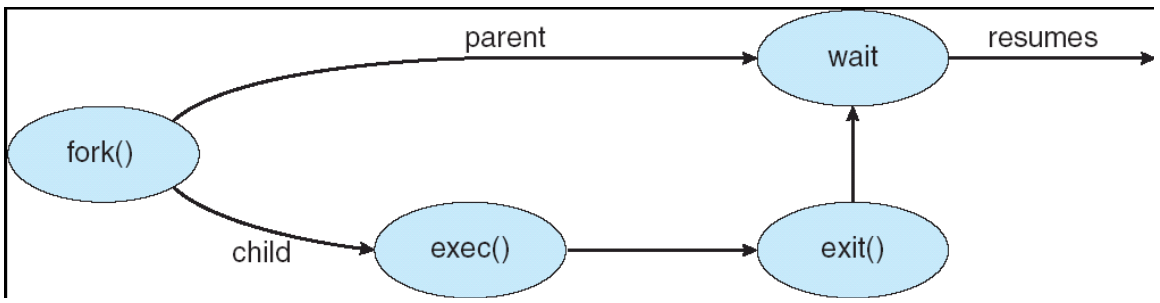
```
sh%  cat prog.c
```

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 return to the setup() function. */
  }`
}
```

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.

4. Testing

A list of Unix commands is provided at the end of this assignment that you may use to test your shell. You will be graded both on your solution and screenshots of testing on your shell which will be handed in as a hardcopy. For Part A, please submit screenshots showing the execution of 5 different commands including one that has arguments. Any added code should be commented in a professional manner.

PART B:

1. Goal

Enhance the shell written for Part A by adding a history feature. Call this version shellB.c

2. Introduction

The second part of this assignment is an extension of the UNIX Shell interface you built in Part A. You will add a history feature into your UNIX Shell which will allow users to access up to twelve most recently entered commands. These commands will be numbered starting at 1 and will continue to grow larger even past 12, e.g. if the user has entered 35 commands, the twelve most recent commands should be numbered 24-35.

3. Assignment

A user will be able to list the twelve most recently entered commands when s/he types the command **history** or its alias/shortcut **h**. With this list of previous commands, the user can execute any of those previous 12 commands by entering **r num** where '**num**' is the number of that command in the history list. Also, the user should be able to execute the most recent command again by entering **rr** for 'run most recent'. You can assume that one space will separate the **r** and the number and that the number will be followed by '\n'. Also, when **rr** is used to execute the most recent command you may assume that it is immediately followed by '\n'. Commands with arguments must execute correctly, for instance, 'ls –a'.

Any command that is executed in this fashion should be echoed on the user's screen and the command placed in the history buffer as the next command. (**r num** and **rr** do not go into the history list; the actual command that they specify, though, does.)

For a better idea of how this actually works in the Linux CSE Environment, execute a few commands in the terminal window then type **history.** To execute a command from the history list type **!num** (no space between the ! and the num). To execute the most recent command type **!!.** Depending on whether it is set up in your aliases for commands, **h** may work as well. Also note that when you type **!num** or **!!**, the full command from the history list is output then executed. If you type **history** again you can see the commands that were repeated with **!!** or **!num** are now in the updated history list. Our implementation of history does not use **!** and does not use the same syntax (no space).

4. Implementation Details

Include the following error checking and features in your shell.

Error Checking

In addition to the error checking from Part A your shell should now print an error message which includes a description of the problem for;

- **r** with no history number
- **r** with non-positive number
- **r** with number that is not in history buffer.

After the error message is printed the user is prompted for another command.

Features

In addition to the requirements from Part A such as being able to execute Linux commands with arguments and execute a command in the background, your shell should now do the following;

- Handle duplicate commands in history by deleting the older reference to a command and inserting the command as the most recent command in history. For instance if command number 5 in history was an **ls** and then for command number 10 **ls** is entered again, the history entry for 5 is deleted. The numbers for other commands do not need to be changed, that is, there will now be no command number 5 in history.
- In addition to executing commands in history using **rr** or **r num**, execute the *most recent* command in history that begins with the string **str** by entering **r str**. The command **r str** does not get inserted into history, the actual command it refers to does with the duplicate removed.
- Change the size of history using the command **sethistory num**, where **num** is the new size of history. If **num** is smaller than the current size of history save only the most recent **num** commands are now in history. If **sethistory num** is executed again with **num** being larger, history will now save more commands but the former commands are not restored.

5. Testing

Obviously, the enhanced shell should work for any of the commands given in Part A. The user should provide screenshots of test cases that illustrate error messages, history, and the history features.

General Instructions

I suggest that you complete your solution in our CSE Linux environment. Use the gcc compiler. The TA will compile and test your solution under this environment and using this compiler.

Any program that does not compile will receive a zero. The TA 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.

Submission

Code for Part A should be in a file called shellA.c, code for Part B in a file called shellB.c. Use the submit command to submit the files to the lab1 directory. At the beginning of each file you should tell the TA your full name, your email address, how to compile your file, and how to run your compiled program. Please make sure that your instructions are clear.

A late penalty of 10% **per hour** will be assessed, for instance, if a lab is submitted at 12:00 am or 12:59 am there is a 10% penalty.

Reminder: It is the student's responsibility to know if the lab was submitted correctly. ***Labs must be the student's own work.***

In addition, please turn in a printout of your completed code, and documentation(such as screenshots) of runs on test cases, at the start of the next class. Your name and email address should be on the printout. Missing or late printouts will also result in a point deduction.

Piazza

Piazza has been set up to serve as a student discussion platform. Suggested topics regarding labs include; syntax and/or execution errors, logic errors, questions regarding system calls, etc. Oftentimes a question that one student has is actually a question that several students have so the post helps everyone. Please feel free to answer each other's questions and to discuss answers, etc. The idea is for students to have a place outside of the classroom where they can discuss topics relevant to the class. Piazza posts and questions/answers can also serve as a good reference for students that may have the similar questions later in the course. Your posts are anonymous to other students unless you choose to give identifying information. ***Please do not post code on Piazza or any online repository that makes files public.***

Plagiarism

Students are required to use the code posted on Carmen to begin their assignment. Submitting other code, for instance, from the Internet or other sources, or code from a previous course (even if it's your code), is considered plagiarism and will be handled accordingly.

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