**CSE 2431**
**SP 23**

# Lab1

**Due: Thursday, March 9, 2023 at 11:30 p.m. on Carmen**

**NOTE: A one-half hour "grace period" is given for submission, so as long as you submit before midnight, the lab will be counted as being on time; but if you submit at midnight or after, no matter what the reason, the lab will be counted as one day late (However, if you are sick or have some other situation beyond your control, please contact the instructor AS EARLY AS POSSIBLE about the possibility of an extension)).**

## 1. Goal

This lab helps you understand the concept of processes, how to create, and how to terminate a process. In addition, you are expected to get familiar with system calls related to processes.

## 2. Introduction

This lab assignment asks you to build a simple SHELL interface (using the C Programming Language; this is required) that accepts user commands, creates (forks) a child process, and executes the user commands in the child process. The SHELL interface provides a user a prompt after which the next command is entered. The example below illustrates the prompt CSE2431Sh$ and the user's next command: cat prog.c. This command displays the file prog.c content on the terminal (assuming the user has a file prog.c in their current working directory) using the UNIX/LINUX cat command.

```
CSE2431Sh$ 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 (that is, in this case, the parent process does not wait) by adding the ampersand (&) at the end of the command. By rewriting the above command as

```
CSE2431Sh$ cat prog.c &
```

the parent and child processes now run concurrently; that is, the parent continues execution after forking the child, and does not wait for the child to exit.

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 this lab, the child will call/invoke execvp(); for more details about this system call, you can use the man command for online documentation. You are expected to see the command documentation using man to determine how to pass parameters to execvp()).

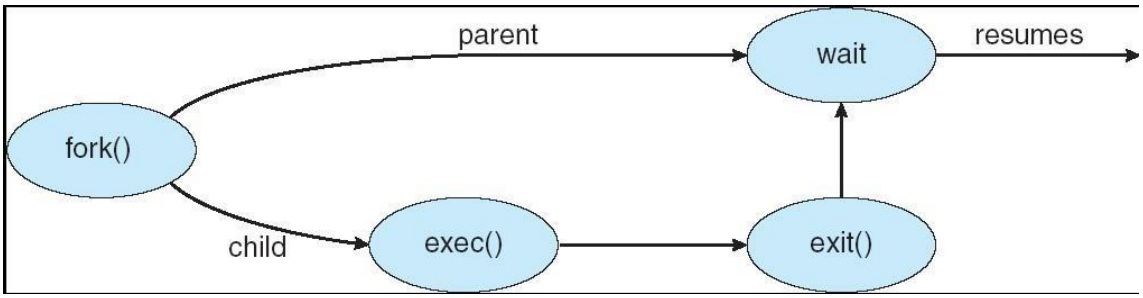


Figure 1. *Process Creation & Termination*

## 3. **A Simple Shell**

First, to do this lab, you should make a cse2431 directory, and then a lab1 directory inside the cse2431 directory for the assignment on stdlinux. A C program that provides the basic code for a command line shell is supplied in the file shellA.c, which you can get on Carmen in the Labs folder. Get the file shellA.c for the assignment with a skeleton of the code on Carmen. To get the file from Carmen, log in to stdlinux, and then start a firefox web browser as follows at your shell prompt:

```
$ firefox https://carmen.osu.edu/#
```

When the browser starts, you can log in to Carmen, and then go to the Lab folder for the course, and then the Lab1 folder, and then download the shellA.c file to your linux ~/Downloads directory (it will be downloaded to this directory in stdlinux by default). You can copy the shellA.c file from ~/Downloads to your cse2431/lab1 directory using the cp command after cd to cse2431 and then cd to lab1):

```
$ cp ~/Downloads/shellA.c ./
```

The shellA.c program is composed of two functions: *main()* and *setup()*. The *setup()* function reads in the user's next command (which can be up to 40 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 shellA.c program is terminated when the user enters <Control><D> at the command line prompt and *setup()* then invokes *exit()* (a system call that will terminate the process running the shell).

The *main()* function presents the prompt CSE2431Sh$ 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 CSE2431Sh$ prompt, *args[0]* will point to the string *ls* and *args[1]* will point to the string *-l*. (By "string", we mean a null-terminated, C-style string variable.)

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

#define MAX_LINE 40

/** setup() reads in the next command line string stored in inputBuffer, separating it into distinct tokens
using whitespace as delimiters. setup() modifies the args parameter so that it holds pointers to the null-
terminated strings that 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 inputBuffer[], char *args[],int *background)
{
        /** full code available in the file shellA.c */
}
int main(void)
{
char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
int background; /* equals 1 if a command is followed by '&' */
char *args[MAX_LINE/2+1]; /* command line arguments */
pid_t ret_val; /* to hold value returned by fork */
while (1){
        background = 0;
        printf("CSE2431Sh$ ");
        setup(inputBuffer,args,&background); /* get next command */
        /* the steps are:
        (1)     fork a child process using fork()
        (2)     the child process will invoke execvp() with appropriate parameters
        (3)     if background == 0, the parent will wait, otherwise return to top of while loop to call 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 shellA.c so that upon returning from *setup()*, a child process is forked. After that, the child process executes the command specified by the user of your shell.

As noted above, the *setup()* function loads the contents of the *args* array with pointers of type char * to the strings in the command entered by the user (that is, setup separates the command into a sequence of strings separated by white space characters, and places a char * in the args array which points to each of the white-space separated strings in the command). ***You do not need to write any code to parse the user command, and you should not attempt to do so!*** setup() does this for you; DO NOT MODIFY THE CODE IN THE *setup()* FUNCTION! Arguments from the *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 (the command is actually (the name of) a PROGRAM in a file on disk which can be run as a process to execute the command which the user enters at the command line prompt) and *params* stores the parameters to this command which were entered by the user on the command line. You can find more information on *execvp()* by issuing the command "man execvp". Note that you should check the value of *background* in the code you write in main() to determine if the parent process is to wait for the child to exit or is not to wait for the child. To have the parent wait, you can use the wait() system call, or waitpid() system call; you can decide which you want to use, as long as your code works correctly; people usually find it is easier to use waitpid(), which takes the pid of the child the parent is waiting on as a parameter, but you can use either one.

## 4. **Instructions**

You are required to complete your solution in our CSE Linux environment. Use the gcc compiler (run "man gcc" to learn about the compiler). The grader will compile and test your solution under this environment and using this compiler with the following compilation command:

```
$ gcc shellA.c -o shellA
```

**Any program that does not compile will receive a zero, and there are no exceptions to this rule; be sure you verify that your code compiles and performs correctly before you submit it**. The TA will not spend *any time* to fix your code (even the most minor errors such as typos), due to simple errors you introduce at the last minute (or even not 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, and to submit by 11:30 pm on the due date (as noted above, as long as you submit by 11:59:59, your lab will not be considered late, but do not wait till the last minute; start working, and complete the lab in advance, at least a day, to avoid problems).** Do yourself a favor and do not leave yourself too little time to complete the assignment enough in advance that you are not making last minute changes which expose you to the risk of having no time to compile and test before submitting; if you do this, any consequences are your responsibility, unfortunately (this is the way things work in a professional environment, so we want to encourage everyone to recognize this and work the way professionals are expected to here as well).

**SYNCHRONIZATION OF PARENT AND CHILD:** In standard Unix/Linux shells, if the child runs concurrently, the parent will return to the top of the while loop and often print the prompt before the child writes output (or in the middle of the child's output; for example, you may see the shell prompt in the middle of the child's output if the parent is not waiting for the child). This is normal behavior in Unix/Linux shells, and you should not be concerned if your shell code behaves this way (there are some ways to try to address this, but we will not worry about it here).

## 5. **Test Cases and Warnings**

Make sure that your shell performs correctly on the following test cases (run these commands in the Linux shell from the command line to see what they do):

```
CSE2431Sh$ mkdir new

CSE2431Sh$ ls

CSE2431Sh$ ls -al
```

CSE2431Sh$ date

CSE2431Sh$ cat shellA.c

CSE2431Sh$ ls &

CSE2431Sh$ ls -al &

CSE2431Sh$ date &

**WARNING: Be sure that you do not attempt to execute commands which change directories** (such as cd, cd .. , etc.)! Also, be use that you do not attempt to test your shell program using input redirection (We will not test it this way)! That is, *you must type the test commands shown above to test your shell*. You cannot put them in an input file and use redirection of the input.

You will be graded both on your solution and on the performance of your code on the provided tests. Any code you write (but not the code you are provided) should also be commented in a professional manner.

6. **Submission**

You should submit your shellA.c file to Carmen Canvas (**WARNING: BE SURE YOU DO NOT SUBMIT AN EXECUTABLE -- SUBMIT ONLY A C SOURCE FILE - YOU WILL GET NO CREDIT IF YOU SUBMIT AN EXECUTABLE!).** PLEASE **DO NOT zip** your file before submitting. You can submit by starting a firefox web browser in stdlinux at the stdlinux command line prompt (if your prompt is different from $, this will not matter):

$ firefox https://carmen.osu.edu/#

After Carmen opens, you can log in and go to the Lab 1 assignment for our class, and submit your shellA.c source file. Submit ONLY your shellA.c source file.

**DO NOT SUBMIT A FOLDER!** SUBMIT *ONLY THE shellA.c SOURCE FILE*!!!!!!

**NOTE:** At the beginning of the file "shellA.c" you need add a comment which tells the grader your full name. **Include other comments which document any code which you add to the shellA.c source code file, but you do not need to add comments for code which you did not write or modify.**

A late penalty of ***25%*** will be assessed for any lab submitted after the deadline given at the top of this document, but within 24 hours after the due date and time stated above; any lab submitted more than 24 hours late will receive no credit. Labs must be the student's own work. **NOTE:** Avoid making multiple submissions (you are limited to 4 submissions maximum). ***Test and check your code thoroughly before submitting*, and resubmission should not be necessary.**

**8. 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 instructor will also monitor answers posted

on Piazza to make sure that they are correct, but 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. ***DO NOT POST CODE or an idea about how to implement a solution in a question unless you leave it private (it will be by default). If you are NOT posting code or ideas about a solution, however, please change your post so that it is not private, so other students will be able to see the posted question and answer. If you do not do this, the instructor will change your post to make it visible to others before an answer is posted.***