**CS 346 – Lab #3 – Creating processes in C**

The goal of today's lab is to gain a keener understanding of the C functions we use when creating processes. The most important functions for this purpose are fork, exec, and wait. From our class discussion, you should remember the purpose of each function. In particular, the difference between fork and exec.

This lab paper consists of several practical questions that you need to answer. Along the way, you will be experimenting with several example C programs. At the end of the lab, I'll ask you to write your own C program, which you will also need to turn in.

Log into the CS Linux server, and create a new directory in your account called lab03. Do all your work in this directory. Copy all the files from my folder ~chealy/www/cs346/lab03. During today's lab you are going to be compiling and running several C programs. As a reminder, here is how you do it. Suppose you have a C program called carrot.c. The compile command is

```
gcc carrot.c –o carrot
```

And the command to run is

```
./carrot
```

Have fun!

**Fork and Exec**

Open the file fork1.c to examine its code. Compile and run the program. In your own words, what does this program do? How many total processes do you see? Describe the two parameters we pass to execv().

(1) _____

_____

_____

**Multiple child processes**

Next, let's turn our attention to fork2.c. Compile and run this program. Run the program several times. Does anything change in the output when you run it again? Looking at the code, how can you tell that the cat process will always finish before the factor process?

(2) _____

_____

_____

**The wait function**

When we create processes, fork() and exec() get a lot of our attention. But the wait() function also plays a role. Let's do a little experiment. Create a duplicate copy of fork2.c and call it fork3.c. Edit the file fork3.c: In the first else clause, comment out the line of code that says "wait(&i);". Now, compile and run the program. Run the program several times to see if the behavior varies.

Sometimes, it appears that one of the child processes completes after the shell prompt appears. Which child process is doing this? Why is this happening?

(3) _____

_____

_____

Let's practice some more with the wait() function. Create a new duplicate copy of fork2.c, and call this one fork4.c. Make the following modifications to fork4.c:

1. Comment out the two calls to `wait(&i);` that appear in the two else clauses.

2. At the end of the program, type these two lines of code:

```
return_value = wait(&i);
printf("wait returned %d\n", return_value);
```

3. In the first if clause, insert the following statement between the printf statement and the execv statement:

```
sleep(1);
```

Compile fork4.c. Run the program a few times. Describe the behavior of the program. The return value printed at the end of the program matches which child process? Why?

(4) _____

_____

_____

Next, we'll make a small change to fork4.c. Make these modifications:

4. Comment out the call to `sleep(1);` to you placed in the first if clause.

5. Type a similar `sleep(1);` statement in the second if clause.

Compile and run fork4.c. Run the program a few times. Describe how the output is different this time. Why is this happening?

(5) _____

_____

_____

**The process table**

One Linux command that is especially useful is the ps command. This command asks Linux to list the processes. The command takes various options. Try the command: `ps –ef`. This is asking for a complete listing of all living processes. In this list of processes, there are certain columns that will interest us:

- UID column: this is the user ID of the owner of the process (which could be you)
- PID column: the process ID
- PPID column: the parent's process ID
- CMD column: the name of the program running in this process

But listing all the processes may be too many for us. Since many people may be logged into the computer at once, we are really only interested in the processes that belong to ourselves. So, you can filter the results using the `grep` command. Try this command:

```
ps –ef | grep <username>
```

where <username> is replaced with your Linux username. You should see a complete listing of only your processes. If you are not currently running any programs yourself, then most of the processes you will see pertain to your login process. The only drawback to filtering like this is that you also lose the column headings, but you should be able to recognize the information that you need.

Here's even better news: We can get the process table to print inside our C program. Because C and Linux are closely related, this is simple to do.

Create a duplicate copy of fork2.c and call it fork5.c. Let's make some small edits to fork5.c:

1. At the top of the program under the two #include directives and before the start of the `main` function, enter this new line of code:

   ```
   #include <stdlib.h>
   ```

2. In each of the two if clauses, immediately before the call to execv(), insert this statement:

   ```
   system("ps –ef | grep <username>");
   ```

   Don't forget to replace username with your actual user name.

Compile fork5.c. Let's run this program and redirect its output to a text file. Enter this command at the shell:

./fork5 > fork5.out

Now, your output is permanently kept in a file called fork5.out. Examining this output text file, what are the parent and child process numbers? (6) _____

Here is another question: We asked our program to print the process table twice: once by our cat process, and the other by our factor process. Why do the cat and factor processes not appear in the process table?

(7) _____

_____

_____

The purpose of printing the process table within our C program is to prove the "ancestry" of our processes: to demonstrate the parent-child relationships. In fact, we can even trace back a processes ancestry even further if we remove the grep filter. So, please make one more change to fork5.c:

3. In the second if clause, the one pertaining to the factor process, let's remove the grep filter in the line of code containing our system() call between the printf statement and the call to execv. In other words, change the function call to say:

```
system("ps -ef");
```

Compile and run the program one more time, and redirect the output to a file fork5.out as before. Examine this file, and find where the child process is listed. Let's discover the ancestry of this factor process. In other words, starting with the factor process, identify its parent process ID, and then its parent, grandparent, etc., until you reach the "root" of the process tree. For each process that you encounter along the way, also take note of the owner (user) of the process, and the name of the command associated with that process.

(8) _____

_____

_____

_____

**An Exercise**

Write a C program, fork6.c., that is based on what you learned in fork5.c. Your new program should do the following. The main program, i.e. original process, will fork two children. Each child process will fork one child. The two grandchildren processes will not fork. Run your program. As it runs, print the process table, so we can verify the tree of processes (children and grandchildren of the original process).

Hint: You will not need to use exec() or wait() in this program. Also, you do not need to print the process table.

**What to turn in:**

Usually I ask you to turn in all the files you changed or created, but for this lab, you only need to turn in fork6.c and your answers to the written questions in this handout.