Lab7. Job and Process Management

Objectives

After completing this lab, the student should be able to:

- Manage several jobs running in the background.
- Understand how processes are created using the `fork` and `exec` calls.
- Control the priority of newly created processes using the `nice` command.
- Identify and use signals for manipulating processes.

1. Job Control

Sometimes we need to execute more than one job on the same terminal, but we are forced to wait until one command is done executing and getting the shell prompt back before we can execute the next command. This is especially a problem if the one of the jobs we are executing takes a long time such as a *backup job.* To get around this, Linux allows us to run several jobs at the same time in the *background*. This is called **job control**.

To be able to understand job control, we need to use a program that will be running for long time or wait indefinitely for user input. For this, we will use the **`xlogo`** command. The **`xlogo`** program is a sample program supplied with the X Window System that makes the graphics on our display. The program simply displays a resizable window containing the **X** logo.

1- Type the **xlogo** command in the terminal to launch the **`xlogo`** program as follows:

```
[me@linux ~]$ xlogo
```

After entering the command, a small window containing the logo should appear somewhere on the screen. On some systems, `xlogo` might print a warning message, but it can be safely ignored. We can verify that `xlogo` is running by resizing its window. If the **X** logo is redrawn in the new size, the program is running.

Notice how our shell prompt in the terminal **has not returned**? *This is because the shell is waiting for the program to finish, just like all the other programs we have used so far*.

Note: *If your system does not include the* `xlogo`*, try using* `gedit` *or* `kwrite` *instead.*

2- Close the **`xlogo`** window and notice that the shell prompt has returned. Now we can use **`xlogo`** command to understand job control.

3- To run a job in the **background**, we follow **the command with an ampersand** (&). In our case we are going to run three **`xlogo`** jobs in the background as follows:

```
[me@linux ~]$ xlogo &
[1][2000]
[me@linux ~]$ xlogo &
[2][2500]
[me@linux ~]$ xlogo &
[3][2503]
```

Note that each time we run a job in the **background** the system displays **two numbers**:

1. The first is the **job id number** or **jobspec** number.
2. The second is the **job process id (PID)**.

These numbers are important to be able to *reference the job later on for manipulation*.

4- Now, we can display **our background jobs** by using the command:

```
[me@linux ~]$ jobs
[1]   Running xlogo &
[2] - Running xlogo &
[3] + Running xlogo &
```

The **first number** is the job id (or *jobspec*) number. The **plus and minus** signs reference the last scheduled job and the one before it, respectively. The **third column** displays the current status of the job. Currently all jobs are in running status. The **last column** corresponds to the command used to create the respective job.

We can manipulate the jobs in several ways, as follows:

5- To get a job back to *foreground* we use the **`fg`** (foreground) command followed by the job id or jobspec number. For example, to get **job 2 to the foreground**, we run the command:

```
[me@linux ~]$ fg %2
```

This brings the job to the **foreground**.

6- To send the job to *background*, we press **CTRL+Z** and the job will be moved back to *background*.

7- Now, run the following command:

```
[me@linux ~]$ jobs
```

*What do you notice different about job #2?*____

8- To resume a **stopped or suspended job**, we use the **bg** (background) command followed by the job id or jobspec number. To resume job 2 (i.e., change its status to **running**) we use the command:

```
[me@linux ~]$ bg %2
```

9- Now, run the command:

```
[me@linux ~]$ jobs
```

What is the status of job #2 now? ____

10- To terminate a job, we use the **`kill`** command followed by the job id or jopspec number. For example, to kill job 3, we run the following command:

```
[me@linux ~]$ kill %3
```

11- If we type the command: **`jobs`** enough, we will see the status of job 3 has changed to **Terminated** or it had been disappeared from the list.

12- Let us now perform the following experiment:

a. Kill all remaining jobs such that none are in the background.

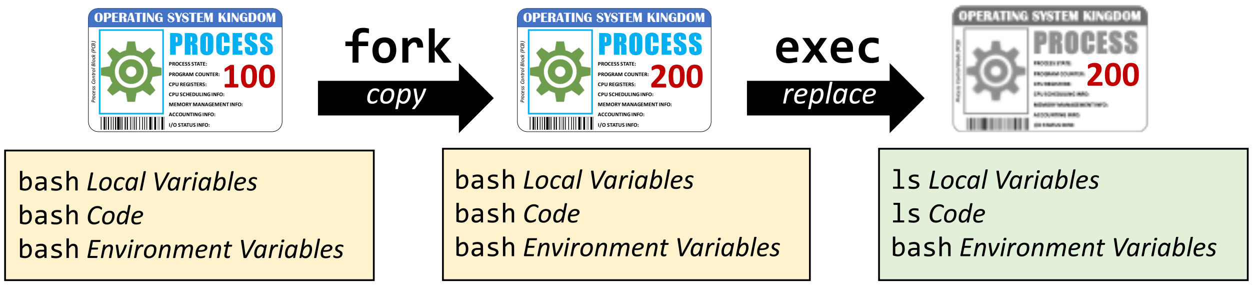
b. Write the sequence of commands needed to have the following output displayed when the command "`jobs`" is executed:

```
[me@linux ~]$ jobs
[1]   Stopped     xlogo
[2] - Terminated  xlogo
[3] + Running     xlogo
```

2. Process Control

A process is simply a *program in execution*. Each command we run results in one or more processes. There are several processes running in the *background* that allow us to use the system and provide us with different services. Interacting and manipulating processes is called **process control**.

When a process is first created, a *duplicate copy of the parent process (aka address space) is created* using the **fork** function. This copy is similar to the original except for its process id (PID). When the system executes the **exec** command, the system basically replaces the duplicated copy of the parent process (aka address space) with the new address space for the newly run command. For example, when we run the command **ls** under the **bash** shell, a copy of **bash** is created first, then it is replaced by **ls**. Notice that the **environment variables** are passed from the parent process (**bash**) to the child process (**ls**).



Let us now run few experiments to understand what is going on when spawning child processes.

1- To view *process information*, we can use the **ps** (process status) command. To see our running processes, we use **ps** with the **`-f`** option as follows:

```
[me@linux ~]$ ps -f
```

*Describe the output?*____

2- Let us create two variables called **var1** and **var2** respectively.

```
[me@linux ~]$ var1=first
[me@linux ~]$ var2=second
```

When new variables are created, they are defined as **local variables**. To change a variable from **local** to **environment**, we **export it** using the **export** command.

3- Let us make **var2** an environment variable as follows:

```
[me@linux ~]$ export var2
```

The **set** command is used to display both *local and environment variables*. The command **env** is used to display the *environment variables only*. Let us check for **var1** and **var2** in our main process (**bash** shell).

4- Run the command:

```
[me@linux ~]$ set | grep “var1\|var2”
```

*Which of the two variables (***var1** *and* **var2***) do you see in the output? Why?* _________

5- Now run the command:

```
[me@linux ~]$ env | grep “var1\|var2”
```

Which do you see now? Why? ____

6- Now, let us create a new child process from the current ***bash*** process. As a child process, we will create a process to execute the **Korn Shell** via the command **ksh**:

```
[me@linux ~]$ ksh
```

Note that your Ubuntu virtual machine may not come equipped with Korn Shell, therefore, you may need to install it via the command: `sudo apt-get install ksh`

7- Now, run the command:

```
[me@linux ~]$ ps -f
```

*What is the output now?*____

Note down the numbers for process id (PID) and parent process id (PPID). Those should tell you that **bash** process is the parent process of the child process **ksh** and you are currently within the child process. Let us check for the variables **var1** and **var2** in the child process (**ksh**).

8- Run the command:

```
[me@linux ~]$ set | grep “var1\|var2”
```

*Which of the two variables (***var1** *and* **var2***) do you see in the output? Why?*___________

9- Now run the command:

```
[me@linux ~]$ env | grep “var1\|var2”
```

Which do you see now? Why? ____

This shows that **only environment variables are passed from parent processes to child processes.**

As shown above any created process goes through the **fork** and **exec** steps explained above. We can use the ***exec*** command to skip the **fork** step and just do the **exec** step and see what happens, as follows:

10- Now, run the command:

```
[me@linux ~]$ ps -f
```

You should have three processes (**bash**, **ksh**, and **ps –f**). When running this command, the process “**ps –f** “ will not exist anymore as it has already run and terminated.

11- Now, note down the PID for the **ksh** process and instead of running the “**ps –f**” command as before, run it as follows:

```
[me@linux ~]$ exec ps -f
```

What processes do you see now? What happened to **ksh** *(write down the PID for* **ps –f** *process)*____

*What would you expect to happen if you run the command “***exec ps –f***” again? Try it. What happened?* ____

This shows that processes do go through both the **fork** and the **exec** steps, otherwise a *new child process will take over its parent process and destroy it.*

3. `nice` Command

Users may decrease the priority of their processes (especially those that take a long time and are not of high priority such as backups) to allow other users to run their processes at a higher priority. When they do that, they are nice and to do that they use the **`nice`** command. The only user that can both decrease and increase the priority of his/her processes is the root (system administrator). Let us see how the **`nice`** command is used.

1- Run the command:

```
[me@linux ~]$ ps -l
```

Note the two new columns displayed namely:

- **PRI** column refers to the priority of the process.
- **NI** column refers to the nice value of the process.

2- Now run the following command:

```
[me@linux ~]$ nice -6 ps -l
```

Notice what happened to the **PRI** and **NI** values for process "**ps -l**". They increased. Increasing the priority number actually makes the priority for that process less.

3- Now try to run the command:

```
[me@linux ~]$ nice --8 ps -l
```

What happened? Why? ____

4. Signals

Users can control their processes through sending signals using the **`kill`** command. There are many signals that may be sent to a process. To get a list you may use the following command:

```
[me@linux ~]$ man 7 signal
```

There are three interesting signals that stand out. Those are namely **TERM** (also called SIGTERM) which has the number 15, **HUP** (also called SIGHUP) which has the number 1, and **KILL** (also called SIGKILL) which has the number 9. The default signal is the TERM signal.

The TERM signal tries to terminate signals cleanly and may be blocked by processes such as shells. The HUP signal is used to restart a process to have it upload any changes in its configuration files. The KILL signal is used to kill a process uncleanly and cannot be blocked. Let us try the TERM and KILL signals:

1- Run the **xlogo** command in the **background** and **write down its PID** (let us assume the process is assigned a PID of **1234**). Check to see that the process is **running** in the background (use the **jobs** command).

2- Try the following command - *use the PID of your process* instead of 1234:

```
[me@linux ~]$ kill 1234
```

Now recheck if the process is running with the **jobs** *command. What did you find?*_____

3- Now repeat the same steps, create **xlogo** job in the background and check its PID (we are assuming its 1234, but it could be anything). For ***each time*** you create the **xlogo** job, try killing it with one of the following commands (*specify the correct PID for your* **xlogo** *process, we are assuming its 1234*):

```
[me@linux ~]$ kill -15 1234
```

```
[me@linux ~]$ kill -TERM 1234
```

```
[me@linux ~]$ kill -SIGTERM 1234
```

What did you notice about each of the three commands above? ____________________

4- Open two terminals and use the **ps** command to determine the PID of the terminal you are not using, as follows:

```
[me@linux ~]$ ps -f
```

What is the PID for the bash process running on the **pts** number different from the **pts** number that your **ps –f** process is running. That is the bashPID you need.

5- Now try running the following command:

```
[me@linux ~]$ kill -15 bashPID
```

What happened? Why? ____

6- Now try the following kill command:

```
[me@linux ~]$ kill -9 bashPID
```

Now what happened? ____