COMP311
Linux OS Laboratory
Lab7: Job and Process Management

By

Alaa’ Omar

Objectives

1. Manage several jobs running in the background.
2. Understand how processes are created using the fork and exec steps.
3. Control the priority of newly created processes using the nice command.
4. Identify and use signals for manipulating processes.

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 create and use a command that will take a long time. To do this, we do the following steps:

Job Control

1. Create a new file called forever using vi as follows:

```
$vi forever
    while true
    do
    echo running > myfile
    done
    :wq
```

This is basically a script file with an infinite loop.

2- Now we have to make sure that our PATH variable includes the current directory (.). This step is important for the shell to locate our newly created command forever. This is done as follows:

PATH=$PATH:. (.bash_profile)

3- The third step is adding the execute (x) permission to the command to make it executable. This is done by adding x to all parts of the mode as follows:

$chmod +x forever

Job Control

Now we have a command called ***forever*** that runs for a long time and that can be used to understand job control.

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

forever&

[1][2000]

forever&

[2][2500]

forever&

[3][2503]

Each time we run a job in the background the system displays **two numbers ([1][2000]**).

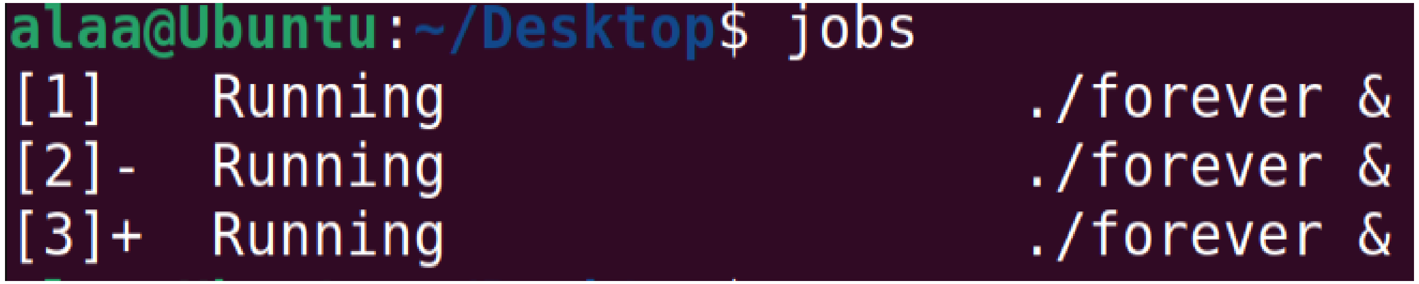
1. The **first ([1])** is **the job id number .**
2. The **second ([2000])** is the **job process number.**

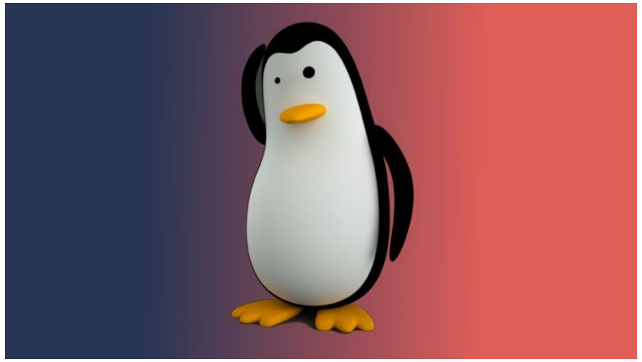
These numbers are important to be able to reference the job later on for manipulation.

Job Control

We can display our background job by using the command:
$jobs
This will display an output similar to the following:
[1]   Running forever
[2] - Running forever   (before last)
[3] + Running forever (last job)

The number is the job id number. The **plus** and **minus** signs reference the **last** and the **one before last jobs**. The status of all jobs is running. The last column is the name of the command used to create the job.





Job Control

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

- To get a job back to the foreground we use the **fg (foreground)** command followed by the job id number. E.g. to get job 2 to the foreground, we run the command:

  $fg %2

  This brings the job to the foreground.
- To send the job to the background, we press **ctrl-z**. The job is moved back to the background.Run the following command:

  $jobs

What do you notice different about job # 2?

Job Control

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

$bg %2

Run the command:

$jobs

What is the status of job # 2 now? ______________________________.

- To terminate a job, we use the kill command followed by the job id number. e.g., to kill job 3, we issue the following command:

$kill %3

If we type the command: jobs quickly enough, we will see the status of job 3 changing to Terminated and if we check again it will disappear.

Job Control

Do the following:
kill all remaining jobs such that none are in the background.
Write the sequence of commands needed to have the following output displayed when the command “jobs” is issued:
[1] Stopped forever
[2] - Terminated forever
[3] + Running forever
Commands:

$fg %1
$Ctrl+z
$kill %2; bg %3

Processes and Jobs control

- A PROCESS is a program that is loaded into memory and ready to run, along with the program's data and the information needed to keep track of the program.

- **All processes are managed by the kernel, the central part of the operating system.**
- **When a process is created, the kernel assigns it a unique identification number called a Process ID or PID.**
- **To keep track of all the processes in the system, the kernel maintains a PROCESS table, indexed by PID, contain one entry for each process. Along with the PID, each entry in the table contains the information necessary to describe and manage the process.**

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 command is run, a duplicate copy of the parent process is created using the fork function. This copy is similar to the original except for its process id number (pid). After that the system executes the command using the exec step which basically loads the new command on top of the copy created as follows:

When we run the command ls under the bash shell, a copy of bash is created and which is replaced by ls.

Process Control

pid (100)	***Fork →***	***pid (200)***	***Exec →***	***pid (200)***
bash (local variable)		bash (local variable)		ls (local variable)
bash code		bash code		ls code
bash (env. variable)		bash (env. variable)		bash (env. variable)

Process Control

Notice that the environment variables are passed from the parent process (bash) to the child process (ls).

Let us now run through to see some details on what happens above.

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:

$ps -f

Describe the output?

Process Control

Let us create two variables called var1 and var2 respectively.

$var1=first

$var2=second

When new variables are created, they are defined as local variables. To change a variable from local to environment, we export it (use the export command). Let us make var2 an environment variable as follows:

$export var2

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

Run the command:

$set | grep var

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

Process Control

Now run the command:
env | grep var
Which do you see now? Why?

Now run a child processes (ksh) as follows:
ksh
Run the command:
ps -f
What is the output now?

Process Control

Notice the numbers pid (process id) and ppid (parent process id). Those should tell you that bash is the parent process and ksh is the child process.

You are now in the child process. Let us check for the variables **var1** and **var2** in the child process (ksh).

Run the command:

set | grep var

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

Now run the command:

env | grep var

Which do you see now? Why?

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

Process Control

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:

Run the command:

ps –f

You should have three processes (bash, ksh, and ps –f). ps –f does not exist anymore. Now register the pid number for the ksh process. Now instead of running the “ps –f” command as before, run is as follows:

exec ps –f

What processes do you see now? What happened to ksh (hint: note the pid number for the 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.

Process Control

$exec ps –f output

```
alaa@Ubuntu:~/Desktop$ exec ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
alaa        2997    2971  0 20:51 pts/0    00:00:00 bash
alaa        3487    2997  0 22:08 pts/0    00:00:00 sh
alaa        3488    3487  0 22:08 pts/0    00:00:00 rbash
alaa        3496    3488  0 22:08 pts/0    00:00:00 dash
alaa        3515    3496  1 22:21 pts/0    00:00:00 ps -f
$ exec ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
alaa        2997    2971  0 20:51 pts/0    00:00:00 bash
alaa        3487    2997  0 22:08 pts/0    00:00:00 sh
alaa        3488    3487  0 22:08 pts/0    00:00:00 rbash
alaa        3496    3488  0 22:08 pts/0    00:00:00 ps -f
```

Process Control

$ps –f command output

```
alaa@Ubuntu:~/Desktop$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
alaa        2997    2971  0 20:51 pts/0    00:00:00 bash
alaa        3487    2997  0 22:08 pts/0    00:00:00 sh
alaa        3488    3487  0 22:08 pts/0    00:00:00 rbash
alaa        3496    3488  0 22:08 pts/0    00:00:00 dash
alaa        3506    3496  0 22:17 pts/0    00:00:00 bash
alaa        3512    3506  0 22:17 pts/0    00:00:00 ps -f
```

Process Control

- The **ps -f** command in Linux is used to display detailed information about processes in a full format. The output of the **ps -f** command consists of multiple columns that provide information about each process. Here is a description of the columns commonly seen in the output of **ps -f**:

  - **UID**: The user ID of the process owner.
  - **PID**: The process ID, a unique identifier assigned to each running process.
  - **PPID**: The parent process ID, indicating the process that spawned the current process.
  - **C**: The processor utilization of the process, measured in CPU time.
  - **STIME**: The start time of the process.
  - **TTY**: The terminal associated with the process.
  - **TIME**: The accumulated CPU time used by the process.
  - **CMD**: The command or program name associated with the process.

  The output of **ps -f** provides a detailed overview of the processes running on the system, including information about the user, process ID, parent process ID, CPU utilization, start time, terminal, CPU time, and command name.

Modify process execution priorities

```
$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    2997    2971  0  80   0 -  5159 do_wai pts/0    00:00:00 bash
0 S  1000    3487    2997  0  80   0 -   722 do_wai pts/0    00:00:00 sh
0 S  1000    3488    3487  0  80   0 -  5030 do_wai pts/0    00:00:00 rbash
0 S  1000    3496    3488  0  80   0 -   722 do_wai pts/0    00:00:00 dash
0 R  1000    3498    3496  0  80   0 -  5411 -      pts/0    00:00:00 ps
```

The **NI** column shows how nice this process is. The nicer the process, the less CPU it asks. The nice values can be from **-20 to 19**. To interpret this value, look at it like this: a process with nice = -20 is gets more priority for CPU and RAM while a process with nice = 19 is SUPER NICE and lets other processes use the resources before her).

Only the root can issue high-priority niceness (below 0)

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.

Run the command:

$ps -l

Note the two new columns displayed namely:

PRI (which refers to the priority of the process)

NI (which refers to the nice value of the process)

Now run the above command as follows:

$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.

Now try to run the command:

$nice --8 ps –l (--8 = two dashes then 8)

What happened? Why?

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:

$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:

- Run the command we created in the beginning of this lab (forever) in the background and note the process id number given (let us assume it is 1234). Check to see that the process is running in the background (use the jobs command).

Signals

Try the following command:

$kill 1234 (use the number shown for your process)

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

Now repeat the same steps (i.e., create the forever job in the background and check its PID (we are assuming its 1234, but it could be anything)).

For each time you create the forever job try killing it with one of the following commands:

$kill -15 1234 (specify the correct PID, we are assuming its 1234)

$kill -TERM 1234

$kill -SIGTERM 1234

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

Signals

Open two terminals (if you are using telnet then open two telnet connections)

Use the ps command to determine the process id number of the terminal you are not using, as follows:

$ps -f

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

Now try running the following command:

kill pidofbash (or kill -15 pidbash)

What happened? Why?

Now try the following kill command:

$Kill -9 pidofbash (-9 is equivalent to –KILL or -SIGKILL)

Now what happened?

Signals

The kill command sends unix signals to processes. Pressing Ctrl+c and Ctrl+z is also sending signals. By default, the kill command sends the signal 15 (which is TERM and tells to process to **terminate itself**). It is also possible to use PIDs instead of job numbers and kill other signals. The general format is **kill -SIGNAL_ID_OR_NAME process_id**

Example : `kill -9 8733 or kill -KILL 8733`

signal number	signal name	meaning
1	HUP	Informing the process that its controlling terminal (like an ssh connection) is terminated
15	TERM	normal termination request
9	KILL	forcefully kills the proccess

The End