

Comp311- Lab Linux

Lab 7

Job and Process Management



Instructor :Murad Njoum

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** steps.
- Control the **priority** of newly created processes using the **nice** command.
- 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:

Use sleep command

```
sleep 100
```

```
ctrl+z
```

```
jobs
```

```
sleep 200 ctrl+z
```

```
sleep 300 ctrl+z
```

```
sleep 400 ctr+z
```

```
jobs %2 %4
```

```
jobs
```

```
fg %3
```

```
bg %4
```

```
jobs
```

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.



What is the difference ?

```
sleep
```

```
sleep&
```

```
forever
```

```
forever&
```

Cont..:

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
```



Cont..

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. The first is the job id number and the second is the job process number. These numbers are important to be able to reference the job later on for manipulation

Cont..

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
[3] +	Running	forever



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

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

Continue

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?

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.



Practice

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&



fg %1

Ctrl+z

kill %2; bg %3

Practice 2

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]	Running	forever
[2]-	Stopped	forever
[3]	Terminated	forever
[4]+	Stopped	forever
[5]	Terminated	forever
[6]	Terminated	forever



fg %2, ctrl+z

fg %4 ctrl+z

kill %3 %5 %6

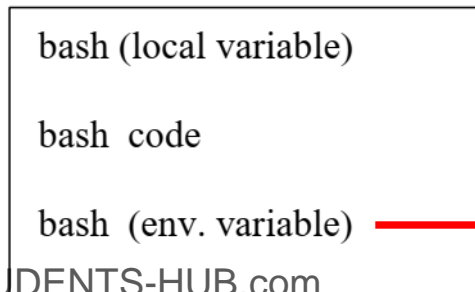
Process Control

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:

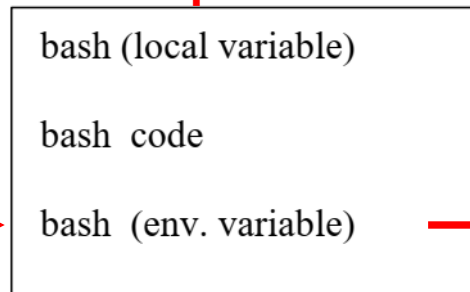
When we run the command **ls** under the **bash** shell, a copy of bash is created and which is replaced by ls.

pid (100)
Original



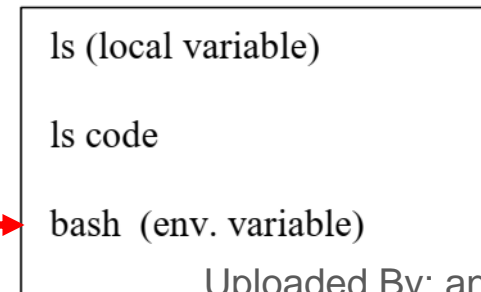
Fork →

pid (200)
parent



Exec →

pid (200)
Child



Continue..

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

Let us now run through to see the 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 (do full format)

Describe the output?

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



Cont..

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?

Now run the command:

env | grep var

Which do you see now? Why?



```
:~$ env|grep var2
```

```
:~$ set|grep var2
```

```
:~$ var1=first
```

```
:~$ var2=second
```

```
:~$ set|grep var2  
var2=second
```

```
:~$ set|grep var1  
var1=first
```

```
:~$ env|grep var1
```

```
:~$ env|grep var2
```

```
:~$ export var2
```

```
:~$ env|grep var2  
var2=second
```



Cont:

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?

Neither of them appears, because they are local in bash shell (parent).

Now run the command:

env | grep var

Which do you see now? Why?

Var2 appears, because it's environment variable (global).

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



Now run a child processes (ksh) as follows:

ksh

Run the command:

ps -f

What is the output now?

```
mnjourn@ubuntu:~$ ksh
```

```
$ ps -f
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
mnjourn	1902	1838	0	23:04	pts/1	00:00:00	-bash
mnjourn	1930	1902	0	23:05	pts/1	00:00:00	ksh
mnjourn	1932	1930	0	23:05	pts/1	00:00:00	ps -f

Continue:

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)



UID	PID	PPID	C	STIME	TTY	TIME	CMD
mnjourn	2354	2267	0	12:26	pts/0	00:00:00	-bash
mnjourn	2373	2354	0	12:32	pts/0	00:00:00	ksh
mnjourn	2377	2373	0	12:33	pts/0	00:00:00	-sh
mnjourn	2378	2377	0	12:33	pts/0	00:00:00	ps -f

mnjourn	2354	2267	0	12:26	pts/0	00:00:00	-bash
mnjourn	2373	2354	0	12:32	pts/0	00:00:00	ksh
mnjourn	2377	2373	0	12:33	pts/0	00:00:00	ps -f

exec ps -f

What happen with ksh shell (parent of ps -f) process?

Destroyed



More ..



What would you expect to happen if you run the command “exec ps -f” again?

_____ Destroy the bash shell (system terminated) _____.

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**.



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 or nice -n 10 ps -l (renice with -p, top,htop)

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?



Niceness values range from

-20 (most favorable to the process) to **19** (least favorable to the process).

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.



Signals :

The **TERM** signal tries to terminate signals **cleanly** and may be **blocked** by processes such as shells. **this signal can be blocked, handled, and ignored. It is the normal way to politely ask a program to terminate.**

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?



EXAMPLES

kill -9 -1

Kill all processes you can kill.

kill -l

List the available signal choices

kill 123 543 2341 3453

Send the default signal, **SIGTERM**, to all those processes.



\$ Kill -L

- 1) SIGHUP
- 2) SIGINT
- 3) SIGQUIT
- 4) SIGILL
- 5) SIGTRAP
- 6) SIGABRT
- 7) SIGBUS
- 8) SIGFPE
- 9) SIGKILL
- 10) SIGUSR1
- 11) SIGSEGV
- 12) SIGUSR2
- 13) SIGPIPE
- 14) SIGALRM
- 15) SIGTERM
- 16) SIGSTKFLT
- 17) SIGCHLD
- 18) SIGCONT
- 19) SIGSTOP
- 20) SIGTSTP
- 21) SIGTTIN
- 22) SIGTTOU
- 23) SIGURG
- 24) SIGXCPU
- 25) SIGXFSZ
- 26) SIGVTALRM
- 27) SIGPROF
- 28) SIGWINCH
- 29) SIGIO
- 30) SIGPWR
- 31) SIGSYS
- 34) SIGRTMIN
- 35) SIGRTMIN+1
- 36) SIGRTMIN+2
- 37) SIGRTMIN+3
- 38) SIGRTMIN+4
- 39) SIGRTMIN+5
- 40) SIGRTMIN+6
- 41) SIGRTMIN+7
- 42) SIGRTMIN+8
- 43) SIGRTMIN+9
- 44) SIGRTMIN+10
- 45) SIGRTMIN+11
- 46) SIGRTMIN+12
- 47) SIGRTMIN+13
- 48) SIGRTMIN+14
- 49) SIGRTMIN+15
- 50) SIGRTMAX-14
- 51) SIGRTMAX-13
- 52) SIGRTMAX-12
- 53) SIGRTMAX-11
- 54) SIGRTMAX-10
- 55) SIGRTMAX-9
- 56) SIGRTMAX-8
- 57) SIGRTMAX-7
- 58) SIGRTMAX-6
- 59) SIGRTMAX-5
- 60) SIGRTMAX-4
- 61) SIGRTMAX-3
- 62) SIGRTMAX-2
- 63) SIGRTMAX-1
- 64) SIGRTMAX

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 pidbash (or kill -15 pidbash) (i.e kill -15 2890)

What happened? Why?

Nothing happen, because the term (-15) signal terminated when finish the current process.

Now try the following kill command:

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

Now what happened?

Killed the process direct, because the KILL (-9) signal terminated immediately. the current process

Quiz (Bonus 5)

Without Using fg and ctrl+z command, show the result at screen

- | | |
|----------------|---------|
| [1] Hangup | forever |
| [2]- Stopped | forever |
| [3] Terminated | forever |
| [4]+ Stopped | forever |
| [5] Hangup | forever |
| [6] Killed | forever |



- 1.Run script 6 times ,i.e: forever
2. kill -SIGSTOP %4 %2
3. kill -SIGHUP %1 %5 ; kill -SIGTERM %3; kill -SIGKILL %6



Thank You for attention !

Published By: Murad Njoum