Unit 5-Introduction to processes

A program/command when executed, a special instance is provided by the system to the process. This instance consists of all the services/resources that may be utilized by the process under execution.

* Whenever a command is issued in Unix/Linux, it creates/starts a new process.
* For example, pwd when issued which is used to list the current directory location the user is in, a process starts.
* Through a 5 digit ID number Unix/Linux keeps an account of the processes, this number is called process ID or PID.
* Each process in the system has a unique PID.
* Used up pid’s can be used in again for a newer process since all the possible combinations are used.
* At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

**Initializing a process**

**A process can be run in two ways:**

**Method 1: Foreground Process :** Every process when started runs in foreground by default, receives input from the keyboard, and sends output to the screen.  When issuing pwd command

**$ ls pwd**

**Output:**

$ /home/geeksforgeeks/root

When a command/process is running in the foreground and is taking a lot of time, no other processes can be run or started because the prompt would not be available until the program finishes processing and comes out. 

**Method 2: Background Process:**It runs in the background without keyboard input and waits till keyboard input is required. Thus, other processes can be done in parallel with the process running in the background since they do not have to wait for the previous process to be completed.   
Adding & along with the command starts it as a background process

**$ pwd &**

Since pwd does not want any input from the keyboard, it goes to the stop state until moved to the foreground and given any data input. Thus, on pressing Enter:  
**Output:**

[1] + Done pwd

$

That first line contains information about the background process – the job number and the process ID. It tells you that the ls command background process finishes successfully. The second is a prompt for another command.

**Process Status Command(PS)**

**Tracking ongoing processes**

ps (Process status) can be used to see/list all the running processes.

**$ ps**

PID TTY TIME CMD

19 pts/1 00:00:00 sh

24 pts/1 00:00:00 ps

For more information -f (full) can be used along with ps

**$ ps –f**

UID PID PPID C STIME TTY TIME CMD

52471 19 1 0 07:20 pts/1 00:00:00f sh

52471 25 19 0 08:04 pts/1 00:00:00 ps -f

For single-process information, ps along with process id is used

**$ ps 19**

PID TTY TIME CMD

19 pts/1 00:00:00 sh

For a running program (named process) **Pidof** finds the process id’s (pids)   
**Fields described by ps are described as:**

* **UID**: User ID that this process belongs to (the person running it)
* **PID**: Process ID
* **PPID**: Parent process ID (the ID of the process that started it)
* **C**: CPU utilization of process
* **STIME**: Process start time
* **TTY**: Terminal type associated with the process
* **TIME**: CPU time is taken by the process
* **CMD**: The command that started this process

**There are other options which can be used along with ps command :**

* **-a**: Shows information about all users
* **-x**: Shows information about processes without terminals
* **-u**: Shows additional information like -f option
* **-e**: Displays extended information

Mechanism of process creation:

**A process is simply a running program. A process is said to be born when the program starts execution and remains alive till the process is active. After the execution the process is said to be died.**

**A process in a Unix system is created by fork system call. Every process except process 0 is created. The process that invokes the fork system call is parent process and the newly created process is the child process.**

**Every process has one parent process but a parent can have many child process. The kernel identifies it process by its process identification number (PID). Process 0 is a special process created "by hand" when the system boots.**

**There are 3 distinct phase in mechanism of process creation and uses 3 system calls: fork(), exec() and wait().**

**fork(): Creates a child process. A new process is created because an existing process creates an exact copy of itself. This child process has the same environment as its parent but only the PID is different. This procedure is known as forking.**

**exec(): After forking the process, the address space of the child process is overwritten by the new process data. This is done through exec call to the system. No new process is created over here. The PID & PPID remains unchanged.**

**wait(): The parent then executes wait system call to wait for the child process to complete its execution.**

**The important attributes that are inherited by the child process from its parents are: Real UID and GID, PGID, Nice value, Environment setting, Current working directory, memory segments etc.**

**When the process dies it immediately moves to the zombie state and remains in that state until the parent picks it up to exit status. A zombie is the harmless dead child.**

**When a child dies kernel send SIGCHLD signal to parent for information. It is also possible for the parent to die before child dies. The child then becomes an orphan process and kernel makes 'init' parent of all the orphans. When this adopted child dies 'init' picks up it to exit status**

**Nice renice Command:**

**nice command** in Linux helps in execution of a program/process with modified scheduling priority. It launches a process with a user-defined scheduling priority. In this, if we give a process a higher priority, then Kernel will allocate more CPU time to that process.

Whereas the **renice command** allows you to change and modify the scheduling priority of an already running process. Linux Kernel schedules the process and allocates CPU time accordingly for each of them.

**nice**: It starts a new process (job) and assigns it a priority (nice) value at the same time.   
Syntax:

nice [-nice value]

nice value ranges from -20 to 19, where -20 is of the highest priority.

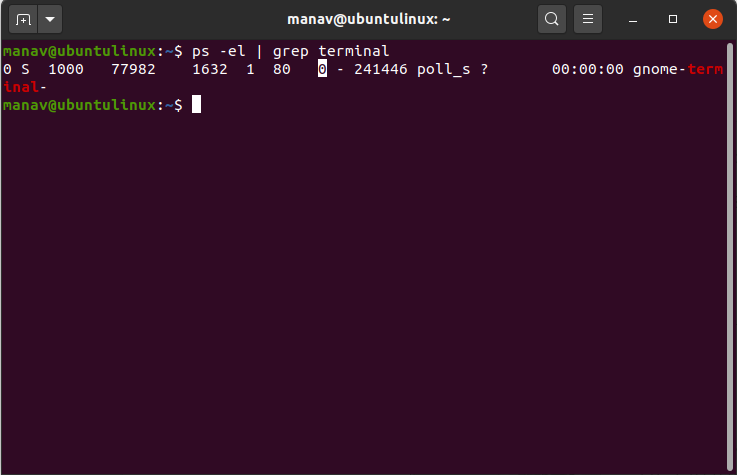
**renice** : To change the priority of an already running process renice is used.   
Syntax:

renice [-nice value] [process id]

### Working with nice and renice Command

**1.** To check the nice value of a process. 

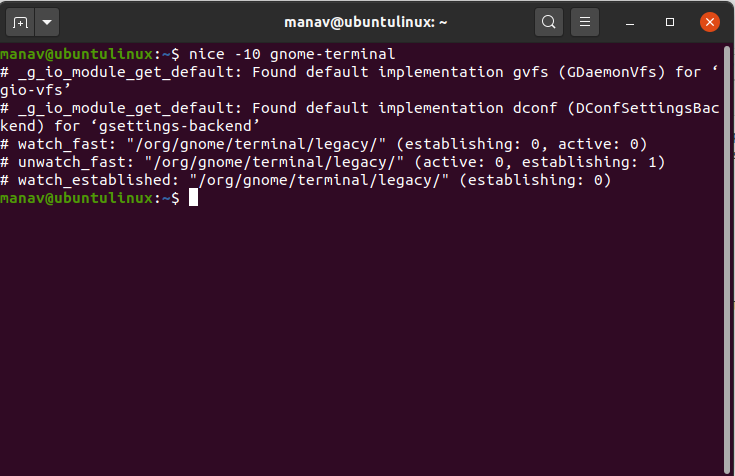
ps -el | grep terminal



The eight highlighted value is the nice value of the process.

**2.** To set the priority of a process

nice -10 gnome-terminal



**Kill command for Stopping a process:**  
When running in foreground, hitting Ctrl + c (interrupt character) will exit the command. For processes running in background kill command can be used if it’s pid is known.

**$ ps –f**

UID PID PPID C STIME TTY TIME CMD

52471 19 1 0 07:20 pts/1 00:00:00 sh

52471 25 19 0 08:04 pts/1 00:00:00 ps –f

**$ kill 19**

Terminated

If a process ignores a regular kill command, you can use kill -9 followed by the process ID.

**$ kill -9 19**

Terminated

Internal and external commands

The UNIX system is command-based *i.e* things happen because of the commands that you key in. All UNIX commands are seldom more than four characters long.  
**They are grouped into two categories:**

* **Internal Commands :** Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn’t have to search the given path for them in the PATH variable, and also no process needs to be spawned for executing it.  
  Examples: source, cd, fg, etc.
* **External Commands :** Commands which aren’t built into the shell. When an external command has to be executed, the shell looks for its path given in the PATH variable, and also a new process has to be spawned and the command gets executed. They are usually located in /bin or /usr/bin. For example, when you execute the “cat” command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.  
  Examples: ls, cat etc.

If you know about UNIX commands, you must have heard about the [ls](https://www.geeksforgeeks.org/practical-applications-ls-command-linux/) command. Since [ls](https://www.geeksforgeeks.org/practical-applications-ls-command-linux/) is a program or file having an independent existence in the /bin directory(or /usr/bin), it is branded as an **external command** that actually means that the ls command is not built into the shell and these are executables present in a separate file. In simple words, when you will key in the ls command, to be executed it will be found in /bin. Most commands are external in nature, but there are some which are not really found anywhere, and some which are normally not executed even if they are in one of the directories specified by PATH. For instance, take [echo](https://www.geeksforgeeks.org/echo-command-in-linux-with-examples/) command:

**$type echo**

echo is a shell builtin

[echo](https://www.geeksforgeeks.org/echo-command-in-linux-with-examples/) isn’t an external command in the sense that, when you type echo, the shell won’t look in its PATH to locate it(even if it is there in /bin). Rather, it will execute it from its own set of built-in commands that are not stored as separate files. These built-in commands, of which **echo** is a member, are known as **internal commands**.

You now might have noticed that it’s the shell that actually does all these works. This program starts running as soon as the user log in and dies when the user logs out. The shell is an external command with a difference, it possesses its own set of internal commands. So, if a command exists both as an internal command of the shell as well as external one(in /bin or /usr/bib), the shell will accord top priority to its own internal command of the same name.

This is exactly the case with **echo** which is also found in /bin, but rarely ever executed because the shell makes sure that the internal **echo** command takes precedence over the external. Now, talk more about the internal and external commands.

**Getting the list of Internal Commands**

If you are using bash shell you can get the list of shell built-in commands with **help** command :

**$help**

// this will list all

the shell built-in commands //

**How to find out whether a command is internal or external?**

In addition to this you can also find out about a particular command *i.e* whether it is internal or external with the help of [type](https://www.geeksforgeeks.org/type-command-in-linux-with-examples/) command :

**$type cat**

cat is /bin/cat

//specifying that cat is

external type//

**$type cd**

cd is a shell builtin

//specifying that cd is

internal type//

**Internal vs External**

The question that when to use which command between internal and external command is of no use cause the user uses a command according to the need of the problem he wants to solve. The only difference that exists between internal and external commands is that internal commands work much faster than the external ones as the shell has to look for the path when it comes to the use of external commands.

There are some cases where you can avoid the use of external by using internal in place of them, like if you need to add two numbers you can do it as:

//use of internal command let

for addition//

**$let c=a+b**

instead of using :

//use of external command expr

for addition//

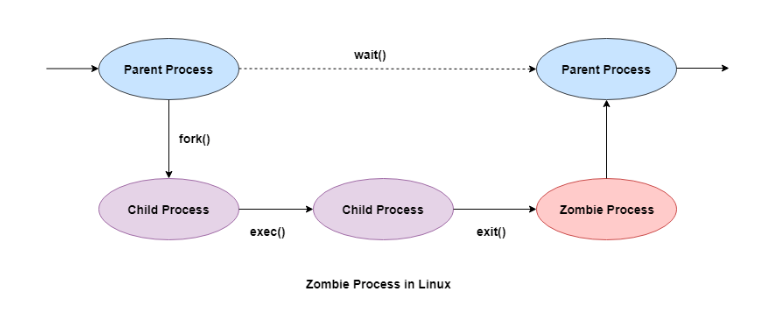
**$c=`expr $a+$b`**

In such a case, the use of let will be a better option as it is a shell built-in command so it will work faster than the expr which is an external command.

Process States and Zombies

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child’s exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process.

A diagram that demonstrates the creation and termination of a zombie process is given as follows −



Some of the salient points related to zombie processes are as follows −

* All the memory and resources allocated to a process are deallocated when the process terminates using the exit() system call. But the process’s entry in the process table is still available. This process is now a zombie process.
* The exit status of the zombie process zombie process can be read by the parent process using the wait() system call. After that, the zombie process is removed from the system. Then the process ID and the process table entry of the zombie process can be reused.
* If the parent process does not use the wait() system call, the zombie process is left in the process table. This creates a resource leak.
* If the parent process is not running anymore, then the presence of a zombie process indicates an operating system bug. This may not be a serious problem if there are a few zombie processes but under heavier loads, this can create issues for the system such as running out of process table entries.
* The zombie processes can be removed from the system by sending the SIGCHLD signal to the parent, using the kill command. If the zombie process is still not eliminated from the process table by the parent process, then the parent process is terminated if that is acceptable.

**Jobs in background**

## Whats a job in Linux

A job is a process that the shell manages. Each job is assigned a sequential job ID. Because a job is a process, each job has an associated PID. There are three types of job statuses:  
1. **Foreground**: When you enter a command in a terminal window, the command occupies that terminal window until it completes. This is a foreground job.  
2. **Background**: When you enter an ampersand (&) symbol at the end of a command line, the command runs without occupying the terminal window. The shell prompt is displayed immediately after you press Return. This is an example of a background job.  
3. **Stopped**: If you press Control + Z for a foreground job, or enter the stop command for a background job, the job stops. This job is called a stopped job.

**Job contol command**

Job control commands enable you to place jobs in the foreground or background, and to start or stop jobs. The table describes the job control commands.

|  |  |
| --- | --- |
| **Option** | **Description** |
| jobs | Lists all jobs |
| bg %n | Places the current or specified job in the background, where n is the job ID |
| fg %n | Brings the current or specified job into the foreground, where n is the job ID |
| Control-Z | Stops the foreground job and places it in the background as a stopped job |

## Running a Job in the Background

To run a job in the background, you need to enter the command that you want to run, followed by an **ampersand (&)** symbol at the end of the command line. For example, run the sleep command in the background.

sleep 100 &

[1] 1302

$

The shell returns the job ID, in brackets, that it assigns to the command and the associated PID. With the job ID, you can use the job control commands to manage the job whereas the kernel uses PIDs to manage jobs.

When a background job is complete and you press Return, the shell displays a message indicating the job is done.

[1] + Done sleep 100 &

$

## Managing the background jobs

You can use the **jobs** command to list the jobs that are currently running or suspended in the background.

$ jobs

[1]+ Running sleep 100 &

You can use the fg command to bring a background job to the foreground.

$ fg % 1

sleep 100

**Note**: The foreground job occupies the shell until the job is completed, suspended, or stopped and placed into the background.

You can use the ‘**Control+Z** keys and **bg** command to return a job to the background. The Control+Z keys suspend the job, and place it in the background as a stopped job. The bg command runs the job in the background. For example:  
**1. Using CTRL+Z**

$ sleep 100

^Z

[1]+ Stopped sleep 100

$ jobs

[1]+ Stopped sleep 100

**2. Using bg**

$ bg % 1

[1]+ sleep 100 &

$ jobs

[1]+ Running sleep 100 &

**cron:Running jobs periodically**

cron is a time-based scheduling utility program. With cron, you can launch routine [background jobs](https://everythingdevops.dev/linux-background-and-foreground-process-management/) at specific times, days, months, etc., on an ongoing basis. The jobs launched are referred to as cron **jobs.**

The cron utility program is driven by a configuration file called /etc/crontab (cron table), which contains various shell commands that need to be run at scheduled times.

There are two types of crontab files, the system-wide crontab files, and the individual user crontab files. Each line of a crontab file represents a job and is composed of a cron expression, followed by a shell command to execute.

In this article, you will:

* see some uses of the cron utility,
* understand the structure of a **crontab** file, and then
* learn how to schedule a periodic task on Linux with cron.

**Uses of cron**

There are several cases which you, as a Linux user, would want to schedule tasks with cron; a few of them are:

* To schedule automated updates.
* To back up data at a specific time of the hour, day, month, year, etc.
* And so much more.

**time: Timing processes**

The time command in Linux and Unix-like operating systems lets you determine how long a specific command will take to run. Usually, it’s used to measure the performance of scripts or commands. The faster a command finishes its task, the better its performance.

In addition, the time command can also display the system resource usage of the process, making it a helpful tool for reviewing the efficiency of a specific command.

For example, you want to choose between two **[cron jobs](https://www.hostinger.in/tutorials/cron-job)** that can run the same task. You can determine how long each cron job will take to complete the tasks by running the time command.

However, before we begin talking about how to use the Linux time command, keep in mind that it differs depending on the shell you use.

Using the time command is very simple – all you have to do is open your shell program and enter:

$ time

To take full advantage of the time command, you have to understand its syntax:

$ time

[arg1] [arg2] ... [argN]

time [options]

[arg1] [arg2] ... [argN]

**Time** runs the given **command** with any given arguments (**arg**). Once the command finishes its task, time outputs information which includes the duration it took to run the command and the resources used by it.

If the program command exits with a non-zero status indicating failure, time will output a warning message and the exit status.

By entering the string format used by the command, you can determine which information to display in the system resource usage section.

If the user doesn’t specify any format, but the time environment variable is set, its value will be used as the format. Otherwise, the default format built into time will be used.

**Options** are resource specifiers that change the output of time. They must appear before the command. Anything entered after the command will be regarded as an argument.

To view all the options you can use for time, enter the following into the command line:

$ man time

Signals are the interrupts that are sent to the program to specify that an important event has occurred. Events can vary from user requests to invalid memory access errors. Various signals, like the interrupt signal, means that the user has asked the program to perform something which is not present in the user flow of control.

There are two kinds of signals:

1. Maskable
2. Non-Maskable  
   ***Maskable: -*** Maskable Signals are the signals that the user can change or ignore, for example, **Ctrl +C.**  
   ***Non-Maskable: -*** Non-Maskable Signals are the signals that the users cannot change or ignore. Non-Maskable signals mainly occur if a user is signaled for non-recoverable hardware errors.
3. There are various processes in different states in the Linux computer system. All these processes belong to either the operating system or the user application. A mechanism is required for the kernel and these processes to coordinate their activities. One method to perform this, for the process, to inform others if anything vital occurs. That's the reason we have signals.
4. Basically, the signal means a one-way notification. The kernel sent the signal to the process, to one process to itself, or another process.  
   Linux signals trace their origin to Unix signals. In later Linus versions, real-time signals were added. Signal inter process is an easy and lightweight form of communication and is therefore compatible with embedded systems.

