Process

A process is simply an instance of a running program. A process is said to be born when a program starts execution and remains alive as long as the program is active. After execution is complete, the process is said to die. A process also has a name, usually the name of the program being executed. For example, when you execute the grep command, a process named grep is created. When two users run the same program, there is one program on disk but two processeses in memory.

Attributes of every process are maintained by the kernel in memory in a separate structure called the **process table.** Two important attributes of a process are:

- **Process-id (PID)** Each process is uniquely identified by a unique integer called the Process-id (PID) that is allotted by the kernel when process is born.
- **Parent PID (PPID)** The PID of the parent is also available as a process attribute. When several processes have same PPID, it often makes sense to kill the parent rather than all its children separately.

$$

The shell maintains a set of environment variables, like PATH and SHELL. The shell's pathname is stored in SHELL, but its PID is stored in a special variable, $$. To know the PID of your current shell used $$.

```
$ echo $$

291
```

Parent and Children

```
$ cat student.lst
```

When you run the above cat command, cat process become active as long as command is active. The shell is said to be the parent of cat.

```
$ cat student.lst | grep 'director'
```

The above command start two processes for the two commands. The process names are cat and grep and both are child of shell.

Wait and Not Wait

When the child process is running, the parent can do:

1) It may wait for the child to die so that it can spawn the next process. The death is informed by the kernel to parent.

   When you execute a command from the shell, the shell process waits for the command to die before it returns the prompt to take up the next command.

2) It may not wait for the child to die at all and may continue to spawn other processes. This is what the init process does, the reason why init is the parent of several processes. Your shell can also create a process without waiting for it to die.

Process status (ps)

ps command display some attribute of running processes. By default, ps displays the processes of user, who run the command.

If you execute the command immediately after logging in, it will show you the process of bash shell. Here PID means process id, TTY means terminal, TIME is cumulative processor time, CMD is a process name.

```
$ ps
PID   TTY       TIME  CMD
260  console  0:00   bash
```

ps options

full listing (-f)

-f option gives you full (detail) list of every process. It will also show UID (user id), PPID (parent process id) etc…

```
$ ps -f
UID  PID  PPID  C    STIME     TTY       TIME  CMD
amit 334  291   0   11:20:50  console  0:00   f1.sh
amit 291  1     0   10:22:55  console  0:00   bash
```

Displaying Proceses of a User (-u)

The system administrator needs to use the –u (user) option to know the activities of any user. –u (user) option give process details of particular user.

```
$ ps –u nilesh
PID   TTY       TIME   CMD
324   pts/3     11:05  vi
322   pts/3     11:02  bash
```

Displaying all user processes (-a)

The –a (all) option lists processes of all users but doesn't display the system processes.

```
$ ps -a
PID   TTY       TIME   CMD
322   pts/3     11:02  bash
324   pts/3     11:05  vi
402   pts/1     12:01  ksh
404   pts/1     12:05  sort
```

System processes (-e or –A)

Apart from the processes a user generates, a number of system processes keeps running all the time, most of them are not associated with any terminal. They are spawned during system startup and some of them start when the system goes to the multiuser state. To list all processes running on your machine use the –e or –A option.

```
$ ps -e
PID   TTY       TIME   CMD
0     ?         10:00  sched                    Take care of swapping
```

1	?	10:00	init	Parent of all shells
2	?	10:00	pageout	Part of the kernel
322	pts/3	11:02	bash	
324	pts/3	11:05	vi	
402	pts/1	12:01	ksh	
404	pts/1	12:05	sort	

Mechanism of Process creation

There are three distinct phases in the creation of a process and uses three important system calls or functions – fork, exec and wait.

The three phases are discussed below:

Fork

A proess in UNIX is created with the fork system call, which creates a copy of the process that invoke it. The process image is practically identical to that of the calling process, except for a few parameters like the PID. When a process s forked in this way, the child gets a new PID.

Exec

Forking creates a process but it is not enough to run a new program. To do that, the forked child needs to overwrite its own image with the code and data of the new program. This mechanism is called exec, and a child process is said to exec a new program. No new process is created here. The PID and PPID of exec'd program is remain unchanged.

Wait

The parent then executes the wait system call to wait for the child process to complete. It picks up the exit status of the child and then continues with its other functions.

Internal and External commands

From the process view point, the shell recognizes three types of commands:

1) External Commands : The most commonly used ones are the UNIX utilities and programs, like cat, ls, etc… The shell creates a process for each of theses commands that it executes while remaining their parent.

2) Shell Scripts : The shell executes these script by spawning another shell, which then executes the commands listed in the script.

3) Internal Commands: The shell has a number of built-in commands as well. It is called internal commands. Some them like cd and echo do not generate a process and are executed directly by the shell. Similarly, variable assignment with the statement x=5. Do not generate process.

Process States

At any instant of time, a process is in a particular state.

A process after creation is in the **runnable state** before it actually runs. While the process is running it is in **running state.** While the process is running, it may be invoke a disk I/O operation when it has nothing to do except wait for the I/O to complete. The process then moves to the **sleeping state** to be woken up when the I/O operation is over. A process can also be suspended by pressing a key (usually [Ctrl-z].

Process whose parents do not wait for their death move to the zombie state. When a process dies, it immediately moves to the **zombie state**. It remains in this state until the parent pick up the child's exit status from the process table. When that is done, the kernel frees the process table entry. A zombie is a harmless dead child but can not kill it.

It is also possible for the parent itself to die before the child dies. The child then becomes an orphan and the kernel makes init the parent of all orphans. When this adopted child dies, init waits for its death.

Running jobs in background

A multitasking system lets a user do more than one job at a time. Since there can be only one job in the forground, the rest of the jobs have to run in background. There are two ways of doing this:

- With the shell's **&** operator
- The **nohup** command

No logging out (&)

The & is the shell's operator used to run a process in the background. The parent in this case doesn't wait for the child's death. Just at the end of the command write &, so now the command will run in the background.

```
$ sort –o student.lst student.lst &

520                                    The job's PID
```

The shell immediately returns a number – The PID of the invoked command (550). The prompt is returned and the shell is ready to accept another command even though the previous command has not been terminated yet. The shell-however, remains the parent of the background process. Using an **&** you can run as many jobs in the background as the system load permits.

Log out Safely (nohup)

When a parent is killed, its childs (background and forground jobs) are also normally killed.

The nohup (no hangup) command, when prefix to a command, permits execution of the process even after the user has logged out. You must use the & with it as well:

```
$ nohup sort student.lst &

525

Sending output to nohup.out
```

The shell return the PID, and display the message. nohup send the standard output to the file nohup.out. If you do not get this message you have to take care of it, by storing the output using redirection if necessary. You can now safely log lout without aborting the command.

The shell died, on logging out but its child did not. It turned in to an orphan. The kernel reassign the PPID of the orphan to the system's init process (PID 1). When the user log out, the init become the parent of all process run with nohup command.

When you use the ps command after using nohup from another terminal, you will notice following:

```
$ ps –f –u kumar
UID      PID   PPID  C     STIME      TTY   TIME  COMMAND
kumar   586   1       45    10:52:08   01     01      sort student.lst
```

If you run more than one command in a pipeline, you should use the nohup command at the beginning of each command in the pipeline.

```
$ nohup  grep “Surat” student.lst  & | nohup sort &
```

Job execution with low priority (nice)

UNIX offers the nice command, which is used with the & operator to reduce the priority of jobs.

To run a job with a low priority, the command name should be prefixed with nice:

```
$ nice wc –l  f1.txt
Or
$ nice  wc –l f1.txt  &
```

nice values are system-dependent and typically range from 1 to 19. Command executes with a nice value that is generally in the middle of the range usually 10. A higher nice value implies a lower priority. nice reduces the priority of any process, thereby raising its nice value.

You can also specify the nice value explicitly with the –n option.

```
$ nice –n 5 wc –l f1.txt
```

A nonprivileged user can’t increase the priority of a process,that power is reserved for the superuser. The nice value is displayed with the **ps –o nice** command.

KILLING PROCESS WITH SIGNALS

You can communicate with the process by sending a signal to the process. Each signal is identified by a number and is designed to perform a specific function. Signals are better represented by their symbolic names having the SIG prefix.

If a program is running longer than you anticipated and you want to terminate it, you normally press the interrupt key. This sends the process the SIGINT signal (number 2). The default action of this signal is to kill the process. A process may also ignore a signal or execute some user defined code written to handle that signal.

Premature termination of a process (kill)

The kill command sends a signal, usually with a intention of killing one or more processes. kill is an internal command in most shells. The command uses one or more PIDs as its arguments, and by default uses the SIGTERM(15) signal.

```
$ kill 105

Or

$ kill  –s  TERM 105
```

Terminate the job having PID 105.

If you do not remember the PID use ps to know that and then use kill.

If you run more than one job – either in the background or in different windows. You can kill them all with a single kill statement. Just specify all their PIDs with kill.

```
$ Kill 131  132  133  136 150
```

The above command kill the above processes specified with process id. If all these processes have the same parent, you may simply kill the parent in order to kill all its children. If you create process with nohup command, and then logout then the init become parent of all processes(created with nohup). But you can not kill the init. So have to individually kill the processes.

Killing the last background job

For most shells, the system variable $! Store the PID of the last background job. So you can kill the last background process without using the ps command to find out its PID.

```
$ sort –o student.lst student.lst &

345

$ kill $!                          Kills the sort command
```

Using kill with other signals

By default kill ues the SIGTERM signal (15) to terminate the process. Sometime some program simply ignore it and continue execution normally. In that case the process can be killed with the SIGKILL (9).

This signal can not be generated at the press of a key, you must use kill with the signal name (without the SIG) preceded by the –s option.

$ kill –s KILL 121	Recommended
Or	
$ Kill -9 121	Not recommended

A simple kill command (with TERM) would not kill the login shell. You can kill your login shell by using any of these commands.

$ kill -9 $$	$$ stores the PID of current shell
Or	
$ kill -s KILL 0	kills all processes including the login shell

Job Control

A job is the name given to a group of processes. The easiest way of creating a job is to run a pipeline of two or more commands. Now consider you expect a job to complete in 10 minutes and it goes on for half an hour. If you kill the job now, you will loss a lot of work. You can use the job control facility to manipulates jobs.

Job control here means you can do following

- Relegate (send) a job to the background (bg)
- Bring it back to the foreground (fg)
- List the active jobs (jobs)
- Suspend a foreground job ([Ctrl –z])
- Kill a job (kill)

If you invoke a command and the prompt has not yet returned, you can suspend the job by pressing [Ctrl-z]. you will then see the following message:

[1] + Stopped	sort student.lst > student.lst

This job is not been terminated yet, it is only stopped (suspended). You can now use the bg command to push the current foreground job to background.

```
$ bg

[1] sort student.lst > student.lst &
```

The & shows that this job is now running in background. Now you can start more jobs.

```
$ wc –l  f1.txt > word_count.txt &

[2] 550
```

Here [2] indicates second job.

```
$ ls –l  > list.txt &

[3]  560
```

Each of these job comprises a single process. Now you have three jobs running.

You can have listing or their status with the jobs command.

```
$ jobs

[3]  +  Running

[1]  -  Running

[2]     Running
```

You can now bring any of the background jobs to the foreground with the fg command. To bring the current job to the foreground, use

```
$  fg
```

This will bring the ls command in the foreground.

The fg and bg command can also be used with the job number, job name or a string as arguments, prefixed by % symbol.

$ fg %1	Bring first job to foreground
$ fg $sort	Brings sort job to foreground
$ bg %2	Sends second job to background
$ bg %?perm	Sends to background job containing string perm

At any time, however, you can terminate a job will the kill command using the same identifiers as above. Thus, kill %1 kills the first background job with SIGTERM.

EXECUTE LATER (at and batch)

UNI provides sophisticated facilities to schedule a job to run at a specified time of day.

One – time execution (at)

at takes as its argument the time the job is to be executed and displays the at> prompt. Input has to be supplied from the standard input.

```
$ at 16:10

at> student.sh

[Ctrl-d]

Commnds will be executed using /user/bin/bash

Job 1051177770.a at Wed Oct 08  16:10:00 2014
```

The job goes to the queue, and at 4:10 pm oct 8 the script student.sh will be executed. at shows the job number, the date and time of scheduled execution.

At does not indicate the name of the script to be executed. that is something the user has to remember. The standard output and standard error of the program are mailed to the user. Who can use any mail reading program to view it. Alternatively, a user may prefer to redirect the output of the command itself.

```
$ at 15:05

stud1.sh > report.lst
```

The output of stud1.sh will store in report.lst

you can also use the –f option to take command from a file. However, any error messages that may be generated when executing a program will, in the absence of redirection, continue to be mailed to the user.

To mail job completion to the user, use the **–m** option.

at also offers the keywords now, moon, midnight, today, tomorrow, hours, days, weeks, months and years. Moreover, it accepts the + symbol to act as an operator.

```
$ at 14

$ at 4pm
```

At 2:10pm	
At noon	12 hours today
at now + 1 year	At current time after 1 year
at 2:10pm + 1day	At 2:10pm tomorrow
at 14:15 January 12, 2014	
at 8am tomorrow	

Jobs can be listed with the **at –l** command and removed with **at -r**. Unfortunately, there is no way you can find out the name of the program scheduled to be executed.

Executed in Batch Queue (batch)

The batch command also schedules job for later execution. Jobs are executed as soon as the system load permits. The command does not take any arguments but uses an internal algorithm to determine the execution time. The response of batch is similar to at otherwise.

```
$ batch < stud.sh
Commnds will be executed using /user/bin/bash
Job 1031134770.a at Wed Oct 09 17:10:00 2014
```

Any job scheduled with batch goes to a special **at** queue from where it can be removed with at –r.

Running job periodically (cron)

cron executes programs at regular intervals. Every minute it wakes up and looks in a control file (the crontab file) in /var/spool/cron/crontabs for instruction to be performed at that instant. After executing them, it goes back to sleep, only to wake up the next minute.

A user may also be permitted to place a crontab file named after her login name in the crontabs directory. Sumit has to place his crontab commands in the file /var/spool/cron/crontabs/sumit .

Format /specimen copy

```
00-10 17 * 3,,6,9,12,5 find / -newer .last_time -print > backuplist
```

Each line contains a set of six field separated by whitespace. The command is shown in the last field. All of these fields together determine when and how often the command will be executed.

The first field (legal values 00 to 59) specifies the number of minutes after the hour when the command is to be executed. The range 00-10 schedules execution every minute in the first 10 minutes of the hour.

The second field (17 means 5 pm) indicates the hour in 24 hour format for scheduling (legal value 1 to 24).

The third field (legal values 1 to 31) controls the day of the month. This field (here an asterisk) read with the other two, implies that the command is to be executed every minute, for the first 10 minutes, starting at 5 pm every day.

The fourth field (3,6,9,12)specifies the month (legal values 1 to 12).

The fifth field (5-Friday) indicates the day of week (legal value 0 to 6), Sunday having the value 0.

The sixth field is a command. The find command will thus be executed every minute in the first 10 minutes after 5. pm, every Friday of the months March, June, September and December of Every Year.

Creating a crontab file (crontab)

You can also create your own crontab file with vi in the format shown above. The crontab command is used to place the file in the directory containing crontab files for cron to read the file again.

$ crontab cron.txt	cron.txt contains cron commands

If sumit runs this command, a file named sumit will be created in /var/spool/cron/crontabs containing the content of cron.txt. In this way different users can have crontab files named after their user-ids.

You can see the contents of your crontab files with crontab –l and remove them with crontab –r.

cron is mainly used by the system administrator to perform housekeeping chores, like removing outdated files or collecting data on system performace. It is also extremely useful to periodically dial up to an internet mail server to send and retrieve mail.