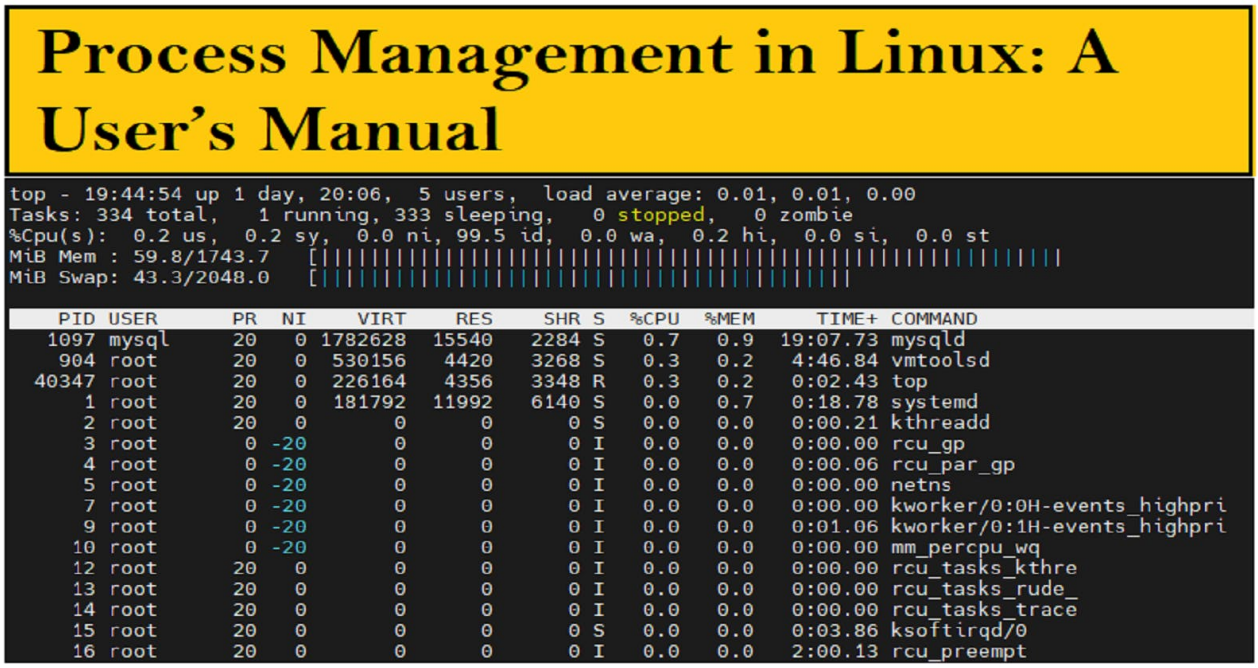
Process Management in Linux: A User's Manual

rakeshkumarjangid.hashnode.dev/process-management-in-linux-a-users-manual

Rakesh Kumar Jangid



Understand the process & How it works?.

1. **Process**: A process is like a task your computer is performing. It's an instance of a program that's currently running. When a process is created. For example, when you create an executable program and run this program, It becomes a process when it running.

   Inside a process, some imp. key things are available :

   - `PID:` "Process ID"
   - `PPID:` "Parent Process ID"
   - `Owner:` "User of the process, who runs this?"
   - `Executable Program:` "How does this program work?"
   - `Memory Time:` "Memory time is taken by the process"
   - `CPU Time:` "CPU time is taken by the process"
   - `Security Context`: "Linux Security Context"

2. Life Cycle of Process

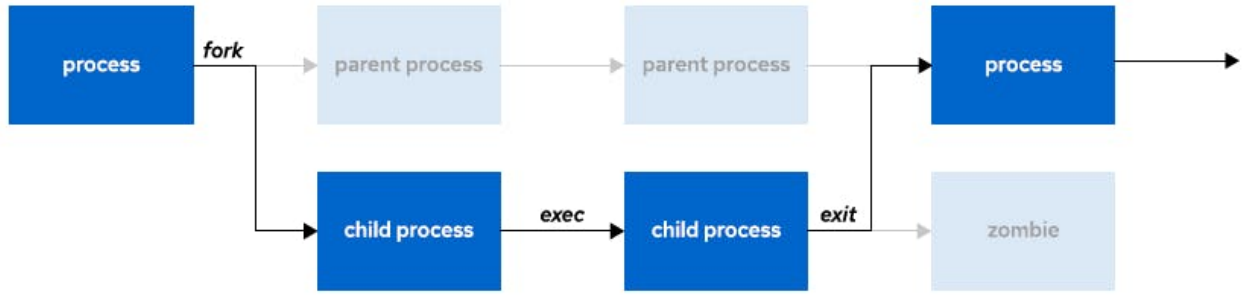


Figure 8.1: Process life cycle

Let's understand the process...

1. **Program to Process**: A program is a set of instructions that are loaded into memory. When these instructions are executed, the program becomes a process. Each process has a unique process ID `(PID)` for tracking and security purposes.

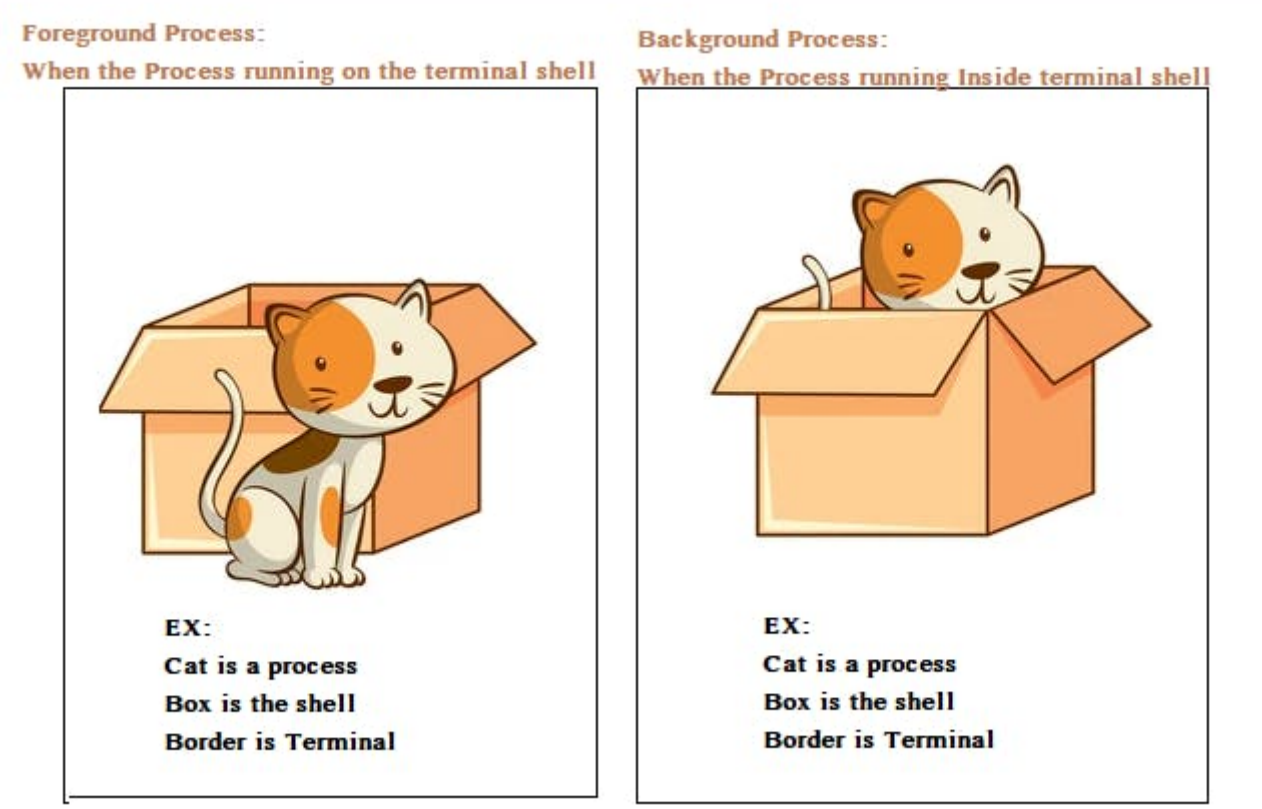
2. **Creating a Child Process**: A parent process can create a child process through a mechanism known as `"forking".` In this process, the parent duplicates its own address space to create a new process structure for the child. The child process inherits various attributes from the parent, such as security identities, file descriptors, resource privileges, environment variables, and program code.

3. **Execution of Child Process**: Once created, the child process can execute its own program code. During this time, the parent process usually sleeps, setting a wait request to be signaled when the child completes.

4. **Zombie Process**: A zombie process, also known as a **defunct process**, is a peculiar state in which a process has completed its execution (via the `exit` system call), but its entry still lingers in the process table. Essentially, it's a process that has finished its job but hasn't been properly cleaned up by its parent process.

Example : Let's consider a real-world example. Imagine you're a chef `(parent process)` in a restaurant. You get an order `(program)` to prepare a dish. You start preparing the dish `(process)`. Now, you need to chop some vegetables, so you ask your assistant `(child process)` to do it. You wait `(sleep)` until your assistant finishes chopping `(child process execution)`. Once the assistant is done, they inform you and go on to their next task `(child process exits, leaving a zombie)`. You then continue preparing the dish (the parent process continues execution after cleaning up the zombie). and server the order (process completed)

What is the Background & Foreground Process ??



1. **Foreground Process**:
    - A **foreground process** is one that **requires user interaction**. When a process runs directly in the terminal shell, it occupies the terminal session, and you interact with it directly.
    - For example, if you execute a command that performs a task and waits for your input or displays output in the terminal, it is a foreground process.
    - While a foreground process is running, you cannot use the terminal for other commands until the process completes or you interrupt it.
2. **Background Process**:
    - A **background process** operates independently of user interaction. It runs **behind the scenes**, allowing you to continue using the terminal for other tasks.
    - When you start a process in the background, it doesn't hold the terminal session hostage. You can execute other commands or even disconnect from an SSH session without affecting the background process.
    - Background processes are useful for long-running tasks, such as monitoring events or performing lengthy computations.

💡

How to check the background process in Linux?

```
# jobs
```

`jobs` command is used to check whether your process is running background or not. if there are any such processes it look like:-

```
# jobs

[1]+  Stopped                 sleep 100
[2]   Running                 sleep 100 &
[3]-  Running                 firefox &
```

💡

How to run a new fresh program/process in the background.

```
# sleep &
```

Suppose you run a command Ex: Firefox on the foreground, first you have to stop this `(ctrl +z)` and then run this process in the background `(bg %job_id)` again to see jobs id using `jobs` command

```
[root@web ~]# firefox
^Z
[1]+  Stopped                 firefox
[root@web ~]# jobs
[1]+  Stopped                 firefox
[root@web ~]# bg %1
[1]+ firefox &
[root@web ~]# jobs
[1]+  Running                 firefox &
[root@web ~]#
```

Let's assume, I am running this command: firefox

Program/Command	Signal	Run in the Background	Run on the Foreground
New Fresh command		firefox &	firefox
Existing command	ctrl + z	bg %jobs-id	fg %jobs-id

What is Process States ??

In a multitasking operating system, each CPU (or CPU core) can be working on one process at a time. As a process runs, its immediate requirements for CPU time and resource allocation change. Processes are assigned a state, which changes as circumstances dictate. A process, during its lifecycle, goes through several stages. These stages or states are:

1. `New:` The process is about to be created but not yet created.

2. `Ready:` After the creation of a process, the process enters the ready state i.e., the process is loaded into the main memory and is waiting to get the CPU time for its execution.

3. `Running:` The process is chosen from the ready queue by the CPU for execution.

4. `Sleeping or Wait:` Whenever the process requests access to I/O or needs input from the user or needs access to a critical region, it enters the blocked or waiting state.

5. `Terminated or Stopped:` The process is killed, and the resources allocated to the process are released or deallocated.

Process State Transitions

A process can move between different states in an operating system based on its execution status and resource availability. Here are some examples of how a process can move between different states:

- `New to Ready:` When a process is created, it is in a new state. It moves to the ready state when the operating system has allocated resources to it and it is ready to be executed.

- `Ready to Running:` When the CPU becomes available, the operating system selects a process from the ready queue depending on various scheduling algorithms and moves it to the running state.

- `Running to Waiting:` If a process requests access to I/O or needs input from the user or needs access to a critical region, it enters the blocked or waits state.

- `Waiting to Ready:` Once the I/O operation is completed the process goes to the ready state.

- `Running to Stopped:` If a process has completed execution, it moves to the terminated/stopped state.

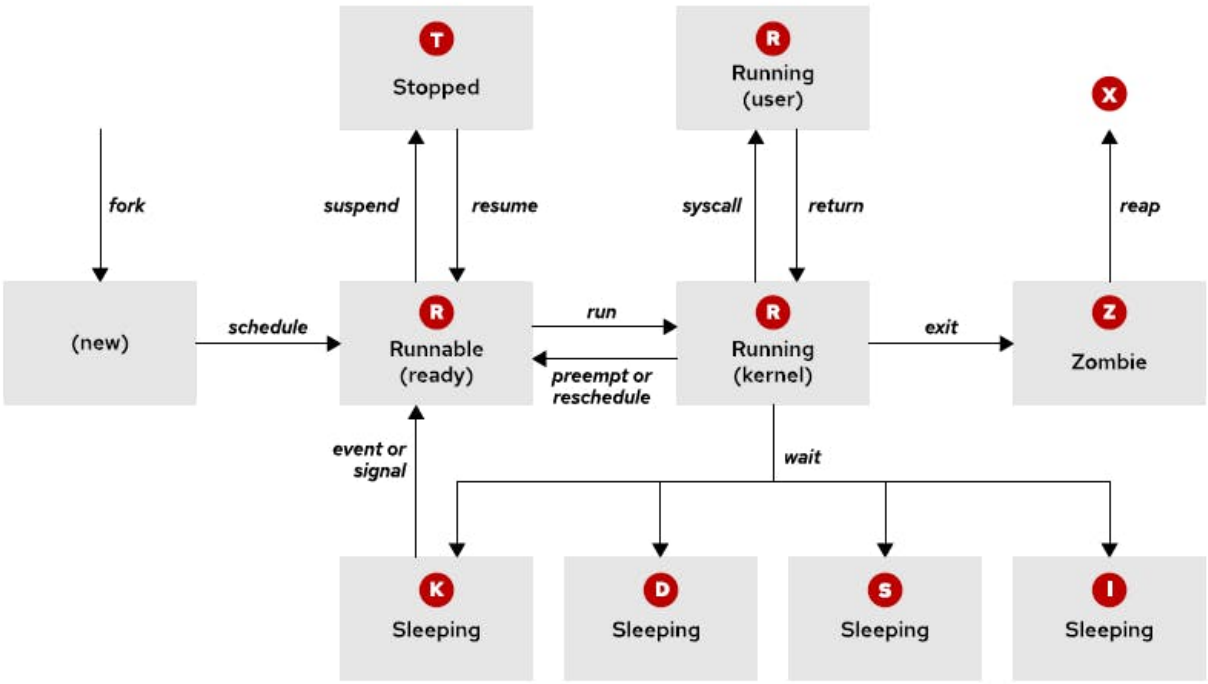


Figure 8.2: Linux process states

Mainly we need to understand the following `5 types` of processes.

Process State	Flags	Description
Running	R	The process is either executing on a CPU or waiting to run.
Sleeping Interruptible	S	The process is waiting for some condition such as a hardware request, system resource access, or signal. It can be awakened by a signal.
Sleeping Uninterruptable	D	This process is also sleeping, but unlike S state, it does not respond to signals. It's used when process interruption might cause an unpredictable device state.
Stopped	T	The process is stopped (suspended), usually by being signaled by a user or another process. It can be resumed by another signal.
Zombie	Z	A child process that has completed execution but still has an entry in the process table to report to its parent process. All resources except for the process identity (PID) are released.

Importance of Process States

We have substantial reasons to understand process states, especially if we are overseeing an application as a monitoring authority.

1. `Performance Analysis:` It helps us figure out if our computer is running smoothly or if something is slowing it down.

2. `Resource Management:` It shows us how our computer's resources (like memory and processing power) are being used.
3. `System Troubleshooting:` If our computer is having problems, understanding process states can help us find out what's going wrong.
4. `Process Optimization:` We can make changes to improve how efficiently our computer runs.
5. `Predicting System Behavior:` It can give us an idea of how our computer might behave under different conditions.

💡

Important Linux commands for understanding process states and parameters include `top` and `ps`, each with numerous options to modify output behavior. Examples include `ps`, `ps -aux`, `ps lax`, and `top`.

`ps`: Reports a snapshot of the current processes on current shell terminal.

```
# ps
    PID TTY          TIME CMD
  37078 pts/0    00:00:00 bash
  40340 pts/0    00:00:00 ps
```

`ps -aux`: Shows all processes for all users.

```
# ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.6 181792 11992 ?        Ss   Feb08   0:18
/usr/lib/systemd/systemd rhgb --switched-root --system --deserialize 31
root           2  0.0  0.0      0     0 ?        S    Feb08   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   Feb08   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   Feb08   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   Feb08   0:00 [netns]
```

`ps lax`: Provides a very detailed and technical information about tasks.

```
# ps lax
F   UID     PID    PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0       1       0  20   0 181792 11992 ep_pol Ss   ?          0:18
/usr/lib/systemd/systemd rhgb --switched-root --system --deserialize 31
1     0       2       0  20   0      0     0 kthrea S    ?          0:00
[kthreadd]
1     0       3       2   0 -20      0     0 rescue I<   ?          0:00 [rcu_gp]
```

`uptime`: Shows the current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes.

```
# uptime
19:29:13 up 1 day, 19:50,  5 users,  load average: 0.01, 0.02, 0.05
```

💡

How to calculate Actual load average of the system in (1 Minutes, 5 Minutes & 15 Minutes)?

To calculate the load average of the system we have to require two values.

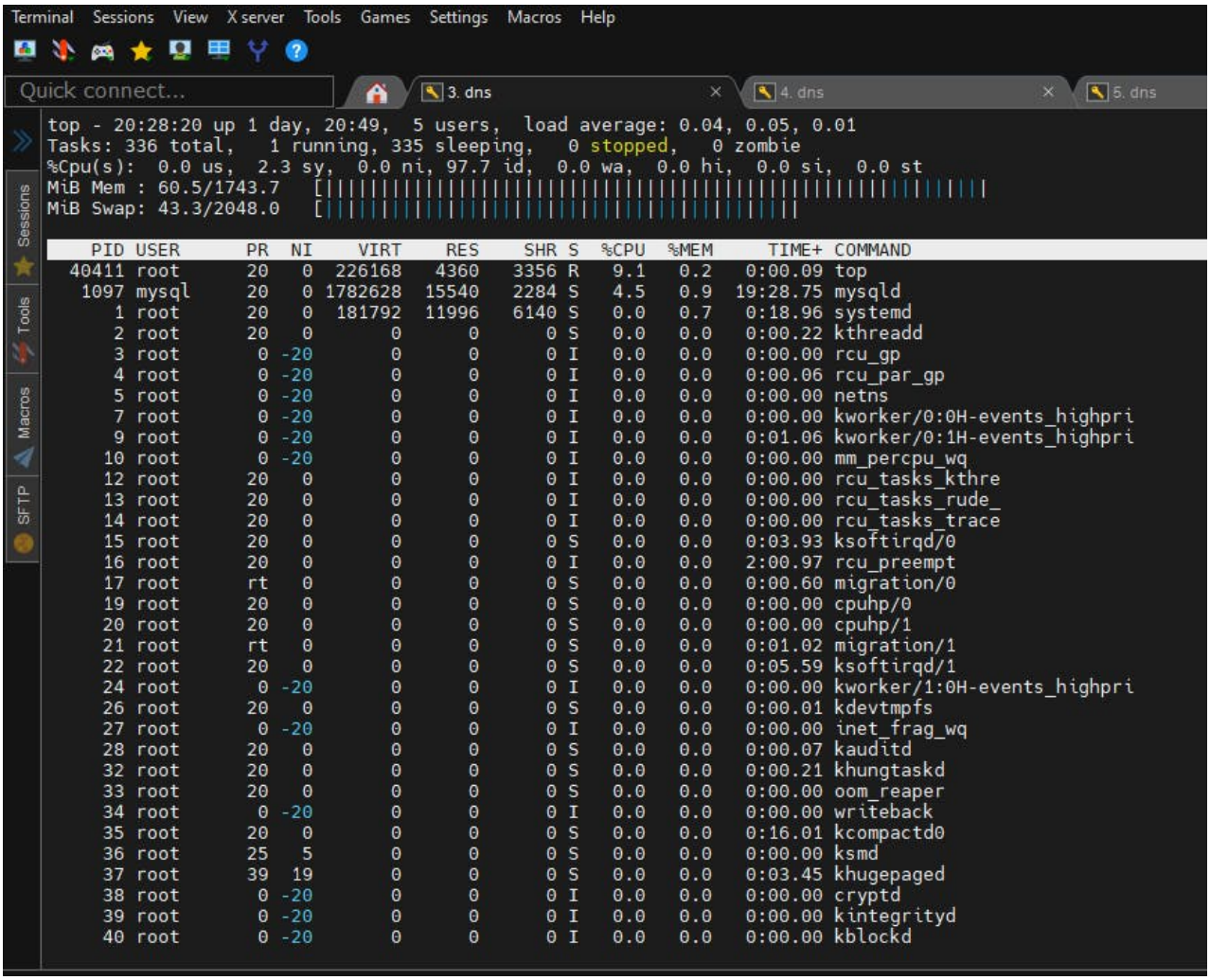
1. load average (One minutes, Five Minutes, Fifteen Minutes), which you can get from command "uptime"
2. Number of CPU, which you can get from command "lscpu"

Calculation Formula: Load Average / Number of CPU

`top`: Provides a dynamic real-time view of the running system. It can display system summary information and a list of processes currently being managed by the kernel.

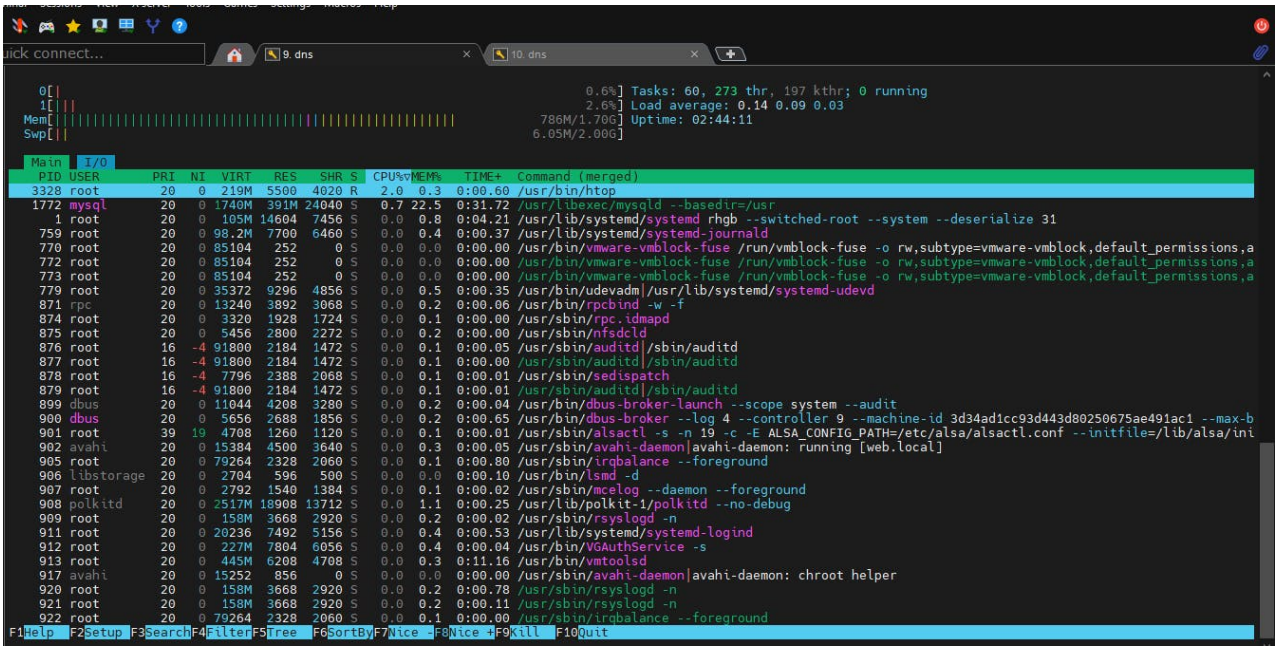
```
# top
```

`top` is a dynamic real-time process monitoring tool with lots of options, where `ps` is a static process monitoring tool at a certain time only.




```
# htop
```

The ‘htop’ command is similar to ‘top’, but it provides a more user-friendly and colorful display. It also supports mouse operations and has options for customization.



💡

What the meaning of parameters ?

Options	Meaning
F	Process flags (e.g., forked, traced, etc.).
UID	User ID of the process owner.
PID	Process ID (unique identifier).
PPID	Parent process ID (ID of the process that spawned this one).
PRI	Priority of the process (scheduling priority).
NI	Nice value (user-defined priority adjustment).
VSZ	Virtual memory size (total memory used by the process).
RSS	Resident set size (actual physical memory used by the process).
WCHAN	Waiting channel (function where the process is waiting).
STAT	Process status (e.g., running, sleeping, zombie, etc.).
TTY	Controlling terminal (if any).
TIME	Cumulative CPU time used by the process.
COMMAND	Command or program associated with the process.
USER	User who owns the process.

Options	Meaning
`%CPU`	Percentage of CPU usage by the process.
`%MEM`	Percentage of memory usage by the process.
`START`	Start time of the process.

Remember to use `man <command>` (replace `<command>` with the command name) to get more details about each command and its options.

```
# man top
# man ps
```

Important keyboard options with `top` command

Here are the uses of each option in the `top` command, each in a separate row:

```
# top
```

Options	Uses
Z	Global: colors
B	Global: bold
E,e	Global: summary/task memory scale
l	Toggle: load avg
t	Toggle: task/cpu
m	Toggle: memory
l	Toggle: Irix mode
0	Toggle: zeros
1,2,3	Toggle: cpu/numa views
4	Toggle: cpus two abreast
f,F	Fields: add/remove/order/sort
X	Fields: increase fixed-width
L,&	Locate: find/again
<,>	Move sort column: left/right
R	Toggle: Sort
H	Toggle: Threads
J	Toggle: Num justify

Options	Uses
C	Toggle: Coordinates
c	Toggle: Cmd name/line
i	Toggle: Idle
S	Toggle: Time
j	Toggle: Str justify
x,y	Toggle highlights: sort field; running tasks
z	Toggle: color/mono
b	Toggle: bold/reverse (only if ‘x’ or ‘y’)
u,U	Filter by: effective/any user
o,O	Filter by: other criteria
n,#	Set: max tasks displayed
^O	Show: other filter(s)
V	Toggle: forest view
v	Toggle: hide/show forest view children
k	Manipulate tasks: kill
r	Manipulate tasks: renice
d or s	Set update interval
W,Y,!	Write config file; Inspect other output; Combine Cpus
q	Quit

What is Process Signals, How Signals Works ??

Process signaling is a method used in operating systems to communicate between processes. It’s like a notification system where one process sends a signal, and another process receives it.

Here are some uses of process signaling:

1. **Interrupt a Process**: If a process is running, a signal can be sent to stop it immediately.
2. **Resume a Process**: A stopped process can be resumed using a signal.

3. **Terminate a Process**: If a process needs to be ended, a signal can be sent to terminate it.

4. **Handle Errors**: If a process encounters an error, it can send a signal to indicate that something went wrong.

💡

As we discussed earlier, a process can be in either an interruptible or non-interruptible sleeping state. Applications can internally send signals to manage the process cycle for work completion. Alternatively, a system administrator can also directly send signals to a specific process to manage it based on certain events using the `kill` and `pkill` commands.

Fundamental process management signals

Signal Number	Signal Name	Description
1	HUP (Hangup)	Ends the controlling process of a terminal. Also asks for process re-initialization without ending it.
2	INT (Keyboard interrupt)	Stops the program. It can be blocked or handled. Triggered by pressing Ctrl+c.
3	QUIT (Keyboard quit)	Similar to INT, but also creates a process dump at termination. Triggered by pressing Ctrl+\.
9	KILL (Kill, unblockable)	Abruptly stops the program. It cannot be blocked, ignored, or handled.
15	TERM (Terminate)	Stops the program. Unlike KILL, it can be blocked, ignored, or handled. It's a polite way to ask a program to end, allowing it to finish important tasks and clean up. By default, `kill` sends a **SIGTERM** signal, which politely asks the process to terminate
18	CONT (Continue)	Sent to a process to resume if stopped. It cannot be blocked. Even if handled, it always resumes the process.
19	STOP (Stop, unblockable)	Pauses the process. It cannot be blocked or handled.
20	TSTP (Keyboard stop)	Pauses the process. Unlike STOP, it can be blocked, ignored, or handled. Triggered by pressing Ctrl+z.

💡

As per the system administrator's need, he can assign process signals using `kill` & `pkill` commands, and Process ID can get from linux commands EX:- `pgrep`, `ps`, `top`

1. **Scenario**: You want to stop a program nicely (Not forcefully).
    - **Question**: What signal do you use to ask a program to stop, but let it finish up important tasks first?
    - **Answer**: Use the `SIGTERM` signal.

```
# pgrep firefox
# pgrep httpd
# pgrep mysql
# ps -aux | grep -e firefox -e httpd -e mysql -e username
# top

# kill -l
# kill -15 <process-id>
# kill -SIGTERM <process-id>
```

1. **Scenario**: You need to stop a program right away because it's causing problems.
    - **Question**: What signal do you use to force a program to stop immediately?
    - **Answer**: Use the `SIGKILL` signal.

```
# kill -l
# kill -9 <process-id>
# kill -SIGKILL <process-id>
```

1. **Scenario**: You want to pause a program for a while.
    - **Question**: What signal do you use to pause a program and then start it again later?
    - **Answer**: Use the `SIGTSTP` signal to pause and the `SIGCOUNT` signal to start again.

```
# kill -l
# kill -20 <process-id>
# kill -18 <process-id>
# kill -SIGTSTP <process-id>
# kill -SIGCOUNT <process-id>
```

1. **Scenario**: You want a program to read its configuration settings again without stopping it.
    - **Question**: What signal do you use to ask a program to read its settings again?
    - **Answer**: Use the `SIGKILL` signal.

```
# kill -l
# kill -1 <process-id>
# kill -SIGHUP <process-id>
```

1. Scenario: You want to prevent an anonymous user from using the shell immediately.
    - Question: What steps or measures can you take to quickly disable shell access for an anonymous user?
    - Answer: Using `SIGKILL` signal

```
# kill -l
# kill -9 <process-id>
# kill -SIGKILL <process-id>
```

💡

Improve the management of processes through `PKILL` signals.

Here are some real-world scenarios where you can use the `pkill` command effectively:

Scenario	Command	Description
Kill a process by name	`pkill firefox`	This command will kill all running processes named ‘firefox’.
Send a different signal	`pkill --signal SIGKILL gedit`	This command sends the SIGKILL signal to all ‘gedit’ processes.
Match full command line	`pkill -f "pinggoogle.com"`	This command kills the ‘ping google.com’ command. The `-f` option matches the complete command line.
Case insensitive match	`pkill -i firefox`	This command will kill all running processes named ‘firefox’, ignoring case.
Kill processes by user	`pkill -u mark`	This command kills all processes being run by the user ‘mark’.
Kill oldest process	`pkill -o firefox`	This command kills the oldest ‘firefox’ process.
Kill newest process	`pkill -n firefox`	This command kills the newest ‘firefox’ process.
Kill processes by group	`pkill -g 1000`	This command kills all processes in the group with ID ‘1000’.
Kill processes by session	`pkill -s 1`	This command kills all processes in the session with ID ‘1’.

Scenario	Command	Description
Kill processes by terminal	`pkill -t pts/1`	This command kills all processes on the terminal ‘pts/1’.

```
# pkill --help
```

💡

Which tool do we prefer between ‘kill’ and ‘pkill’?

- Use `kill` when you know the PID.
- Use `pkill` when you want to terminate processes by their names and patterns.

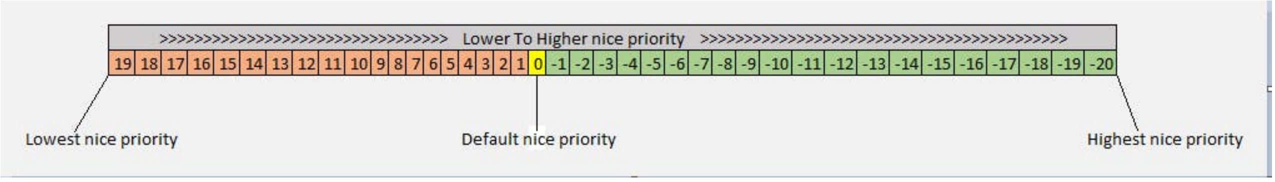
What is Process Priority, How to Modify Processes Priority??

```
# ps lax
```

```
F   UID     PID    PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0       1       0  20   0 106364 15912 ep_pol Ss   ?          0:02 /usr/lib/systemd/systemd rh
gb --switched-root --system --deserialize 31
1     0       2       0  20   0      0     0 kthrea S    ?          0:00 [kthreadd]
1     0       3       2   0 -20      0     0 rescue I<   ?          0:00 [rcu_gp]
1     0       4       2   0 -20      0     0 rescue I<   ?          0:00 [rcu_par_gp]
1     0       5       2   0 -20      0     0 rescue I<   ?          0:00 [netns]
1     0       6       2  20   0      0     0 worker I    ?          0:00 [kworker/0:0-events]
1     0       7       2   0 -20      0     0 worker I<   ?          0:00 [kworker/0:0H-events_highpr
i]
1     0       8       2  20   0      0     0 worker I    ?          0:00 [kworker/u256:0-events_unbo
und]
1     0       9       2   0 -20      0     0 worker I<   ?          0:00 [kworker/0:1H-events_highpr
i]
1     0      10       2   0 -20      0     0 rescue I<   ?          0:00 [mm_percpu_wq]
5     0      11       2  20   0      0     0 worker I    ?          0:01 [kworker/u256:1-events_unbo
und]
1     0      12       2  20   0      0     0 rcu_ta I    ?          0:00 [rcu_tasks_kthre]
1     0      13       2  20   0      0     0 rcu_ta I    ?          0:00 [rcu_tasks_rude_]
1     0      14       2  20   0      0     0 rcu_ta I    ?          0:00 [rcu_tasks_trace]
1     0      15       2  20   0      0     0 smpboo S    ?          0:00 [ksoftirqd/0]
1     0      16       2  20   0      0     0 rcu_gp I    ?          0:00 [rcu_preempt]
1     0      17       2 -100   -      0     0 smpboo S    ?          0:00 [migration/0]
1     0      18       2  20   0      0     0 worker I    ?          0:00 [kworker/0:1-events]
1     0      19       2  20   0      0     0 smpboo S    ?          0:00 [cpuhp/0]
1     0      20       2  20   0      0     0 smpboo S    ?          0:00 [cpuhp/1]
1     0      21       2 -100   -      0     0 smpboo S    ?          0:00 [migration/1]
1     0      22       2  20   0      0     0 smpboo S    ?          0:00 [ksoftirqd/1]
1     0      23       2  20   0      0     0 worker I    ?          0:00 [kworker/1:0-events]
1     0      24       2   0 -20      0     0 worker I<   ?          0:00 [kworker/1:0H-events_highpr
i]
1     0      26       2  20   0      0     0 worker I    ?          0:00 [kworker/u256:2-events_unbo
und]
5     0      27       2  20   0      0     0 devtmp S    ?          0:00 [kdevtmpfs]
1     0      28       2   0 -20      0     0 rescue I<   ?          0:00 [inet_frag_wq]
1     0      29       2  20   0      0     0 kaudit S    ?          0:00 [kauditd]
```

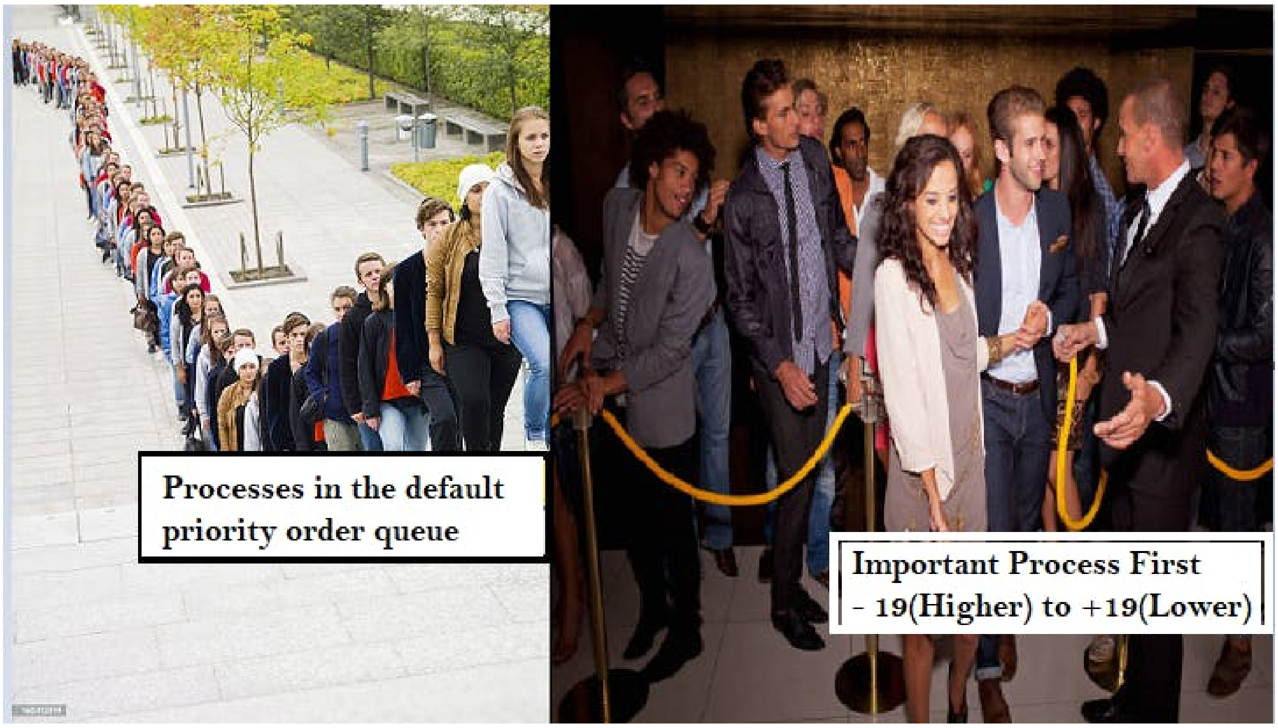
Process Priority is a characteristic of a process that determines how much CPU time it is allocated for execution. The `NI` column displays the niceness of processes, indicating their priority.It’s important because it helps the operating system manage resources efficiently. If all processes were given equal priority, a long-running or resource-intensive

process could monopolize the CPU, causing other processes to slow down or even halt. By assigning different priorities, the operating system can ensure that important processes get the resources they need, while less important processes are made to wait.



The `nice`**value** is a way to influence process priority in Unix-like operating systems. It's a value that can be assigned to a process to either increase or decrease its priority. The `nice` value ranges `from -20 (highest priority) to +19 (lowest priority)`. By default, the Nice value is `zero`, which gives the process a neutral priority.

- The `nice` command in Unix-like systems is used to start a process with a certain nice value.
- while the `renice` command is used to change the nice value of an already running process.
- The lower the nice value (i.e., more negative), the higher the priority of the process.
- The higher nice value (i.e., more positive) gives the process a lower priority.



So, the Nice and renice values are directly connected with process priority because they are tools that allow users to influence the scheduling priority of processes. This can be useful in a variety of situations, such as ensuring that a critical process gets the CPU time it needs or preventing a resource-intensive process from monopolizing the CPU.

What is NICE & RENICE value in the process Priority?

In Linux, the `nice` and `renice` commands are used to influence the scheduling priority of processes. Here's a simple explanation:

> `nice`: This command is used `when you're starting a new process and you want to set its priority`. The `nice` value can range `from -20 (highest priority) to 19 (lowest priority)`.

💡

For example, if you want to start a process with a lower & higher priority, you can use the `nice` command like this:

```
# nice -n 10 command
# nice -n -20 firefox
# nice -n 1 command
```

Above command will start the `command` with a `nice` value of 10, -20, 1 , which is lower & Higher priority.

> `renice`: This command is used when you want to change the priority of an already running process. For example, if you have a process with process ID (PID) 15784 and you want to lower its priority, you can use the `renice` command like this:

```
# renice -n 15 -p 15784
# renice -n -19 -p 45125
```

This will change the `nice` value of the process with PID 15784, 45125 to 15, -19 which is a lower & higher nice priority.

💡

Remember, only the superuser (root) can increase the priority (set a negative nice value). Normal users can only decrease the priority (set a positive nice value) or keep it the same. Normal users can only affect processes they **own** or have permission to modify

---

Thank you!