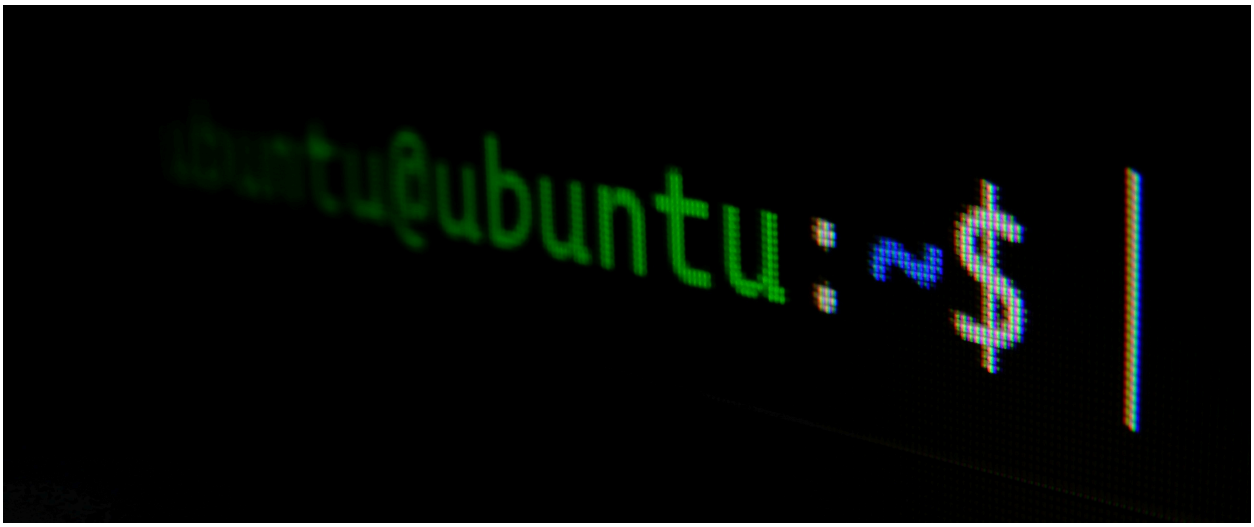


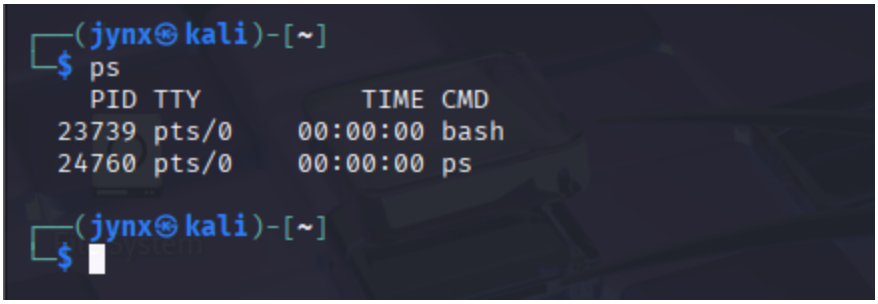
CHAPTER 6 - PROCESS MANAGEMENT

A **process** is essentially a program/thread that's running and using resources [CPU, GPU, RAM etc.], an efficient Linux administrator, digital forensic investigator or ***hackers*** need to have a particularly relevant and profound understanding of processes and how they work.

▼ Viewing Processes.

The ***'ps'*** command is used to observe and analyze which background processes are currently running on the system.

example:



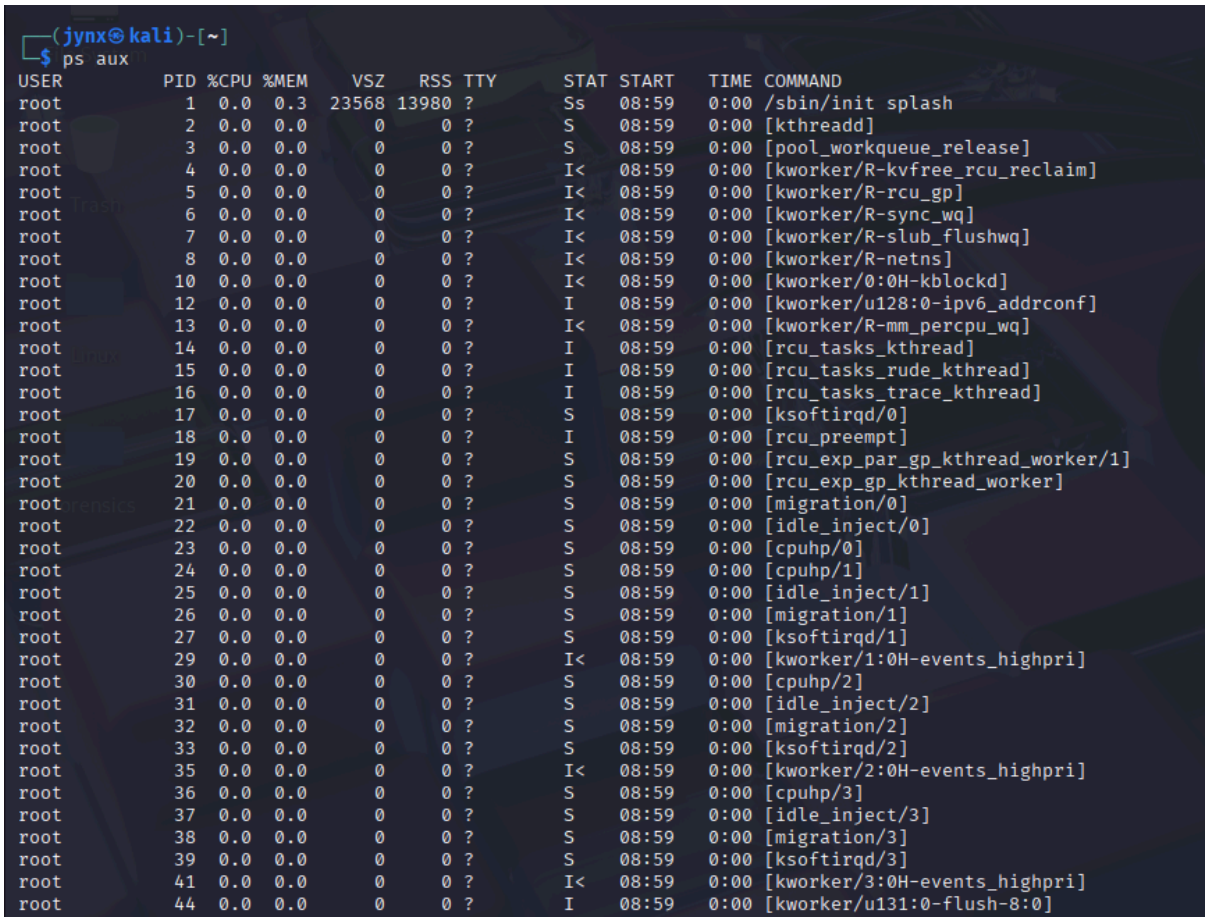
The Linux kernel assigns unique **Process IDs** (PIDs) sequentially to each process as it's created. When managing processes, PIDs are more important

than process names for identification.
Running
`ps` alone provides minimal information—only processes started by the current user on the current terminal (typically just the bash shell and the `ps` command itself).

For complete system visibility, use; ***`ps aux`*** - this command displays all processes running on the system for all users, including:

- System background processes
- Processes from other users
- Detailed process information

Note: The options `aux` are used ***without*** a dash (-) and must be lowercase, as Linux is case-sensitive.



```
┌──(jynx㉿kali)-[~]
└─$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.3  23568 13980 ?        Ss   08:59   0:00 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    08:59   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    08:59   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/R-kvfree_rcu_reclaim]
root           5  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/R-rcu_gp]
root           6  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/R-sync_wq]
root           7  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/R-slub_flushwq]
root           8  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/R-netns]
root          10  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/0:0H-kblockd]
root          12  0.0  0.0      0     0 ?        I    08:59   0:00 [kworker/u128:0-ipv6_addrconf]
root          13  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/R-mm_percpu_wq]
root          14  0.0  0.0      0     0 ?        I    08:59   0:00 [rcu_tasks_kthread]
root          15  0.0  0.0      0     0 ?        I    08:59   0:00 [rcu_tasks_rude_kthread]
root          16  0.0  0.0      0     0 ?        I    08:59   0:00 [rcu_tasks_trace_kthread]
root          17  0.0  0.0      0     0 ?        S    08:59   0:00 [ksoftirqd/0]
root          18  0.0  0.0      0     0 ?        I    08:59   0:00 [rcu_preempt]
root          19  0.0  0.0      0     0 ?        S    08:59   0:00 [rcu_exp_par_gp_kthread_worker/1]
root          20  0.0  0.0      0     0 ?        S    08:59   0:00 [rcu_exp_gp_kthread_worker]
root          21  0.0  0.0      0     0 ?        S    08:59   0:00 [migration/0]
root          22  0.0  0.0      0     0 ?        S    08:59   0:00 [idle_inject/0]
root          23  0.0  0.0      0     0 ?        S    08:59   0:00 [cpuhp/0]
root          24  0.0  0.0      0     0 ?        S    08:59   0:00 [cpuhp/1]
root          25  0.0  0.0      0     0 ?        S    08:59   0:00 [idle_inject/1]
root          26  0.0  0.0      0     0 ?        S    08:59   0:00 [migration/1]
root          27  0.0  0.0      0     0 ?        S    08:59   0:00 [ksoftirqd/1]
root          29  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/1:0H-events_highpri]
root          30  0.0  0.0      0     0 ?        S    08:59   0:00 [cpuhp/2]
root          31  0.0  0.0      0     0 ?        S    08:59   0:00 [idle_inject/2]
root          32  0.0  0.0      0     0 ?        S    08:59   0:00 [migration/2]
root          33  0.0  0.0      0     0 ?        S    08:59   0:00 [ksoftirqd/2]
root          35  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/2:0H-events_highpri]
root          36  0.0  0.0      0     0 ?        S    08:59   0:00 [cpuhp/3]
root          37  0.0  0.0      0     0 ?        S    08:59   0:00 [idle_inject/3]
root          38  0.0  0.0      0     0 ?        S    08:59   0:00 [migration/3]
root          39  0.0  0.0      0     0 ?        S    08:59   0:00 [ksoftirqd/3]
root          41  0.0  0.0      0     0 ?        I<   08:59   0:00 [kworker/3:0H-events_highpri]
root          44  0.0  0.0      0     0 ?        I    08:59   0:00 [kworker/u131:0-flush-8:0]
```

COLUMNS (understanding) in the output:

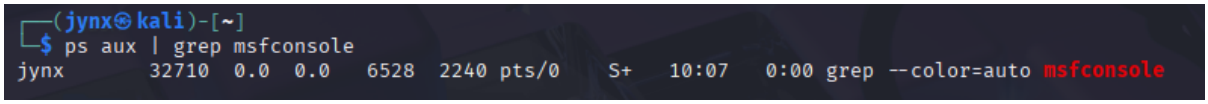
1. ***USER*** - The user who invoked the process.
2. ***PID*** - The process ID [unique].
3. ***%CPU*** - Percentage of CPU being utilized by the process.
4. ***%MEM*** - Percentage of memory being utilized by the process.
5. ***COMMAND*** - The name of the command that started the process.

→ To perform any manipulation commands on a particular process, we must specify its ***PID***.

▼ Filtering by Process Name.

The ***grep*** command can be used to filter out processes on the basis of name(s).

example:

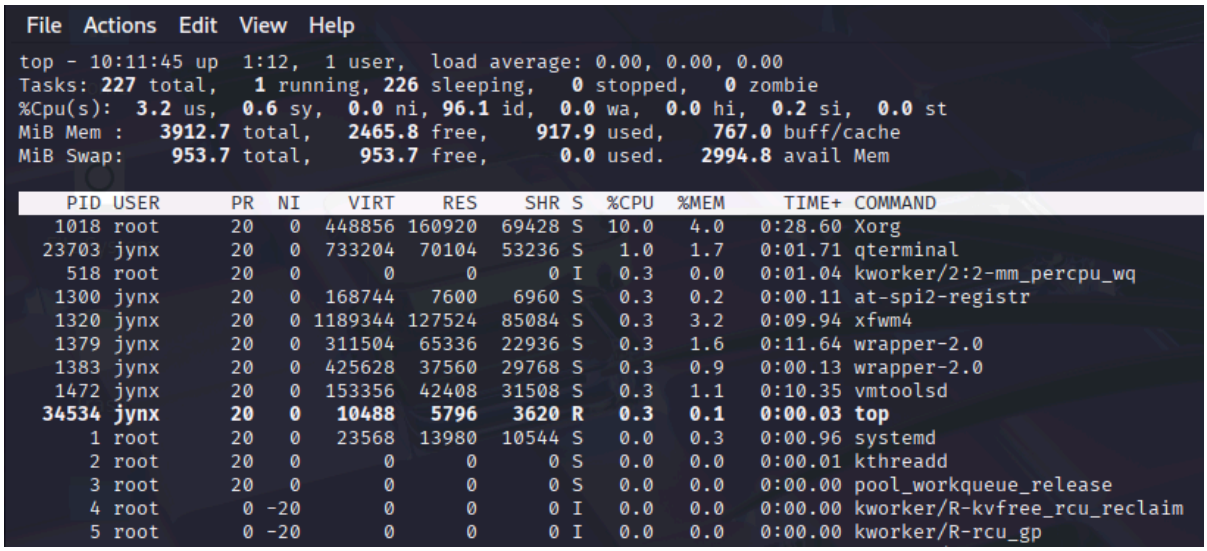


▼ *'top'* command.

By default, the `ps` command displays processes in order they were started (chronologically), essentially by PIDs. If you wish to see the processes consuming most system resources use 'top' command.

> `top`

example:



→ While you have top running, pressing the `H` or `?` key will bring up a list of interactive commands, and pressing `Q` will quit top.

▼ Managing Processes.

The `nice` command is used to change the priority of a process to the kernel.

The values for nice range from –20 to +19, with zero being the default value.

-20 - Most likely to receive priority

0 - Default priority

-19 - Least likely to receive priority

→ You can also alter the priority using `renice` command.

→ The `nice` command requires to increment the value, on the other hand the `renice` command wants an absolute value.

example:

```
nice -n -15 /bin/slowproces123
```

This command above, would increment by the `nice` value by 15 and prioritize process- slowprocess123.

```
nice -n 10 /bin/slowproces123
```

This command above, would decrement by the `nice` value by 10 and de-prioritize the process- slowprocess123.

▼ Killing Processes.

Processes can become problematic when they:

- Consume excessive system resources
- Display abnormal behavior
- Become unresponsive or freeze

Such problematic processes waste valuable system resources that could be allocated to functional processes, they are called ***zombie processes***.

When you identify a problematic process, use the `kill` command to terminate it. The kill command offers multiple termination methods, each identified by a specific signal number that determines how the process should be stopped.

Syntax: `kill -signal PID` , *signal switch is optional.*

→ Common KILL Signals:

Signal Name	Number	Description
SIGHUP	1	**Hangup Signal** - Terminates the process and immediately restarts it using the same Process ID. Commonly used to reload configuration files.
SIGINT	2	**Interrupt Signal** - A gentle termination request that allows the process to clean up before exiting. Similar to pressing Ctrl+C. May not work if the process is unresponsive.
SIGQUIT	3	**Quit Signal** - Forces process termination and creates a core dump file in the current directory. The core dump contains the process's memory state for debugging purposes.
SIGTERM	15	**Termination Signal** - The standard and default kill signal. Politely requests the process to terminate, allowing it to perform cleanup operations before shutting down.
SIGKILL	9	**Kill Signal** - The most forceful termination method. Immediately destroys the process without allowing

		cleanup. Use as a last resort when other signals fail.

EXAMPLE:

→ To restart a process with the HUP signal for a process with PID 6996, enter the -1 option with kill:

> `kill -1 6996`

▼ Running Processes in the Background.

In Linux systems, all user interactions occur through a shell environment, regardless of whether you're using the command line interface or graphical desktop. Even GUI applications ultimately execute their commands through the underlying shell.

When you run a command, the shell operates synchronously by default - it waits for the current command to finish executing before displaying the next command prompt and accepting new input.

Sometimes you need to run a process while continuing to use the terminal for other tasks. For example, if you start a file download with `wget`:

```
wget https://example.com/largefile.zip
```

The terminal becomes occupied with the download process, showing progress updates but preventing you from entering new commands until the download completes.

Instead of opening multiple terminals or waiting for the process to finish, you can run processes in the background. Background processes continue running independently while freeing up the terminal for other commands.

To run a command in the background, append an ampersand (&) to the end:

```
wget https://example.com/largefile.zip &
```

Now the download runs in the background, and you immediately get a new command prompt to execute other tasks while the file continues downloading.

Benefits

This approach is particularly useful for:

- Long-running downloads or file transfers
- System monitoring tools
- Any process that takes significant time to complete
- Maximizing terminal efficiency without opening multiple windows

---

If you want to move a process running in the background to the ***foreground***, you can
use the fg (foreground) command. The
`fg` command requires the PID of the process you
want to return to the foreground.

> `fg 6996`

▼ Scheduling Processes

System administrators and developers frequently need to automate tasks at specific times. Common scenarios include:

- Running system backups during off-peak hours
- Performing regular maintenance tasks
- Executing monitoring scripts at intervals
- Automating data processing jobs

Scheduling Tools

Linux provides two primary scheduling mechanisms:

at Command

Used for **one-time** scheduled tasks. The `at` daemon runs commands at a specified future time.

cron/crond

Better suited for **recurring** tasks that need to run daily, weekly, or monthly.

The `at` command accepts various time formats for flexible scheduling:

Time Format	Meaning
`at 2:30pm`	Run at 2:30 PM today
`at 2:30pm July 15`	Run at 2:30 PM on July 15
`at midnight`	Run at midnight today
`at midnight July 15`	Run at midnight on July 15
`at tomorrow`	Run tomorrow at current time
`at now + 30 minutes`	Run in 30 minutes
`at now + 2 hours`	Run in 2 hours
`at now + 3 days`	Run in 3 days
`at now + 2 weeks`	Run in 2 weeks
`at 2:30pm 07/15/2025`	Run at 2:30 PM on July 15, 2025

Example Usage

To schedule a backup script to run at 3:00 AM:

```
$ at 3:00am
at> /home/user/backup_script.sh
at> <Ctrl+D>
```

The `at` command enters interactive mode where you specify the commands to execute, then press Ctrl+D to save the scheduled task.