Module 6 - Processes and performance

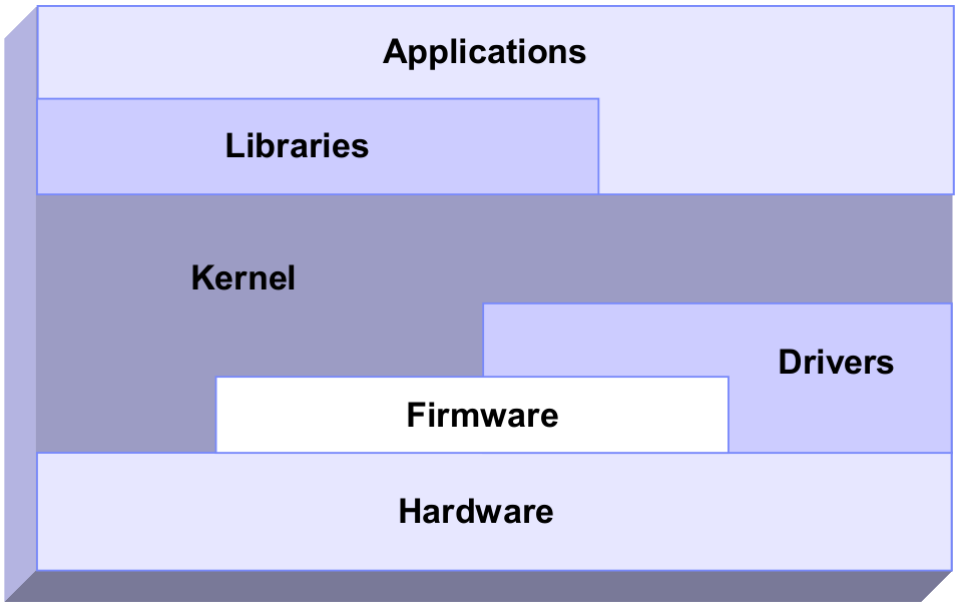
By Juan Medina

jmedina@collin.edu

COLLIN COLLEGE

System Performance

Linux performance tuning & analysis is a challenging task that requires an in-depth understanding of the hardware, operating system, and applications. Performance is affected by multiple factors.

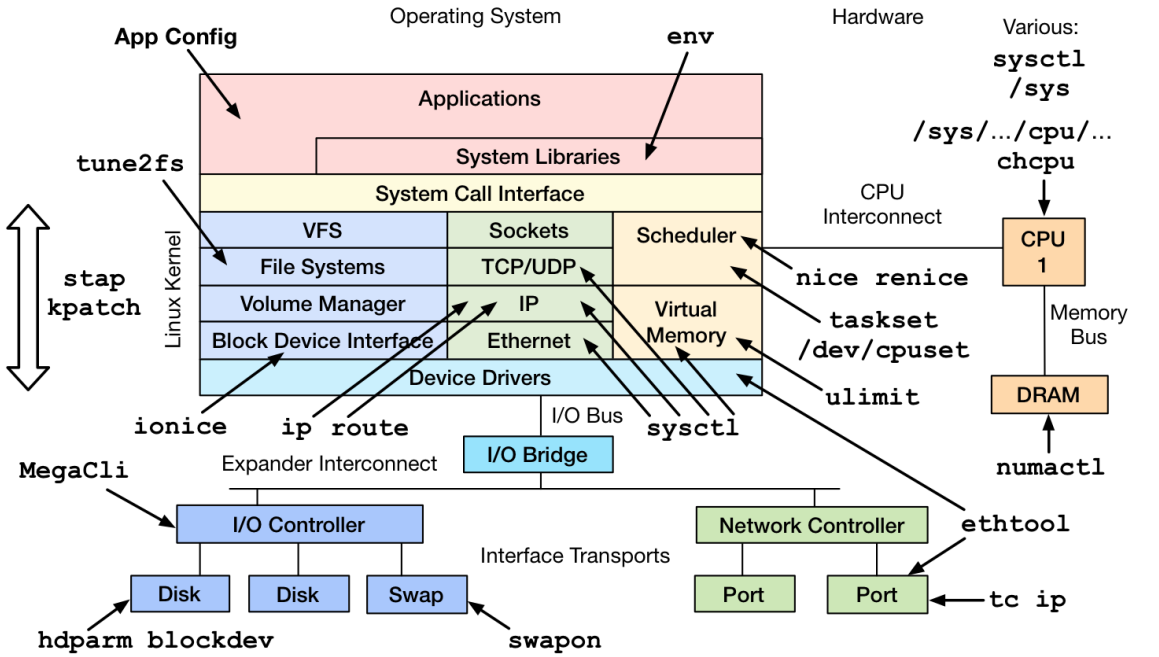


You could tune your system for weeks in vain if there is an upgrade on a disk driver that changes your scenario, keep in mind the whole picture of systems performance. Understanding the way an operating system manages the system resources helps us understand what subsystems we need to tune in any application scenario.

Linux process management

Process management is one of the most important roles of any operating system. Effective process management enables an application to operate steadily and effectively. Linux process management includes process scheduling, interrupts handling, signaling, process prioritization, process switching, process state, process memory, and so on. Understanding how the performance of your system works is an essential job of a good system administrator, whenever something goes wrong or slow everybody will turn at you looking for solutions and answers!. There is a huge amount of tools to debug system performance in Linux.

Linux Performance Tuning Tools



http://www.brendangregg.com/linuxperf.html 2016

What is a process?

Modern operating systems are usually multitasking, meaning that they create the illusion of doing more than one thing at once by rapidly switching from one executing program to another. The Linux kernel manages this through the use of processes waiting for their turn at the CPU. Sometimes a computer will become slow or an application will stop responding because of the things that affect these processes.

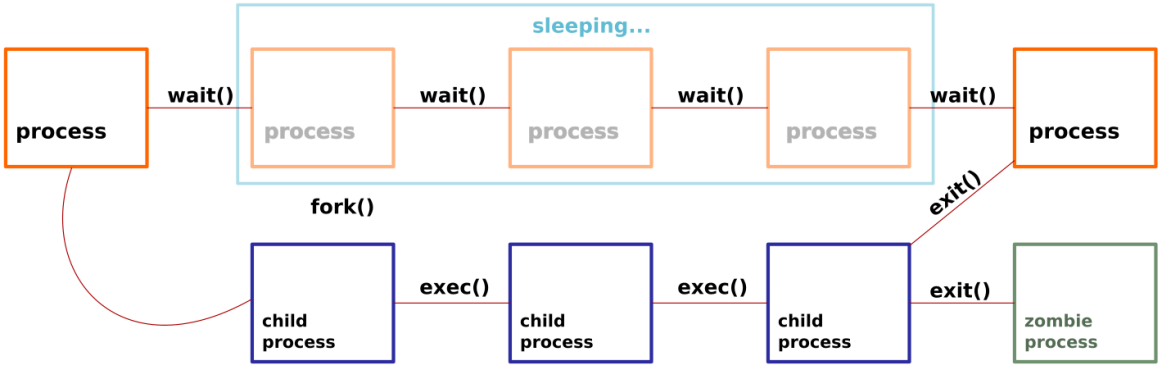
A process is a running instance of a launched, executable program. A process consists of:

- An address space of allocated memory.
- Security properties including ownership credentials and privileges
- One or more execution threads of program code
- Process state

The environment of a process includes:

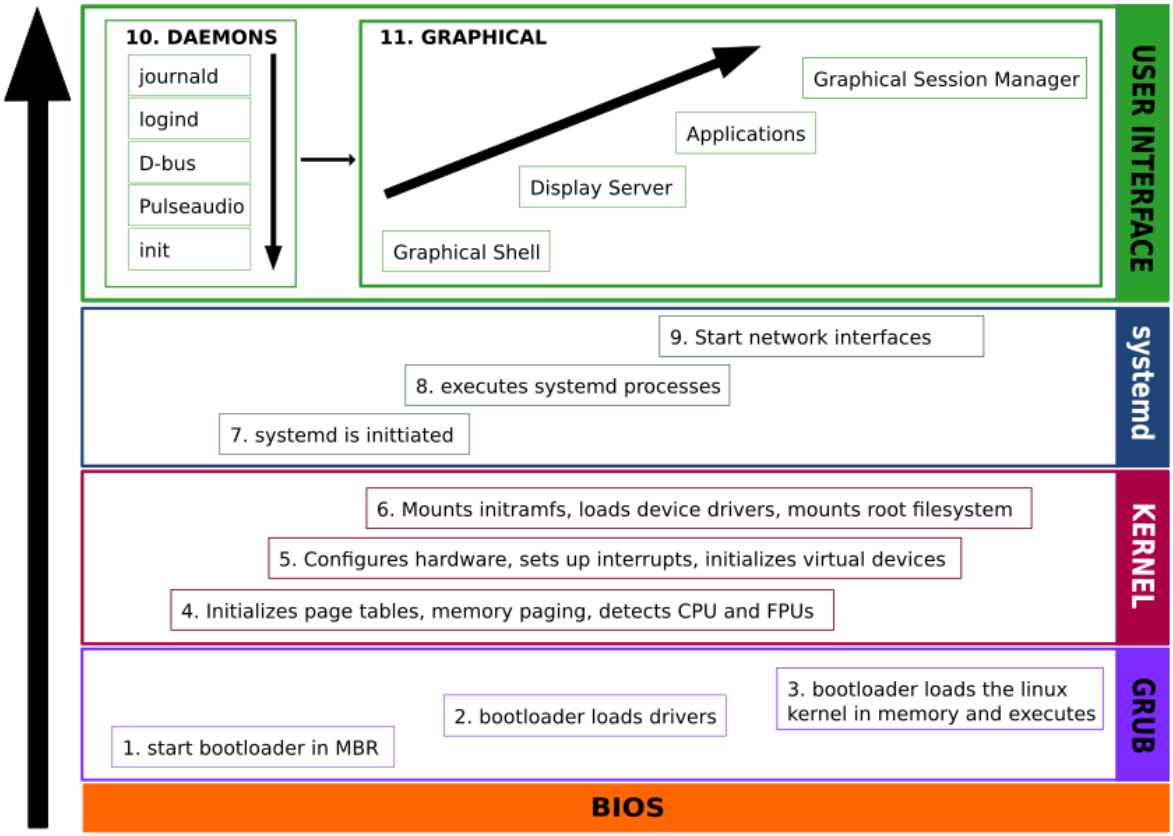
- Local and global variables
- A current scheduling context
- Allocated system resources, such as file descriptors and network ports

An existing (parent) process duplicates its own address space (fork) to create a new (child) process structure. Every new process is assigned a unique process ID (PID) for tracking and security. The PID and the parent's process ID (PPID) are elements of the new process environment. Any process may create a child process.



Through the fork routine, a child process inherits security identities, previous and current file descriptors, port and resource privileges, environment variables, and program code. A child process may then exec its own program code. Normally, a parent process sleeps while the child process runs, setting a request (wait) to be signaled when the child completes. Upon exit, the child process has already closed or discarded its resources and environment. The only remaining resource, called a zombie, is an entry in the process table. The parent, signaled awake when the child exited, cleans the process table of the child's entry, thus freeing the last resource of the child process. The parent process then continues with its own program code execution.

Linux start-up process



The BIOS of course is the first thing that runs when you turn on a computer and is this little chip with just enough embedded code to start the hardware and load the Master Boot Record (MBR).

The MBR is located in the boot sector in your hard drive and has the information about the disk partitions, filesystems and a little program called the bootloader that will help to load the operative system.

The bootloader will load the basic hardware drives and load the Linux Kernel into the memory.

The Kernel prepares the rest of the hardware, loads drivers, mounts the Linux filesystems and runs the initial services (init or systemd).

The systemd daemon runs all the pre-defined processes and network configurations and continues to run until the user interface is loaded by running all the needed daemons for sound, bluetooth, etc etc and run the display manager and applications.

Viewing Processes

The most commonly used command to view processes is ps. The ps program has a lot of options, but in it simplest form it is used like this:

```
[jmedinar@localhost ~]$ ps
    PID TTY          TIME CMD
 129753 pts/0    00:00:05 bash
 381649 pts/0    00:00:00 ps
```

The result in this example lists two processes, which are `bash` and `ps` respectively, without options `ps` will only list processes associated with the current terminal session. TTY is short for "Teletype," and refers to the controlling terminal for the process. The TIME field is the amount of CPU time consumed by the process. As we can see, neither process makes the computer work very hard.

The ps command is used for listing current processes. It can provide detailed process information, including:

- User identification (UID), which determines process privileges
- Unique process identification (PID)
- CPU and real time already expended
- How much memory the process has allocated in various locations
- The location of process stdout, known as the controlling terminal
- The current process state

If we add the flag `x` rhe `ps` command will show all of our processes regardless of what terminal they are controlled by.

```
[jmedinar@localhost ~]$ ps x | head
    PID TTY      STAT   TIME COMMAND
  27598 ?        Ss     0:02 /usr/lib/systemd/systemd --user
  27601 ?        S      0:00 (sd-pam)
  27698 ?        Sl     0:13 /usr/bin/gnome-keyring-daemon --
daemonize --login
  27753 tty2     Ssl+   0:00 /usr/libexec/gdm-wayland-session
/usr/bin/gnome-session
  27759 ?        Ss     0:00 /usr/bin/dbus-broker-launch --scope
user
```

The `?` in the TTY column indicates that there is no controlling terminal for that process.

Perhaps the most common set of flag combinations for the `ps` command are:

flags `aux` to display all processes including processes without a controlling terminal

```
[jmedinar@localhost ~]$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
root         4  0.0  0.0      0     0 ?        S<   Oct22   0:01
[kworker/0:0H]
root         6  0.0  0.0      0     0 ?        S    Oct22   0:01
[ksoftirqd/0]
root         7  0.0  0.0      0     0 ?        S    Oct22   0:00
[migration/0]
root         8  0.0  0.0      0     0 ?        S    Oct22   0:00
[rcu_bh]
root         9  0.0  0.0      0     0 ?        S    Oct22   3:30
[rcu_sched]
root        10  0.0  0.0      0     0 ?        S<   Oct22   0:00
[lru-add-drain]
root        11  0.0  0.0      0     0 ?        S    Oct22   0:00
[watchdog/0]
```

flags `lax` provides a long listing with more technical detail, but may display faster by avoiding user name lookups.

```
[jmedinar@localhost ~]$ ps lax
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME
COMMAND
1     0     4     2   0 -20      0     0 worker S<   ?          0:01
[kworker/0:0H]
1     0     6     2  20   0      0     0 smpboo S    ?          0:01
[ksoftirqd/0]
1     0     7     2 -100  -      0     0 smpboo S    ?          0:00
[migration/0]
1     0     8     2  20   0      0     0 rcu_gp S    ?          0:00
[rcu_bh]
1     0     9     2  20   0      0     0 rcu_gp S    ?          3:30
[rcu_sched]
1     0    10     2   0 -20      0     0 rescue S<   ?          0:00
[lru-add-drain]
5     0    11     2 -100  -      0     0 smpboo S    ?          0:00
[watchdog/0]
```

The similar UNIX syntax uses the options -ef to display all processes.

```
[jmedinar@localhost ~]$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         6     2  0 Oct22 ?        00:00:01 [ksoftirqd/0]
root         7     2  0 Oct22 ?        00:00:00 [migration/0]
root         8     2  0 Oct22 ?        00:00:00 [rcu_bh]
root         9     2  0 Oct22 ?        00:03:30 [rcu_sched]
root        10     2  0 Oct22 ?        00:00:00 [lru-add-drain]
root        11     2  0 Oct22 ?        00:00:00 [watchdog/0]
```

- Processes in brackets (usually at the top of the list) are scheduled kernel threads.
- Zombies are listed as exiting or defunct.

- The output of ps displays once. Use top for a process display that dynamically updates.
- ps can display in tree format so you can view relationships between parent and child processes.
- The default output is sorted by process ID number. At first glance, this may appear to be chronological order. However, the kernel reuses process IDs, so the order is less structured than it appears.

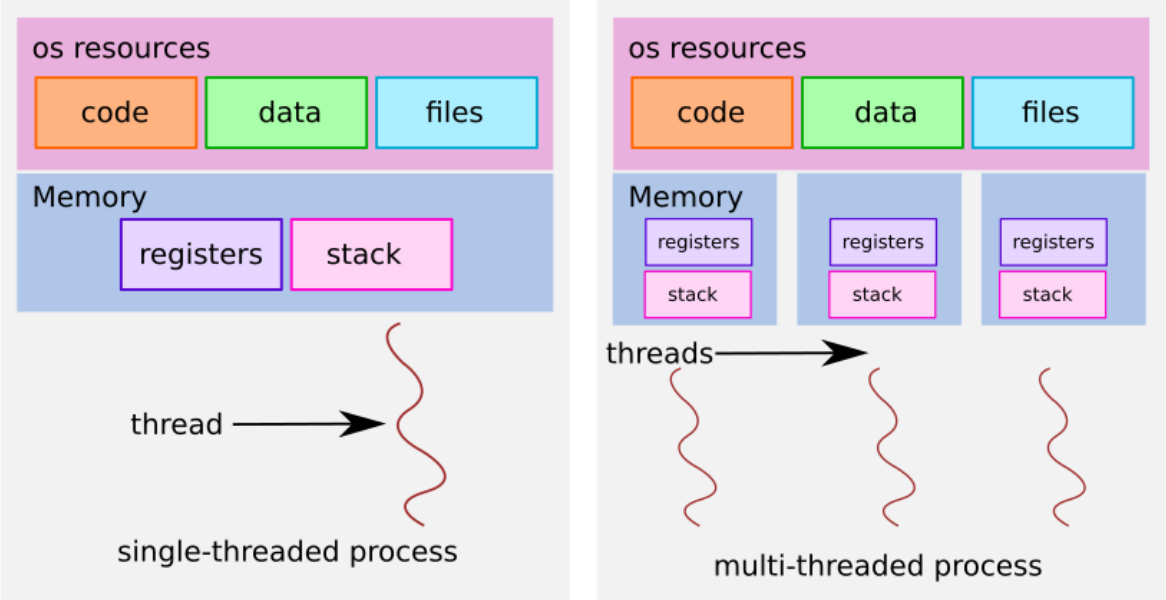
> ***TIP: How to create a Zombie process:***
>
> ```
> 1  $(sleep 1 & exec /bin/sleep 10)
> ```
>
> *The above will create a zombie process that will last 10 seconds and die!*

Threads

In Linux, some processes are divided into pieces called threads. A thread is very similar to a process—it has an identifier (TID), and the kernel schedules and runs threads just like processes. However, unlike processes, which usually do not share system resources such as memory and I/O connections with other processes, all threads are executed inside a single process and share their system resources.

Single-Threaded and Multithreaded Processes

All processes start out single-threaded. This thread is known as the main thread. The main thread may then start new threads in order for the process to become multithreaded. The primary advantage of a multithreaded process is that when the process has a lot to do, threads can run simultaneously on multiple CPUs, potentially speeding up computation.



Viewing Threads

By default, the output from the `ps` and `top` commands show only processes. To display the thread information in ps, add the m option.

```
[jmedinar@localhost ~]$ ps m
    PID TTY      STAT   TIME COMMAND
  27753 tty2     -      0:00 /usr/libexec/gdm-wayland-session
/usr/bin/gnome-session
      - -        Ssl+   0:00 -
      - -        Ssl+   0:00 -
      - -        Ssl+   0:00 -
  27762 tty2     -      0:00 /usr/libexec/gnome-session-binary
      - -        Sl+    0:00 -
```

Each line with a number in the PID column represents a `process`, as in the normal `ps` output. The lines with the dashes in the PID column represent the `threads` associated with the process.

Normally, you won't interact with individual threads as you would with processes. You need to know a lot about how a multi-threaded program was written in order to act on one thread at a time, and even then, doing so might not be a good idea.

Threads can confuse things when it comes to resource monitoring because individual threads in a multi-threaded process can consume resources simultaneously.

Process states

When troubleshooting a system, it is important to understand how the kernel communicates with processes and how processes communicate with each other. At process creation, the system assigns the process a state. The **S** column of the `top` command or the **STAT** column of the `ps aux` show the state of each process.

On a single CPU system, only one process can run at a time. It is possible to see several processes with a state of R. However, not all of them will be running consecutively, some of them will be in status waiting.

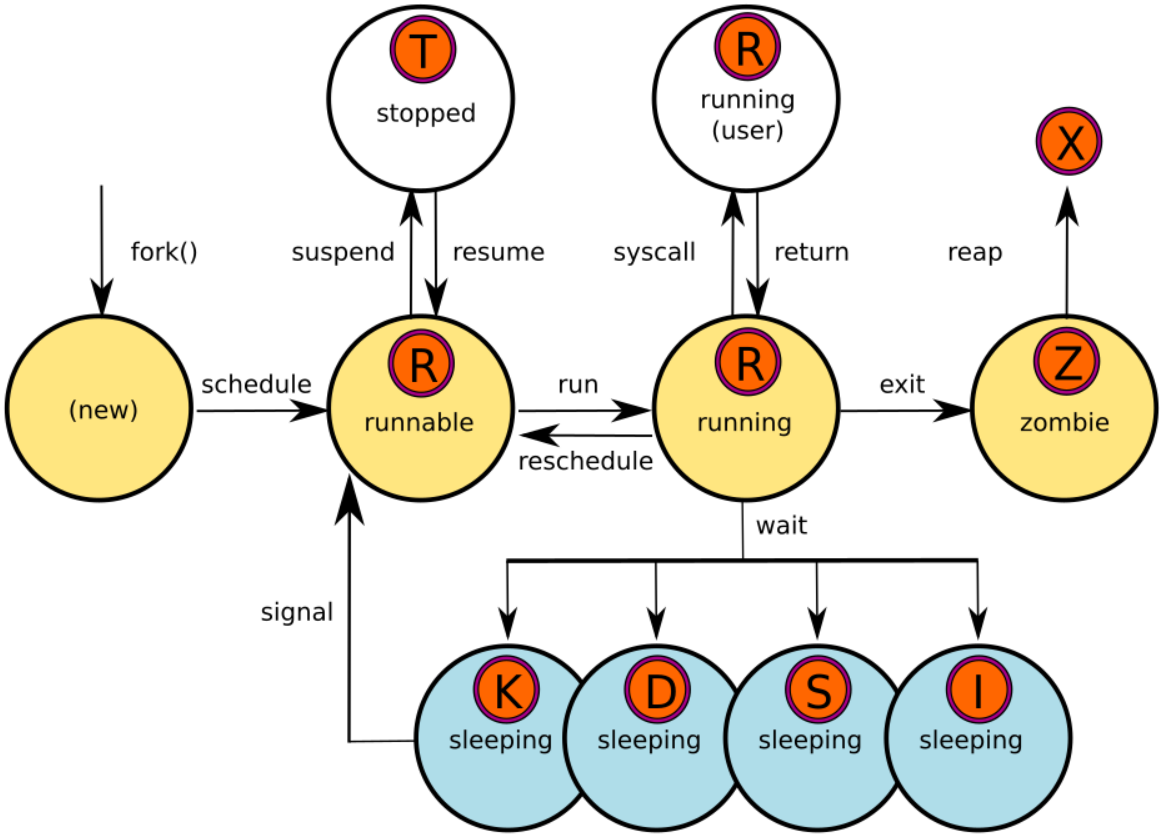
```
[jmedinar@localhost ~]$ top
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
TIME+ COMMAND
      1 root      20   0  176104  13184   5960 S   0.0   0.1
0:14.20 systemd
      2 root      20   0       0      0      0 S   0.0   0.0
0:00.29 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0
0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0
0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0
0:00.00 kworker/0:0H-kblockd
```

```
[jmedinar@localhost ~]$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
root           1  0.0  0.0 176104 13184 ?        Ss   Dec19   0:14
/usr/lib/systemd/systemd
root           2  0.0  0.0      0     0 ?        S    Dec19   0:00
[kthreadd]
root           3  0.0  0.0      0     0 ?        I<   Dec19   0:00
[rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   Dec19   0:00
[rcu_par_gp]
```

Process can be suspended, stopped, resumed, terminated, and interrupted using signals. Signals can be used by other processes, by the kernel itself, or by users logged into the system.

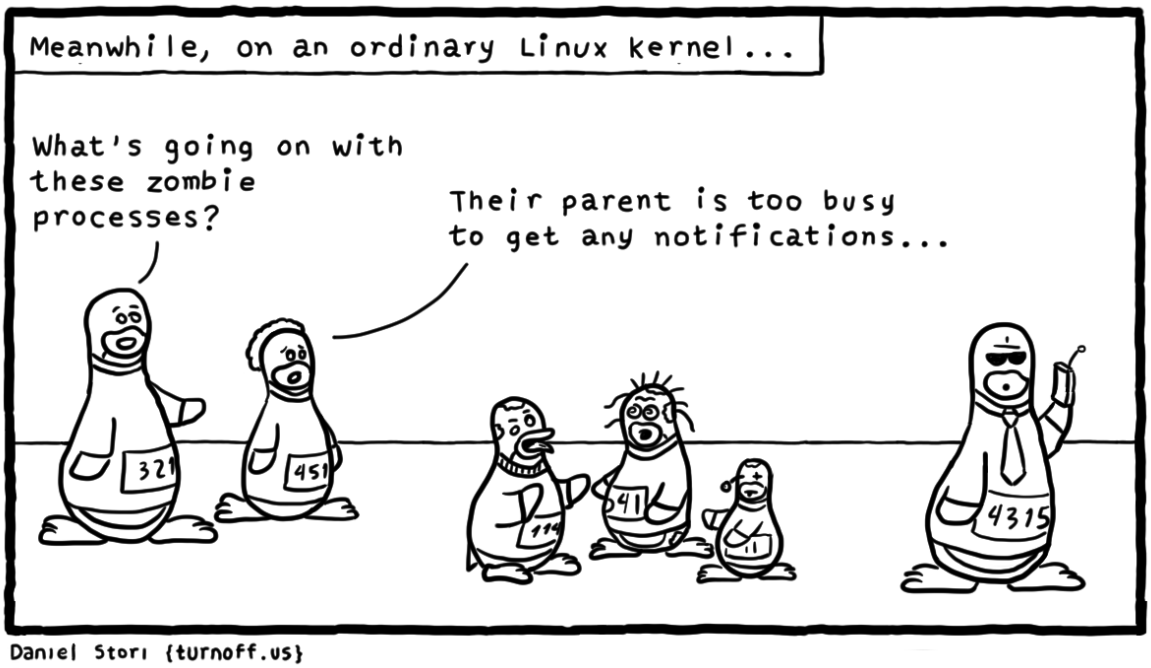
In a multitasking operating system, each CPU (or CPU core) can be working on one process at a single point in time. As a process runs, its immediate requirements for CPU time and resource allocation change. Processes are assigned a state, which changes as circumstances dictate.

Linux process states are illustrated in the following diagram:



NAME	FLAG	KERNEL-DEFINED STATE NAME AND DESCRIPTION
Running	R	TASK_RUNNING: The process is either executing on a CPU or waiting to run.
Sleeping	S	TASK_INTERRUPTIBLE: The process is waiting for a hardware request, system resource access, or signal to then return to Running.
Sleeping	D	TASK_UNINTERRUPTIBLE: This process is also Sleeping, but unlike S state, does not respond to signals.
Sleeping	K	TASK_KILLABLE: Identical to the uninterruptible D state, but modified to allow it to respond to the kill signal.
Sleeping	I	TASK_REPORT_IDLE: A subset of state D. The kernel does not count these processes when calculating load average. It accepts fatal signals.
Stopped	T	TASK_STOPPED: The process has been Stopped (suspended), usually by being signaled by a user or another process. The process can be continued (resumed) by another signal to return to Running.
Stopped	T	TASK_TRACED: A process that is being debugged is also temporarily Stopped and shares the same T state flag.

NAME	FLAG	KERNEL-DEFINED STATE NAME AND DESCRIPTION
Zombie	Z	EXIT_ZOMBIE: A child process signals its parent as it exits. All resources except for the process identity (PID) are released.
Zombie	X	EXIT_DEAD: When the parent kills the remaining childs (reaps), the process is now released completely. This state will never be observed in process-listing utilities.



Signals

Now that we can see and monitor processes, let's gain some control over them.

A signal is a software interrupt delivered to a process. Signals report events to an executing program.

The following table lists the fundamental signals used by system administrators for routine process management.

SIGNAL	SHORT NAME	DEFINITION	PURPOSE
1	HUP	Hangup	Used to report termination of the controlling process of a terminal. Also used to request process reinitialization (configuration reload) without termination.
2	INT	Keyboard interrupt	Causes program termination. Can be blocked or handled. Sent by pressing INTR key combination (Ctrl+C).
3	QUIT	Keyboard quit	Similar to SIGINT, but also produces a process dump at termination. Sent by pressing QUIT key combination (Ctrl+).
9	KILL	Kill, unblockable	Causes abrupt program termination. Cannot be blocked, ignored, or handled; always fatal.
15	TERM	Terminate	Causes program termination. Unlike SIGKILL, can be blocked, ignored, or handled. The “polite” way to ask a program to terminate; allows self-cleanup.
18	CONT	Continue	Sent to a process to resume, if stopped. Cannot be blocked. Even if handled, always resumes the process.

SIGNAL	SHORT NAME	DEFINITION	PURPOSE
19	STOP	Stop, unblockable	Suspends the process. Cannot be blocked or handled.
20	TSTP	Keyboard stop	Unlike SIGSTOP, can be blocked, ignored, or handled. Sent by pressing SUSP key combination (Ctrl+Z).

Background and foreground processes

Before we get into sending signals we need to understand where is our process running. It has two possible places:

- Foreground: The process or program is currently attached to our terminal session or graphical interface. We can see it! We can stop it by closing the terminal or send signals using keyboard control sequences `Ctrl+Z` to suspend, `Ctrl+C` to kill, or clicking the X in the corner of your window.
- Background: The process is attached to a background "hidden" terminal session and it is invisible to us!. Background processes of cannot read input or receive keyboard generated interrupts, but may be able to write to the terminal. We will use signal-sending commands to send signals to a background process.

Running jobs in the background

Any command can be started in the background by appending an ampersand `&` to the end of the command line.

```
[jmedinar@localhost ~]$ sleep 1000 &
[1] 424038
```

The Bash shell displays a job number (unique to the session) and the PID of the new child process. The shell does not wait for the child process to terminate, but rather goes back to the shell prompt.

You can display the list of jobs that Bash is tracking for a particular session with the jobs command.

```
[jmedinar@localhost ~]$ jobs
[1]+  Running                 sleep 1000 &
```

A background job can be brought to the foreground by using the fg command with its job ID (%job number).

```
[jmedinar@localhost ~]$ fg %1
sleep 1000
```

In the preceding example, the sleep command is now running in the foreground on the controlling terminal. The shell itself is again asleep, waiting for this child process to exit.

To send a foreground process again to the background, first press the keyboard generated suspend request (Ctrl+Z) in the terminal. The job is immediately placed in the background and is suspended.

```
[jmedinar@localhost ~]$ fg %1
sleep 1000
^Z
[1]+  Stopped                 sleep 1000
```

The `ps j` command displays information relating to jobs. The PID is the unique process ID of the process. The PPID is the PID of the parent process of this process, the process that started (forked) it. The PGID is the PID of the process group leader, normally the first process in the job's pipeline. The SID is the PID of the session leader, which (for a job) is normally the interactive shell that is running on its controlling terminal.

Since the example sleep command is currently suspended, its process state is T.

```
[jmedinar@localhost ~]$ ps j
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
 129753  424038  424038  129753 pts/0     424202 T     1000   0:00 sleep 1000
```

To resume the suspended process running in the background, use the bg command with the same job ID.

```
[jmedinar@localhost ~]$ bg %1
[1]+ sleep 1000 &
```

Sending signals

Signals can be specified as options either by name (for example, -HUP or -SIGHUP) or by number (the related -1). Users may kill their own processes, but root privilege is required to kill processes owned by others.

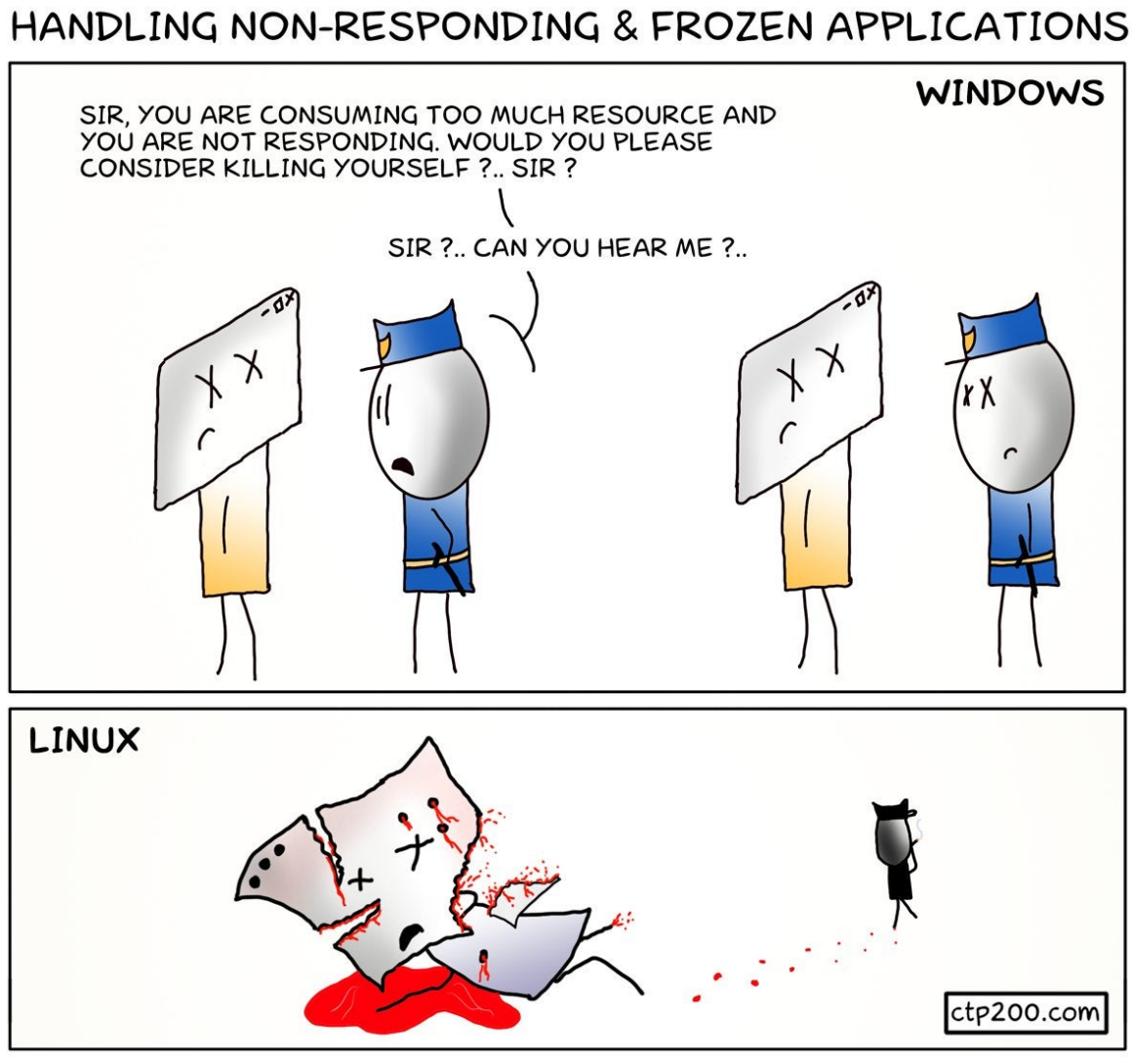
The kill command sends a signal to a process by PID number. Despite its name, the kill command can be used for sending any signal, not just those for terminating programs. You can use the kill -l command to list the names and numbers of all available signals.

```
[jmedinar@localhost ~]$ 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
```

```
 6  21) SIGTTIN 22) SIGTTOU 23) SIGURG  24) SIGXCPU 25) SIGXFSZ
 7  26) SIGVTALRM  27) SIGPROF 28) SIGWINCH   29) SIGIO   30) SIGPWR
 8  31) SIGSYS  34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37)
    SIGRTMIN+3
 9  38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42)
    SIGRTMIN+8
10  43) SIGRTMIN+9 44) SIGRTMIN+10   45) SIGRTMIN+11   46) SIGRTMIN+12
    47) SIGRTMIN+13
11  48) SIGRTMIN+14   49) SIGRTMIN+15   50) SIGRTMAX-14   51) SIGRTMAX-
    13   52) SIGRTMAX-12
12  53) SIGRTMAX-11   54) SIGRTMAX-10   55) SIGRTMAX-9 56) SIGRTMAX-8
    57) SIGRTMAX-7
13  58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62)
    SIGRTMAX-2
14  63) SIGRTMAX-1 64) SIGRTMAX
```

The kill command is used to “kill” processes. This allows us to terminate programs that need killing by providing the PID or jobspec ID "%1". Here's an example:

```
1  [jmedinar@localhost ~]$ kill 28401
2  [1]+ Terminated xlogo
```



The kill command is used to send signals to programs. Its most common syntax looks like this:

```
[jmedinar@localhost ~]$ kill [-signal] PID...
```

If no signal is specified on the command line, then the TERM (Terminate) signal is sent by default.

Signaling multiple processes killall

It's also possible to send signals to multiple processes matching a specified program or username by using the killall command. Here is the syntax:

```
$ killall [-u user] [-signal] name...
```

To demonstrate let's run a few instances of xlogo in the background

```
$ xlogo &
[1] 18801
$ xlogo &
[2] 18802
```

and now let's kill them all!

```
$ killall xlogo
[1]- Terminated xlogo
[2]+ Terminated xlogo
```

Tracking Processes

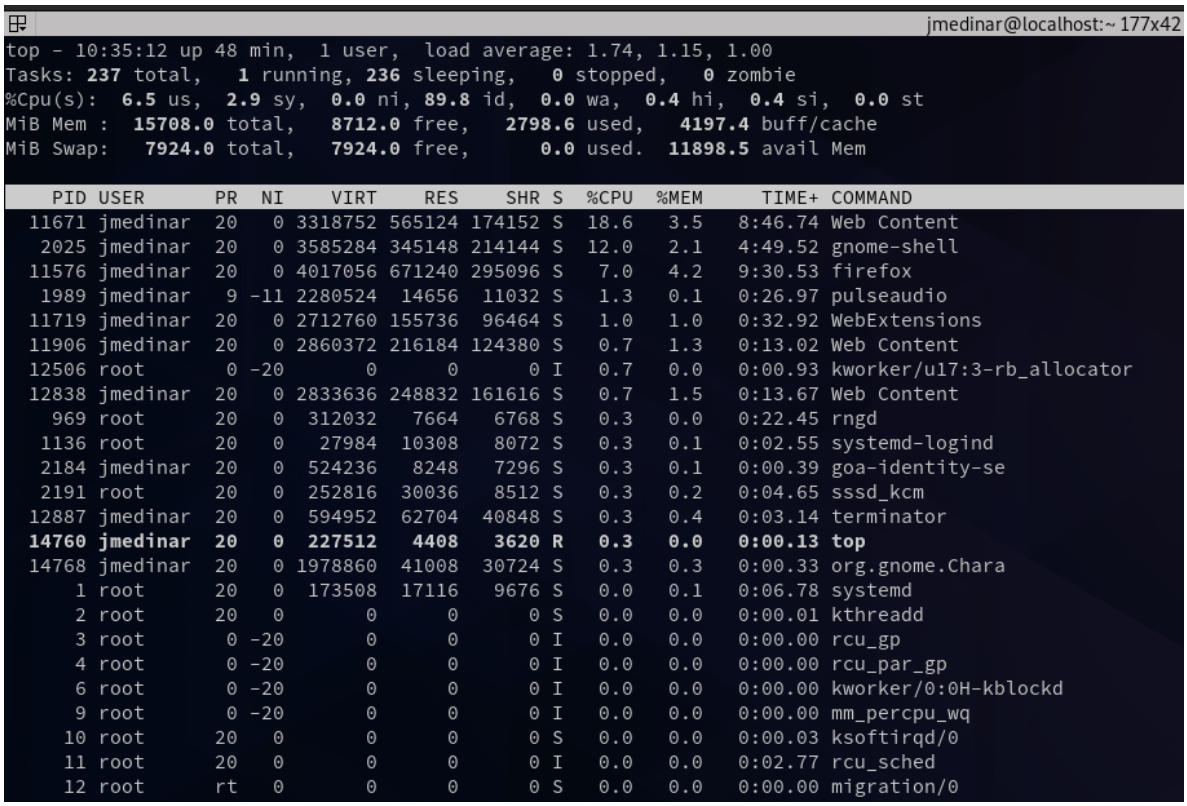
You learned how to use `ps` to list processes running on your system at a particular time. The `ps` command lists current processes, but it does little to tell you how processes change over time. Therefore, it won't really help you to determine which process is using too much CPU time or memory over time.

So let's discuss some other options.

top

The top program is often more useful than `ps` because it displays the current system status as well as many of the fields in a `ps` listing, and it updates the display every second. Perhaps most important is that top shows the most active processes at the top of its display. So this is interactive Live monitoring!

```
1 [jmedinar@localhost ~]$ top
```



You can send commands to top with keystrokes. These are some of the most important commands:

- **Spacebar** - Updates the display immediately.
- **M** - Sorts by current resident memory usage.
- **T** - Sorts by total (cumulative) CPU usage.
- **P** - Sorts by current CPU usage (the default).
- **u** - Displays only one user's processes.
- **f** - Selects different statistics to display.
- **?** - Displays a usage summary for all top commands.

lsof

The `lsof` command lists open files and the processes using them. Because Unix places a lot of emphasis on files, `lsof` is among the most useful tools for finding trouble spots. But `lsof` doesn't stop at regular files it can list network resources, dynamic libraries, pipes, and more.

```
[root@localhost ~]# lsof
COMMAND   PID TID TASKCMD  USER   FD   TYPE    DEVICE  SIZE/OFF
NODE NAME
systemd     1             root  cwd   DIR     253,0       4096
  2 /
systemd     1             root  rtd   DIR     253,0       4096
  2 /
systemd     1             root  txt   REG     253,0    1776584
1193883 /usr/lib/systemd/systemd
systemd     1             root  mem   REG     253,0     142064
1187615 /usr/lib64/libnl-3.so.200.26.0
systemd     1             root  mem   REG     253,0     532392
1187630 /usr/lib64/libnl-route-3.so.200.26.0
systemd     1             root  mem   REG     253,0     130816
1188822 /usr/lib64/libibverbs.so.1.11.32.0
systemd     1             root  mem   REG     253,0     175336
1193020 /usr/lib64/libudev.so.1.6.18
systemd     1             root  mem   REG     253,0      40328
1187770 /usr/lib64/libffi.so.6.0.2
```

There are two basic approaches to running lsof:

- List everything and pipe the output to a command like `less`, and then search for what you're looking for.
- Narrow down the list that lsof provides with command-line options.

You can use command-line options to provide a filename as an argument and have lsof list only the entries that match the argument. For example, the following command displays entries for open files in /usr:

```
[jmedinar@localhost ~]$ lsof /usr
```

To list the open files for a particular process ID, run:

```
[jmedinar@localhost ~]$ lsof -p pid
```

Strace

`strace` is a very handy and useful tool used by system administrators for debugging and diagnostic systems and process-related problems where the source is not really clear and available when taking a quick and first look. This debug tool allows programmers and system users to quickly find out how a program is interacting with the OS. It does this by monitoring system calls and signals.

strace outputs each line in the trace that includes the system call name, its arguments, and the return value.

Let's run strace against the `/bin/ls` program that will just list our files, and save the output of the system calls into a file called ls.txt

```
[jmedinar@localhost ~]$ strace -o ls.txt /bin/ls
```

Now let's explore the content of the `ls.txt` file

```
[jmedinar@localhost ~]$ vim ls.txt
```

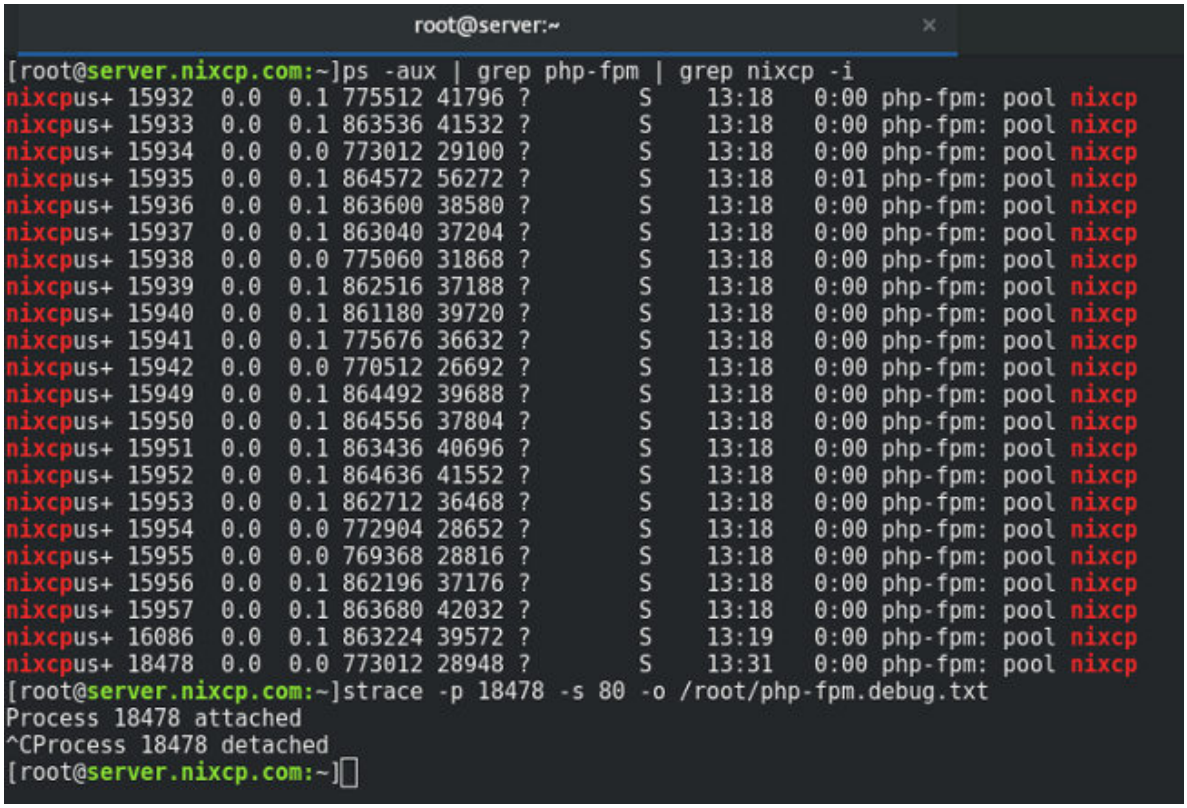
```
execve("/bin/ls", ["/bin/ls"], 0x7ffcd40a2f50 /* 62 vars */) = 0
brk(NULL)                               = 0x55b68a679000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fffd242ad50) = -1 EINVAL
(Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file
or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=135254, ...}) = 0
mmap(NULL, 135254, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f1f0a6b6000
close(3)                                = 0
openat(AT_FDCWD, "/lib64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3,
"\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\213\0\0\0\0\0\0"..
., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=175960, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7f1f0a6b4000
mmap(NULL, 181736, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f1f0a687000
mmap(0x7f1f0a68e000, 106496, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x7000) = 0x7f1f0a68e000
mmap(0x7f1f0a6a8000, 32768, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x21000) = 0x7f1f0a6a8000
mmap(0x7f1f0a6b0000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f1f0a6b0000
mmap(0x7f1f0a6b2000, 5608, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f1f0a6b2000
```

Do no worry about the content of the file now it's enough for the moment to understand the tool is there.

That is just a basic strace example. The interesting part comes when you can strace a process and find out what it is doing exactly in real time.

```
[jmedinar@localhost ~]$ strace -p 18478 -s 80 -o /root/php-fpm.debug.txt
```

By pressing CTRL + C you will terminate the trace and it will be detached.



You can also specify what you need to trace, for example, if you only need to trace the open and read system calls, you should specify that in the strace syntax, as you see below:

```
$ strace -e trace=open,read -p 18478 -s 80 -o /root/php-fpm.debug.txt
```

Now you know how to trace a Linux process easily with two simple commands, with this information you can easily track a Linux process to find out what is doing exactly inside your server. strace takes a little bit more of time to understand from the manual, but it's the definitive tool to trace a Linux process.

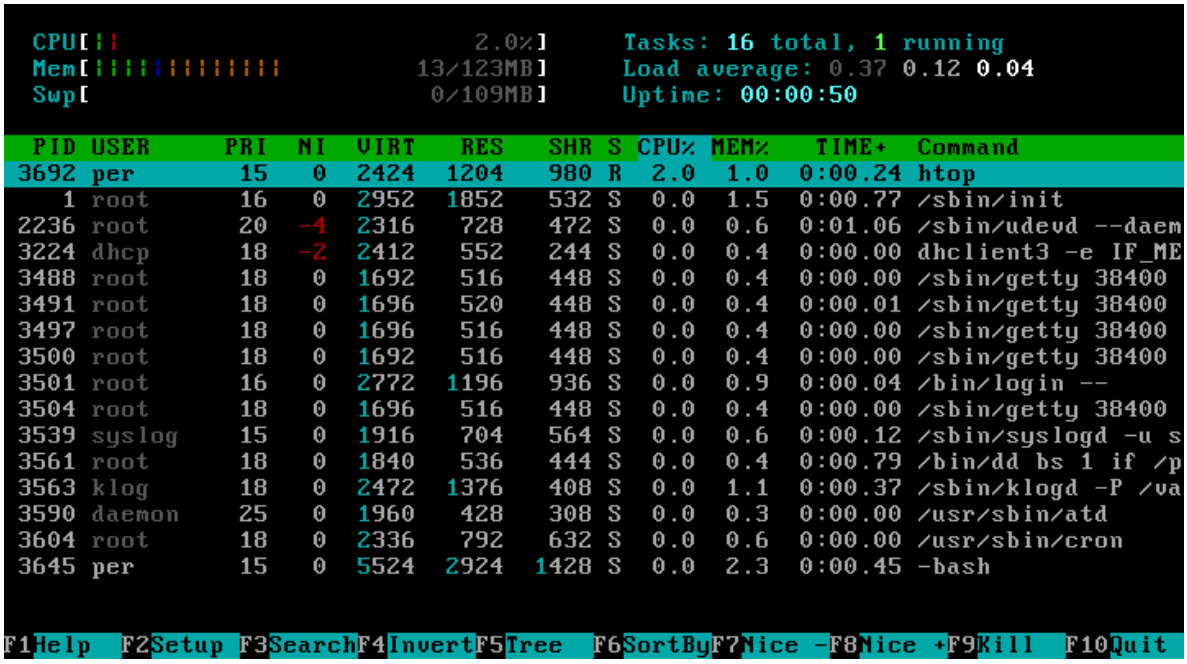
htop

The first great thing about htop is that it will show you your usage per CPU, as well as a meaningful text graph of your memory and swap usage right at the top. I find this much easier to understand at a glance than the default output from `top`.

First we have to install it!

```
[jmedinar@localhost ~]$ sudo dnf install htop
```

Once installed, just type `htop` at a terminal to launch it, and notice the great text-mode graph at the top of the display:



Just use your Up/Down arrow keys to select a process, and then you can kill it with the `F9 key` if you'd like, or you can change the priority by using the `F7 and F8 keys.`

https://hisham.hm/htop/

nmon

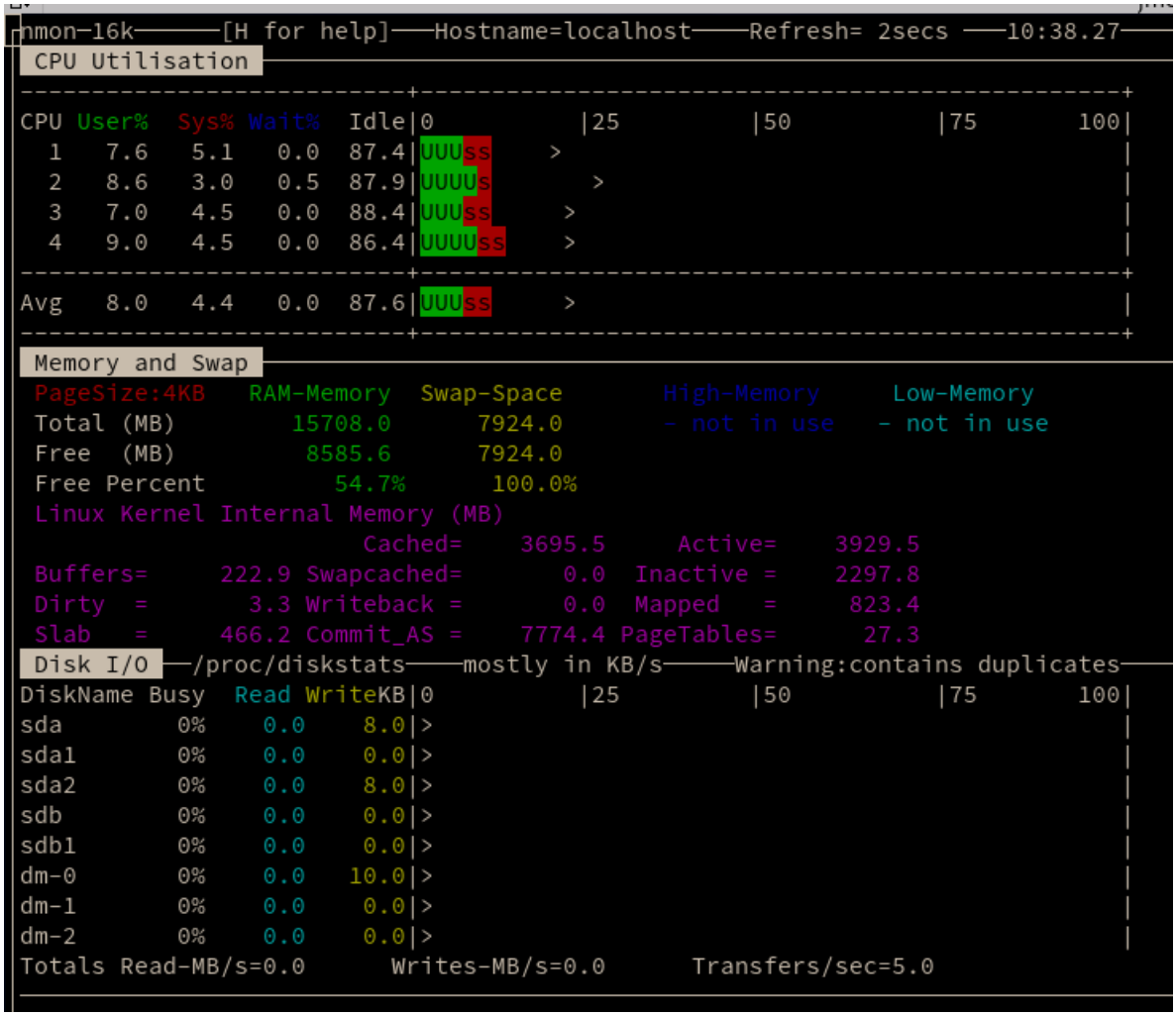
The `nmon` tool will using a simple Ncurses interface to display the usage for CPU, memory, network, disks, file system, NFS, top processes, resources, and power micro-partition. What's best about this tool is that you get to choose what nmon displays. And since it's text-based, you can secure shell into your servers and get a quick glimpse from anywhere.

Installation

```
$ sudo dnf install nmon
```

Usage

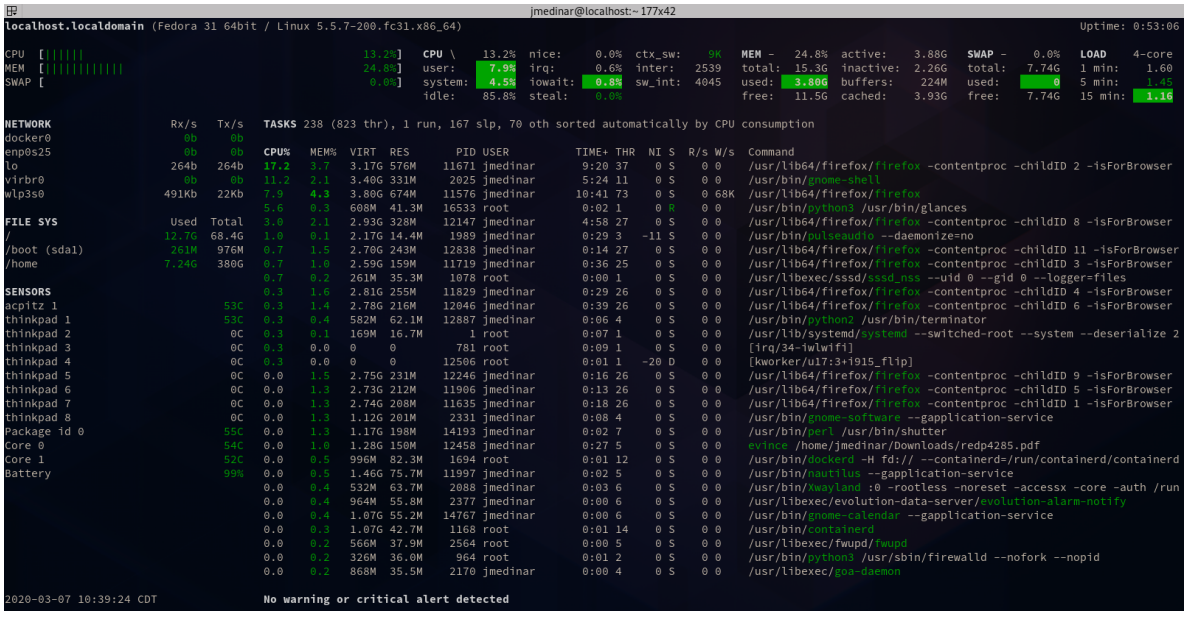
```
$ nmon
```



http://nmon.sourceforge.net/pmwiki.php

Glances

A more modern approach!



https://nicolargo.github.io/glances/

vmstat

vmstat is a very useful tool because it allows you to have a quick overall performance view of your system and it is available in all Unix type systems here some quick a dirty explanations on how to understand the output of the vmstat command:

vmstat is installed by default in all systems and you can run it as follows

```
[jmedinar@localhost ~]$ vmstat
```

or have it to repeat every X interval of seconds

```
[jmedinar@localhost ~]$ vmstat 3
```

or have it give a more human-readable output

```
[jmedinar@localhost ~]$ vmstat -w 3
```

1. r should never be smaller than b or we may have a CPU bottleneck due processes suspended due to memory load control
2. if fre is really small and if any paging is going on pi or po this is most likely a cause of a bottleneck
3. if b and wa are high we may have an I/O bottleneck due to the number of blocking processes
4. if b is low or normal and free is small and us + sy = (close to 100) then we have a memory bottleneck
5. if us+sy average more than 80% we may have a CPU bottleneck if you are at 100 our system is breathing heavily
6. if us+sy is small but wa is greater than 25 we may have I/O intensive activity or disk subsystem might not be balanced properly which turns on CPU not being able to work as hard as he can
7. if us+sy is over 80% and r is larger than [5 * (Number of processors - Number of bound processors)] then we have a CPU bound
8. if r is greater than the number of CPUs, there is at least one thread waiting for a CPU and this is causing of a performance impact.
9. if sy raises over 10000 per second per processor we may be polling subroutines likes select() indicates a bad code it is advisable to have a baseline measurement that gives a count for a normal sy value.

Daemons

A **daemon** (also known as background processes) is a Linux program that runs in the background. Almost all daemons have names that end with the letter "**d**". For example, 'httpd' the daemon that handles the Apache server, or, 'sshd' which handles SSH remote access connections. Linux often starts daemons at boot time. Shell scripts stored in **/etc/init.d** directory are used to start and stop daemons.

Typical functions of daemons

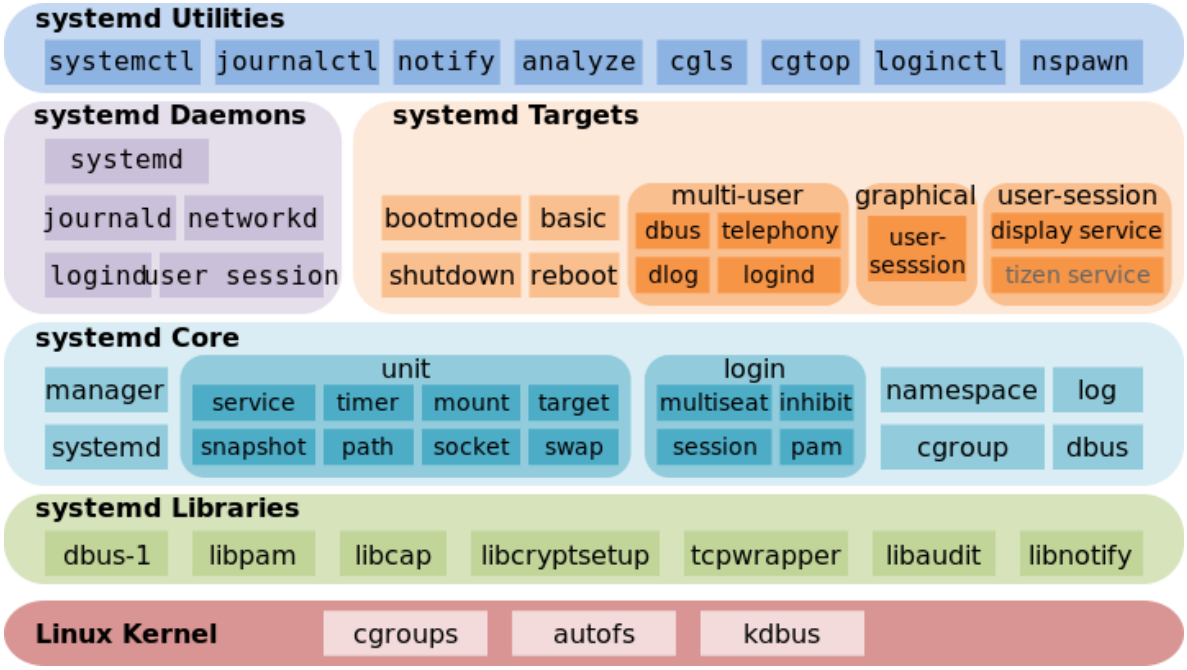
- Open network port (such as port 80) and respond to network requests.
- Monitor system such as hard disk health or RAID array.
- Run scheduled tasks such as cron.

systemd

The systemd daemon manages startup for Linux, including service startup and service management in general. It activates system resources, server daemons, and other processes both at boot time and on a running system.

In Linux, the first process that starts (PID 1) is **systemd**. A few of the features provided by systemd include:

- Starting multiple services simultaneously, which increase the boot speed of a system.
- On-demand starting of daemons without requiring a separate service.
- Automatic service dependency management, which can prevent long timeouts. For example, a network-dependent service will not attempt to start up until the network is available.
- A method of tracking related processes together by using Linux control groups.



Units

systemd uses units to manage different types of objects. Some common unit types are listed below:

- **Service units** have a .service extension and represent system services. This type of unit is used to start frequently accessed daemons, such as a web server.
- **Socket units** have a .socket extension and represent inter-process communication (IPC) sockets that systemd should monitor. If a client connects to the socket, systemd will start a daemon and pass the connection to it. Socket units are used to delay the start of a service at boot time and to start less frequently used services on demand.
- **Path units** have a .path extension and are used to delay the activation of a service until a specific file system change occurs. This is commonly used for services which use spool directories such as a printing system.

The systemctl command is used to manage units. To display available unit types use the `systemctl -t help` command.

```
[jmedinar@localhost ~]$ systemctl -t help
Available unit types:
service
mount
swap
socket
target
device
automount
timer
path
slice
scope
```

Listing service units

You use the `systemctl` command to explore the current state of the system. To lists all currently loaded service units:

```
[root@localhost ~]# systemctl list-units --type=service
  UNIT                                LOAD   ACTIVE SUB
DESCRIPTION
  accounts-daemon.service             loaded active running Accounts
Service
  agnclientd.service                  loaded active running SYSV: AGNS
agnclientd
  agnLogd.service                     loaded active running SYSV: AGNS
agnLogd
  alsa-state.service                  loaded active running Manage
Sound Card State (restore and store)
  atd.service                         loaded active running Job
spooling tools
  auditd.service                      loaded active running Security
Auditing Service
```

The above output limits the type of unit listed to service units with the `--type=service` option.

The output has the following columns:

Column	Description
UNIT	The service unit name.
LOAD	Whether systemd properly parsed the unit's configuration and loaded the unit into memory.
ACTIVE	The high-level activation state of the unit. This information indicates whether the unit has started successfully or not.
SUB	The low-level activation state of the unit. This information indicates more detailed information about the unit. The information varies based on unit type, state, and how the unit is executed.
DESCRIPTION	The short description of the unit.

By default, the `systemctl list-units --type=service` command lists only the service units with active activation states. The `--all` option lists all service units regardless of the activation states. Use the `--state=` option to filter by the values in the LOAD, ACTIVE, or SUB fields.

```
[root@localhost ~]# systemctl list-units --type=service --all | head
  UNIT                                     LOAD      ACTIVE   SUB
    DESCRIPTION
  accounts-daemon.service                  loaded    active
running Accounts Service
  agnclientd.service                       loaded    active
running SYSV: AGNS agnclientd
  agnLogd.service                          loaded    active
running SYSV: AGNS agnLogd
  alsa-restore.service                     loaded    inactive dead
   Save/Restore Sound Card State
  atd.service                              loaded    active
running Job spooling tools
  auditd.service                           loaded    active
running Security Auditing Service
  auth-rpcgss-module.service               loaded    inactive dead
   Kernel Module supporting RPCSEC_GSS
```

The systemctl command without any arguments lists units that are both loaded and active.

```
[root@localhost ~]# systemctl
  UNIT                                LOAD   ACTIVE     SUB
DESCRIPTION
  proc-sys-fs-binfmt_misc.automount    loaded active     running
Arbitrary Executable File Formats...
  sys-devices-pci0000:00-0000:0...     loaded active     plugged
/sys/devices/pci0000:00/0000:00:02....
  sys-devices-pci0000:00-0000:00...    loaded active     plugged
Broadwell-U Audio Controller
  sys-devices-pci0000:00-0000:00:...   loaded active     plugged
PC-LM1E_Camera
  sys-devices-pci0000:00-0000:00:...   loaded active     plugged
/sys/devices/pci0000:00/0000:0...
  sys-devices-pci0000:00-0000:00:...   loaded active     plugged
/sys/devices/pci0000:00/0000:00:1.0/bl...
```

The `systemctl list-units` command displays units that the systemd service attempts to parse and load into memory; it does not display installed, but not enabled, services. To see the state of all unit files installed, use the `systemctl list-unit-files` command.

```
[root@localhost ~]# systemctl list-unit-files --type=service
UNIT FILE                                  STATE
accounts-daemon.service                    enabled
alsa-restore.service                       static
alsa-state.service                         static
anaconda-direct.service                    static
anaconda-nm-config.service                 static
anaconda-noshell.service                   static
anaconda-pre.service                       static
anaconda-shell@.service                    static
anaconda-sshd.service                      static
...output omitted...
```

In the output of the `systemctl list-units-files` command, valid entries for the STATE field are enabled, disabled, static, and masked

Controlling system services

Status of a service

```
[root@localhost ~]# systemctl status sshd.service
● sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled;
vendor preset: enabled)
   Active: active (running) since Thu 2020-10-22 08:09:58 CDT; 6
days ago
     Docs: man:sshd(8)
           man:sshd_config(5)
 Main PID: 1271 (sshd)
    Tasks: 1
   Memory: 1.1M
   CGroup: /system.slice/sshd.service
           └─1271 /usr/sbin/sshd -D

Oct 22 08:09:58 oc4083742478.ibm.com systemd[1]: Started OpenSSH
server daemon.
Oct 22 09:42:51 oc4083742478.ibm.com sshd[13495]: error: Could not
load host key: /etc/ssh/ssh_host_dsa_key
Oct 22 09:42:51 oc4083742478.ibm.com sshd[13495]: Did not receive
identification string from 127.0.0.1 port 42338
Oct 23 10:23:53 oc4083742478.ibm.com sshd[13281]: error: Could not
load host key: /etc/ssh/ssh_host_dsa_key
```

This command displays the current status of the service. The meaning of the fields are:

FIELD	DESCRIPTION
Loaded	Whether the service unit is loaded into memory.

FIELD	DESCRIPTION
Active	Whether the service unit is running and if so, how long it has been running.
Main PID	The main process ID of the service, including the command name.
Status	Additional information about the service.

Several keywords indicating the state of the service can be found in the status output:

KEYWORD	DESCRIPTION
loaded	Unit configuration file has been processed.
active (running)	Running with one or more continuing processes.
active (exited)	Successfully completed a one-time configuration.
active (waiting)	Running but waiting for an event.
inactive	Not running.
enabled	Is started at boot time.
disabled	Is not set to be started at boot time.
static	Cannot be enabled, but may be started by an enabled unit automatically.

The systemctl command provides methods for verifying the specific states of a service. For example, use the following command to verify that the a service unit is currently active (running):

```
1 [root@localhost ~]# systemctl is-active sshd.service
2 active
```

The command returns state of the service unit, which is usually active or inactive.

Run the following command to verify whether a service unit is enabled to start automatically during system boot:

```
[root@localhost ~]# systemctl is-enabled sshd.service
enabled
```

The command returns whether the service unit is enabled to start at boot time, which is usually enabled or disabled.

To verify whether the unit failed during startup, run the following command:

```
[root@localhost ~]# systemctl is-failed sshd.service
active
```

The command either returns active if it is properly running or failed if an error has occurred during startup. In case the unit was stopped, it returns unknown or inactive. To list all the failed units, run the `systemctl --failed --type=service` command.

Starting and stopping services

To start a service, first verify that it is not running with systemctl status. Then, use the `systemctl start` command as the root user.

```
[root@localhost ~]# systemctl start sshd.service
```

The systemd service looks for .service files for service management in commands in the absence of the service type with the service name. Thus the above command can be executed as:

```
[root@localhost ~]# systemctl start sshd
```

To stop a currently running service, use the stop argument with the systemctl command. The example below shows how to stop the sshd.service service:

```
[root@localhost ~]# systemctl stop sshd.service
```

Restarting and reloading services

On the restart of service, the process ID changes and a new process ID gets associated during the startup. To restart a running service, use the restart argument with the systemctl command.

```
[root@localhost ~]# systemctl restart sshd.service
```

Some services have the ability to reload their configuration files without requiring a restart. This process is called a service reload. Reloading a service does not change the process ID associated with various service processes. To reload a running service, use the reload argument with the systemctl command.

```
[root@localhost ~]# systemctl reload sshd.service
```

In case you are not sure whether the service has the functionality to reload the configuration file changes, use the reload-or-restart argument with the systemctl command.

```
[root@localhost ~]# systemctl reload-or-restart sshd.service
```

Enabling services to start or stop at boot

To start a service at boot, use the systemctl enable command.

```
[root@localhost ~]# systemctl enable sshd.service
Created symlink /etc/systemd/system/multi-
user.target.wants/sshd.service → /usr/
lib/systemd/system/sshd.service.
```

The above command creates a symbolic link from the service unit file, usually in the `/usr/lib/systemd/system` directory, to the location on disk where systemd looks for files, which is in the `/etc/systemd/system/TARGETNAME.target.wants` directory.

To disable the service from starting automatically, use the following command, which removes the symbolic link created while enabling a service. Note that disabling a service does not stop the service.

```
[root@localhost ~]# systemctl disable sshd.service
Removed /etc/systemd/system/multi-user.target.wants/sshd.service.
```

To verify whether the service is enabled or disable, use the systemctl is-enabled command.

Listing unit dependencies

Some services require that other services be running first, creating dependencies on the other services. Other services are not started at boot time but rather only on demand. In both cases, systemd and systemctl start services as needed whether to resolve the dependency or to start an infrequently used service. For example, if the CUPS print service is not running and a file is placed into the print spool directory, then the system will start CUPS-related daemons or commands to satisfy the print request.

```
[root@localhost ~]# systemctl stop cups.service
Warning: Stopping cups, but it can still be activated by:
   cups.path
   cups.socket
```

To completely stop printing services on a system, stop all three units. Disabling the service disables the dependencies.

The `systemctl list-dependencies UNIT` command displays a hierarchy mapping of dependencies to start the service unit. To list reverse dependencies (units that depend on the specified unit), use the `--reverse` option with the command.

```
[root@localhost ~]# systemctl list-dependencies sshd.service
sshd.service
● ├─system.slice
● ├─sshd-keygen.target
● │ ├─sshd-keygen@ecdsa.service
● │ ├─sshd-keygen@ed25519.service
● │ └─sshd-keygen@rsa.service
● └─sysinit.target
...output omitted...
```

Masking and unmasking services

At times, a system may have different services installed that are conflicting with each other.

For example, there are multiple methods to manage mail servers (postfix and sendmail, for example). Masking a service prevents an administrator from accidentally starting a service that conflicts with others. Masking creates a link in the configuration directories to the `/dev/null` file which prevents the service from starting.

```
[root@localhost ~]# systemctl mask sendmail.service
Created symlink /etc/systemd/system/sendmail.service → /dev/null.
```

```
[root@localhost ~]# systemctl list-unit-files --type=service
UNIT FILE                                   STATE
sendmail.service                            masked
...output omitted...
```

Attempting to start a masked service unit fails with the following output:

```
[root@localhost ~]# systemctl start sendmail.service
Failed to start sendmail.service: Unit sendmail.service is masked.
```

Use the systemctl unmask command to unmask the service unit.

```
[root@localhost ~]# systemctl unmask sendmail
Removed /etc/systemd/system/sendmail.service.
```

Load Averages

The `load average` is the measurement of processes in R state (Running). That is, it is an estimate of the number of processes that are capable of using the CPU at any given time. When thinking about a load average, keep in mind that most processes on your system are usually on some form of waiting state, meaning that most processes are not actually running and should contribute nothing to the load of the system. Only processes that are actually doing something affect the load average.

It can be used as a rough gauge of how many system resource requests are pending, and to determine whether system load is increasing or decreasing over time. Every five seconds, the kernel collects the current load number, based on the number of processes in runnable and uninterruptible states. This number is accumulated and reported as an exponential moving average over the most recent 1, 5, and 15 minutes.

uptime

The `uptime` command displays the current load average. It prints the current time, how long the machine has been up, how many user sessions are running, and the current load average.

```
1 [jmedinar@localhost ~]$ uptime
2  21:06:12 up 8 days,  7:47,  1 user,  load average: 0.55, 0.67, 0.77
```

The last three numbers represent the load averages for the past **1 minute**, **5 minutes**, and **15 minutes**, respectively. A quick glance indicates whether system load appears to be increasing or decreasing.

If the main contribution to load average is from processes waiting for the CPU, you can calculate the approximate per CPU load value to determine whether the system is experiencing significant waiting.

As you can see, this system isn't very busy: An average of only `0.77` processes have been running across all processors for the past 15 minutes. In other words, if you had just one CPU in your system, it was only using around 77% on the last 15 minutes.

But how many CPUs do you really have?

The lscpu command can help you determine this.

```
[jmedinar@localhost ~]$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   36 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              2
Core(s) per socket:              2
Socket(s):                       1
... Output omitted ...
```

In the above example, the system has 4 CPUs in the system. If a load average goes up to around 1, a single process is probably using the CPU nearly all of the time 100% of 1 CPU but since we have 4 CPUs we can think on this as 25% of load!

High Loads

A high load average does not necessarily mean that your system is having trouble. A system with enough memory and I/O resources can easily handle many running processes. If your load average is high and your system still responds well, don't panic.

The system just has a lot of processes sharing the CPU. The processes have to compete with each other for processor time, and as a result, they'll take longer to perform their computations than they would if they were each allowed to use the CPU all of the time.

CPU Cycles

Monitoring the performance of CPU is essential to debug processes inside any system. There are a lot of tools available to monitor and display CPU performance and these tools rely on system cycles that are built into the operating system to extract the performance readings.

Processor metrics

- **CPU utilization:** This is probably the most straightforward metric. It describes the overall utilization per processor. If the CPU utilization exceeds 80% for a sustained period of time, a processor bottleneck is likely.
- **User time:** Depicts the CPU percentage spent on user processes, including nice time. High values in user time are generally desirable because, in this case, the system performs actual work.
- **System time:** Depicts the CPU percentage spent on kernel operations including IRQ and softirq time. High and sustained system time values can point you to bottlenecks in the network and driver stack. A system should generally spend as little time as possible in kernel time.
- **Waiting:** Total amount of CPU time spent waiting for an I/O operation to occur. Like the blocked value, a system should not spend too much time waiting for I/O operations; otherwise, you should investigate the performance of the respective I/O subsystem.
- **Idle time:** Depicts the CPU percentage the system was idle waiting for tasks.
- **Nice time:** Depicts the CPU percentage spent on re-nicing processes that change the execution order and priority of processes.

SAR

SAR stands for System Activity Report, as its name suggests SAR is used to collect, report & save **CPU**, **Memory**, **I/O** usage. SAR produce the reports on the fly and can also save the reports in the log files as well.

Generating CPU Report on the Fly 5 times every 2 seconds.

```
[jmedinar@localhost ~]$ sar 2 5
```

Saving sar output to a file using -o

```
[jmedinar@localhost ~]$ sar 2 5 -o /tmp/sar.out > /dev/null 2>&1
```

Display report

```
[jmedinar@localhost ~]$ sar -f /tmp/sar.out
```

Generating Memory Usage report using -r

```
[jmedinar@localhost ~]$ sar -r 2 5
```

Generating Paging Statistics Report using -B

```
[jmedinar@localhost ~]$ sar -B 2 5
```

Generating block device statistics report using -d

```
[jmedinar@localhost ~]$ sar -d -p 2 4
```

Generating Network statistic report using -n

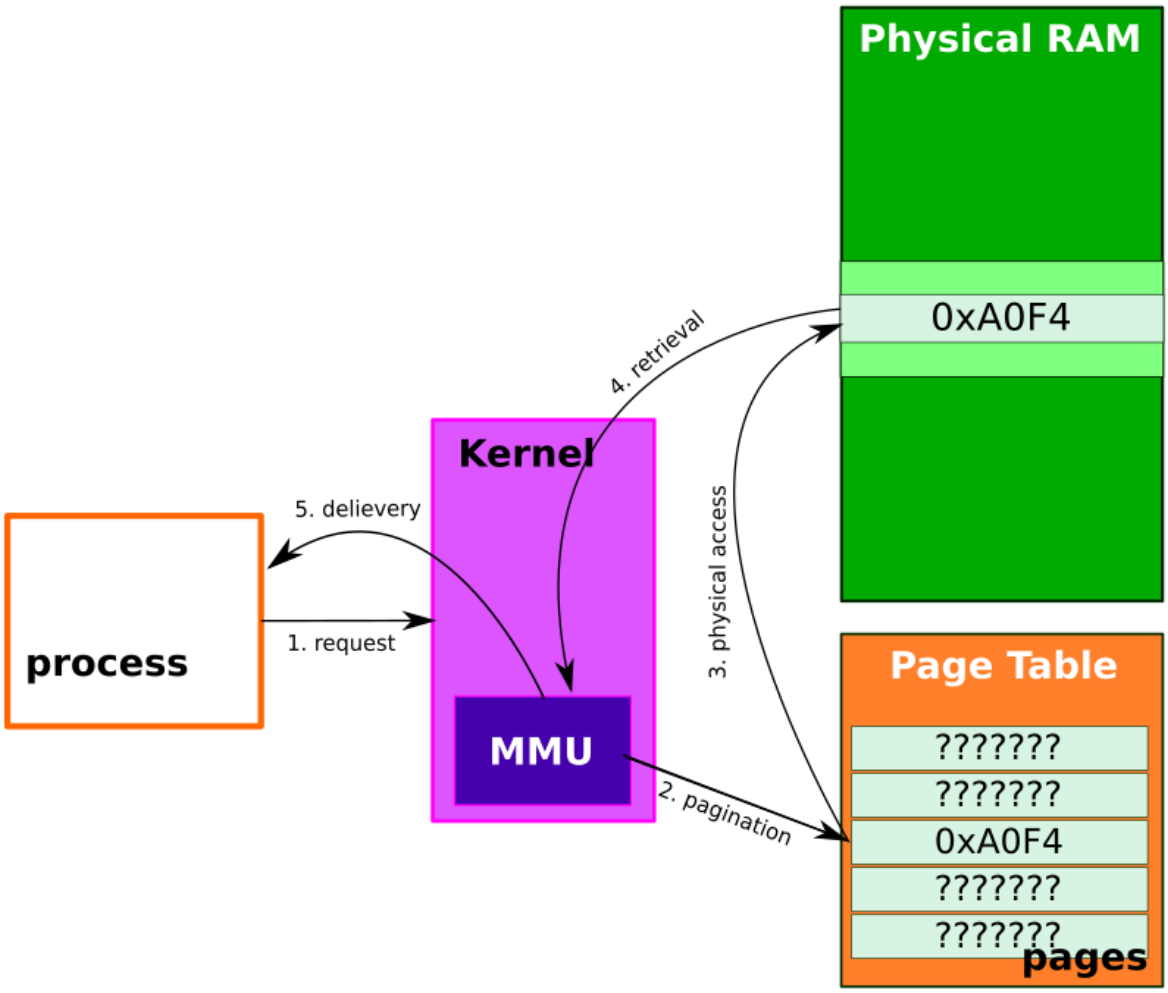
```
[jmedinar@localhost ~]$ sar -n ALL
```

Understanding the Memory

One of the simplest ways to check your system memory status is to run the `free` command or view the `/proc/meminfo` file to see how much real memory is being used for caches and buffers. Performance problems can arise from memory shortages. If there isn't much cache or buffer free, you may need more memory. However, it's too easy to blame a shortage of memory for every performance problem on your machine.

How Memory Works

The kernel breaks the memory used by each process into smaller pieces called **pages**. The kernel maintains a data structure, called a **page table**, that contains a mapping of a process's virtual page addresses to real page addresses in memory. When a process accesses memory, the **Memory Management Unit** (MMU) translates the virtual addresses used by the process into real addresses based on the kernel's page table.



So a process does not actually need all of its pages to be immediately available in order to run. The kernel generally loads and allocates pages as a process needs them.

To see how this works, consider how a program starts:

1. The kernel loads the program into memory.
2. The kernel may allocate some pages to the new process.
3. As the process runs, it might reach a point where the next required instruction is not in any of the pages already loaded. So the kernel takes over, loads the necessary pages into memory, and then the program resume execution.
4. Similarly, if the program requires more working memory than was initially allocated, the kernel handles it by finding free pages and assigning them to the process.

Memory metrics

- **Free memory:** The free memory value in Linux should not be a cause for concern. The Linux kernel allocates most unused memory as cache, so subtract the amount of buffers and cache from the used memory to determine (effectively) free memory.
- **Swap usage:** This value depicts the amount of swap space used. The swap usage only tells you that Linux manages memory really efficiently. Swap In/Out is a reliable means of identifying a memory bottleneck. Values above 200 to 300 pages per second for a sustained period of time express a likely memory bottleneck.
- **Buffer and cache**: Cache allocated as a filesystem and block device cache.
- **Active versus inactive memory:** Provides you with information about the active use of the system memory. Inactive memory is a likely candidate to be swapped out to disk.

Check Memory Usage

Linux comes with a different set of commands to check memory usage. The free command displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel. The vmstat command reports information about processes, memory, paging, block IO, traps, and CPU activity. Finally, you can use the top, and/or atop/htop commands which provide a dynamic real-time view of a running system. top and friends can display system summary information as well as a list of tasks currently being managed by the Linux kernel.

Linux by default tries to use RAM in order to speed up disk operations by making use of available memory for creating buffers (file system metadata) and cache (pages with actual contents of files or block devices), helping the system to run faster because disk information is already in memory which saves I/O operations.

/proc/meminfo

The /proc/meminfo file stores statistics about memory usage on the Linux based system. The same file is used by free and other utilities to report the amount of free and used memory (both physical and swap) on the system as well as the shared memory and buffers used by the kernel.

free command

To display free memory size in MB (megabytes):

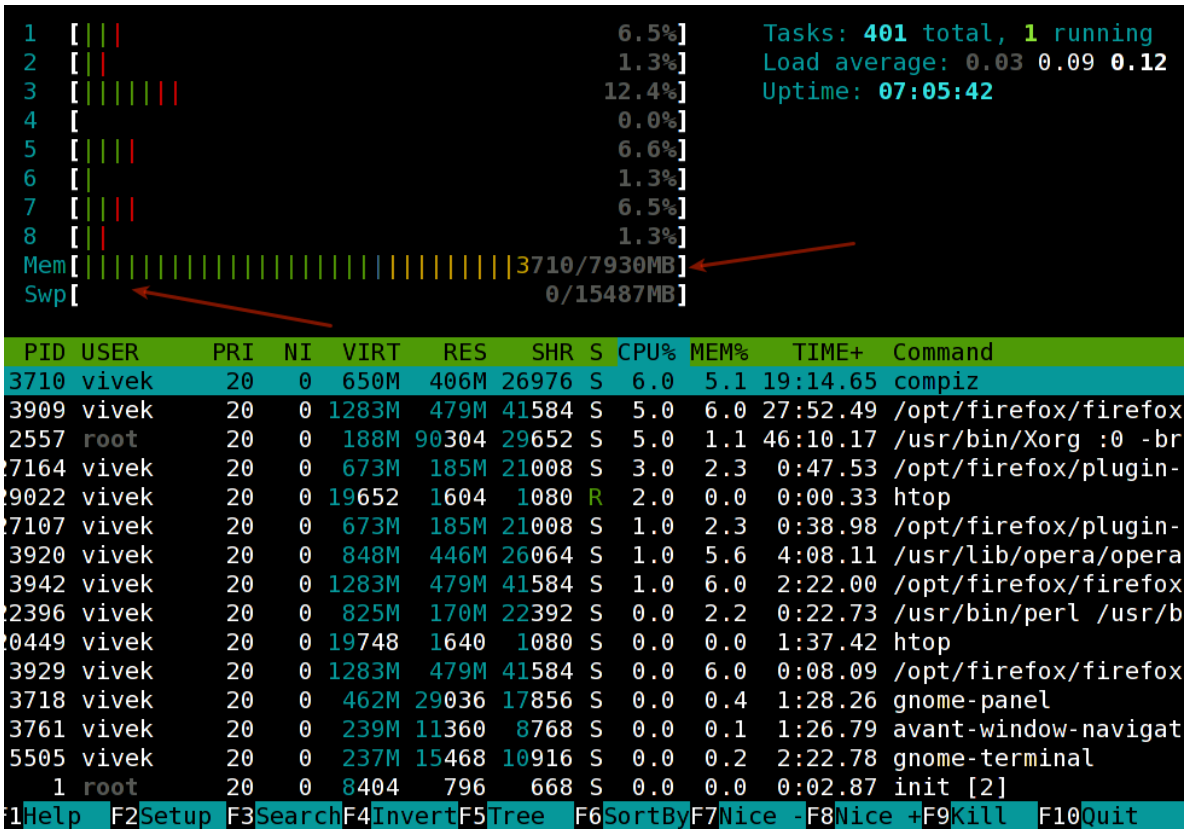
```
$ free -m
```

Displays a line containing the totals memory in MB:

```
$ free -t -m
```

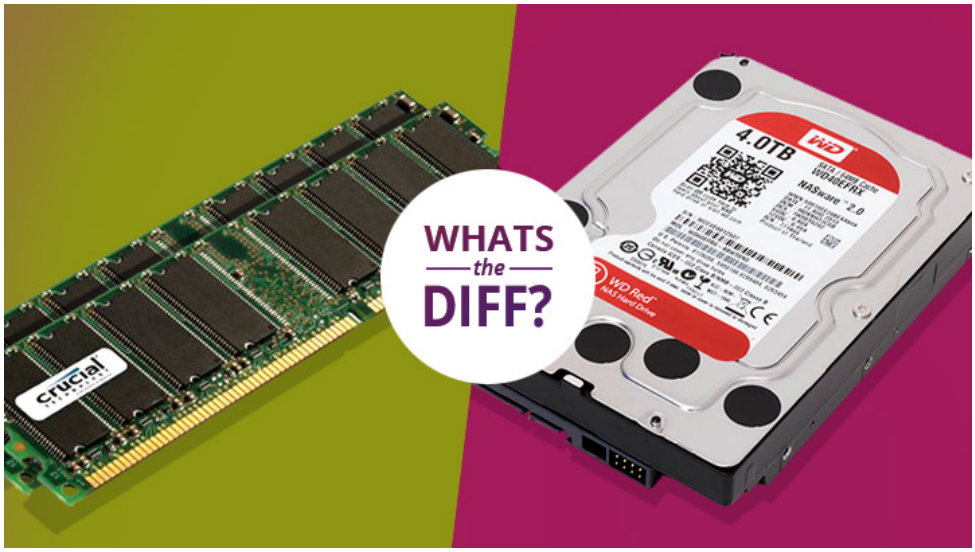
htop

The program htop is an interactive process viewer. It is similar to top, but allows to scroll the list vertically and horizontally to see all processes and their full command lines. By default, the htop command is not installed on most Linux distributions.



I/O Monitoring

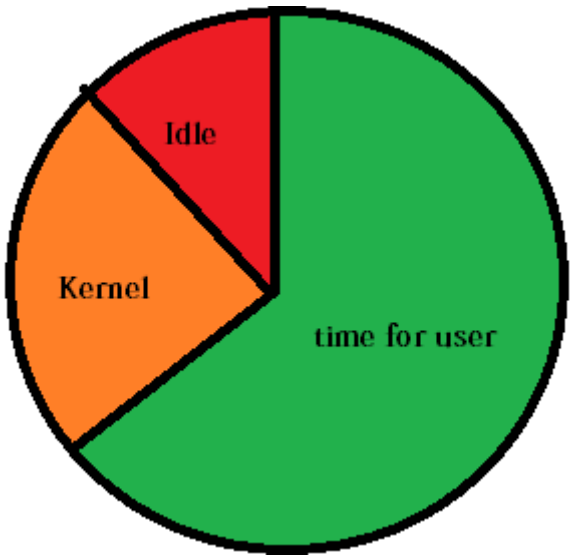
"Everything is a file", is a very famous Linux philosophy. There is a reason for this philosophy to get famous. The main reason behind this is the fact that the Linux operating system in itself works on the same philosophy.



A user can do operations on these devices, by exactly the same way, he does operations on a regular file (read / write / move / copy / paste). The main advantages of block devices are the fact that they can be read randomly. However, serial devices can be operated only serially (data from them can only be accessed as the order they come, not in a random manner.)

Your computer's RAM is a block device. But your storage hard disk is a serial device. The data stored in RAM can be accessed almost instantly regardless of where in memory it is stored, so it's milliseconds fast. But we need some form of non-volatile storage and that consists of a drive, either a hard drive or a solid-state drive. No matter what type of drive you have, storage is almost always slower than RAM.

The slowest part of any Linux system is the disk I/O systems. There is a large difference between, the speed and the duration taken to complete an input/output request of CPU, RAM, and Hardisk. Sometime's if one of the processes running on your system, do a lot of reading/writing operations on the disk, there will be an intense lag or slow response from other processes because they are all waiting for their respective I/O operations to get completed. In an ideal situation, the processor time is largely devoted to the user, and some to the kernel and some are idle.



Block device metrics

- **Iowait**: Time the CPU spends waiting for an I/O operation to occur. High and sustained values most likely indicate an I/O bottleneck.
- **Average queue length**: Amount of outstanding I/O requests. In general, a disk queue of 2 to 3 is optimal; higher values might point toward a disk I/O bottleneck.
- **Average wait**: A measurement of the average time in ms it takes for an I/O request to be serviced. The wait time consists of the actual I/O operation and the time it waited in the I/O queue.
- **Transfers per second**: Depicts how many I/O operations per second are performed (reads and writes). The transfers per/second metric in conjunction with the kBytes per second value helps you to identify the average transfer size of the system. The average transfer size generally should match with the stripe size used by your disk subsystem.
- **Blocks read/write per second**: This metric depicts the reads and writes per second expressed in blocks of 1024 bytes
- **Kilobytes per second read/write**: Reads and writes from/to the block device in kilobytes represent the amount of actual data transferred to and from the block device.

iostat

iostat is used to get the input/output statistics for storage devices and partitions. iostat is a part of the sysstat package. With iostat, you can monitor the read/write speeds of your storage devices (such as hard disk drives, SSDs) and partitions (disk partitions).

Installation

```
1 [jmedinar@localhost ~]$ sudo dnf install sysstat
```

You can monitor all the storage devices and disk partitions of your computer with iostat as follows:

```
1 [jmedinar@localhost ~]$ sudo iostat
```

iostat generates a report of read/write speeds (in kilobytes/second or kB/s) and total reads/writes (in kB) of every storage device and partitions at that time.

```
[jmedinar@localhost ~]$ sudo iostat
Linux 5.5.7-200.fc31.x86_64 (localhost.localdomain)     03/07/2020      _x86_64_        (4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          15.25    0.17    5.84    0.98    0.00   77.76

Device             tps    kB_read/s    kB_wrtn/s    kB_dscd/s    kB_read    kB_wrtn    kB_dscd
dm-0             48.48       582.98       835.41         0.00    1424313    2041052          0
dm-1              0.04         0.91         0.00         0.00       2228          0          0
dm-2             29.48       107.06       217.80         0.00     261568     532120          0
sda              32.06       588.59       834.28         0.00    1438024    2038296          0
sdb              11.86       108.55       216.08         0.00     265214     527908          0
```

If you want to see real-time statistics, then you can tell iostat to update the report every n seconds (let's say 2 seconds) interval as follows:

```
1 [jmedinar@localhost ~]$ sudo iostat -d 2
```

Monitoring Specific Storage Devices or Partitions with iostat:

By default, iostat monitors all the storage devices of your computer. But, you can monitor specific storage devices (such as sda, sdb) or specific partitions (such as sda1, sda2, sdb4) with iostat as well.

```
1 [jmedinar@localhost ~]$ sudo iostat sda
```

You can also monitor multiple storage devices with iostat.

```
1 [jmedinar@localhost ~]$ sudo iostat sda sdb
```

If you want to monitor specific partitions, then you can do so as well.

```
1 [jmedinar@localhost ~]$ sudo iostat sda1 sda2
```

Changing the Units of iostat:

By default, iostat generates reports in kilobytes (kB) unit. But there are options that you can use to change the unit.

To change the unit to megabytes, run iostat as follows:

```
1 [jmedinar@localhost ~]$ sudo iostat -m -d 2 sda
```

To change the unit to human readable format, run iostat as follows:

```
1 [jmedinar@localhost ~]$ sudo iostat -h -d 2 sda
```

Extended Display of iostat:

If you want, you can display a lot more information about disk i/o with iostat. To do that, use the **-x** option of iostat.

```
[jmedinar@localhost ~]$ sudo iostat -x -d 2 sda
```