

Steal time (ST) also referred to as “Stolen CPU”, exists in virtualized computing environments –It is the time that the CPU uses to run internal virtual machine tasks, with the hypervisor allocating CPU cycles to other “external tasks” that are probably caused by one of your noisy neighbors.

CPU steal is primarily relevant in virtualized environments in which a single host (physical) machine runs multiple guest (virtual) machines. Each virtual machine receives of fraction of the physical CPU resources (and other physical resources). That is, the physical CPU is multiplexed across the multiple virtual CPUs used by the guest VMs. The operating system kernel detects when it has work available but does not have access to the CPU to perform that work.

These are measurements of how long a virtual CPU remains idle while it waits for a physical CPU to provide support for its virtual processes.

*Steal time is the percentage of time a virtual CPU waits for a real CPU while the hypervisor is servicing another virtual processor.*

Your virtual machine (VM) shares resources with other instances on a single host in a virtualized environment. One of the resources it shares is CPU Cycles. If your VM is one of four equally sized VMs on a physical server, its CPU usage isn't capped at 25% of all CPU cycles - it can be allowed to use more than its proportion of CPU cycles

st, "steal time", is only relevant in virtualized environments. It represents time when the real CPU was not available to the current virtual machine - it was "stolen" from that VM by the hypervisor (either to run another VM, or for its own needs).

- **Steal time** (for the whole system only), on [virtualized](#) hardware, is the amount of time the [operating system](#) wanted to execute, but was not allowed to by the [hypervisor](#).<sup>[1]</sup> This can happen if the physical hardware runs multiple guest operating system and the hypervisor chose to allocate a CPU time slot to another one

## CPU Steal

For example, Linux instances will not report the proper values for CPU usage due the virtualization layer on the underlying infrastructure. For accurate values for CPU usage on EC2 instances, the cloud user should rely only on the CloudWatch metrics.

# CPU Steal – Why AWS CloudWatch Metrics Are Different Than Agent Metrics

CloudWatch says you're at 90% CPU utilization, while your installed agent tells you you're at 60%.

both CloudWatch and your agent are right. The difference lies in what they are reporting, and that difference also defines “CPU Steal.”

When Amazon reports on your CPU usage, the reported number is a percentage of your assigned CPU. Since EC2 instances are shared among multiple end users, Amazon determines how much of the CPU “belongs” to you and reports on how much of that you’re using.

Agent-based reporting, on the other hand, is based on a core's total CPU usage. It does not take into account any other users attached to the instance or how much Amazon thinks should be available specifically for your app – rather, it simply reports on how many cycles your instance's CPU is using at a given point in time.

The difference between these two metrics is what's known as "CPU Steal." Essentially, CPU steal is how much of your instance's CPU is engaged by something other than your own virtual machine. This phenomenon is often chalked up to 'noisy neighbors' sharing your instance pulling significant usage.

**Large stolen time** - Basically this means that the host system running the hypervisor is too busy. If possible, check the other virtual machines running on the hypervisor, and/or migrate to your virtual machine to another host.

<http://www.stackdriver.com/cpu-steal-why-aws-cloudwatch-metrics-are-different-than-agent-metrics/>

<https://anturis.com/blog/stolen-cpu-explained-and-how-to-troubleshoot-stolen-cpu-issues/>

<http://iamondemand.com/blog/who-stole-my-cpu/>

<http://blog.scoutapp.com/articles/2013/07/25/understanding-cpu-steal-time-when-should-you-be-worried>

<http://www.stackdriver.com/understanding-cpu-steal-experiment/>

<http://www.stackdriver.com/cpu-steal-why-aws-cloudwatch-metrics-are-different-than-agent-metrics/>

## CPU Nice

the time the CPU has spent running users' processes that have been "niced".

It is the CPU scheduling priority, higher values (+19) mean lower priority, and lower values (-20) mean higher priority

You can set the nice value when launching a process with the `nice` command and then change it with the `renice` command. Only the superuser (root) can specify a priority increase of a process.

A "niced" process is one with a positive nice value. So if the processor's nice value is high, that means it is working with some low priority processes.

- High CPU utilization with high nice value: Nothing to worry, not so important tasks doing their job, important processes will easily get CPU time if they need. This situation is not a real bottleneck.
- High CPU utilization with low nice value: Something to worry because the CPU is stressed with important processes so these or new processes will have to wait. This situation is a real bottleneck.

The CPU priority in Linux are expressed in nice values, between -20 and 20, the higher the nice value, the lowest the priority in the CPU.

You can run `ps -eo nice,pid,args | grep '^ \s*[1-9]'` to get a list of positive nice (low priority) commands.

Processes on Linux are started with a niceness of 0 by default. The nice command (without any additional parameters) will start a process with a niceness of 10. At that level the scheduler will see it as a lower priority task and give it less CPU resources.

Start two matho-primes tasks, one with nice and one without:

```
sudo nice matho-primes 0 9999999999 > /dev/null &
matho-primes 0 9999999999 > /dev/null &
```

Now run top.

top - 11:39:23 up 3:33, 4 users, load average: 1.25, 0.93, 0.62										
Tasks: 159 total, 4 running, 155 sleeping, 0 stopped, 0 zombie										
%Cpu(s): 88.1 us, 0.3 sy, 9.5 ni, 0.0 id, 0.0 wa, 0.0 hi, 2.0 si, 0.0 st										
KiB Mem: 1018256 total, 794236 used, 224020 free, 0 buffers										
KiB Swap: 2129916 total, 15632 used, 2114284 free. 185252 cached Mem										
PID	USER	PR	NI	VIRT	RES	SHR	%CPU	%MEM	TIME+	COMMAND
6919	gary	20	0	9208	2540	492	90.0	0.2	0:17.12	matho-primes
6918	gary	30	10	9208	2540	500	9.9	0.2	0:01.83	matho-primes
6864	root	20	0	0	0	0	0.0	0.0	0:00.17	kworker/0:1
1	root	20	0	52848	4856	2860	S 0.0	0.5	0:04.58	systemd
2	root	20	0	0	0	0	S 0.0	0.0	0:00.01	kthreadd
3	root	20	0	0	0	0	S 0.0	0.0	0:02.41	ksoftirqd/0
5	root	0	-20	0	0	0	S 0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	S 0.0	0.0	0:00.00	kworker/u2:0
7	root	rt	0	0	0	0	S 0.0	0.0	0:00.00	migration/0
8	root	20	0	0	0	0	S 0.0	0.0	0:00.00	rcu_bh
9	root	20	0	0	0	0	S 0.0	0.0	0:00.00	rcuob/0
10	root	20	0	0	0	0	S 0.0	0.0	0:12.55	rcu_sched
11	root	20	0	0	0	0	R 0.0	0.0	0:32.36	rcuos/0
12	root	rt	0	0	0	0	S 0.0	0.0	0:00.66	watchdog/0
13	root	0	-20	0	0	0	S 0.0	0.0	0:00.00	khelper
14	root	20	0	0	0	0	S 0.0	0.0	0:00.00	kdevtmpfs
15	root	0	-20	0	0	0	S 0.0	0.0	0:00.00	netns

Nice has an associated command called renice. It changes the niceness level of an already running process. To use it, find out the PID of process hogging all the CPU time (using ps) and then run renice:  
renice +10 1234

Where 1234 is the PID.

<http://blog.scoutapp.com/articles/2014/11/04/restricting-process-cpu-usage-using-nice-cpulimit-and-cgroups>

<http://blog.scoutapp.com/articles/2015/02/24/understanding-linuxs-cpu-stats>

## CPU idle

**Idle**, which means it has nothing to do.

the id statistic tell us that the **processor was idle** just over 73% of the time during the last sampling period.

Percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request.

- **Idle time** (for the whole system only) is the amount of time the CPU was not busy, or, otherwise, the amount of time it executed the [System Idle process](#). Idle time actually measures unused CPU capacity.

“System Idle Process” is the software that runs when the computer has absolutely nothing better to do.

It effectively runs at the lowest possible priority so that if anything, anything at all, comes along for the CPU to work on, it can. When there’s nothing left to do, back to idle it goes.

So having the System Idle Process using 90% of your CPU is a good thing ... it means that that 90% is readily available should there be any real work to do.

the **System Idle Process** contains one or more kernel **threads** which run when no other runnable thread can be scheduled on a CPU. In a multiprocessor system, there is one idle thread associated with each CPU core.

Modern processors use idle time to save power. Common methods are reducing the clock speed along with the CPU voltage and sending parts of the processor into a sleep state.

Most **operating systems** [which?] will display an **idle task**, which is a special task loaded by the OS **scheduler** only when there is nothing for the computer to do. The idle task can be hard-coded into the scheduler, or it can be implemented as a separate task with the lowest possible priority.

# IRQ

- Interrupts are signals sent across IRQ (Interrupt Request Line) by hardware or software.
- Interrupts allow devices like keyboard, serial cards and parallel ports to indicate that it needs CPU attention.
- Once the CPU receives the Interrupt Request, CPU will temporarily stop execution of running program and invoke a special program called Interrupt Handler or ISR (Interrupt Service Routine).
- The Interrupt Service or Interrupt Handler Routine can be found in Interrupt Vector table that is located at fixed address in the memory. After the interrupt is handled CPU resumes the interrupted program.
- At boot time, system identifies all devices, and appropriate interrupt handlers are loaded into the interrupt table.

## /proc/interrupts File

On a Linux machine, the file /proc/interrupts contains information about the interrupts in use and how many times processor has been interrupted

```
# cat /proc/interrupts
          CPU0   CPU1   CPU2   CPU3
 0: 3710374484      0      0      0  IO-APIC-edge  timer
 1:        20      0      0      0  IO-APIC-edge  i8042
 6:         5      0      0      0  IO-APIC-edge  floppy
 7:         0      0      0      0  IO-APIC-edge  parport0
 8:         0      0      0      0  IO-APIC-edge  rtc
 9:         0      0      0      0  IO-APIC-level acpi
 12:       240      0      0      0  IO-APIC-edge  i8042
 14: 112000026      0      0      0  IO-APIC-edge  ide0
 51: 61281329      0      0      0  IO-APIC-level ioc0
 59:         1      0      0      0  IO-APIC-level vmci
 67: 19386473      0      0      0  IO-APIC-level eth0
 75: 94595340      0      0      0  IO-APIC-level eth1
NMI:        0      0      0      0
LOC: 3737150067 3737142382 3737145101 3737144204
ERR:        0
MIS:        0
```

In the above file:

- The first Column is the IRQ number.
- The Second column says how many times the CPU core has been interrupted. In the above example timer is interrupt name [System clock] and 3710374484 is the number of times CPUo has been interrupted. I8042 is Keyboard controller that controls PS/2 keyboards and mouse in Pc's.
- For interrupt like rtc [Real time clock] CPU has not being interrupted. RTC are present in electronic devices to keep track of time.
- NMI and LOC are drivers used on system that are not accessible/configured by user.

IRQ number determines the priority of the interrupt that needs to be handled by the CPU.

A small IRQ number value means higher priority.

For example if CPU receives interrupt from Keyboard and system clock simultaneously. CPU will serve System Clock first since it has IRQ number 0.

- IRQ 0 – system timer (cannot be changed);
- IRQ 1 – keyboard controller (cannot be changed)
- IRQ 3 – serial port controller for serial port 2 (shared with serial port 4, if present);
- IRQ 4 – serial port controller for serial port 1 (shared with serial port 3, if present);
- IRQ 5 – parallel port 2 and 3 or sound card;
- IRQ 6 – floppy disk controller;
- IRQ 7 – parallel port 1. It is used for printers or for any parallel port if a printer is not present.

For devices like joystick CPU doesn't wait for the device to send interrupt. Since Joystick used for gaming and the movement of joystick will be fast it will be ideal to use polling and check whether device needs attention. The disadvantage behind this method is CPU can get into busy wait, checking the device many times.

## Hardware Interrupts

All of the above discussed scenarios are example of Hardware interrupts.

Hardware interrupts are further classified into two major categories:

1. Non-maskable interrupts [NMI]: As the name suggests these types of interrupts cannot be ignored or suppressed by the CPU. NMI's are sent over separate interrupt line and it's generally used for critical hardware errors like memory error, Hardware traps indicating Fan failure, Temperature Sensor failure etc.
2. Maskable interrupts: These interrupts can be ignored or delayed by CPU.

## Software Interrupts

These interrupts are generated when the CPU executes an instruction which can cause an exception condition in the CPU [ALU unit] itself.

For example, divide a number by zero which is not possible, it will lead to divide-by-zero exception, causing the computer to abandon the calculation or display an error message.

The file /proc/stat is also a file part of the [/proc filesystem](#), which has information about system kernel statistics, also holds some interrupt information.

```
# cat /proc/stat
cpu 17028082 5536753 5081493 1735530500 42592308 90006 479750 0
cpu0 5769176 1170683 1495750 403368354 39406374 90006 284864 0
cpu1 3714389 1451937 1186134 444082258 1084780 0 64876 0
cpu2 3791544 1471013 1211868 443988514 1056981 0 64764 0
cpu3 3752971 1443119 1187740 444091373 1044172 0 65244 0
intr 417756956 --- Output Truncated
```

The line intr shows the count of the interrupt serviced since boot time. The first column is total of all interrupts serviced. Each subsequent column is the total for a particular interrupt.

## IRQ Balance

Irqbalance is a Linux utility that distributes interrupts over the processor cores in your computer system which helps to improve performance.

Irqbalance goal is to find a balance between power saving and optimal performance.

```
# rpm -qa | grep irqbalance
```

```
irqbalance-0.55-15.el5
```

```
# yum search irqbalance
```

```
# yum install irqbalance.x86_64
```

Start the irqbalance service:

```
service irqbalance start
```

The following is a sample output from a Linux machine where irqbalance is installed. We could see that interrupts are now being distributed between CPUs.

```
# cat /proc/interrupts
          CPU0   CPU1   CPU2   CPU3
 0: 950901695      0      0      0  IO-APIC-edge  timer
 1:      13      0      0      0  IO-APIC-edge  i8042
 6:      96  10989      0      0  IO-APIC-edge  floppy
 7:      0      0      0      0  IO-APIC-edge  parport0
 8:      1      0      0      0  IO-APIC-edge  rtc
 9:      0      0      0      0  IO-APIC-level  acpi
12:     109    1787      0      0  IO-APIC-edge  i8042
15:     99 84813914      0      0  IO-APIC-edge  ide1
51: 17371      0 46689970      0  IO-APIC-level  ioc0
67:    1741      0      0 225409160 PCI-MSI  eth0
83:      0      0      0      0 PCI-MSI  vmci
NMI:      0      0      0      0
LOC: 950902917  950903742  950901202  950901400
ERR:      0
MIS:      0
```

Irqbalance is especially useful on systems with multi-core processors, as interrupts will typically only be serviced by the first core.

<http://www.thegeekstuff.com/2014/01/linux-interrupts/>

## Wait

```
wa -- iowait  
Amount of time the CPU has been waiting for I/O to complete.
```

The higher the number the more cpu resources are waiting for I/O access.

To identify whether I/O is causing system slowness you can use several commands but the easiest is the unix command **top**.

```
# top  
top - 14:31:20 up 35 min, 4 users, load average: 2.25, 1.74, 1.68  
Tasks: 71 total, 1 running, 70 sleeping, 0 stopped, 0 zombie  
Cpu(s): 2.3%us, 1.7%sy, 0.0%ni, 0.0%id, 96.0%wa, 0.0%hi, 0.0%si, 0.0%st
```

## Finding which disk is being written to

The above top command shows I/O Wait from the system as a whole but it does not tell you what disk is being affected; for this we will use the **iostat** command.

```
$ iostat -x 2 5
avg-cpu: %user %nice %system %iowait %steal %idle
3.66 0.00 47.64 48.69 0.00 0.00

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrrq-sz avgqu-sz await r_await w_await svctm %util
sda 44.50 39.27 117.28 29.32 11220.94 13126.70 332.17 65.77 462.79 9.80 2274.71 7.60 111.41
dm-0 0.00 0.00 83.25 9.95 10515.18 4295.29 317.84 57.01 648.54 16.73 5935.79 11.48 107.02
dm-1 0.00 0.00 57.07 40.84 228.27 163.35 8.00 93.84 979.61 13.94 2329.08 10.93 107.02
```

The iostat command in the example will print a report every 2 seconds for 5 intervals; the **-x** tells iostat to print out an extended report.

The 1st report from iostat will print statistics based on the last time the system was booted; **for this reason in most circumstances the first report from iostat should be ignored.**

Every sub-sequential report printed will be based on the time since the previous interval. For example in our command we will print a report 5 times, the 2nd report are disk statistics gathered since the 1st run of the report, the 3rd is based from the 2nd and so on.

Aside from %utilized there is a wealth of information in the output of iostat; items such as read and write requests per millisecond(rrqm/s & wrqm/s), reads and writes per second (r/s & w/s) and plenty more. In our example our program seems to be read and write heavy this information will be helpful when trying to identify the offending process.

# Finding the processes that are causing high I/O

**iostop**

```
# iostop
Total DISK READ: 8.00 M/s | Total DISK WRITE: 20.36 M/s
TID PRIO USER DISK READ DISK WRITE SWAPIN IO> COMMAND
15758 be/4 root 7.99 M/s 8.01 M/s 0.00 % 61.97 % bonnie++ -n 0 -u 0 -r 239 -s 478 -f -b -d /tmp
```

The simplest method of finding which process is utilizing storage the most is to use the command iostop. After looking at the statistics it is easy to identify bonnie++ as the process causing the most I/O utilization on this machine.

<http://bencane.com/2012/08/06/troubleshooting-high-io-wait-in-linux/>

<http://www.yellow-bricks.com/2012/07/17/why-is-wait-so-high/>

<http://veithen.github.io/2013/11/18/iowait-linux.html>

- **User time** is the amount of time the CPU was busy executing code in [user space](#).

**24.8 us** - This tells us that the processor is spending 24.8% of its time **running user space processes**. A user space program is any process that doesn't belong to the kernel. Shells, compilers, databases, web servers, and the programs associated with the desktop are all user space processes. If the processor isn't idle, it is quite normal that the majority of the CPU time should be spent running user space processes.

- **System time** is the amount of time the CPU was busy executing code in [kernel space](#).

**Kernel space** is strictly reserved for running a privileged [operating system kernel](#), kernel extensions, and most [device drivers](#). In contrast, user space is the memory area where [application software](#) and some drivers execute.

**0.5 sy** - This is the amount of time that the CPU spent **running the kernel**. All the processes and system resources are handled by the Linux kernel. When a user space process needs something from the system, for example when it needs to allocate memory, perform some I/O, or it needs to create a child process, then the kernel is running. In fact the scheduler itself which determines which process runs next is part of the kernel. The amount of time spent in the kernel should be as low as possible. In this case, just 0.5% of the time given to the different processes was spent in the kernel. This number can peak much higher, especially when there is a lot of I/O happening.

## Process

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

## Parent Process

In the operating system Unix, every process except process 0 (the swapper) is created when another process executes the `fork()` system call. The **process that invoked `fork` is the parent process** and the newly-created process is the child process. Every process (except process 0) has one parent process, but can have many child processes.

- The operating system kernel identifies each process by its process identifier.
- Process 0 is a special process that is created when the system boots
- after forking a child process (process 1), process 0 becomes the swapper process (sometimes also known as the “idle task”).

- Process 1, known as `init`, is the ancestor of every other process in the system.

## Child process

A child process in computing is a **process created by another process (the parent process)**.

A child process inherits most of its attributes, such as open files, from its parent. In UNIX, a child process is in fact created (using `fork`) as a copy of the parent

- if a process does not have a parent this usually indicates that it was created directly by the kernel.
- System call `fork()` is used to create processes. The purpose of `fork()` is to create a new process, which becomes the child process of the caller.

## Zombie Process

a **zombie process or defunct process is a process that has completed execution but still has an entry in the process table**.

- This entry is still needed to allow the parent process to read its child's exit status.
- the kill command has no effect on a zombie process.
- When a process dies on Linux, it isn't all removed from memory immediately — its process descriptor stays in memory (the process descriptor only takes a tiny amount of memory).
- The process's status becomes EXIT\_ZOMBIE and the process's parent is notified that its child process has died with the SIGCHLD signal.
- The parent process is then supposed to execute the wait() system call to read the dead process's exit status and other information.
- this allows the parent process to get information from the dead process. After wait() is called, the zombie process is completely removed from memory.
- This normally happens very quickly, so you won't see zombie processes accumulating on your system. However, if a parent process isn't programmed properly and never calls wait(), its zombie children will stick around in memory until they're cleaned up.

```
top - 22:47:18 up 33 min, 2 users, load average: 3.61, 1.51, 0.86
Tasks: 201 total, 7 running, 157 sleeping, 0 stopped, 37 zombie
Cpu(s): 13.5%us, 7.7%sy, 0.0%ni, 78.8%id, 0.0%wa, 0.0%hi, 0.0%si,
Mem: 1024792k total, 940672k used, 84120k free, 84300k buffers
Swap: 1046524k total, 2388k used, 1044136k free, 428356k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1191	root	20	0	106m	70m	11m	S	11.2	7.0	0:38.93	Xorg
2349	howtogee	20	0	72636	12m	10m	R	3.7	1.3	0:02.59	metacity
2381	howtogee	20	0	50276	9564	7492	S	3.7	0.9	0:03.69	bamfdaemon
2959	howtogee	20	0	90892	24m	18m	R	3.7	2.4	2:23.76	gnome-syste
3551	howtogee	20	0	91544	15m	11m	R	1.9	1.6	0:01.42	gnome-termi
3772	howtogee	20	0	2836	1184	880	R	1.9	0.1	0:00.15	top
1	root	20	0	3628	1708	1348	S	0.0	0.2	0:02.76	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.17	ksoftirqd/0
5	root	20	0	0	0	0	S	0.0	0.0	0:01.11	kworker/u:6

## Dangers of Zombie Processes

- each zombie process retains its process ID (PID).
- Linux systems have a finite number of process IDs – 32767 by default on 32-bit systems.
- If zombies are accumulating at a very quick rate

the entire pool of available PIDs will eventually become assigned to zombie processes, preventing other processes from launching.

## Getting Rid of Zombie Processes

One way is by sending the SIGCHLD signal to the parent process. This signal tells the parent process to execute the wait() system call and clean up its zombie children. Send the signal with the **kill** command, replacing *pid* in the command below with the parent process PID:

```
kill -s SIGCHLD pid
```

# Sleeping State

- A process enters a Sleeping state when it needs resources that are not currently available.
- At that point, it either goes voluntarily into Sleep state or the kernel puts it into Sleep state.
- Going into Sleep state means the process immediately gives up its access to the CPU.
- When the resource the process is waiting on becomes available a signal is sent to the CPU.
- The next time the scheduler gets a chance to schedule this sleeping process, the scheduler will put the process either in Running or Runnable state.

Here is an example of how a login shell goes in and out of sleep state:

- You type a command and the shell goes into Sleep state and waits for an event to occur.
- The shell process sleeps on a particular wait channel (WCHAN).
- When an event occurs, such as an interrupt from the keyboard, every process waiting on that wait channel wakes up.

To find out what wait channels processes are waiting on for your system, type

**ps -l** (to see processes associated with the current shell)

**ps -el**

(to see all processes on the system). If a process is in Sleep state, the WCHAN field shows the system call that the process is waiting on

- The Linux kernel uses the **sleep()** function, which takes a time value as a parameter that specifies the minimum amount of time (in seconds that the process is set to sleep before resuming execution)
- This causes the CPU to suspend the process and continue executing other processes until the sleep cycle has finished.

There are two types of sleep states: **Interruptible** and **Uninterruptible** sleep states

### **Interruptible Sleep State**

An Interruptible sleep state means the process is waiting either for a particular time slot or for a particular event to occur.

- Output from the ps command will show as S in the state field

### **Uninterruptible Sleep State**

It will wake only as a result of a waited-upon resource becoming available or after a time-out occurs during that wait (if the time-out is specified when the process is put to sleep).

- The Uninterruptible state is mostly used by device drivers waiting for disk or network I/O.
- the command **ps -l** uses the letter D in the state field (S) to indicate that the process is in an Uninterruptible sleep state

## Running State

The process that is executing and using the CPU at a particular moment is called a running process.

You can run the ps and top commands to see the state of each process. If a process is running, the Running state is shown as R in the state field.

### How a process reaches a Running state.

When you fire off a command such as ls , a shell (bash)searches the directories in the search path stored in the PATH environment variable to find where the ls command is located.

- Once the ls file is found, the shell clones itself using the forking method
- then the new child process replaces the binary image it was executing (the shell) with the ls command's executable binary image

[https://access.redhat.com/sites/default/files/attachments/processstates\\_20120831.pdf](https://access.redhat.com/sites/default/files/attachments/processstates_20120831.pdf)

<http://www.linux-tutorial.info/modules.php?name=MContent&pageid=84>

These running processes run in the following spaces:

- In user space
- In system space

- When a user initiates a process, the process starts working in user mode. That user mode process does not have access to kernel data structures or algorithms .
- Each CPU type provides special instructions to switch from user mode to kernel mode. If a user-level process wants to access kernel data structures or algorithms, then it requests that information through system calls that deal with the file subsystem or the process control subsystem. Examples of these system
- calls include:

**File subsystem system calls:**open(),close(),read(),write(),chmod(),chown()

**Process control system calls:**fork(),exec(),exit(),wait(),brk(),signal()