



[Our blog](#) - musings on Ruby performance and beautiful data.

Write faster Ruby.

Learn how to build Ruby apps at scale. We'll deliver a curated selection of optimization tips right to your inbox each month.

Recent Posts

- [Introducing free app monitoring for Elixir apps](#)
- [Alerting: get notified when performance goes bad](#)
- [RailsConf 2017: 5 standout performance sessions](#)
- [The danger of Rails.env.production?](#)
- [Overhead Benchmarks: New Relic vs. Scout](#)

Restricting process CPU usage using nice, cpulimit, and cgroups

November 04 • By Derek • Posted in [HowTo](#) • [Comments](#)



The Linux kernel is an incredible circus performer, carefully juggling many processes and their resource needs to keep your server humming along. The kernel is also all about equity: when there is competition for resources, the kernel tries to distribute those resources fairly.

However, what if you've got an important process that needs priority? What about a low-priority process? Or what about limiting resources for a group of processes?

The kernel can't determine what CPU processes are important without your help.

Most processes are started at the same priority level and the Linux kernel schedules time for each task evenly on the processor. Have a CPU intensive process that can be run at a lower priority? Then you need to tell the scheduler about it!

There are at least three ways in which you can control how much CPU time a process gets:

- Use the **nice** command to manually lower the task's priority.
- Use the **cpulimit** command to repeatedly pause the process so that it doesn't exceed a certain limit.
- Use Linux's built-in **control groups**, a mechanism which tells the scheduler to limit the amount of resources available to the process.

Let's look at how these work and the pros and cons of each.

Simulating high CPU usage

Before looking at these three techniques, we need to find a tool that will simulate high CPU usage on a system. We will be using CentOS as our base system, and to artificially load the processor we can use the prime number generator from the [Mathomatic toolkit](http://mathomatic.orgserve.de/mathomatic-16.0.5.tar.bz2).

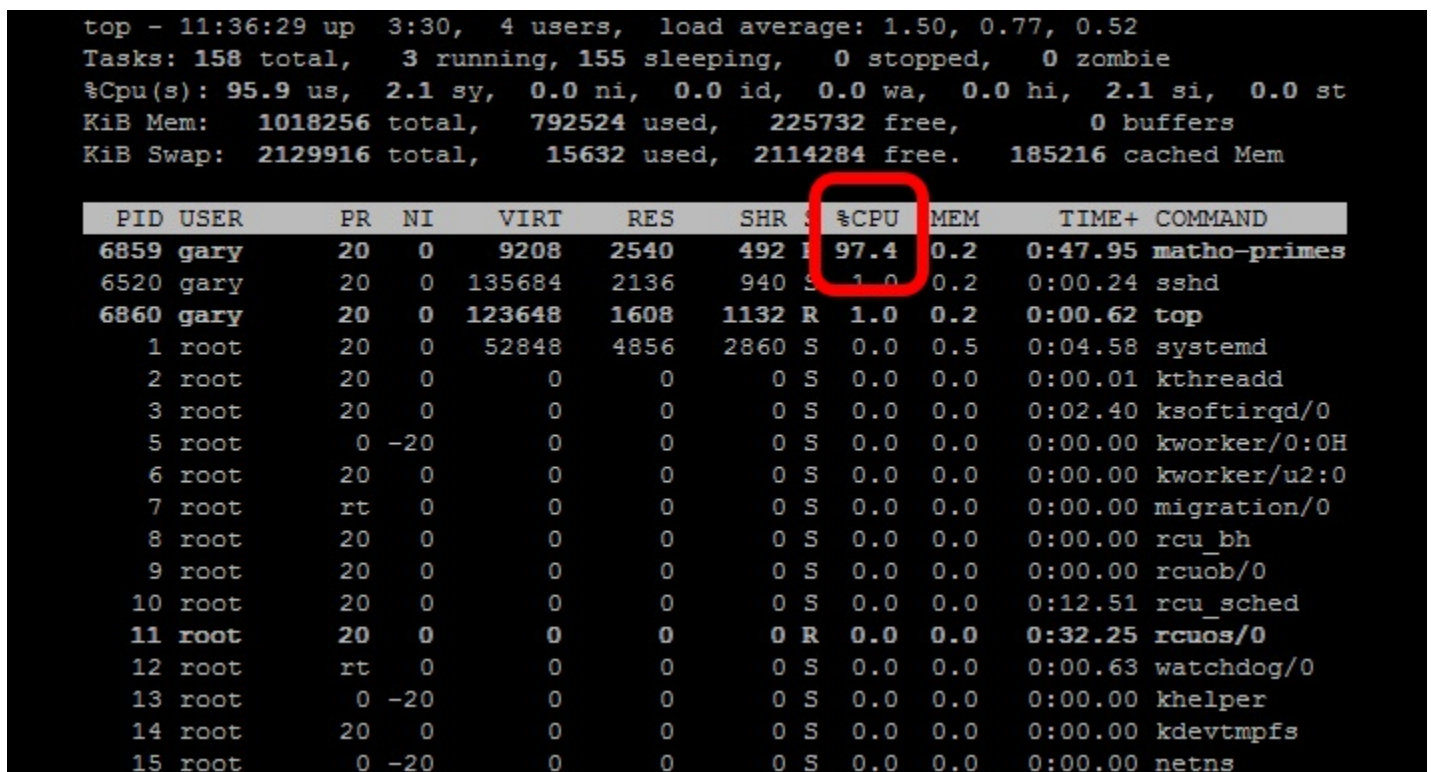
There isn't a prebuilt package for CentOS so you will need to build it yourself. Download the source code from <http://mathomatic.orgserve.de/mathomatic-16.0.5.tar.bz2> and then unpack the archive file. Change directory into `mathomatic-16.0.5/primes`. Run `make` and `sudo make install` to build and install the binaries. You will now have the `matho-primes` binary in `/usr/local/bin`.

Run the command like this:

```
/usr/local/bin/matho-primes 0 9999999999 > /dev/null &
```

This will generate a list of prime numbers from zero to nine billion nine hundred ninety-nine million nine hundred ninety-nine thousand nine hundred ninety-nine. Since we don't really want to keep the list, the output is redirected to `/dev/null`.

Now run `top` and you will see that the `matho-primes` process is using all the available CPU.



```
top - 11:36:29 up 3:30, 4 users, load average: 1.50, 0.77, 0.52
Tasks: 158 total, 3 running, 155 sleeping, 0 stopped, 0 zombie
%Cpu(s): 95.9 us, 2.1 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 2.1 si, 0.0 st
KiB Mem: 1018256 total, 792524 used, 225732 free, 0 buffers
KiB Swap: 2129916 total, 15632 used, 2114284 free. 185216 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	%CPU	MEM	TIME+	COMMAND
6859	gary	20	0	9208	2540	492	97.4	0.2	0:47.95	matho-primes
6520	gary	20	0	135684	2136	940	1.0	0.2	0:00.24	sshd
6860	gary	20	0	123648	1608	1132	1.0	0.2	0:00.62	top
1	root	20	0	52848	4856	2860	0.0	0.5	0:04.58	systemd
2	root	20	0	0	0	0	0.0	0.0	0:00.01	kthreadd
3	root	20	0	0	0	0	0.0	0.0	0:02.40	ksoftirqd/0
5	root	0	-20	0	0	0	0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	0.0	0.0	0:00.00	kworker/u2:0
7	root	rt	0	0	0	0	0.0	0.0	0:00.00	migration/0
8	root	20	0	0	0	0	0.0	0.0	0:00.00	rcu_bh
9	root	20	0	0	0	0	0.0	0.0	0:00.00	rcuob/0
10	root	20	0	0	0	0	0.0	0.0	0:12.51	rcu_sched
11	root	20	0	0	0	0	0.0	0.0	0:32.25	rcuos/0
12	root	rt	0	0	0	0	0.0	0.0	0:00.63	watchdog/0
13	root	0	-20	0	0	0	0.0	0.0	0:00.00	khelper
14	root	20	0	0	0	0	0.0	0.0	0:00.00	kdevtmpfs
15	root	0	-20	0	0	0	0.0	0.0	0:00.00	netns

Exit `top` (press the `q` key) and kill the `matho-primes` process (`fg` to bring the process to the foreground and press `CTRL+C`).

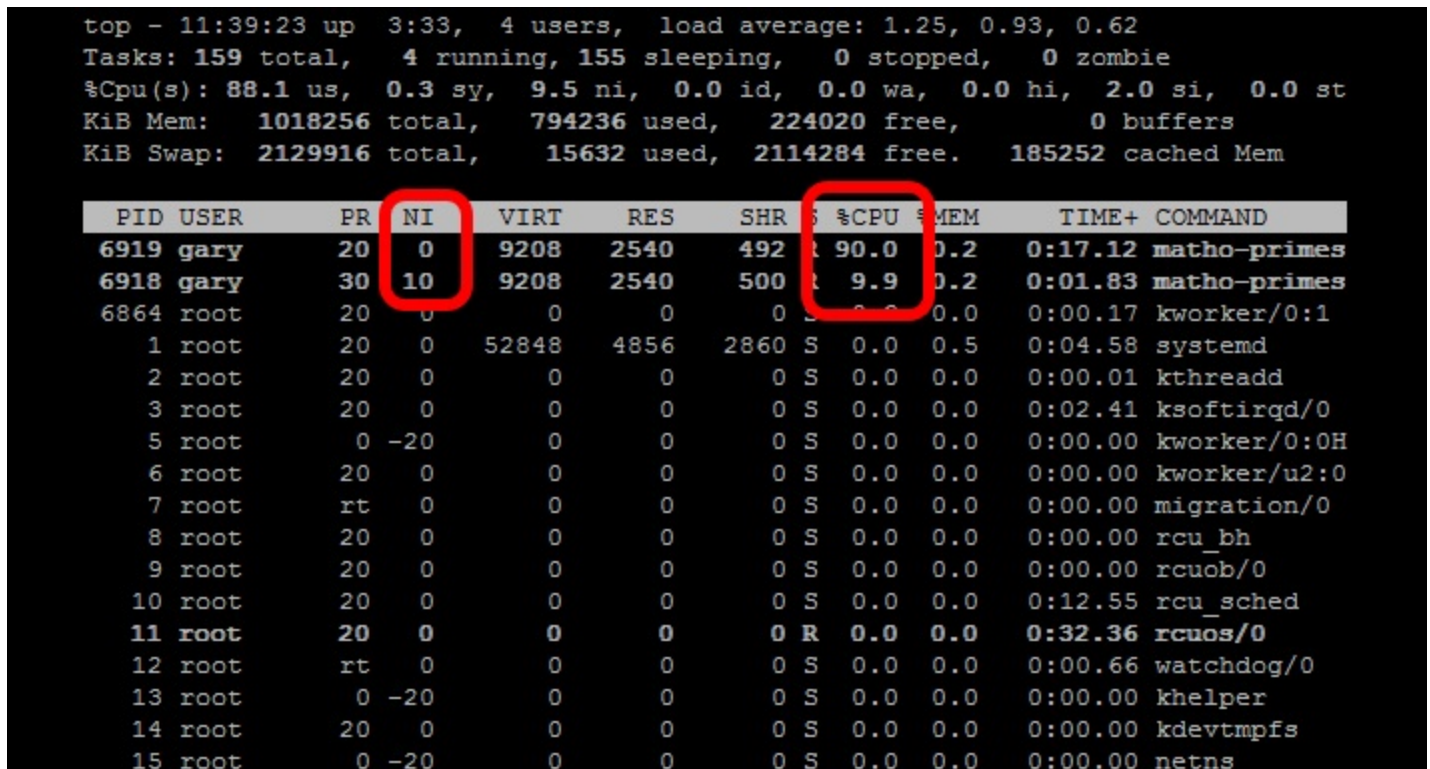
nice

The `nice` command tweaks the priority level of a process so that it runs less frequently. **This is useful when you need to run a CPU intensive task as a background or batch job.** The niceness level ranges from -20 (most favorable scheduling) to 19 (least favorable). 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:

```
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	0.0	0.5	0:04.58	systemd
2	root	20	0	0	0	0	0.0	0.0	0:00.01	kthreadd
3	root	20	0	0	0	0	0.0	0.0	0:02.41	ksoftirqd/0
5	root	0	-20	0	0	0	0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	0.0	0.0	0:00.00	kworker/u2:0
7	root	rt	0	0	0	0	0.0	0.0	0:00.00	migration/0
8	root	20	0	0	0	0	0.0	0.0	0:00.00	rcu_bh
9	root	20	0	0	0	0	0.0	0.0	0:00.00	rcuob/0
10	root	20	0	0	0	0	0.0	0.0	0:12.55	rcu_sched
11	root	20	0	0	0	0	0.0	0.0	0:32.36	rcuos/0
12	root	rt	0	0	0	0	0.0	0.0	0:00.66	watchdog/0
13	root	0	-20	0	0	0	0.0	0.0	0:00.00	khelper
14	root	20	0	0	0	0	0.0	0.0	0:00.00	kdevtmpfs
15	root	0	-20	0	0	0	0.0	0.0	0:00.00	netns

Observe that the process started without `nice` (at niceness level 0) gets more processor time, whereas the process with a niceness level of 10 gets less.

What this means in real terms is that if you want to run a CPU intensive task you can start it using `nice` and the scheduler will always ensure that other tasks have priority over it. This means that the server (or desktop) will remain responsive even when under heavy load.

`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.

Don't forget to kill the `matho-primes` processes once you have finished experimenting with the `nice` and `renice` commands.

cpulimit

The `cpulimit` tool curbs the CPU usage of a process by pausing the process at different intervals to keep it under the defined ceiling. It does this by sending `SIGSTOP` and `SIGCONT` signals to the process. It does not change the `nice` value of the process, instead it monitors and controls the real-world CPU usage.

cpulimit is useful when you want to ensure that a process doesn't use more than a certain portion of the CPU. The disadvantage over *nice* is that the process can't use all of the available CPU time when the system is idle.

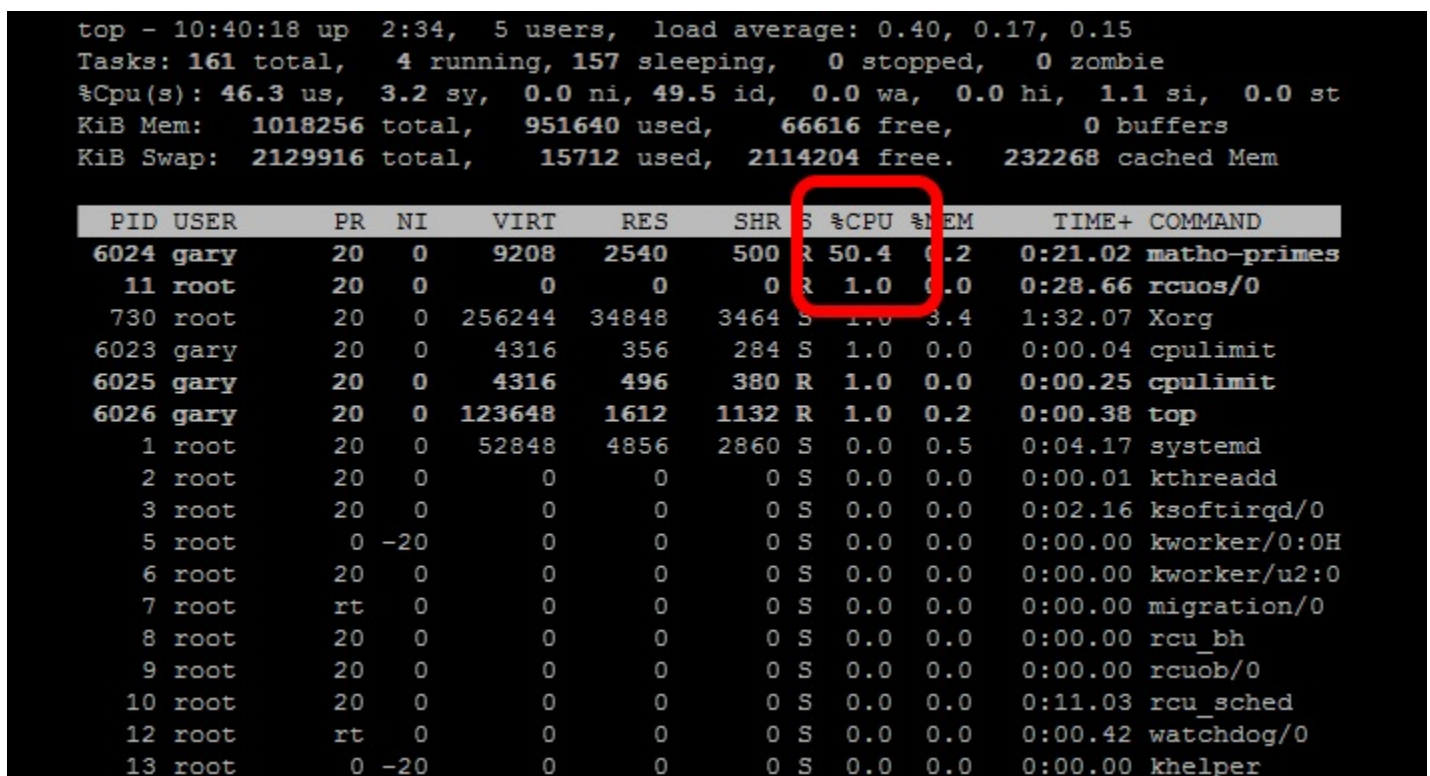
To install it on CentOS type:

```
wget -O cpulimit.zip https://github.com/opsengine/cpulimit/archive/master.zip
unzip cpulimit.zip
cd cpulimit-master
make
sudo cp src/cpulimit /usr/bin
```

The commands above will download the source code from GitHub, unpack the archive file, build the binary, and copy it to `/usr/bin`.

cpulimit is used in a similar way to *nice*, however you need to explicitly define the maximum CPU limit for the process using the `-l` parameter. For example:

```
cpulimit -l 50 matho-primes 0 999999999 > /dev/null &
```



```
top - 10:40:18 up 2:34, 5 users, load average: 0.40, 0.17, 0.15
Tasks: 161 total, 4 running, 157 sleeping, 0 stopped, 0 zombie
%Cpu(s): 46.3 us, 3.2 sy, 0.0 ni, 49.5 id, 0.0 wa, 0.0 hi, 1.1 si, 0.0 st
KiB Mem: 1018256 total, 951640 used, 66616 free, 0 buffers
KiB Swap: 2129916 total, 15712 used, 2114204 free. 232268 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
6024	gary	20	0	9208	2540	500	R	50.4	0.2	0:21.02	matho-primes
11	root	20	0	0	0	0	R	1.0	0.0	0:28.66	rcuos/0
730	root	20	0	256244	34848	3464	S	1.0	3.4	1:32.07	Xorg
6023	gary	20	0	4316	356	284	S	1.0	0.0	0:00.04	cpulimit
6025	gary	20	0	4316	496	380	R	1.0	0.0	0:00.25	cpulimit
6026	gary	20	0	123648	1612	1132	R	1.0	0.2	0:00.38	top
1	root	20	0	52848	4856	2860	S	0.0	0.5	0:04.17	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	0:02.16	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	0:11.03	rcu_sched
12	root	rt	0	0	0	0	S	0.0	0.0	0:00.42	watchdog/0
13	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	khelper

Note how the *matho-primes* process is now only using 50% of the available CPU time. On my example system the rest of the time is spent in idle.

You can also limit a currently running process by specifying its PID using the `-p` parameter. For example

```
cpulimit -l 50 -p 1234
```

Where 1234 is the PID of the process.

cgroups

Control groups (cgroups) are a Linux kernel feature that allows you to specify how the kernel should allocate specific resources to a group of processes. With cgroups you can specify how much CPU time, system memory, network bandwidth, or combinations of these resources can be used by the processes residing in a certain group.

The advantage of control groups over *nice* or *cpulimit* is that the limits are applied to a set of processes, rather than to just one. Also, *nice* or *cpulimit* only limit the CPU usage of a process, whereas cgroups can limit

other process resources.

By judiciously using cgroups the resources of entire subsystems of a server can be controlled. For example in CoreOS, the minimal Linux distribution designed for massive server deployments, the upgrade processes are controlled by a cgroup. This means the downloading and installing of system updates doesn't affect system performance.

To demonstrate cgroups, we will create two groups with different CPU resources allocated to each group. The groups will be called 'cpulimited' and 'lesscpulimited'.

The groups are created with the cgcreate command like this:

```
sudo cgcreate -g cpu:/cpulimited
sudo cgcreate -g cpu:/lesscpulimited
```

The "-g cpu" part of the command tell cgroups that the groups can place limits on the amount of CPU resources given to the processes in the group. Other controllers include cpuset, memory, and blkio. The cpuset controller is related to the cpu controller in that it allows the processes in a group to be bound to a specific CPU, or set of cores in a CPU.

The cpu controller has a property known as cpu.shares. It is used by the kernel to determine the share of CPU resources available to each process across the cgroups. The default value is 1024. By leaving one group (lesscpulimited) at the default of 1024 and setting the other (cpulimited) to 512, we are telling the kernel to split the CPU resources using a 2:1 ratio.

To set the cpu.shares to 512 in the cpulimited group, type:

```
sudo cgset -r cpu.shares=512 cpulimited
```

To start a task in a particular cgroup you can use the cgexec command. To test the two cgroups, start matho-primes in the cpulimited group, like this:

```
sudo cgexec -g cpu:cpulimited /usr/local/bin/matho-primes 0 9999999999 > /dev/null &
```

If you run top you will see that the process is taking all of the available CPU time.

top - 09:02:57 up 56 min, 5 users, load average: 1.26, 0.89, 0.85
 Tasks: 163 total, 3 running, 160 sleeping, 0 stopped, 0 zombie
 %Cpu(s): 96.0 us, 2.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 2.0 si, 0.0 st
 KiB Mem: 1018256 total, 929208 used, 89048 free, 0 buffers
 KiB Swap: 2129916 total, 6976 used, 2122940 free. 306096 cached Mem

PID	USER	PR	NI	VIRT	RES	SHR	%CPU	%MEM	TIME+	COMMAND
4219	root	20	0	9208	2540	492 R	98.1	0.2	4:15.23	matho-primes
3510	gary	20	0	123640	1656	1172 S	0.0	0.2	0:10.78	top
4246	gary	20	0	135684	2264	1064 S	0.0	0.2	0:01.16	sshd
4327	gary	20	0	123680	1664	1132 R	0.0	0.2	0:01.10	top
1	root	20	0	52848	4932	2892 S	0.0	0.5	0:03.35	systemd
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	0:00.51	ksoftirqd/0
5	root	0	-20	0	0	0 S	0.0	0.0	0:00.00	0:0H
6	root	20	0	0	0	0 S	0.0	0.0	0:00.00	2:0
7	root	rt	0	0	0	0 S	0.0	0.0	0:00.00	/0
8	root	20	0	0	0	0 S	0.0	0.0	0:00.00	
9	root	20	0	0	0	0 S	0.0	0.0	0:00.00	
10	root	20	0	0	0	0 S	0.0	0.0	0:00.00	
11	root	20	0	0	0	0 S	0.0	0.0	0:00.00	
12	root	rt	0	0	0	0 S	0.0	0.0	0:00.21	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

Only one process running across all the control groups.

This is because when a single process is running, it uses as much CPU as necessary, regardless of which cgroup it is placed in. The CPU limitation only comes into effect when two or more processes compete for CPU resources.

Now start a second matho-primes process, this time in the lesscpulimited group:

```
sudo cgexec -g cpu:lesscpulimited /usr/local/bin/matho-primes 0 9999999999 > /dev/null &
```

The top command shows us that the process in the cgroup with the greater cpu.shares value is getting more CPU time.

```
top - 09:03:59 up 58 min,  5 users,  load average: 1.53, 1.02, 0.90
Tasks: 164 total,  4 running, 160 sleeping,  0 stopped,  0 zombie
%Cpu(s): 99.0 us,  1.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  1018256 total,  932108 used,   86148 free,    0 buffers
KiB Swap: 2129916 total,   6976 used, 2122940 free. 306100 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	%CPU	%MEM	TIME+	COMMAND
4502	root	20	0	9208	2540	492	65.2	0.2	0:12.86	matho-primes
4219	root	20	0	9208	2540	492	32.6	0.2	5:02.29	matho-primes
1	root	20	0	52848	4932	2892	0.0	0.5	0:03.35	systemd
2	root	20	0	0	0	0	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	0.0	0.0	0:00.51	ksoftirqd/0
5	root	0	-20	0	0	0	0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	0.0	0.0	0:00.00	kworker/u2:0
7	root	rt	0	0	0	0	0.0	0.0	0:00.00	migration/0
8	root	20	0	0	0	0	0.0	0.0	0:00.00	rcu_bh
9	root	20	0	0	0	0	0.0	0.0	0:00.00	rcuob/0
10	root	20	0	0	0	0	0.0	0.0	0:03.08	rcu_sched
11	root	20	0	0	0	0	0.0	0.0	0:06.90	rcuos/0
12	root	rt	0	0	0	0	0.0	0.0	0:00.22	watchdog/0
13	root	0	-20	0	0	0	0.0	0.0	0:00.00	khelper
14	root	20	0	0	0	0	0.0	0.0	0:00.00	kdevtmpfs
15	root	0	-20	0	0	0	0.0	0.0	0:00.00	netns
16	root	0	-20	0	0	0	0.0	0.0	0:00.00	writeback

2:1 ratio

Now start another matho-primes process in the cpulimited group:

```
sudo cgexec -g cpu:cpulimited /usr/local/bin/matho-primes 0 9999999999 > /dev/null &
```

```
top - 09:04:39 up 58 min,  5 users,  load average: 1.96, 1.19, 0.96
Tasks: 166 total,  5 running, 161 sleeping,  0 stopped,  0 zombie
%Cpu(s): 94.8 us,  2.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  3.1 si,  0.0 st
KiB Mem:  1018256 total,  935720 used,   82536 free,    0 buffers
KiB Swap: 2129916 total,   6976 used, 2122940 free. 306100 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	%CPU	%MEM	TIME+	COMMAND
4502	root	20	0	9208	2540	492	65.5	0.2	0:39.21	matho-primes
4219	root	20	0	9208	2540	492	14.9	0.2	5:13.03	matho-primes
4504	root	20	0	9208	2540	492	14.9	0.2	0:02.12	matho-primes
2248	gary	20	0	628388	17724	10476	1.0	1.1	0:10.69	gnome-termi+
3510	gary	20	0	123640	1656	1172	1.0	0.2	0:11.41	top
1	root	20	0	52848	4932	2892	0.0	0.5	0:03.35	systemd
2	root	20	0	0	0	0	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	0.0	0.0	0:00.51	ksoftirqd/0
5	root	0	-20	0	0	0	0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	0.0	0.0	0:00.00	kworker/u2:0
7	root	rt	0	0	0	0	0.0	0.0	0:00.00	migration/0
8	root	20	0	0	0	0	0.0	0.0	0:00.00	rcu_bh
9	root	20	0	0	0	0	0.0	0.0	0:00.00	rcuob/0
10	root	20	0	0	0	0	0.0	0.0	0:03.10	rcu_sched
11	root	20	0	0	0	0	0.0	0.0	0:06.93	rcuos/0
12	root	rt	0	0	0	0	0.0	0.0	0:00.22	watchdog/0
13	root	0	-20	0	0	0	0.0	0.0	0:00.00	khelper

Same 2:1 ratio, split among 3 processes