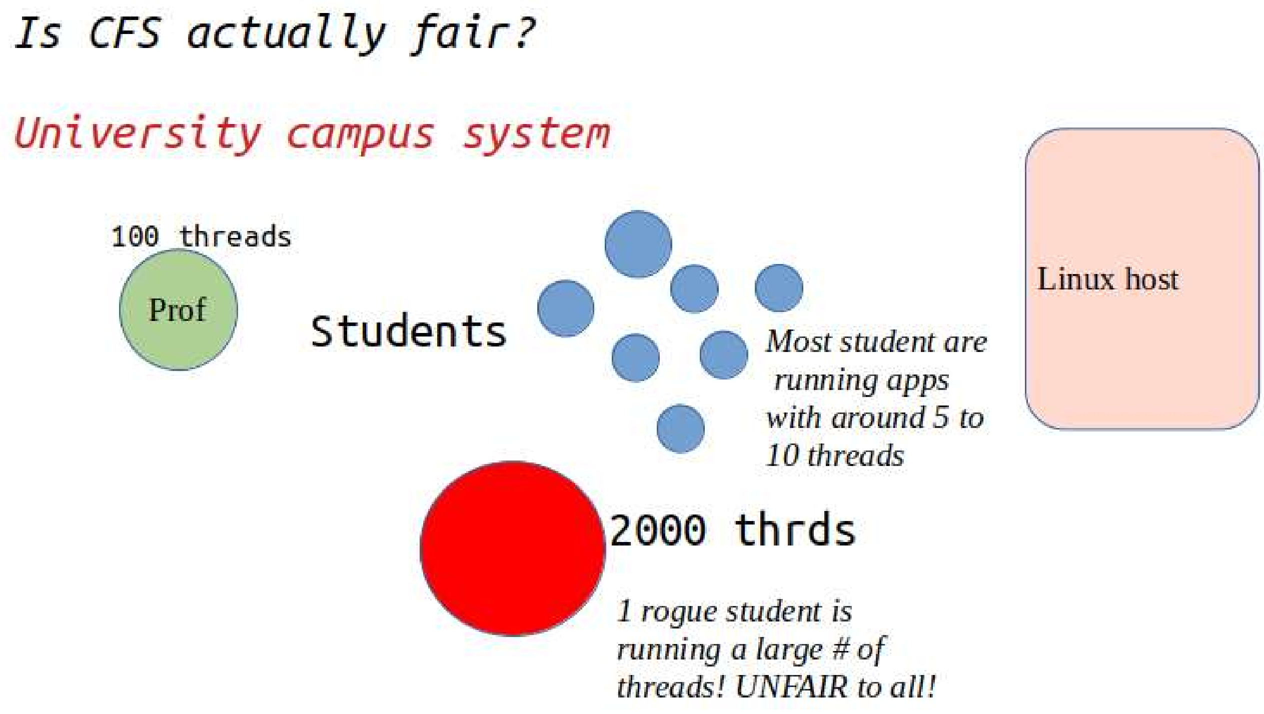
Control Groups - cgroups

Motivation



Is CFS actually fair?

University campus system

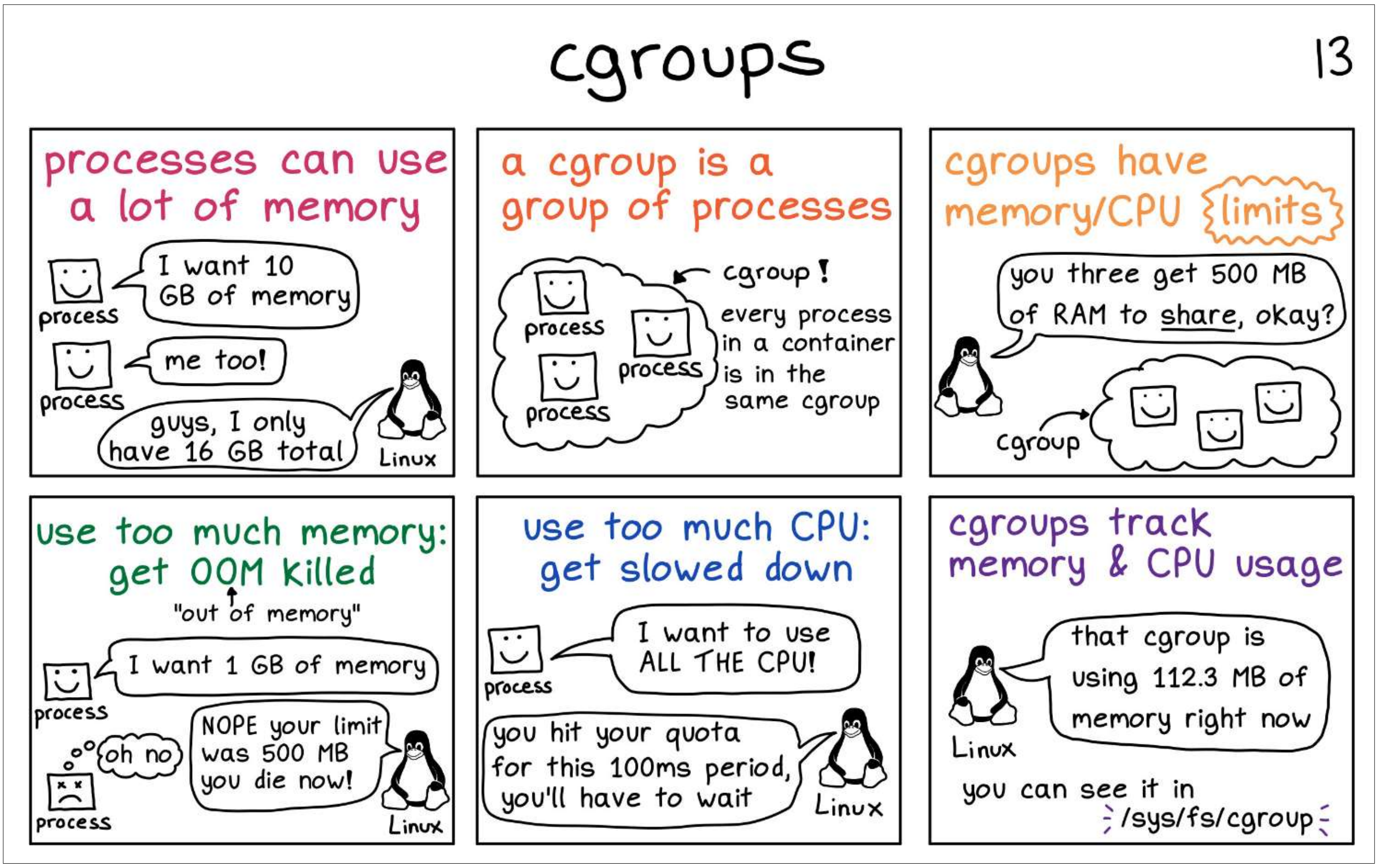
100 threads
Prof
Students
Most student are running apps with around 5 to 10 threads
Linux host
2000 thrds
1 rogue student is running a large # of threads! UNFAIR to all!

Source

...

A cgroup is a logical grouping of processes that can be used for resource management in the kernel. Once a cgroup has been created, processes can be migrated in and out of the cgroup via a pseudo-filesystem API (details can be found in the kernel source file Documentation/cgroups/cgroups.txt).

Src: Julia Evans – sample [ezine page on Cgroups]


cgroups
13
processes can use a lot of memory
process
I want 10 GB of memory
process
me too!
guys, I only have 16 GB total
Linux
a cgroup is a group of processes
cgroup!
every process in a container is in the same cgroup
process
process
process
cgroups have memory/CPU limits
you three get 500 MB of RAM to share, okay?
cgroup
use too much memory: get OOM killed
"out of memory"
I want 1 GB of memory
process
oh no
NOPE your limit was 500 MB you die now!
process
Linux
use too much CPU: get slowed down
I want to use ALL THE CPU!
process
you hit your quota for this 100ms period, you'll have to wait
Linux
cgroups track memory & CPU usage
that cgroup is using 112.3 MB of memory right now
Linux
you can see it in /sys/fs/cgroup

Resource usage within cgroups is managed by attaching controllers to a cgroup. Glauber briefly looked at two of these controllers.

<<

See all the Cgroups:

$ cat /proc/cgroups
…

Modern:

```
$ cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
```

```
$ lssubsys        << subsystem is another name for a resource controller >>
cpuset
cpu,cpuacct          << slink 'cpu' points to this >>
memory
devices
```

```
freezer
net_cls,net_prio
blkio
perf_event
hugetlb
$
```

FYI, the *lssubsys* utility is part of the *cgroup-tools* package on Debian/Ubuntu, and the *libcgroup-tools* package on SuSe
>>

The CPU controller mechanism allows a system manager to control the percentage of CPU time given to a cgroup. The CPU controller can be used both to guarantee that a cgroup gets a guaranteed minimum percentage of CPU on the system, regardless of other load on the system, and also to set an upper limit on the amount of CPU time used by a cgroup, so that a rogue process can't consume all of the available CPU time.

CPU scheduling is first of all done at the cgroup level, and then across the processes within each cgroup. As with some other controllers, CPU cgroups can be nested, so that the percentage of CPU time allocated to a top-level cgroup can be further subdivided across cgroups under that top-level cgroup.

The memory controller mechanism can be used to limit the amount of memory that a process uses. If a rogue process runs over the limit set by the controller, the kernel will page out *that* process, rather than some other process on the system.

...

Resources

See <*kernel-src-tree*>/*Documentation/cgroups*

```
$ ls git-kernel/Documentation/cgroups/ |col
00-INDEX
blkio-controller.txt
cgroups.txt
cpuacct.txt
cpusets.txt
devices.txt
freezer-subsystem.txt
hugetlb.txt
memcg_test.txt
memory.txt
net_cls.txt
net_prio.txt
unified-hierarchy.txt
```

$

https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

Control groups series by Neil Brown on LWN

Resource management: Linux kernel Namespaces and cgroups, Rami Rosen [PDF]

Similar to below: Fedora documentation: http://docs.fedoraproject.org/en-US/Fedora/17/html-single/Resource_Management_Guide/index.html#sec-How_Control_Groups_Are_Organized

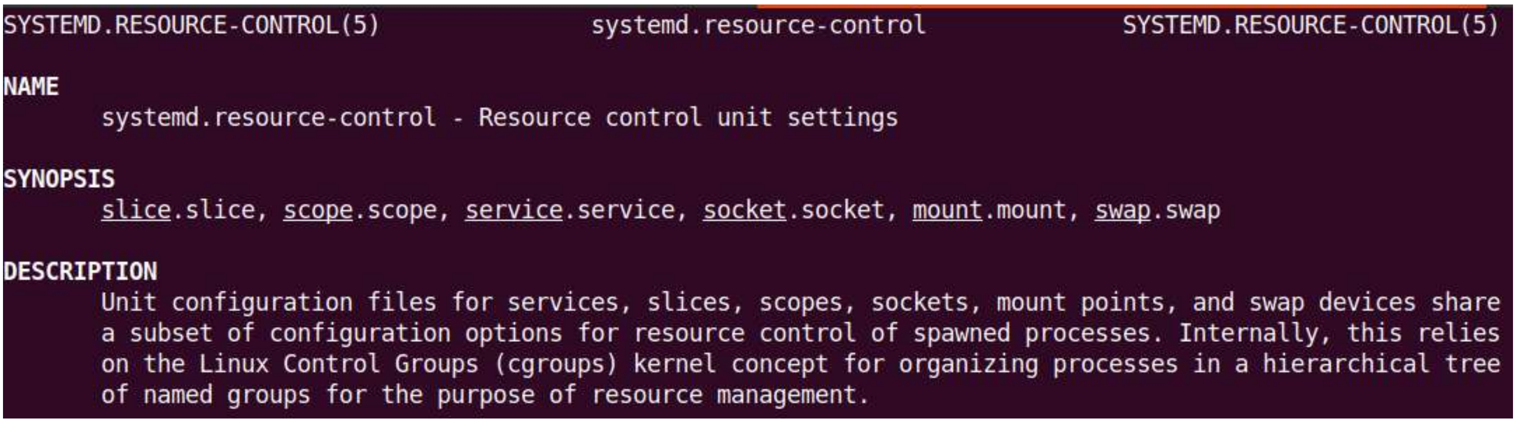
New! Cgroups v2 ; only fully merged in 4.15
A milestone for control groups, Jon Corbet, LWN, 31 July 2017

"… When Heo first raised the issue of fixing the control-group interface in 2012, he identified what he saw as two key problems: the ability to create multiple control-group hierarchies and allowing a control group to contain both processes and other control groups. Both interface features complicated the implementation of controllers, especially in cases where multiple controllers need to be able to cooperate with each other. His proposal was that the new ("V2") control-group API should dispense with these features.

Fast-forward to 2017, and those changes have been made. The V2 interface supports a single control-group hierarchy, and it requires that processes only appear in the leaf nodes of that hierarchy. ..."

Systemd : can specify cgroup parameters when launching services at boot!

```
man systemd.resource-control
```



```
SYSTEMD.RESOURCE-CONTROL(5)          systemd.resource-control          SYSTEMD.RESOURCE-CONTROL(5)

NAME
       systemd.resource-control - Resource control unit settings

SYNOPSIS
       slice.slice, scope.scope, service.service, socket.socket, mount.mount, swap.swap

DESCRIPTION
       Unit configuration files for services, slices, scopes, sockets, mount points, and swap devices share
       a subset of configuration options for resource control of spawned processes. Internally, this relies
       on the Linux Control Groups (cgroups) kernel concept for organizing processes in a hierarchical tree
       of named groups for the purpose of resource management.
```

Source (below): RHEL 7 Resource Management Guide :

[Note: this material below is taken from the public RHEL 7 Resource Guides; thus, it pertains specifically to RHEL 7].

1.1. What are Control Groups

The *control groups*, abbreviated as *cgroups* in this guide, are a Linux kernel feature that allows you to allocate resources — such as CPU time, system memory, network bandwidth, or combinations of these resources — among hierarchically ordered groups of processes running on a system.

By using cgroups, system administrators gain fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources. Hardware resources can be smartly divided up among applications and users, increasing overall efficiency.

Control Groups provide a way to hierarchically group and label processes, and to apply resource limits to them. Traditionally, all processes received similar amount of system resources that administrator could modulate with the process *niceness* value. With this approach, applications that involved a large number of processes got more resources than applications with few processes, regardless of the relative importance of these applications.

Red Hat Enterprise Linux 7 moves the resource management settings from the process level to the application level by binding the system of cgroup hierarchies with the systemd unit tree. Therefore, you can manage system resources with `systemctl` commands, or by modifying systemd unit files. See Chapter 2, *Using Control Groups* for details.

In previous versions of Red Hat Enterprise Linux, system administrators built custom cgroup hierarchies with use of the `cgconfig` command from the libcgroup package. This package is now deprecated and it is not recommended to use it since it can easily create conflicts with the default cgroup hierarchy. However, libcgroup is still available to cover for certain specific cases, where **systemd** is not yet applicable, most notably for using the *net-prio* subsystem. See Chapter 3, *Using libcgroup Tools*.

The aforementioned tools provide a high-level interface to interact with cgroup controllers (also known as subsystems) in Linux kernel. The main cgroup controllers for resource management are *cpu*, *memory* and *blkio*, see Available Controllers in Red Hat Enterprise Linux 7 for the list of controllers enabled by default. For detailed description of resource controllers and their configurable parameters, refer to Controller-Specific Kernel Documentation.

<<

Cgroup Terminology:

- A *cgroup* associates a set of tasks with a set of parameters for one or more subsystems.
- A *subsystem* is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways.
- A subsystem is also called as a cgroup resource *controller*.
  Eg. *cpu, cpuset, memory, blkio, net_cls, freezer, etc.*
- A *hierarchy* is a set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy, and a set of subsystems (or resource controllers); each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the cgroup virtual filesystem associated with it (IOW, it's mounted into the root filesystem via the “cgroup” VFS type).

>>

...

1.3. Resource Controllers in Linux Kernel

A resource controller, also called a cgroup subsystem, represents a single resource, such as CPU time or memory. The Linux kernel provides a range of resource controllers, that are mounted automatically by **systemd**. Find the list of currently mounted resource controllers in `/proc/cgroups`, or use the **lssubsys** monitoring tool. In Red Hat Enterprise Linux 7, **systemd** mounts the following controllers by default:

Available Controllers in Red Hat Enterprise Linux 7

- `blkio` — sets limits on input/output access to and from block devices;
- `cpu` — uses the CPU scheduler to provide cgroup tasks an access to the CPU. It is mounted together with the `cpuacct` controller on the same mount;
- `cpuacct` — creates automatic reports on CPU resources used by tasks in a cgroup. It is mounted together with the `cpu` controller on the same mount;
- `cpuset` — assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup;
- `devices` — allows or denies access to devices for tasks in a cgroup;
- `freezer` — suspends or resumes tasks in a cgroup;
- `memory` — sets limits on memory use by tasks in a cgroup, and generates automatic reports on memory resources used by those tasks;
- `net_cls` — tags network packets with a class identifier (classid) that allows the Linux traffic controller (the `tc` command) to identify packets originating from a particular cgroup task;
- `perf_event` — enables monitoring cgroups with the **perf** tool;
- `hugetlb` — allows to use virtual memory pages of large sizes, and to enforce resource limits on these pages.

The Linux Kernel exposes a wide range of tunable parameters for resource controllers that can be configured with **systemd**. See the kernel documentation (list of references in Controller-Specific Kernel Documentation) for detailed description of these parameters.

<<
Eg. On an Ubuntu 16.04.2 LTS desktop class system:

```
# cat /etc/issue
Ubuntu 16.04.2 LTS \n \l

# lssubsys      << subsystem is another name for a resource controller >>
cpuset
cpu,cpuacct
blkio
memory
devices
freezer
net_cls,net_prio
perf_event
hugetlb
pids
#
```

```
# cat /proc/cgroups
#subsys_name hierarchy    num_cgroups  enabled
cpuset        5           1                  1
cpu                6          75                  1
cpuacct             6         75                  1
blkio             11          73                  1
memory       8          95                   1
devices            2          73                  1
freezer            3           8                  1
net_cls            9           1                  1
perf_event         7           1                  1
net_prio           9           1                  1
hugetlb            4           1                  1
pids              10          74                  1
#
```

>>

...

IMP Note- Limiting and Throttling

Source: Throttling CPU usage with Linux cgroups

...

cpu.shares - The default value is 1024. This gives any process in this cgroup 1024 out of 1024 "CPU shares". In other words if you lower this value it will **limit** the process. For example: if I set this value to 512 then the process will receive a maximum of 50% of the CPU if and only if another process is also requesting CPU time (ignoring any nice and realtime values you may have set). It still has the option to consume 100% of the idle CPU time.

cpu.cfs_period_us - The default value is 100000 and refers to the time period in which the standard scheduler "accounts" the process in microseconds. It does little on its own. (100 ms).

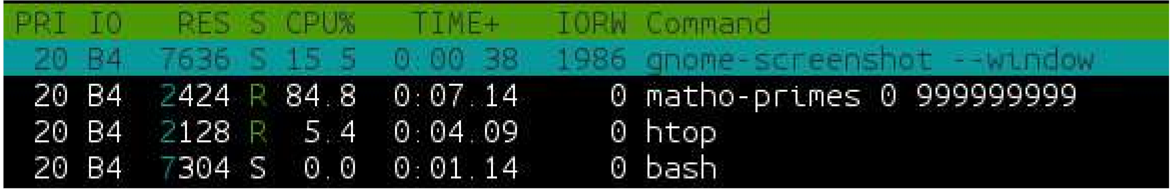
cpu.cfs_quota_us - The default value is -1 which means it has no effect. Any valid value here is in microseconds and must not exceed *cpu.cfs_period_us*.

The trick to throttling a process is to manipulate the last two values,
namely *cpu.cfs_period_us* and *cpu.cfs_quota_us*. The quota is the amount of CPU bandwidth (time) in which a process in the cgroup will be allowed to use per the period. So a process that is given 100000 out of 100000 is allowed to use 100% of the CPU (again, ignoring nice and realtime values). A process given 90000 out of 100000 is allowed to run 90%, 50000 out of 100000 is allowed to run 50% and so on. In the words of the official Linux documentation:

```
The bandwidth allowed for a group is specified using a quota and period. Within
each given "period" (microseconds), a group is allowed to consume only up to
"quota" microseconds of CPU time.  When the CPU bandwidth consumption of a group
exceeds this limit (for that period), the tasks belonging to its
hierarchy will be throttled and are not allowed to run again until the next
period.
```

Example

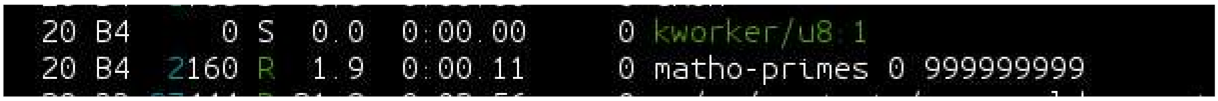
In this first example I have set *cpu.shares = 100* for the *matho-primes* process, which gives the process 100 out of 1024 arbitrary cycles.



```
PRI IO   RES S CPU%    TIME+    IORW Command
 20 B4  7636 S 15.5  0:00.38   1986 gnome-screenshot --window
 20 B4  2424 R 84.8  0:07.14      0 matho-primes 0 999999999
 20 B4  2128 R  5.4  0:04.09      0 htop
 20 B4  7304 S  0.0  0:01.14      0 bash
```

As you can see this has not throttled my process. Because the system has CPU time to spare the process still consumes all that it can.

In this next example I set *cpu.cfs_period_us = 50000* and *cpu.cfs_quota_us = 1000* for the same process.



```
 20 B4     0 S  0.0  0:00.00      0 kworker/u8:1
 20 B4  2160 R  1.9  0:00.11      0 matho-primes 0 999999999
```

This has had the desired effect. For every 50,000 µs time slice, the process is only allowed to use 1,000 µs (2%) and is paused until the next time slice is available. This is true regardless of the current system demand. (Note: the process can still receive less than its 2% allotted time if the system is heavily loaded or a higher priority process demands the time.)

I can check the amount of throttling that has been done at any time:

```
$ cat cpu.stat
nr_periods 336
nr_throttled 334
throttled_time 16181179709
```

To launch the process within the cgroup I used the following command:

```
$ cgexec -g cpu:brian matho-primes 0 999999999
```

Summary

1. Create the cgroup.
2. Set *cpu.cfs_period_us*.
3. Set *cpu.cfs_quota_us*.
4. Use *cgexec* to launch the application.

Once running the cgroup can be edited by the owner of that group, or the process can be moved to a different cgroup. This may be handy using a Cron job to give a process more time at certain times of the day.

Notes:

- The current (April 2015) Linux cgroup documentation doesn't mention the *cpu* subsystem at all and has introduced the *cpusets* subsystem, but the two do not do the same job. It is not clear if this type of throttling capability has been removed altogether or moved to a different area of the kernel.
- There are, of course, bugs.
- Just because you have a process in one cgroup, doesn't mean it cannot also be in another.

• Different process subsystems (*cpu, memory, freezer*, etc.) can be in different in cgroups.

• Multiple processes can and do share cgroups, but only when told to do so.

• Child processes remain inside the cgroups unless moved out of them, and only if they are allowed to be moved (set by policy).

• cgroups are hierarchical and you can create sub-cgroups to limit certain processes further. A sub-cgroup cannot receive more resources than its parent cgroup.

• There may be a slight performance penalty depending on your choice for period and quota. You will only really need to worry about this on highly optimised or incredibly large systems.

• cgroups do not offer virtualisation or jailing of a process, though they can be used alongside these systems, and are in many circumstances (such as Android and LXC).

• You can still set the nice and realtime values to give processes certain priorities. This will not affect the maximum CPU time allowed by the cgroup and the CPU time will be shared in a complex manner between all processes, as One would expect from a decent operating system.

• The *cpu.cfs** values of course refer to the CFS scheduler, the *cpu.rt** values refer to the realtime scheduler. It is unlikely you will want to change the realtime values unless you want fine-grained control over realtime processes.

Also see:
Linux cgroups: limit CPU usage in absolute values which do not depend on CPU speed

---

Control CPU bandwidth:

Simpler to more complex + superior:

1. nice/renice
2. cpulimit app
3. cgroups : can apply to a set of processes under a given cgroup

CPU Partitioning – with (re)nice, cpulimit and Cgroups

Ref article: *Restricting process CPU usage using nice, cpulimit, and cgroups*

"... 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.

..."

The program that stresses a cpu(s): genprime.c
Compiled and built.

Eg. run:

```
$ ./primegen
Usage: ./primegen first_num to_num [range]
$ ./primegen 1 50
Prime numbers between 1 and 50 are:
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
$
```

I Using the nice (and renice) CLI

Nice value range on the Linux OS:

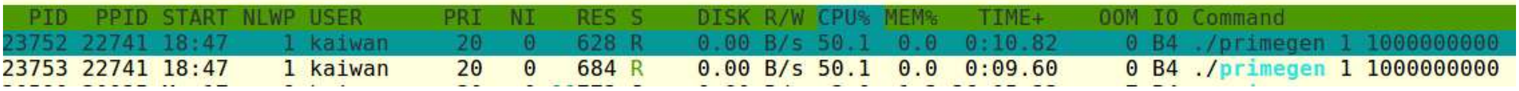
```
-20                 0                 +19
Best             Default             Worst
```

See the man page for details.

Sample Runs:
Notice the output CPU %age from htop.

1.1 Restrict primegen to 1 CPU and run 2 instances:

```
<< restrict each instance to run on CPU #1 only (available CPUs: [0-3]) >>
$ taskset 02 ./primegen 1 1000000000 >/dev/null &
[1] 23752
$ taskset 02 ./primegen 1 1000000000 >/dev/null &
[2] 23753
```



```
  PID  PPID START NLWP USER       PRI  NI   RES S    DISK R/W CPU% MEM%   TIME+     OOM IO Command
23752 22741 18:47    1 kaiwan      20   0   628 R    0.00 B/s 50.1  0.0  0:10.82      0 B4 ./primegen 1 1000000000
23753 22741 18:47    1 kaiwan      20   0   684 R    0.00 B/s 50.1  0.0  0:09.60      0 B4 ./primegen 1 1000000000
```

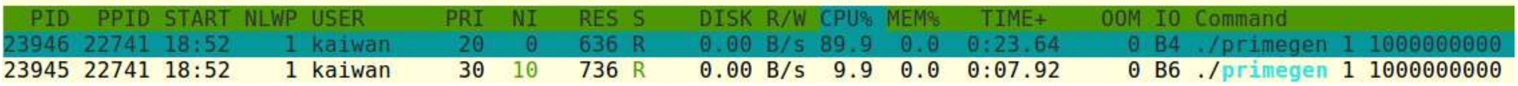
[Above: clipped output of the 'htop' CLI]

Notice how they together take up 100% load on that CPU, and equally share CPU bandwidth between them.

```
$ kill %1 %2   << we kill ‘em off >>
$
[1]-  Terminated              taskset 02 ./primegen 1 1000000000 > /dev/null
[2]+  Terminated              taskset 02 ./primegen 1 1000000000 > /dev/null
$
```

1.2 Restrict primegen to 1 CPU, run 2 instances but use (re)nice to lower the priority of one of them:

```
<< restrict each instance to run on CPU #1 only (available CPUs: [0-3]) >>
$ taskset 02 ./primegen 1 1000000000 >/dev/null &
[1] 23945
$ taskset 02 ./primegen 1 1000000000 >/dev/null &
[2] 23946
$ renice -n 10 -p 23945   << restrict cpu bandwidth of PID 23945 via the (re)nice ! >>
23945 (process ID) old priority 0, new priority 10  << 10 is worse! >>
$
```



```
  PID  PPID START NLWP USER       PRI  NI   RES S     DISK R/W CPU% MEM%    TIME+      OOM IO Command
23946 22741 18:52    1 kaiwan      20   0   636 R     0.00 B/s 89.9  0.0  0:23.64        0 B4 ./primegen 1 1000000000
23945 22741 18:52    1 kaiwan      30  10   736 R     0.00 B/s  9.9  0.0  0:07.92        0 B6 ./primegen 1 1000000000
```

Notice how now they together take up 100% load on that CPU, but do not equally share CPU bandwidth between them – the one with the lower nice value is relegated a (much) lower CPU share.

```
$ kill %1 %2     << we kill ‘em off >>
$
[1]-  Terminated              taskset 02 ./primegen 1 1000000000 > /dev/null
[2]+  Terminated              taskset 02 ./primegen 1 1000000000 > /dev/null
$
```

II With the ‘cpulimit’ CLI

If not already installed, install it with
`sudo apt install cpulimit`
(on Debian/Ubuntu).
(Can see the man page for some sample usage).

Source
“ ... 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. ...”

Note- (from the man page):

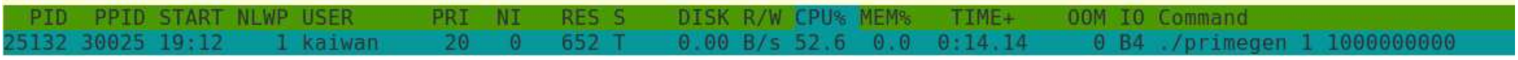
```
“... cpulimit always sends the SIGSTOP and SIGCONT signals to a process, both to verify
that it can control it and  to  limit the average amount of CPU it consumes.  This can
result in misleading (annoying) job control messages that indicate that the job has
been stopped (when actually it was, but immediately restarted).

This can also cause issues with interactive shells that detect or otherwise depend on
SIGSTOP/SIGCONT.   For example, you may place a job in the foreground, only to see it
immediately stopped and restarted in the background.  (See also
<http://bugs.debian.org/558763>.) ...”
```

```
$ cpulimit
Error: You must specify a target process
CPUlimit version 2.1
Usage: cpulimit TARGET [OPTIONS...] [-- PROGRAM]
   TARGET must be exactly one of these:
      -p, --pid=N        pid of the process
      -e, --exe=FILE     name of the executable program file
                         The -e option only works when
                         cpulimit is run with admin rights.
      -P, --path=PATH    absolute path name of the
                         executable program file
   OPTIONS
      -b  --background   run in background
      -c  --cpu=N        override the detection of CPUs on the machine.
      -l, --limit=N      percentage of cpu allowed from 1 up.
                         Usually 1 - 1200, but can be higher
                         on multi-core CPUs (mandatory)
      -q, --quiet        run in quiet mode (only print errors).
      -k, --kill         kill processes going over their limit
                         instead of just throttling them.
      -r, --restore      Restore processes after they have
                         been killed. Works with the -k flag.
      -s, --signal=SIG   Send this signal to the watched process when cpulimit exits.
                         Signal should be specificed as a number or
                         SIGTERM, SIGCONT, SIGSTOP, etc. SIGCONT is the default.
      -v, --verbose      show control statistics
      -z, --lazy         exit if there is no suitable target process,
                         or if it dies
          --             This is the final CPUlimit option. All following
                         options are for another program we will launch.
      -h, --help         display this help and exit
$
```

*2.1 Restrict primegen to 1 CPU (via taskset), run one instance, wrapped around **cpulimit** forcing 50% max CPU bandwidth:*

```
$ taskset 02 cpulimit -l 50 ./primegen 1 1000000000 >/dev/null
$
```

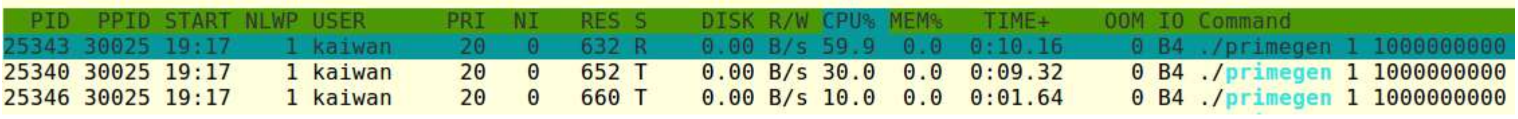


```
  PID   PPID START NLWP USER       PRI   NI    RES S     DISK R/W CPU% MEM%    TIME+     OOM IO Command
25132 30025 19:12     1 kaiwan      20    0    652 T     0.00 B/s 52.6  0.0  0:14.14       0 B4 ./primegen 1 1000000000
```

```
$ pkill primegen
$ Process 25132 dead!
```

2.2 Run three instances of primegen on 1 CPU (via taskset), restricting them to 30%, 60% and 10% CPU bandwidth respectively:

```
$ taskset 02 cpulimit -l 30 ./primegen 1 1000000000 >/dev/null
$ taskset 02 cpulimit -l 60 ./primegen 1 1000000000 >/dev/null
$ taskset 02 cpulimit -l 10 ./primegen 1 1000000000 >/dev/null
```



```
  PID  PPID START NLWP USER        PRI  NI   RES S    DISK R/W CPU% MEM%   TIME+     OOM IO Command
25343 30025 19:17    1 kaiwan       20   0   632 R   0.00 B/s 59.9  0.0  0:10.16       0 B4 ./primegen 1 1000000000
25340 30025 19:17    1 kaiwan       20   0   652 T   0.00 B/s 30.0  0.0  0:09.32       0 B4 ./primegen 1 1000000000
25346 30025 19:17    1 kaiwan       20   0   660 T   0.00 B/s 10.0  0.0  0:01.64       0 B4 ./primegen 1 1000000000
```

```
$ pkill primegen
$ Process 25340 dead!
Process 25343 dead!
Process 25346 dead!

$
```

III With Control Groups

Source

"… 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.
…"

Source
"Control groups can be used << accessed >> in multiple ways:

- By accessing the `cgroup` filesystem directly. *<< via /sys/fs/cgroup >>*
- Using the `cgm` client (part of the **`cgmanager`** package).
- Via tools like `cgcreate`, `cgexec` and `cgclassify` (part of the **`libcgroup`**$^{\text{AUR}}$ package). ***<< we'll use this approach here >>***
- the "rules engine daemon", to automatically move certain users/groups/commands to groups (`/etc/cgrules.conf` and `/usr/lib/systemd/system/cgconfig.service`) (part of the **`libcgroup`**$^{\text{AUR}}$ package).
- through other software such as **Linux Containers** (LXC) virtualization, tools like **`playpen`** or systemd.

…"

Demo: on an Ubuntu VM with only 1 vCPU.
Run:

```
$ ls /sys/fs/cgroup/
blkio/  cpuacct@      cpuset/   freezer/  memory/   net_cls,net_prio/  perf_event/
systemd/
cpu@    cpu,cpuacct/  devices/  hugetlb/  net_cls@  net_prio@          pids/
$
```

```
<< Create a first custom cpu-cgroup >>

$ cgcreate -g cpu:/cgcpu1_more
cgcreate: can't create cgroup /cgcpu1_more: Cgroup, operation not allowed
$ sudo cgcreate -g cpu:/cgcpu1_more
$ ls /sys/fs/cgroup/cpu,cpuacct/
cgcpu1_more/           cgroup.sane_behavior  cpuacct.usage_percpu  cpu.shares
notify_on_release  tasks
cgroup.clone_children  cpuacct.stat          cpu.cfs_period_us     cpu.stat
release_agent      user.slice/
cgroup.procs           cpuacct.usage         cpu.cfs_quota_us      init.scope/
system.slice/
$ cat /sys/fs/cgroup/cpu,cpuacct/cgcpu1_more/cpu.shares
1024    << Leave it's 'cpu share' as default (1024) => it can use 100% cpu bandwidth >>
$
```

```
<< Create a second custom cpu-cgroup >>

$ sudo cgcreate -g cpu:/cgcpu2_less
<< Lessen it's possible CPU bandwidth - give approx 33% (1024/3) to this cgroup >>
$ sudo cgset -r cpu.shares=341 cgcpu2_less
$ cat /sys/fs/cgroup/cpu,cpuacct/cgcpu2_less/cpu.shares
341
$
```

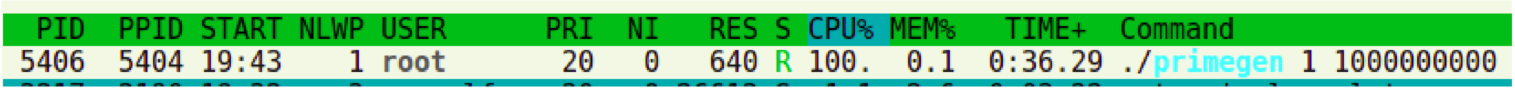
3.1: execute our 'primegen' app within the first CPU Cgroup

```
$ sudo cgexec -g cpu:cgcpu1_more ./primegen 1 1000000000 >/dev/null &
[1] 5404
```

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

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.

<< Note: FAQ- One can move a process(es) into a given cgroup via the cgclassify(1) command. >>



```
  PID  PPID START NLWP USER       PRI  NI   RES S CPU% MEM%   TIME+  Command
 5406  5404 19:43    1 root        20   0   640 R 100.  0.1  0:36.29 ./primegen 1 1000000000
```

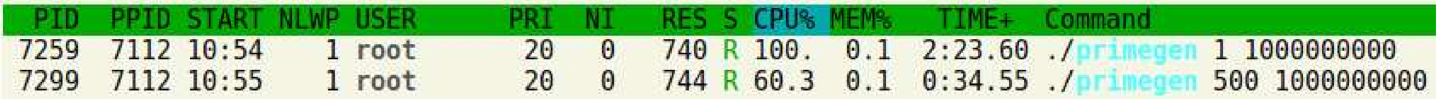
```
$ sudo pkill primegen
$
```

3.2: Execute the ‘primegen’ app 2 instances: one within the first ‘more’ CPU Cgroup, 2nd within the ‘less’ cgroup

```
$ sudo /bin/bash
Password: xxx
#

# cgexec -g cpu:cgcpu1_more ./primegen 1 1000000000 >/dev/null &
[1] 7259
#
# cgexec -g cpu:cgcpu2_less ./primegen 500 1000000000 >/dev/null &
[2] 7299
#
```

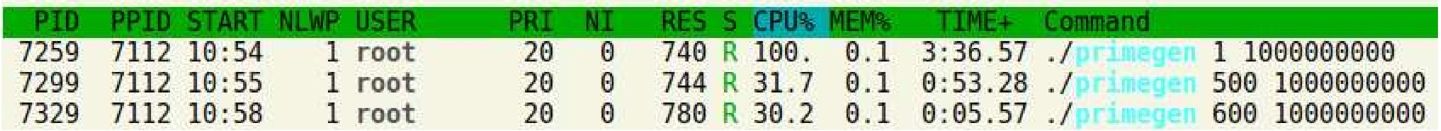
You will notice that the first process consumes pretty much 100% cpu bandwidth; the second instance consumes a lot less (it varies and that too when there’s a chance for it to run.



```
  PID  PPID START NLWP USER       PRI  NI   RES S CPU% MEM%   TIME+  Command
 7259  7112 10:54    1 root        20   0   740 R 100.  0.1  2:23.60 ./primegen 1 1000000000
 7299  7112 10:55    1 root        20   0   744 R 60.3  0.1  0:34.55 ./primegen 500 1000000000
```

3.3 Now we run a third instance of the process - within the ‘less’ cgroup:

```
# cgexec -g cpu:cgcpu2_less ./primegen 600 1000000000 >/dev/null &
[3] 7329
#
```

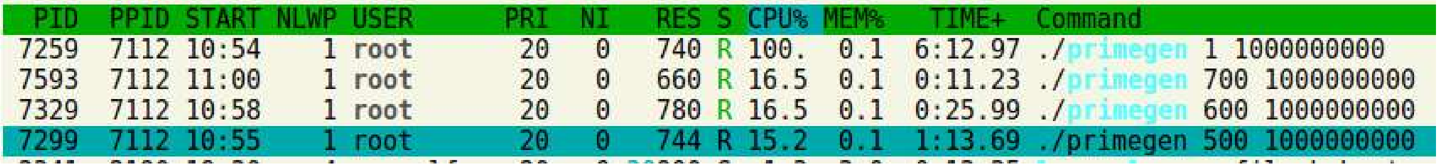


```
  PID  PPID START NLWP USER       PRI  NI   RES S CPU% MEM%   TIME+  Command
 7259  7112 10:54    1 root        20   0   740 R 100.  0.1  3:36.57 ./primegen 1 1000000000
 7299  7112 10:55    1 root        20   0   744 R 31.7  0.1  0:53.28 ./primegen 500 1000000000
 7329  7112 10:58    1 root        20   0   780 R 30.2  0.1  0:05.57 ./primegen 600 1000000000
```

Interestingly, the latter two processes will now split their available CPU bandwidth within their cgroup!

3.4 Run a fourth instance, again within the ‘less’ cgroup:

```
# cgexec -g cpu:cgcpu2_less ./primegen 700 1000000000 >/dev/null &
[4] 7593
#
```



```
  PID  PPID START NLWP USER       PRI  NI    RES S CPU% MEM%    TIME+  Command
 7259  7112 10:54    1 root        20   0    740 R 100.   0.1  6:12.97 ./primegen 1 1000000000
 7593  7112 11:00    1 root        20   0    660 R 16.5   0.1  0:11.23 ./primegen 700 1000000000
 7329  7112 10:58    1 root        20   0    780 R 16.5   0.1  0:25.99 ./primegen 600 1000000000
 7299  7112 10:55    1 root        20   0    744 R 15.2   0.1  1:13.69 ./primegen 500 1000000000
```

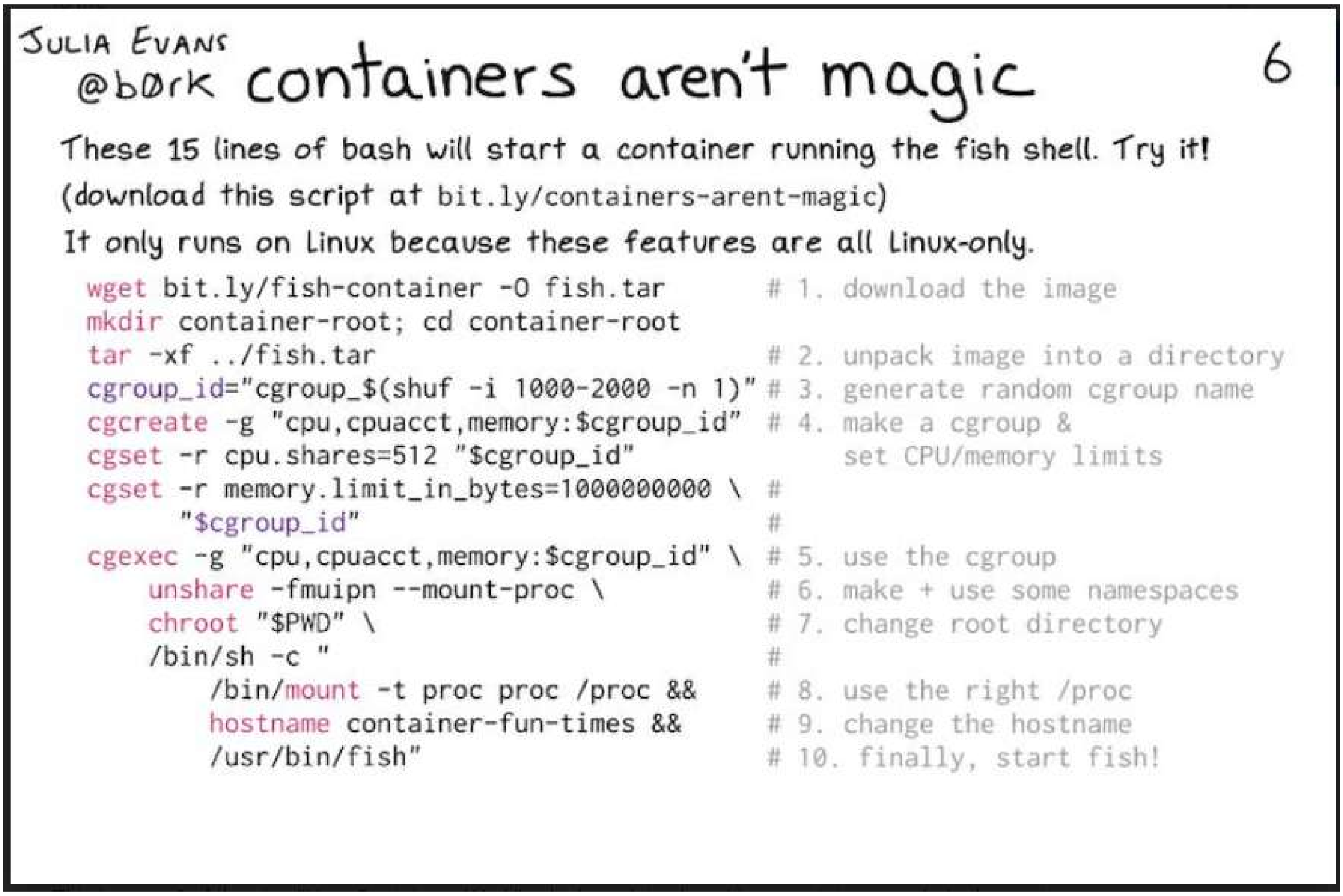
A further splitting of CPU occurs now between the 3 processes in the ‘less’ cgroup, each getting approximately 15%-17% CPU bandwidth.
The first process in the ‘more’ cgroup happily continues to consume 100%.

```
# kill %1 %2 %3 %4
#
[1]   Terminated      cgexec -g cpu:cgcpu1_more ./primegen 1 1000000000 > /dev/null
[2]   Terminated      cgexec -g cpu:cgcpu2_less ./primegen 500 1000000000 > /dev/null
[3]-  Terminated      cgexec -g cpu:cgcpu2_less ./primegen 600 1000000000 > /dev/null
[4]+  Terminated      cgexec -g cpu:cgcpu2_less ./primegen 700 1000000000 > /dev/null
#
```

<<
See the Cgroups v2 CPU example here:
https://github.com/PacktPublishing/Linux-Kernel-Programming/tree/master/ch11/cgroups_v2_cpu_eg

>>

Src: Julia Evans ezine on Containers



JULIA EVANS
@b0rk

containers aren't magic

6

These 15 lines of bash will start a container running the fish shell. Try it!
(download this script at bit.ly/containers-arent-magic)
It only runs on Linux because these features are all Linux-only.

```
wget bit.ly/fish-container -O fish.tar        # 1. download the image
mkdir container-root; cd container-root
tar -xf ../fish.tar                           # 2. unpack image into a directory
cgroup_id="cgroup_$(shuf -i 1000-2000 -n 1)"  # 3. generate random cgroup name
cgcreate -g "cpu,cpuacct,memory:$cgroup_id"   # 4. make a cgroup &
cgset -r cpu.shares=512 "$cgroup_id"          #    set CPU/memory limits
cgset -r memory.limit_in_bytes=1000000000 \   #
      "$cgroup_id"                            #
cgexec -g "cpu,cpuacct,memory:$cgroup_id" \   # 5. use the cgroup
    unshare -fmuipn --mount-proc \            # 6. make + use some namespaces
    chroot "$PWD" \                           # 7. change root directory
    /bin/sh -c "                              #
        /bin/mount -t proc proc /proc &&      # 8. use the right /proc
        hostname container-fun-times &&       # 9. change the hostname
        /usr/bin/fish"                        # 10. finally, start fish!
```

CGManager

One would expect some management utilities for administrators and even users to manage cgroups. There are several: a well-accepted one from the LXC project (?) is Cgroup Manager (cgm). From the website:
“CGManager is a central privileged daemon that manages all your cgroups for you
through a simple D-Bus API. It's designed to work with nested LXC containers
as well as accepting unprivileged requests including resolving user namespaces UIDs/GIDs. ...”

Install the package, and:

```
$ man -k cgm
cgm (1)              - a client script for cgmanager
cgmanager (8)        - a daemon to manage cgroups
cgproxy (8)          - a proxy for cgmanager
$
```

Resource: Thoughts on Linux Containers (LXC), Stefan Hajnoczi

(also has some information on Cgroups usage with cgmanager)

<< RHEL 7 uses *systemd* for cgroup management (and not libcgroup, which was the preferred interface upto RHEL 6 >>

“This guide focuses on utilities provided by **systemd** that are preferred as a way of cgroup management and will be supported in the future. Previous versions of Red Hat Enterprise Linux used the libcgroup package for the same purpose. This package is still available to assure backward compatibility (see Warning), but it will not be supported in the future versions of Red Hat Enterprise Linux.”

...

Because cgroups “live” in a pseudo-filesystem (of type cgroup), no cgroups exist when the system first boots up. The cgroups have to be mounted and setup each time.

By default, when a Linux system boots up, all processes belong in a control group called the “root hierachy”, with equal resource sharing and prioritization.

The user (or more likely, scripts) can then create additional cgroups or hierarchies under the root cgroup, partitioning processes and performing resource sharing according to the project's goals.

Ref:
http://www.janoszen.com/2013/02/06/limiting-linux-processes-cgroups-explained/

...

Example:
On a QEMU-emulated ARM-Linux:

First mount a pseudo-filesystem of type tmpfs, which wil act as the root or base point for the cgroup hierachy.

```
ARM # mount -t tmpfs cgroup_root /sys/fs/cgroup/
ARM # mount
rootfs on / type rootfs (rw)
/dev/root on / type ext4 (rw,relatime,data=ordered)
none on /proc type proc (rw,nodev,noexec,relatime)
none on /sys type sysfs (rw,noexec,relatime)
none on /sys/kernel/debug type debugfs (rw,relatime)
cgroup_root on /sys/fs/cgroup type tmpfs (rw,relatime)
ARM #
```

```
Setup a symbolic link /cgroup to /sys/fs/cgroup

ARM # ln -s /sys/fs/cgroup/ /cgroup
ARM # cd /cgroup/
ARM # ls -l
total 0
ARM #
```

CPUset

CPU Allocation to tasks of a particular CG using the cpuset cgroup

We've booted the system as a dual-core ARM Cortex-A9 (by using the “-smp 2,sockets=1” QEMU parameter) :

```
ARM # cat /proc/cpuinfo
processor     : 0
model name    : ARMv7 Processor rev 0 (v7l)
BogoMIPS      : 496.43
Features      : swp half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer     : 0x41
CPU architecture: 7
CPU variant   : 0x0
CPU part      : 0xc09
CPU revision  : 0

processor     : 1
model name    : ARMv7 Processor rev 0 (v7l)
BogoMIPS      : 474.31
Features      : swp half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer     : 0x41
CPU architecture: 7
CPU variant   : 0x0
CPU part      : 0xc09
CPU revision  : 0

Hardware      : ARM-Versatile Express
Revision      : 0000
Serial        : 0000000000000000
ARM #

ARM # cd /cgroup
```

We'd like to setup this cgroup hierachy:

```
/--
  |-- voip--
           |-- cpuset_cg
```

Thus we now create and mount folders appropriately, which will in effect become the cgroup “hierarchy”.

```
ARM # mkdir voip
ARM # mount -t tmpfs cgroup_root /sys/fs/cgroup/voip/
ARM # cd voip


ARM # mkdir cpuset_cg
ARM # ls -l
total 0
drwxr-xr-x    2 0        0              40 Jul 18 13:52 cpuset_cg/
ARM # ls cpuset_cg/
ARM # mount -t cgroup -o cpuset none cpuset_cg/  << don't miss the option fstype “none”
>>
ARM # mount |grep cgroup
cgroup_root on /sys/fs/cgroup type tmpfs (rw,relatime)
cgroup_root on /sys/fs/cgroup/voip type tmpfs (rw,relatime)
none on /sys/fs/cgroup/voip/cpuset_cg type cgroup (rw,relatime,cpuset)
ARM #
```

Which tasks belong to this CG?

```
ARM # cd cpuset_cg/
ARM # cat tasks
1
2
3
4
5
6
7
8
9
10
11
12
206
209
211
219
230
331
332
341
347
432
461
576
592
651
653
658
659
665
741
ARM #
```

So, by default, ALL tasks belong to this CG!

However, also by default, all share the CPU resources equally, i.e., the allocation / prioritization is identical. In this particular case, we're referring to the CPU affinity of tasks to CPU cores.

```
ARM # cat cpuset_cg/cpuset.cpus
0-1
ARM #
```

We can change both these: the tasks in the CG and the CPU affinity of those tasks, simply by echo'ing different values into the (virtual) files.

```
ARM # ls cpuset_cg/
cgroup.clone_children             cpuset.memory_pressure_enabled
cgroup.procs                      cpuset.memory_spread_page
cgroup.sane_behavior              cpuset.memory_spread_slab
cpuset.cpu_exclusive              cpuset.mems
cpuset.cpus                       cpuset.sched_load_balance
cpuset.mem_exclusive              cpuset.sched_relax_domain_level
cpuset.mem_hardwall               notify_on_release
cpuset.memory_migrate             release_agent
cpuset.memory_pressure            tasks
ARM #
```

What do the above (virtual) files represent?

From *Documentation/cgroups/cpusets.txt*

```
...
Each cpuset is represented by a directory in the cgroup file system
containing (on top of the standard cgroup files) the following
files describing that cpuset:

 - cpuset.cpus: list of CPUs in that cpuset
 - cpuset.mems: list of Memory Nodes in that cpuset
 - cpuset.memory_migrate flag: if set, move pages to cpusets nodes
 - cpuset.cpu_exclusive flag: is cpu placement exclusive?
 - cpuset.mem_exclusive flag: is memory placement exclusive?
 - cpuset.mem_hardwall flag:  is memory allocation hardwalled
 - cpuset.memory_pressure: measure of how much paging pressure in cpuset
 - cpuset.memory_spread_page flag: if set, spread page cache evenly on allowed nodes
 - cpuset.memory_spread_slab flag: if set, spread slab cache evenly on allowed nodes
 - cpuset.sched_load_balance flag: if set, load balance within CPUs on that cpuset
 - cpuset.sched_relax_domain_level: the searching range when migrating tasks

In addition, only the root cpuset has the following file:
- cpuset.memory_pressure_enabled flag: compute memory_pressure?

...
```

Another example:

```
ARM # pwd
/sys/fs/cgroup
ARM #
```

Mount overall “cgroup_root” filesystem of type tmpfs

```
ARM # mount -t tmpfs cgroup_root .
ARM # mount
```

```
rootfs on / type rootfs (rw)
/dev/root on / type ext4 (rw,relatime,data=ordered)
none on /proc type proc (rw,nodev,noexec,relatime)
none on /sys type sysfs (rw,noexec,relatime)
none on /sys/kernel/debug type debugfs (rw,relatime)
cgroup_root on /sys/fs/cgroup type tmpfs (rw,relatime)
ARM #
```

Create dir and mount the “cpu” subsystem (resource controller) filesystem

```
ARM # mkdir root_cpu
ARM # ls
mem/      root_cpu/
ARM # mount -t cgroup -o cpu none root_cpu/
ARM # ls
mem/      root_cpu/
ARM # ls root_cpu/
cgroup.clone_children  cpu.shares            tasks
cgroup.procs           notify_on_release
cgroup.sane_behavior   release_agent
ARM # mount |grep cgroup
cgroup_root on /sys/fs/cgroup type tmpfs (rw,relatime)
none on /sys/fs/cgroup/root_cpu type cgroup (rw,relatime,cpu)
ARM #
```

CPU shares:

```
ARM # cat root_cpu/cpu.shares
1024
ARM #
```

1024 is the base, implies 100% cpu share.

Note:

Automatic process grouping (a.k.a. "the patch that does wonders")

Recommended LWN article :Group scheduling and alternatives

The most impacting feature in this release is the so-called "patch that does wonders", a patch that changes substantially how the process scheduler assigns shares of CPU time to each process. With this feature the system will group all processes with the same session ID as a single scheduling entity.

Example: Let's imagine a system with six CPU-hungry processes, with the first four sharing the same session ID and the other using another two different sessions each one.

Without automatic process
grouping: [proc. 1 | proc. 2 | proc. 3 | proc. 4 | proc. 5 | proc. 6]

With automatic process
grouping: [proc. 1, 2, 3, 4 | proc. 5 | proc. 6]

The session ID is a property of processes in Unix systems (you can see it with commands like ps -eo session,pid,cmd). It is inherited by forked child processes, which can start a new session using setsid(3). The bash shell uses setsid(3) every time it is started, which means you can run a "make -j 20" inside a shell in your desktop and not notice it while you browse the web.

This feature is implemented on top of group scheduling (merged in [2.6.24). You can disable it in */proc/sys/kernel/sched_autogroup_enabled*

Code: (commit)

```
<<
Src: https://wiki.archlinux.org/index.php/Cgroups
```

Memory Cgroup Example

```
# mount -t cgroup -o memory none memcg/
# mount|grep cgroup
none on /cg/voip/memcg type cgroup (rw,relatime,memory)
#
# ls memcg/
cgroup.clone_children            memory.pressure_level
cgroup.event_control             memory.soft_limit_in_bytes
cgroup.procs                     memory.stat
cgroup.sane_behavior             memory.swappiness
memory.failcnt                   memory.usage_in_bytes
memory.force_empty               memory.use_hierarchy
memory.limit_in_bytes            notify_on_release
memory.max_usage_in_bytes        release_agent
memory.move_charge_at_immigrate  tasks
memory.oom_control
#
...

# echo 10000000 > /sys/fs/cgroup/memory/groupname/foo/memory.limit_in_bytes
```

Note that the memory limit applies to RAM use only -- once tasks hit this limit, they will begin to swap. But it won't affect the performance of other processes significantly.

Similarly you can change the CPU priority ("shares") of this group. By default all groups have 1024 shares. A group with 100 shares will get a ~10% portion of the CPU time:

```
# echo 100 > /sys/fs/cgroup/cpu/groupname/foo/cpu.shares
...

>>
```

FAQ:

A process can belong to exactly one cgroup at a time.

Q. How to lookup the control groups a process belongs to?
A. Look them up:
cat /proc/<pid>/cgroups

Eg. (on an x86_64 running Ubuntu 15.04):

```
# ps -A|grep qemu
 5002 pts/17   00:18:18 qemu-system-arm
# cat /proc/5002/cgroup
10:hugetlb:/user.slice/user-1000.slice/session-c2.scope
9:net_cls,net_prio:/user.slice/user-1000.slice/session-c2.scope
8:devices:/user.slice/user-1000.slice/session-c2.scope
7:freezer:/user.slice/user-1000.slice/session-c2.scope
6:perf_event:/user.slice/user-1000.slice/session-c2.scope
5:cpuset:/user.slice/user-1000.slice/session-c2.scope
```

```
4:memory:/user.slice/user-1000.slice/session-c2.scope
3:cpu,cpuacct:/user.slice/user-1000.slice/session-c2.scope
2:blkio:/user.slice/user-1000.slice/session-c2.scope
1:name=systemd:/user.slice/user-1000.slice/session-c2.scope
#
```

Cgroups : Memory Subsystem Example

How to interpret, for example,

```
4:memory:/user.slice/user-1000.slice/session-c2.scope
```

It's wrt the memory subsystem; the path “/user.slice/user-1000.slice/session-c2.scope” must be appended to the cgroup hierachy path + subsystem.

Cgroup is mounted under:
[tmpfs on /sys/fs/cgroup type tmpfs]

Subsystem is: memory

Thus, the path is:
`/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope`

```
# ls /sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope
cgroup.clone_children  memory.kmem.limit_in_bytes
memory.kmem.tcp.usage_in_bytes    memory.oom_control          memory.use_hierarchy
cgroup.event_control   memory.kmem.max_usage_in_bytes      memory.kmem.usage_in_bytes
memory.pressure_level       notify_on_release
cgroup.procs           memory.kmem.slabinfo                memory.limit_in_bytes
memory.soft_limit_in_bytes  tasks
memory.failcnt         memory.kmem.tcp.failcnt             memory.max_usage_in_bytes
memory.stat
memory.force_empty     memory.kmem.tcp.limit_in_bytes
memory.move_charge_at_immigrate  memory.swappiness
memory.kmem.failcnt    memory.kmem.tcp.max_usage_in_bytes  memory.numa_stat
memory.usage_in_bytes
#
```

```
From Documentation/cgroups/memory.txt
...
Brief summary of control files.

 tasks                       # attach a task(thread) and show list of threads
 cgroup.procs                      # show list of processes
 cgroup.event_control              # an interface for event_fd()
 memory.usage_in_bytes             # show current usage for memory
                             (See 5.5 for details)
 memory.memsw.usage_in_bytes       # show current usage for memory+Swap
                             (See 5.5 for details)
 memory.limit_in_bytes             # set/show limit of memory usage
 memory.memsw.limit_in_bytes       # set/show limit of memory+Swap usage
 memory.failcnt                    # show the number of memory usage hits limits
 memory.memsw.failcnt              # show the number of memory+Swap hits limits
 memory.max_usage_in_bytes         # show max memory usage recorded
 memory.memsw.max_usage_in_bytes # show max memory+Swap usage recorded
 memory.soft_limit_in_bytes      # set/show soft limit of memory usage
 memory.stat                 # show various statistics
 memory.use_hierarchy            # set/show hierarchical account enabled
 memory.force_empty          # trigger forced move charge to parent
 memory.pressure_level           # set memory pressure notifications
 memory.swappiness           # set/show swappiness parameter of vmscan
                             (See sysctl's vm.swappiness)
 memory.move_charge_at_immigrate # set/show controls of moving charges
 memory.oom_control          # set/show oom controls.
 memory.numa_stat            # show the number of memory usage per numa node

 memory.kmem.limit_in_bytes      # set/show hard limit for kernel memory
 memory.kmem.usage_in_bytes      # show current kernel memory allocation
 memory.kmem.failcnt             # show the number of kernel memory usage hits limits
 memory.kmem.max_usage_in_bytes  # show max kernel memory usage recorded

 memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory
 memory.kmem.tcp.usage_in_bytes  # show current tcp buf memory allocation
 memory.kmem.tcp.failcnt            # show the number of tcp buf memory usage hits
limits
 memory.kmem.tcp.max_usage_in_bytes # show max tcp buf memory usage recorded
...
```

Run a script that recursively prints the contents of readable files under a given starting folder:

```
# xplore_fs.sh /sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
===================== SUMMARY LIST of Files ==========================

Note: Max Depth is 4.

/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.pressure_level
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
memory.kmem.max_usage_in_bytes
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.use_hierarchy
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.swappiness
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/tasks

...
 -----------------------------------------------------------------------
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/ : <dir>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.pressure_level
: cat: /sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
memory.pressure_level: Invalid argument
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
memory.kmem.max_usage_in_bytes :            0
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
memory.use_hierarchy :           1
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.swappiness :
60
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/tasks : 933
1054
1064
1065
1066

...
<<  memory.limit_in_bytes        # set/show limit of memory usage >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.limit_in_bytes
```

```
: 9223372036854771712 (9007199254740988 KB)(8796093022207.00 MB)(8589934591.00 GB) << 8
EB ! >>
...
...
<< memory.usage_in_bytes         # show current usage for memory >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.usage_in_bytes
:   4593033216 (4485384 KB)(4380.00 MB)(  4.00 GB)
...

<< memory.max_usage_in_bytes     # show max memory usage recorded >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
memory.max_usage_in_bytes :   6998577152 (6834548 KB)(6674.00 MB)(  6.00 GB)

<<  memory.numa_stat             # show the number of memory usage per numa node >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.numa_stat :
total=1121292 N0=1121292
file=278154 N0=278154
anon=843138 N0=843138
unevictable=0 N0=0
hierarchical_total=1121292 N0=1121292
hierarchical_file=278154 N0=278154
hierarchical_anon=843138 N0=843138
hierarchical_unevictable=0 N0=0
...

<< memory.oom_control            # set/show oom controls. >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.oom_control :
oom_kill_disable 0
under_oom 0
...

<< memory.stat                   # show various statistics >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/memory.stat : cache
1363783680
rss 3229122560
rss_huge 1243611136
mapped_file 240046080
writeback 0
pgpgin 83895757
pgpgout 84990138
pgfault 66702779
pgmajfault 34733
inactive_anon 943763456
active_anon 2509733888
inactive_file 329670656
active_file 809648128
unevictable 0
hierarchical_memory_limit 9223372036854771712
total_cache 1363783680
total_rss 3229122560
total_rss_huge 1243611136
total_mapped_file 240046080
total_writeback 0
total_pgpgin 83895757
total_pgpgout 84990138
total_pgfault 66702779
total_pgmajfault 34733
total_inactive_anon 943763456
total_active_anon 2509733888
total_inactive_file 329670656
total_active_file 809648128
```

```
total_unevictable 0
...

<< memory.soft_limit_in_bytes    # set/show soft limit of memory usage >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
memory.soft_limit_in_bytes : 9223372036854771712 (9007199254740988 KB)(8796093022207.00
MB)(8589934591.00 GB) << 8 EB >>
/sys/fs/cgroup/memory/user.slice/user-1000.slice/session-c2.scope/
memory.kmem.tcp.failcnt :            0
#
```

Resource-
Hands on Linux sandbox with namespaces and cgroups

Source: RHEL 6 Resource Management Guide : Control Groups

...

1.2. Relationships Between Subsystems, Hierarchies, Control Groups and Tasks

Remember that system processes are called tasks in cgroup terminology.
Here are a few simple rules governing the relationships between subsystems, hierarchies of cgroups, and tasks, along with explanatory consequences of those rules.

Rule 1

A single hierarchy can have one or more subsystems attached to it.
As a consequence, the **`cpu`** and **`memory`** subsystems (or any number of subsystems) can be attached to a single hierarchy, as long as each one is not attached to any other hierarchy which has any other subsystems attached to it already (see Rule 2).

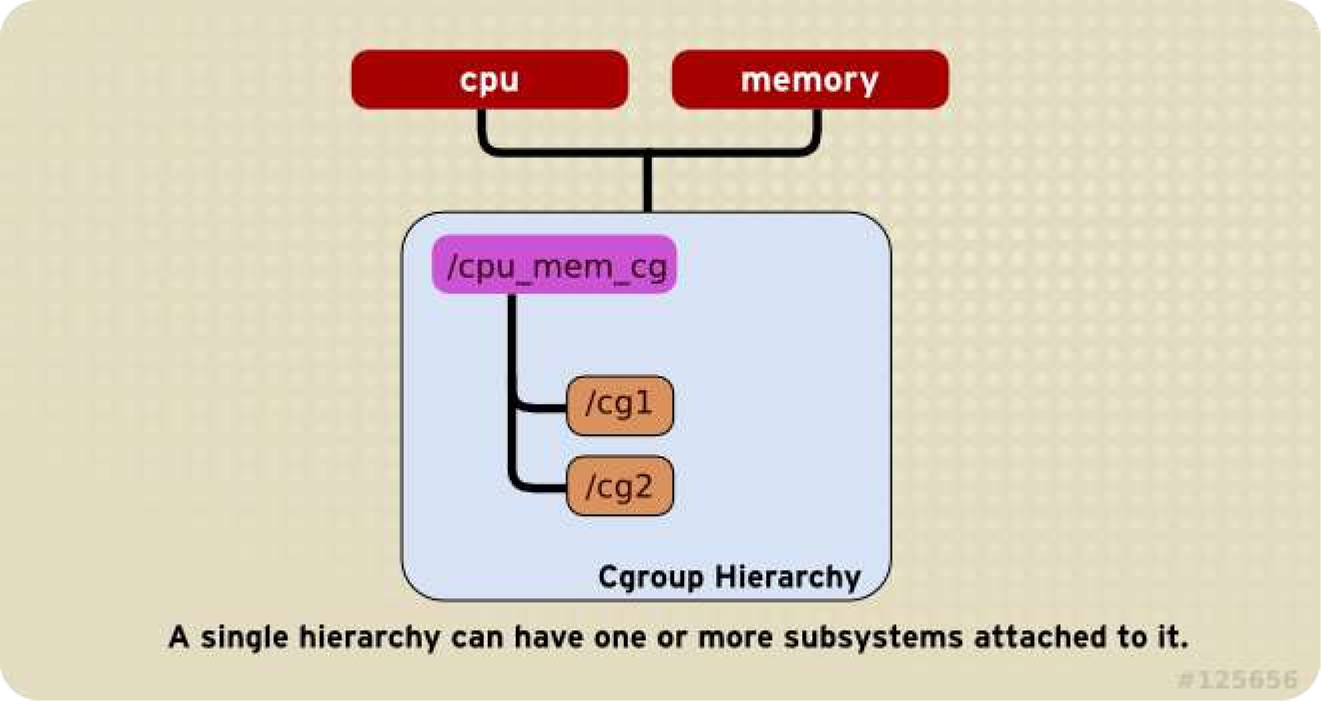

cpu
memory
/cpu_mem_cg
/cg1
/cg2
Cgroup Hierarchy
A single hierarchy can have one or more subsystems attached to it.
#125656

Figure 1.1. Rule 1

Rule 2

Any single subsystem (such as **`cpu`**) cannot be attached to more than one hierarchy if one of those hierarchies has a different subsystem attached to it already.

As a consequence, the **`cpu`** subsystem can never be attached to two different hierarchies if one of those hierarchies already has the **`memory`** subsystem attached to it. However, a single subsystem can be attached to two hierarchies if both of those hierarchies have only that subsystem attached.

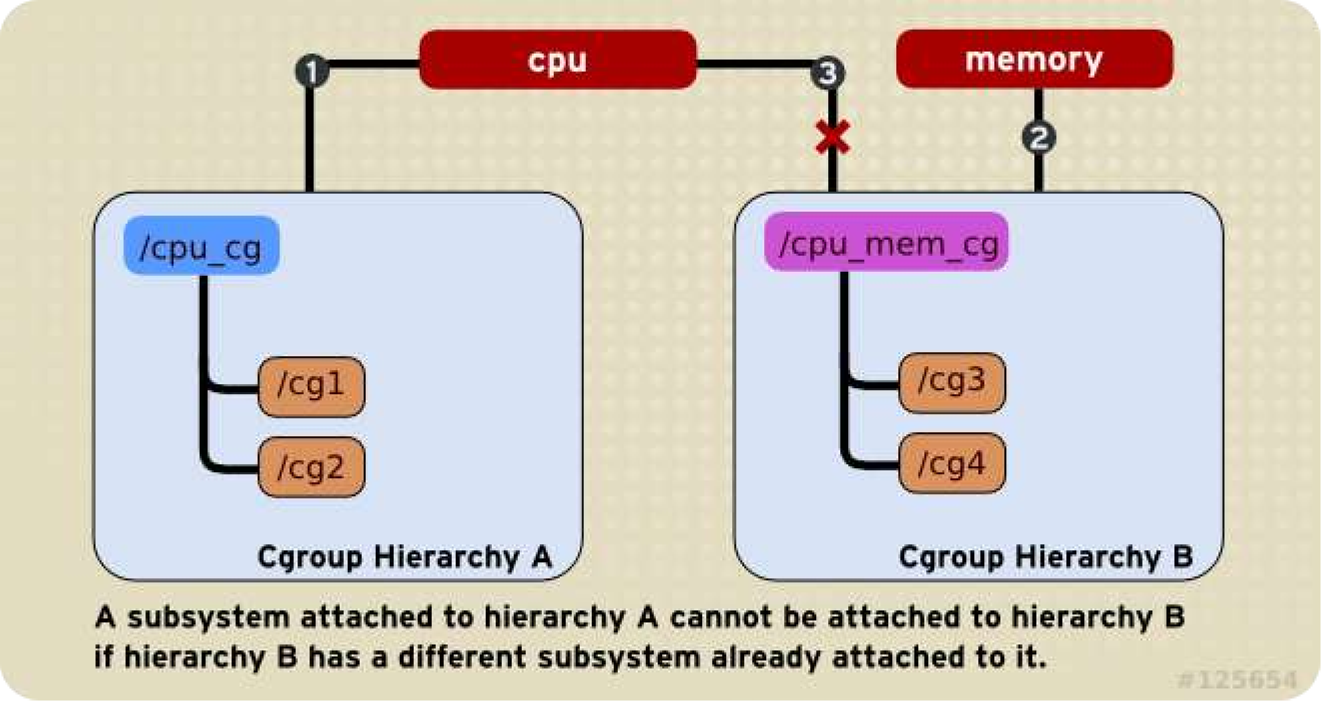

cpu
memory
1
2
3
/cpu_cg
/cg1
/cg2
Cgroup Hierarchy A
/cpu_mem_cg
/cg3
/cg4
Cgroup Hierarchy B
A subsystem attached to hierarchy A cannot be attached to hierarchy B
if hierarchy B has a different subsystem already attached to it.
#125654

Figure 1.2. Rule 2—The numbered bullets represent a time sequence in which the subsystems are attached.

Rule 3

Each time a new hierarchy is created on the systems, all tasks on the system are initially members of the default cgroup of that hierarchy, which is known as the root cgroup. For any single hierarchy you create, each task on the system can be a member of exactly one cgroup in that hierarchy. A single task may be in multiple cgroups, as long as each of those cgroups is in a different hierarchy. As soon as a task becomes a member of a second cgroup in the same hierarchy, it is removed from the first cgroup in that hierarchy. At no time is a task ever in two different cgroups in the same hierarchy.

As a consequence, if the **`cpu`** and **`memory`** subsystems are attached to a hierarchy named **`cpu_mem_cg`**, and the **`net_cls`** subsystem is attached to a hierarchy named **`net`**, then a running **`httpd`** process could be a member of any one cgroup in **`cpu_mem_cg`**, and any one cgroup in **`net`**.
The cgroup in **`cpu_mem_cg`** that the **`httpd`** process is a member of might restrict its CPU time to half of that allotted to other processes, and limit its memory usage to a maximum of **`1024`** MB. Additionally, the cgroup in **`net`**that it is a member of might limit its transmission rate to **`30`** megabytes per second.

When the first hierarchy is created, every task on the system is a member of at least one cgroup: the root cgroup. When using cgroups, therefore, every system task is always in at least one cgroup.

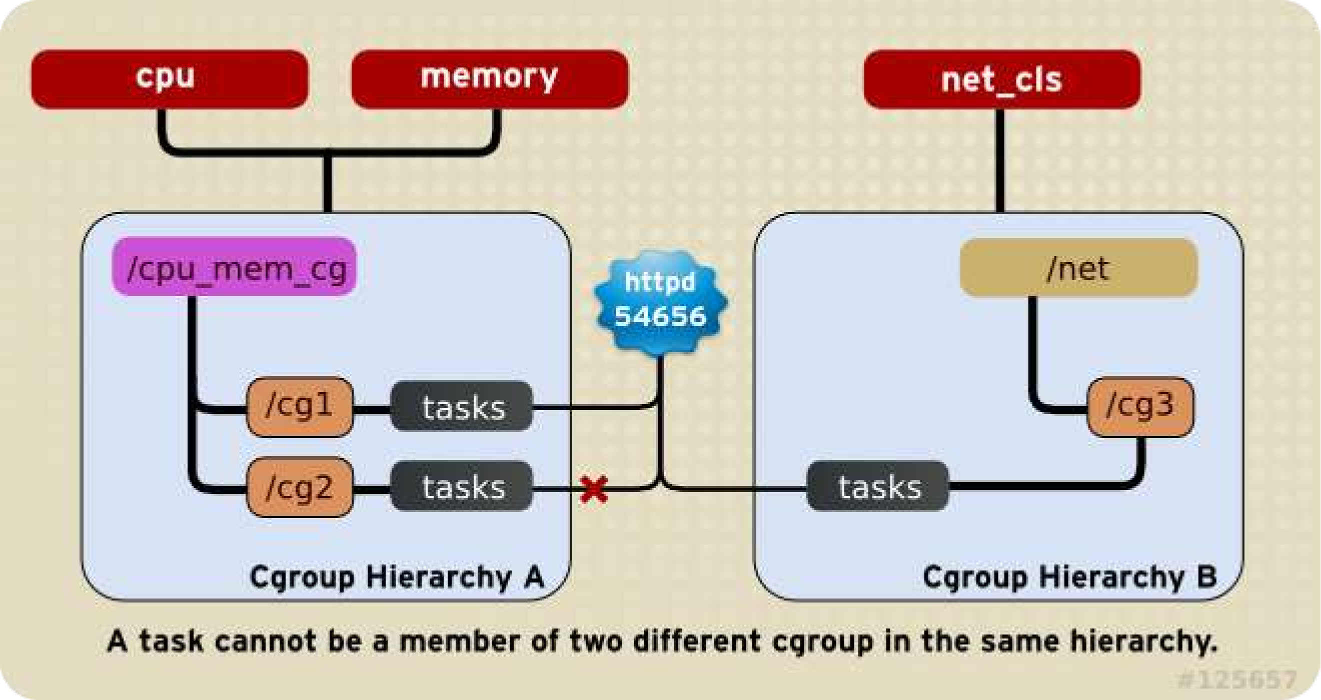


Figure 1.3. Rule 3

Rule 4

Any process (task) on the system which forks itself creates a child task. A child task automatically inherits the cgroup membership of its parent but can be moved to different cgroups as needed. Once forked, the parent and child processes are completely independent.

As a consequence, consider the **`httpd`** task that is a member of the cgroup named **`half_cpu_1gb_max`** in the **`cpu_and_mem`** hierarchy, and a member of the cgroup **`trans_rate_30`** in the **`net`** hierarchy. When that **`httpd`** process forks itself, its child process automatically becomes a member of the **`half_cpu_1gb_max`** cgroup, and the **`trans_rate_30`** cgroup. It inherits the exact same cgroups its parent task belongs to.
From that point forward, the parent and child tasks are completely independent of each other: changing the cgroups that one task belongs to does not affect the other. Neither will changing cgroups of a parent task affect any of its grandchildren in any way. To summarize: any child task always initially inherit memberships to the exact same cgroups as their parent task, but those memberships can be changed or removed later.

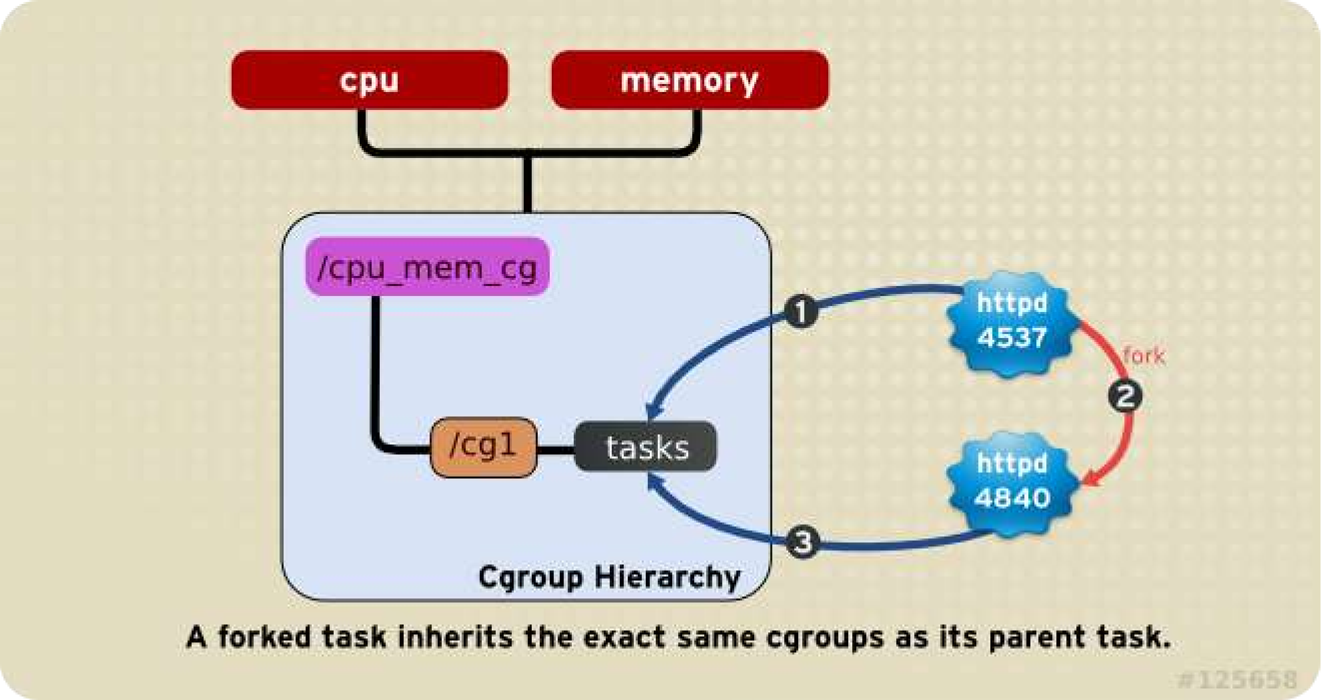


Figure 1.4. Rule 4—The numbered bullets represent a time sequence in which the task forks.

1.3. Implications for Resource Management

- Because a task can belong to only a single cgroup in any one hierarchy, there is only one way that a task can be limited or affected by any single subsystem. This is logical: a feature, not a limitation.
- You can group several subsystems together so that they affect all tasks in a single hierarchy. Because cgroups in that hierarchy have different parameters set, those tasks will be affected differently.
- It may sometimes be necessary to refactor a hierarchy. An example would be removing a subsystem from a hierarchy that has several subsystems attached, and attaching it to a new, separate hierarchy.
- Conversely, if the need for splitting subsystems among separate hierarchies is reduced, you can remove a hierarchy and attach its subsystems to an existing one.
- The design allows for simple cgroup usage, such as setting a few parameters for specific tasks in a single hierarchy, such as one with just the **`cpu`** and **`memory`** subsystems attached.

- The design also allows for highly specific configuration: each task (process) on a system could be a member of each hierarchy, each of which has a single attached subsystem. Such a configuration would give the system administrator absolute control over all parameters for every single task.

Ref:
https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-Creating_a_Hierarchy_and_Attaching_Subsystems.html

...

Example 2.3. Using the mount command to attach subsystems

In this example, a directory named `/cgroup/cpu_and_mem` already exists, which will serve as the mount point for the hierarchy that you create. Attach the `cpu`, `cpuset` and `memory` subsystems to a hierarchy named `cpu_and_mem`, and `mount` the `cpu_and_mem` hierarchy on `/cgroup/cpu_and_mem`:

```
~]# mount -t cgroup -o cpu,cpuset,memory cpu_and_mem /cgroup/cpu_and_mem
```

You can list all available subsystems along with their current mount points (i.e. where the hierarchy they are attached to is mounted) with the `lssubsys` [3] command:

```
~]# lssubsys -am
cpu,cpuset,memory /cgroup/cpu_and_mem
net_cls
ns
cpuacct
devices
freezer
blkio
```

This output indicates that:

- the `cpu`, `cpuset` and `memory` subsystems are attached to a hierarchy mounted on `/cgroup/cpu_and_mem`, and

the `net_cls`, `ns`, `cpuacct`, `devices`, `freezer` and `blkio` subsystems are as yet unattached to any hierarchy, as illustrated by the lack of a corresponding mount point.

Use REHL 7 documentation now:

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/

Resource_Management_Guide/chap-Introduction_to_Control_Groups.html#sec-What_are_Control_Groups

systemd-cgls(1)