



Red Hat Enterprise Linux 6.5 GA Resource Management Guide

Managing system resources on Red Hat Enterprise Linux 6
Edition 4

Martin Prpič
Douglas Silas

Martin Prpič

Rüdiger Landmann

Red Hat Enterprise Linux 6.5 GA Resource Management Guide

Managing system resources on Red Hat Enterprise Linux 6 Edition 4

Martin Prpič
Red Hat Engineering Content Services
mprpic@redhat.com

Rüdiger Landmann
Red Hat Engineering Content Services
r.landmann@redhat.com

Douglas Silas
Red Hat Engineering Content Services
dhensley@redhat.com

Legal Notice

Copyright 2013 Red Hat, Inc. This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed. Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law. Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries. Linux is the registered trademark of Linus Torvalds in the United States and other countries. Java is a registered trademark of Oracle and/or its affiliates. XFS is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries. MySQL is a registered trademark of MySQL AB in the United States, the European Union and other countries. Node.js is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project. The OpenStack Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community. All other trademarks are the property of their respective owners.

Keywords

Abstract

Managing system resources on Red Hat Enterprise Linux 6.

Table of Contents

Preface	4
1. Document Conventions	4
1.1. Typographic Conventions	4
1.2. Pull-quote Conventions	5
1.3. Notes and Warnings	6
2. Getting Help and Giving Feedback	6
2.1. Do You Need Help?	6
2.2. We Need Feedback	7
Chapter 1. Introduction to Control Groups (Cgroups)	8
1.1. How Control Groups Are Organized	8
The Linux Process Model	8
The Cgroup Model	8
1.2. Relationships Between Subsystems, Hierarchies, Control Groups and Tasks	9
Rule 1	9
Rule 2	10
Rule 3	11
Rule 4	11
1.3. Implications for Resource Management	12
Chapter 2. Using Control Groups	14
2.1. The cgconfig Service	14
2.1.1. The /etc/cgconfig.conf File	14
2.2. Creating a Hierarchy and Attaching Subsystems	17
Alternative method	17
2.3. Attaching Subsystems to, and Detaching Them From, an Existing Hierarchy	18
Alternative method	18
2.4. Unmounting a Hierarchy	19
2.5. Creating Control Groups	20
Alternative method	21
2.6. Removing Control Groups	21
2.7. Setting Parameters	21
Alternative method	22
2.8. Moving a Process to a Control Group	23
Alternative method	23
2.8.1. The cgroupd Service	23
2.9. Starting a Process in a Control Group	24
Alternative method	25
2.9.1. Starting a Service in a Control Group	25
2.9.2. Process Behavior in the Root Control Group	25
2.10. Generating the /etc/cgconfig.conf File	26
2.10.1. Blacklisting Parameters	28
2.10.2. Whitelisting Parameters	28
2.11. Obtaining Information About Control Groups	28
2.11.1. Finding a Process	28
2.11.2. Finding a Subsystem	29
2.11.3. Finding Hierarchies	29
2.11.4. Finding Control Groups	29
2.11.5. Displaying Parameters of Control Groups	30
2.12. Unloading Control Groups	30
2.13. Using the Notification API	30
2.14. Additional Resources	31

Chapter 3. Subsystems and Tunable Parameters	33
3.1. blkio	33
3.1.1. Proportional Weight Division Tunable Parameters	33
3.1.2. I/O Throttling Tunable Parameters	34
3.1.3. blkio Common Tunable Parameters	35
3.1.4. Example Usage	37
3.2. cpu	39
3.2.1. CFS Tunable Parameters	39
3.2.2. RT Tunable Parameters	41
3.2.3. Example Usage	41
3.3. cpuacct	42
3.4. cpuset	43
3.5. devices	45
3.6. freezer	46
3.7. memory	47
3.7.1. Example Usage	50
3.8. net_cls	54
3.9. net_prio	54
3.10. ns	55
3.11. perf_event	55
3.12. Common Tunable Parameters	56
3.13. Additional Resources	57
Chapter 4.. Control Group Application Examples	59
4.1. Prioritizing Database I/O	59
4.2. Prioritizing Network Traffic	60
4.3. Per-group Division of CPU and Memory Resources	62
Alternative method	65
Revision History	66

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](#) set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later include the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keys and key combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, select the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** →

Character Map from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic** or **Proportional Bold Italic

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh *john@example.com***.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount */home***.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
static int kvm_vm_ioctl_deassign_device(struct kvm *kvm,
                                       struct kvm_assigned_pci_dev *assigned_dev)
{
    int r = 0;
    struct kvm_assigned_dev_kernel *match;

    mutex_lock(&kvm->lock);

    match = kvm_find_assigned_dev(&kvm->arch.assigned_dev_head,
                                assigned_dev->assigned_dev_id);
    if (!match) {
        printk(KERN_INFO "%s: device hasn't been assigned before, "
                    "so cannot be deassigned\n", __func__);
        r = -EINVAL;
        goto out;
    }

    kvm_deassign_device(kvm, match);

    kvm_free_assigned_device(kvm, match);

out:
    mutex_unlock(&kvm->lock);
    return r;
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer

Portal at <http://access.redhat.com>. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at <https://www.redhat.com/mailman/listinfo>. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. We Need Feedback

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you. Please submit a report in Bugzilla: <http://bugzilla.redhat.com/> against the product **Red Hat Enterprise Linux 6**.

When submitting a bug report, be sure to mention the manual's identifier: *doc-Resource_Management_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Chapter 1. Introduction to Control Groups (Cgroups)

Red Hat Enterprise Linux 6 provides a new kernel feature: *control groups*, which are called by their shorter name *cgroups* in this guide. Cgroups allow you to allocate resources—such as CPU time, system memory, network bandwidth, or combinations of these resources—among user-defined groups of tasks (processes) running on a system. You can monitor the cgroups you configure, deny cgroups access to certain resources, and even reconfigure your cgroups dynamically on a running system. The **cgconfig** (*control group config*) service can be configured to start up at boot time and reestablish your predefined cgroups, thus making them persistent across reboots.

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

1.1. How Control Groups Are Organized

Cgroups are organized hierarchically, like processes, and child cgroups inherit some of the attributes of their parents. However, there are differences between the two models.

The Linux Process Model

All processes on a Linux system are child processes of a common parent: the **init** process, which is executed by the kernel at boot time and starts other processes (which may in turn start child processes of their own). Because all processes descend from a single parent, the Linux process model is a single hierarchy, or tree.

Additionally, every Linux process except **init** inherits the environment (such as the `PATH` variable) ^[1] and certain other attributes (such as open file descriptors) of its parent process.

The Cgroup Model

Cgroups are similar to processes in that:

- they are hierarchical, and
- child cgroups inherit certain attributes from their parent cgroup.

The fundamental difference is that many different hierarchies of cgroups can exist simultaneously on a system. If the Linux process model is a single tree of processes, then the cgroup model is one or more separate, unconnected trees of tasks (i.e. processes).

Multiple separate hierarchies of cgroups are necessary because each hierarchy is attached to *one or more subsystems*. A subsystem ^[2] represents a single resource, such as CPU time or memory. Red Hat Enterprise Linux 6 provides ten cgroup subsystems, listed below by name and function.

Available Subsystems in Red Hat Enterprise Linux

- **blkio** — this subsystem sets limits on input/output access to and from block devices such as physical drives (disk, solid state, USB, etc.).
- **cpu** — this subsystem uses the scheduler to provide cgroup tasks access to the CPU.
- **cpuacct** — this subsystem generates automatic reports on CPU resources used by tasks in a cgroup.
- **cpuset** — this subsystem assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
- **devices** — this subsystem allows or denies access to devices by tasks in a cgroup.

- **freezer** — this subsystem suspends or resumes tasks in a cgroup.
- **memory** — this subsystem sets limits on memory use by tasks in a cgroup, and generates automatic reports on memory resources used by those tasks.
- **net_cls** — this subsystem tags network packets with a class identifier (classid) that allows the Linux traffic controller (**tc**) to identify packets originating from a particular cgroup task.
- **net_prio** — this subsystem provides a way to dynamically set the priority of network traffic per network interface.
- **ns** — the *namespace* subsystem.



Subsystems are also known as resource controllers

You may come across the term *resource controller* or simply *controller* in cgroup literature such as the man pages or kernel documentation. Both of these terms are synonymous with “subsystem”, and arise from the fact that a subsystem typically schedules a resource or applies a limit to the cgroups in the hierarchy it is attached to.

The definition of a subsystem (resource controller) is quite general: it is something that acts upon a group of tasks, i.e. processes.

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).*

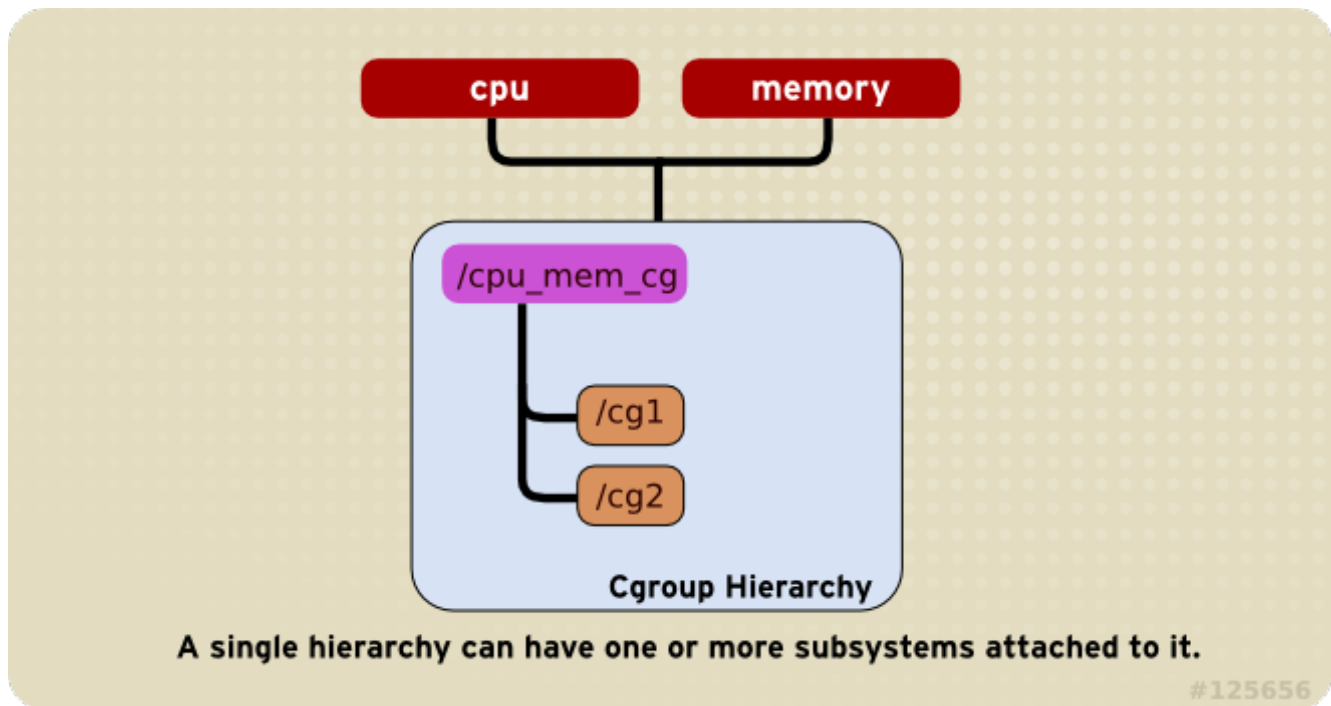


Figure 1.1. Rule 1

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.

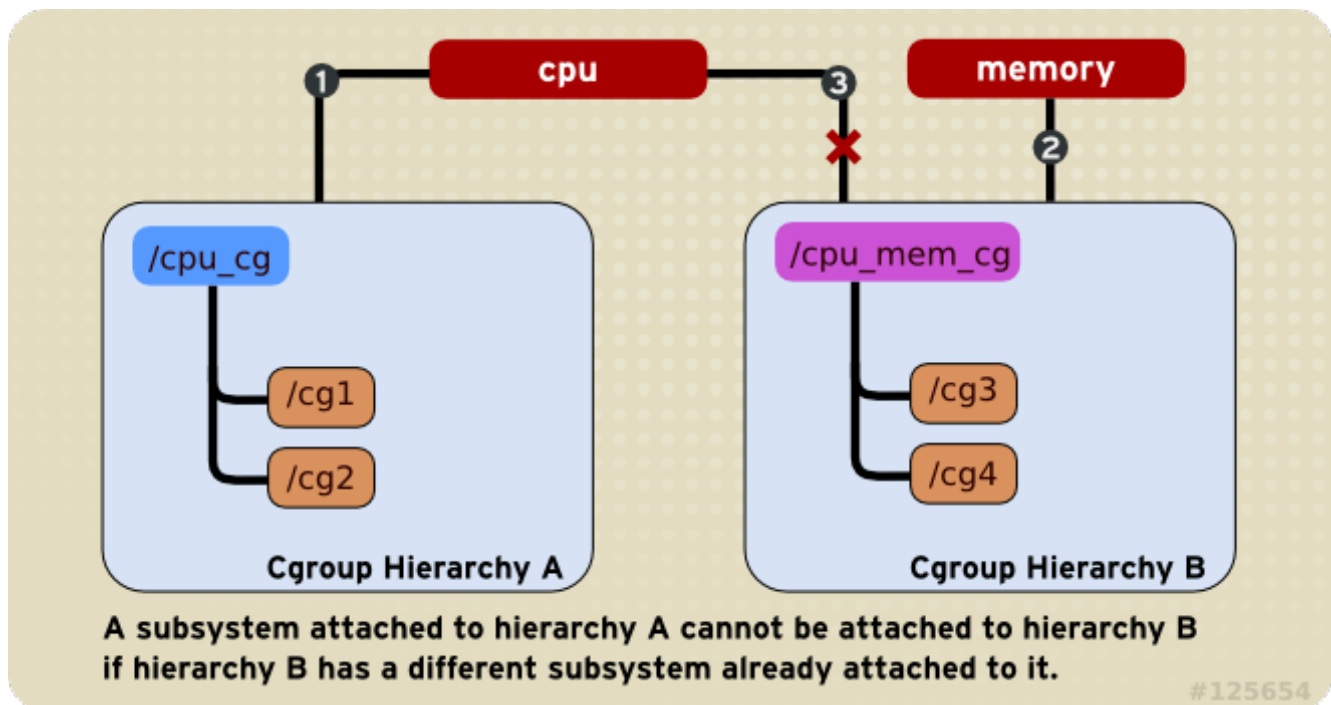


Figure 1.2. Rule 2—The numbered bullets represent a time sequence in which the subsystems are attached.

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.

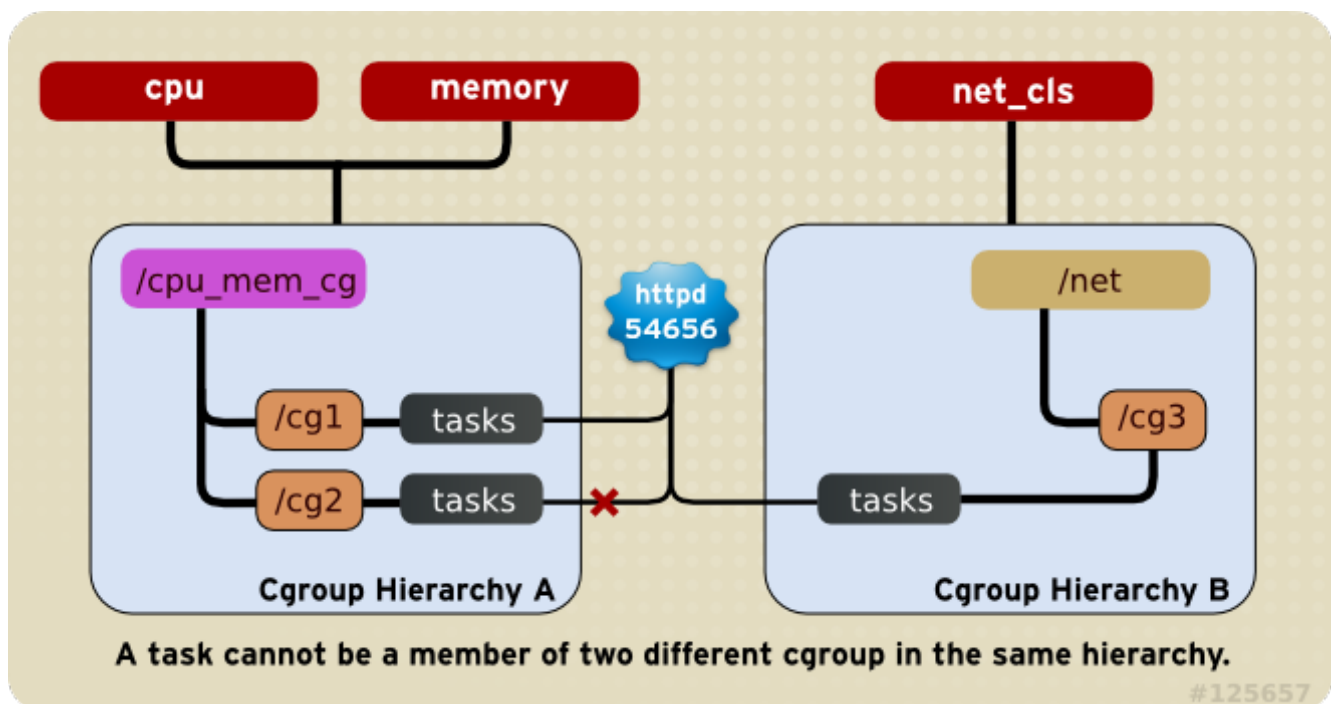


Figure 1.3. Rule 3

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.

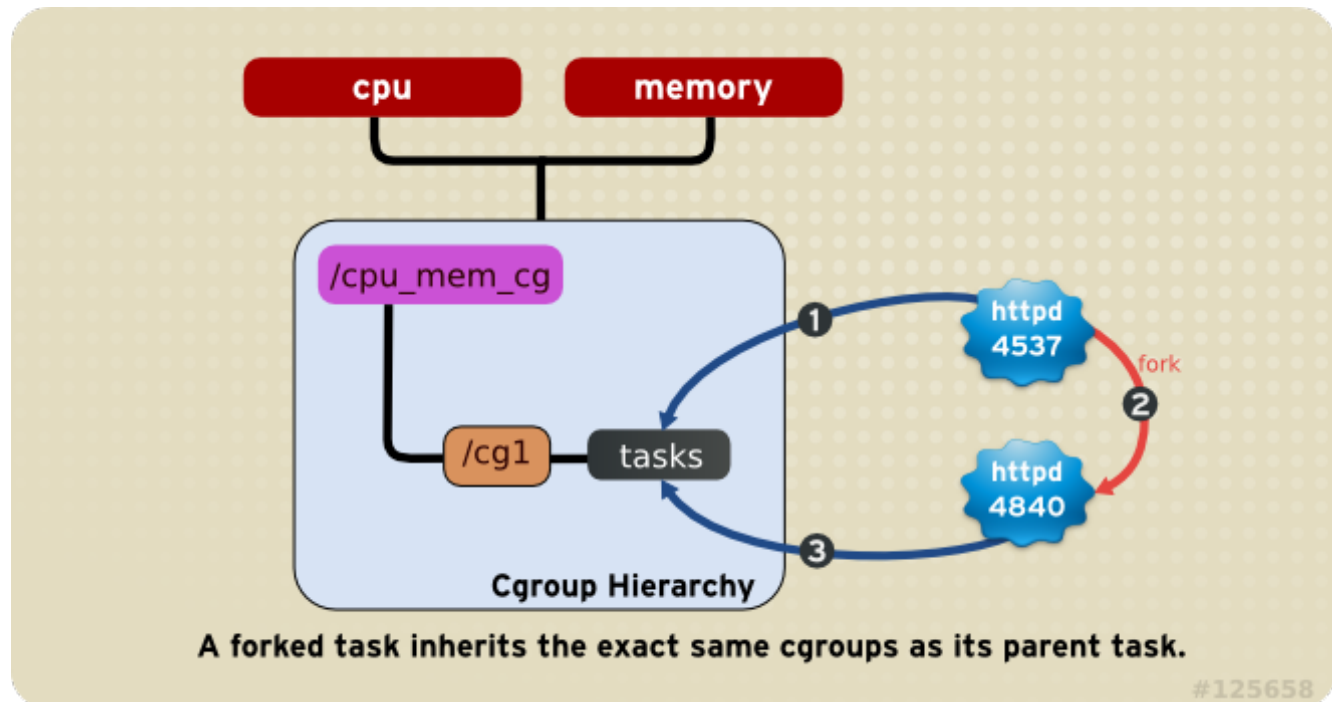


Figure 1.4. Rule 4—The numbered bullets represent a time sequence in which the task forks.

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.

.....
[1] The parent process is able to alter the environment before passing it to a child process.

[2] You should be aware that subsystems are also called *resource controllers*, or simply *controllers*, in the *libcgroup* man pages and other documentation.

Chapter 2. Using Control Groups

As explained in [Chapter 3, Subsystems and Tunable Parameters](#), control groups and the subsystems to which they relate can be manipulated using shell commands and utilities. However, the easiest way to work with cgroups is to install the *libcgroup* package, which contains a number of cgroup-related command line utilities and their associated man pages. It is possible to *mount* hierarchies and set cgroup parameters (non-persistently) using shell commands and utilities available on any system. However, using the *libcgroup*-provided utilities simplifies the process and extends your capabilities. Therefore, this guide focuses on *libcgroup* commands throughout. In most cases, we have included the equivalent shell commands to help describe the underlying mechanism. However, we recommend that you use the *libcgroup* commands wherever practical.



Installing the libcgroup package

In order to use cgroups, first ensure the *libcgroup* package is installed on your system by running, as root:

```
~]# yum install libcgroup
```

2.1. The cgconfig Service

The **cgconfig** service installed with the *libcgroup* package provides a convenient way to create hierarchies, attach subsystems to hierarchies, and manage cgroups within those hierarchies. It is recommended that you use **cgconfig** to manage hierarchies and cgroups on your system.

The **cgconfig** service is not started by default on Red Hat Enterprise Linux 6. When you start the service with **chkconfig**, it reads the cgroup configuration file — */etc/cgconfig.conf*. Cgroups are therefore recreated from session to session and remain persistent. Depending on the contents of the configuration file, **cgconfig** can create hierarchies, mount necessary file systems, create cgroups, and set subsystem parameters for each group.

The default */etc/cgconfig.conf* file installed with the *libcgroup* package creates and mounts an individual hierarchy for each subsystem, and attaches the subsystems to these hierarchies.

If you stop the **cgconfig** service (with the **service cgconfig stop** command), it unmounts all the hierarchies that it mounted.

2.1.1. The /etc/cgconfig.conf File

The */etc/cgconfig.conf* file contains two major types of entry — *mount* and *group*. Mount entries create and mount hierarchies as virtual file systems, and attach subsystems to those hierarchies. Mount entries are defined using the following syntax:

```
mount {  
    subsystem = /cgroup/hierarchy;  
    ...  
}
```

See [Example 2.1, “Creating a mount entry”](#) for an example usage.

Example 2.1. Creating a mount entry

The following example creates a hierarchy for the **cpuset** subsystem:

```
mount {
    cpuset = /cgroup/red;
}
```

the equivalent of the shell commands:

```
~]# mkdir /cgroup/red
~]# mount -t cgroup -o cpuset red /cgroup/red
```

Group entries create cgroups and set subsystem parameters. Group entries are defined using the following syntax:

```
group <name> {
    [<permissions>]
    <controller> {
        <param name> = <param value>;
        ...
    }
    ...
}
```

Note that the **permissions** section is optional. To define permissions for a group entry, use the following syntax:

```
perm {
    task {
        uid = <task user>;
        gid = <task group>;
    }
    admin {
        uid = <admin name>;
        gid = <admin group>;
    }
}
```

See [Example 2.2, “Creating a group entry”](#) for example usage:

Example 2.2. Creating a group entry

The following example creates a cgroup for SQL daemons, with permissions for users in the **sqladmin** group to add tasks to the cgroup and the **root** user to modify subsystem parameters:

```
group daemons {
    cpuset {
        cpuset.mems = 0;
        cpuset.cpus = 0;
    }
}
group daemons/sql {
    perm {
        task {
            uid = root;
            gid = sqladmin;
        } admin {
            uid = root;
            gid = root;
        }
    }
    cpuset {
        cpuset.mems = 0;
        cpuset.cpus = 0;
    }
}
```

When combined with the example of the mount entry in [Example 2.1, “Creating a mount entry”](#), the equivalent shell commands are:

```
~]# mkdir -p /cgroup/red/daemons/sql
~]# chown root:root /cgroup/red/daemons/sql/*
~]# chown root:sqladmin /cgroup/red/daemons/sql/tasks
~]# echo 0 > /cgroup/red/daemons/cpuset.mems
~]# echo 0 > /cgroup/red/daemons/cpuset.cpus
~]# echo 0 > /cgroup/red/daemons/sql/cpuset.mems
~]# echo 0 > /cgroup/red/daemons/sql/cpuset.cpus
```



Restart the cgconfig service for the changes to take effect

You must restart the **cgconfig** service for the changes in the **/etc/cgconfig.conf** to take effect. However, note that restarting this service causes the entire cgroup hierarchy to be rebuilt, which removes any previously existing cgroups (for example, any existing cgroups used by **libvirtd**). To restart the **cgconfig** service, use the following command:

```
~]# service cgconfig restart
```

When you install the **libcgroup** package, a sample configuration file is written to **/etc/cgconfig.conf**. The hash symbols (**#**) at the start of each line comment that line out and make it invisible to the **cgconfig** service.

2.2. Creating a Hierarchy and Attaching Subsystems



Effects on running systems

The following instructions, which cover creating a new hierarchy and attaching subsystems to it, assume that cgroups are not already configured on your system. In this case, these instructions will not affect the operation of the system. Changing the tunable parameters in a cgroup with tasks, however, may immediately affect those tasks. This guide alerts you the first time it illustrates changing a tunable cgroup parameter that may affect one or more tasks.

On a system on which cgroups are already configured (either manually, or by the **cgconfig** service) these commands will fail unless you first unmount existing hierarchies, which will affect the operation of the system. Do not experiment with these instructions on production systems.

To create a hierarchy and attach subsystems to it, edit the **mount** section of the **/etc/cgconfig.conf** file as root. Entries in the **mount** section have the following format:

```
subsystem = /cgroup/hierarchy;
```

When **cgconfig** next starts, it will create the hierarchy and attach the subsystems to it.

The following example creates a hierarchy called **cpu_and_mem** and attaches the **cpu**, **cpuset**, **cpuacct**, and **memory** subsystems to it.

```
mount {
    cpuset  = /cgroup/cpu_and_mem;
    cpu     = /cgroup/cpu_and_mem;
    cpuacct = /cgroup/cpu_and_mem;
    memory  = /cgroup/cpu_and_mem;
}
```

Alternative method

You can also use shell commands and utilities to create hierarchies and attach subsystems to them.

Create a *mount point* for the hierarchy as root. Include the name of the cgroup in the mount point:

```
~]# mkdir /cgroup/name
```

For example:

```
~]# mkdir /cgroup/cpu_and_mem
```

Next, use the **mount** command to mount the hierarchy and simultaneously attach one or more subsystems. For example:

```
~]# mount -t cgroup -o subsystems name /cgroup/name
```

Where ***subsystems*** is a comma-separated list of subsystems and ***name*** is the name of the hierarchy. Brief descriptions of all available subsystems are listed in [Available Subsystems in Red Hat Enterprise Linux](#), and [Chapter 3, Subsystems and Tunable Parameters](#) provides a detailed reference.

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 **ls subsystems** ^[3] command:

```
~]# ls subsystems -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.

2.3. Attaching Subsystems to, and Detaching Them From, an Existing Hierarchy

To add a subsystem to an existing hierarchy, detach it from an existing hierarchy, or move it to a different hierarchy, edit the **mount** section of the `/etc/cgconfig.conf` file as root, using the same syntax described in [Section 2.2, “Creating a Hierarchy and Attaching Subsystems”](#). When **cgconfig** next starts, it will reorganize the subsystems according to the hierarchies that you specify.

Alternative method

To add an unattached subsystem to an existing hierarchy, remount the hierarchy. Include the extra subsystem in the **mount** command, together with the **remount** option.

Example 2.4. Remounting a hierarchy to add a subsystem

The **lssubsys** command shows **cpu**, **cpuset**, and **memory** subsystems attached to the **cpu_and_mem** hierarchy:

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

Remount the **cpu_and_mem** hierarchy, using the **remount** option, and include **cpuacct** in the list of subsystems:

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

The **lssubsys** command now shows **cpuacct** attached to the **cpu_and_mem** hierarchy:

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

Analogously, you can detach a subsystem from an existing hierarchy by remounting the hierarchy and omitting the subsystem name from the **-o** options. For example, to then detach the **cpuacct** subsystem, simply remount and omit it:

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

2.4. Unmounting a Hierarchy

You can *unmount* a hierarchy of cgroups with the **umount** command:

```
~]# umount /cgroup/name
```

For example:

```
~]# umount /cgroup/cpu_and_mem
```

If the hierarchy is currently empty (that is, it contains only the root cgroup) the hierarchy is deactivated when it is unmounted. If the hierarchy contains any other cgroups, the hierarchy remains active in the kernel even though it is no longer mounted.

To remove a hierarchy, ensure that all child cgroups are removed before you unmount the hierarchy, or

use the **cgclear** command which can deactivate a hierarchy even when it is not empty — refer to [Section 2.12, “Unloading Control Groups”](#).

2.5. Creating Control Groups

Use the **cgcreate** command to create cgroups. The syntax for **cgcreate** is:

```
cgcreate -t uid:gid -a uid:gid -g subsystems:path
```

where:

- **-t** (optional) — specifies a user (by user ID, uid) and a group (by group ID, gid) to own the **tasks** pseudo-file for this cgroup. This user can add tasks to the cgroup.



Removing tasks

Note that the only way to remove a task from a cgroup is to move it to a different cgroup. To move a task, the user must have write access to the *destination* cgroup; write access to the source cgroup is unimportant.

- **-a** (optional) — specifies a user (by user ID, uid) and a group (by group ID, gid) to own all pseudo-files other than **tasks** for this cgroup. This user can modify the access that the tasks in this cgroup have to system resources.
- **-g** — specifies the hierarchy in which the cgroup should be created, as a comma-separated list of the **subsystems** associated with those hierarchies. If the subsystems in this list are in different hierarchies, the group is created in each of these hierarchies. The list of hierarchies is followed by a colon and the **path** to the child group relative to the hierarchy. Do not include the hierarchy mount point in the path.

For example, the cgroup located in the directory **/cgroup/cpu_and_mem/lab1/** is called just **lab1** — its path is already uniquely determined because there is at most one hierarchy for a given subsystem. Note also that the group is controlled by all the subsystems that exist in the hierarchies in which the cgroup is created, even though these subsystems have not been specified in the **cgcreate** command — refer to [Example 2.5, “cgcreate usage”](#).

Because all cgroups in the same hierarchy have the same controllers, the child group has the same controllers as its parent.

Example 2.5. cgcreate usage

Consider a system where the **cpu** and **memory** subsystems are mounted together in the **cpu_and_mem** hierarchy, and the **net_cls** controller is mounted in a separate hierarchy called **net**. Run the following command:

```
~]# cgcreate -g cpu,net_cls:/test-subgroup
```

The **cgcreate** command creates two groups named **test-subgroup**, one in the **cpu_and_mem** hierarchy and one in the **net** hierarchy. The **test-subgroup** group in the **cpu_and_mem** hierarchy is controlled by the **memory** subsystem, even though it was not specified in the **cgcreate** command.

Alternative method

To create a child of the cgroup directly, use the **mkdir** command:

```
~]# mkdir /cgroup/hierarchy/name/child_name
```

For example:

```
~]# mkdir /cgroup/cpuset/lab1/group1
```

2.6. Removing Control Groups

Remove cgroups with the **cgdelete**, which has a syntax similar to that of **cgcreate**. Run the following command:

```
cgdelete subsystems:path
```

where:

- ▶ **subsystems** is a comma-separated list of subsystems.
- ▶ **path** is the path to the cgroup relative to the root of the hierarchy.

For example:

```
~]# cgdelete cpu,net_cls:/test-subgroup
```

cgdelete can also recursively remove all subgroups with the option **-r**.

When you delete a cgroup, all its tasks move to its parent group.

2.7. Setting Parameters

Set subsystem parameters by running the **cgset** command from a user account with permission to modify the relevant cgroup. For example, if **/cgroup/cpuset/group1** exists, specify the CPUs to which this group has access with the following command:

```
cpuset]# cgset -r cpuset.cpus=0-1 group1
```

The syntax for **cgset** is:

```
cgset -r parameter=value path_to_cgroup
```

where:

- ▶ **parameter** is the parameter to be set, which corresponds to the file in the directory of the given cgroup
- ▶ **value** is the value for the parameter
- ▶ **path_to_cgroup** is the path to the cgroup *relative to the root of the hierarchy*. For example, to set the parameter of the root group (if **/cgroup/cpuacct/** exists), run:

```
cpuacct]# cgset -r cpuacct.usage=0 /
```

Alternatively, because `.` is relative to the root group (that is, the root group itself) you could also run:

```
cpuacct]# cgset -r cpuacct.usage=0 .
```

Note, however, that `/` is the preferred syntax.



Setting parameters for the root group

Only a small number of parameters can be set for the root group (such as the **cpuacct.usage** parameter shown in the examples above). This is because a root group owns all of the existing resources, therefore, it would make no sense to limit all existing processes by defining certain parameters, for example the **cpuset.cpu** parameter.

To set the parameter of **group1**, which is a subgroup of the root group, run:

```
cpuacct]# cgset -r cpuacct.usage=0 group1
```

A trailing slash on the name of the group (for example, **cpuacct.usage=0 group1/**) is optional.

The values that you can set with **cgset** might depend on values set higher in a particular hierarchy. For example, if **group1** is limited to use only CPU 0 on a system, you cannot set **group1/subgroup1** to use CPUs 0 and 1, or to use only CPU 1.

You can also use **cgset** to copy the parameters of one cgroup into another, existing cgroup. For example:

```
~]# cgset --copy-from group1/ group2/
```

The syntax to copy parameters with **cgset** is:

```
cgset --copy-from path_to_source_cgroup path_to_target_cgroup
```

where:

- ▶ ***path_to_source_cgroup*** is the path to the cgroup whose parameters are to be copied, relative to the root group of the hierarchy
- ▶ ***path_to_target_cgroup*** is the path to the destination cgroup, relative to the root group of the hierarchy

Ensure that any mandatory parameters for the various subsystems are set before you copy parameters from one group to another, or the command will fail. For more information on mandatory parameters, refer to [Mandatory parameters](#).

Alternative method

To set parameters in a cgroup directly, insert values into the relevant subsystem pseudo-file using the **echo** command. For example, this command inserts the value **0-1** into the **cpuset.cpus** pseudo-file of the cgroup **group1**:

```
~]# echo 0-1 > /cgroup/cpuset/group1/cpuset.cpus
```

With this value in place, the tasks in this cgroup are restricted to CPUs 0 and 1 on the system.

2.8. Moving a Process to a Control Group

Move a process into a cgroup by running the **cgclassify** command, for example:

```
~]# cgclassify -g cpu,memory:group1 1701
```

The syntax for **cgclassify** is:

```
cgclassify -g subsystems:path_to_cgroup pidlist
```

where:

- ▶ **subsystems** is a comma-separated list of subsystems, or ***** to launch the process in the hierarchies associated with all available subsystems. Note that if cgroups of the same name exist in multiple hierarchies, the **-g** option moves the processes in each of those groups. Ensure that the cgroup exists within each of the hierarchies whose subsystems you specify here.
- ▶ **path_to_cgroup** is the path to the cgroup within its hierarchies
- ▶ **pidlist** is a space-separated list of *process identifier* (PIDs)

You can also add the **--sticky** option before the **pid** to keep any child processes in the same cgroup. If you do not set this option and the **cgrd** service is running, child processes will be allocated to cgroups based on the settings found in **/etc/cgrules.conf**. The process itself, however, will remain in the cgroup in which you started it.

Using **cgclassify**, you can move several processes simultaneously. For example, this command moves the processes with PIDs **1701** and **1138** into cgroup **group1/**:

```
~]# cgclassify -g cpu,memory:group1 1701 1138
```

Note that the PIDs to be moved are separated by spaces and that the groups specified should be in different hierarchies.

Alternative method

To move a process into a cgroup directly, write its PID to the **tasks** file of the cgroup. For example, to move a process with the PID **1701** into a cgroup at **/cgroup/lab1/group1/**:

```
~]# echo 1701 > /cgroup/lab1/group1/tasks
```

2.8.1. The cgrd Service

Cgrd is a service (which starts the **cgrulesengd** daemon) that moves tasks into cgroups according to parameters set in the **/etc/cgrules.conf** file. Entries in the **/etc/cgrules.conf** file can take one of the two forms:

- ▶ **user subsystems control_group**
- ▶ **user:command subsystems control_group**

For example:

```
maria    devices    /usergroup/staff
```

This entry specifies that any processes that belong to the user named **maria** access the **devices**

subsystem according to the parameters specified in the `/usergroup/staff` cgroup. To associate particular commands with particular cgroups, add the **command** parameter, as follows:

```
maria:ftp devices /usergroup/staff/ftp
```

The entry now specifies that when the user named **maria** uses the **ftp** command, the process is automatically moved to the `/usergroup/staff/ftp` cgroup in the hierarchy that contains the **devices** subsystem. Note, however, that the daemon moves the process to the cgroup only after the appropriate condition is fulfilled. Therefore, the **ftp** process might run for a short time in the wrong group. Furthermore, if the process quickly spawns children while in the wrong group, these children might not be moved.

Entries in the `/etc/cgrouules.conf` file can include the following extra notation:

- ▶ **@** — when prefixed to **user**, indicates a group instead of an individual user. For example, **@admins** are all users in the **admins** group.
- ▶ ***** — represents "all". For example, ***** in the **subsystem** field represents all subsystems.
- ▶ **%** — represents an item the same as the item in the line above. For example:

```
@adminstaff devices /admingroup
@labstaff % %
```

2.9. Starting a Process in a Control Group



Mandatory parameters

Some subsystems have mandatory parameters that must be set before you can move a task into a cgroup which uses any of those subsystems. For example, before you move a task into a cgroup which uses the **cpuset** subsystem, the **cpuset.cpus** and **cpuset.mems** parameters must be defined for that cgroup.

The examples in this section illustrate the correct syntax for the command, but only work on systems on which the relevant mandatory parameters have been set for any controllers used in the examples. If you have not already configured the relevant controllers, you cannot copy example commands directly from this section and expect them to work on your system.

Refer to [Chapter 3, Subsystems and Tunable Parameters](#) for a description of which parameters are mandatory for given subsystems.

Launch processes in a cgroup by running the **cgexec** command. For example, this command launches the **lynx** web browser within the **group1** cgroup, subject to the limitations imposed on that group by the **cpu** subsystem:

```
~]# cgexec -g cpu:group1 lynx http://www.redhat.com
```

The syntax for **cgexec** is:

```
cgexec -g subsystems:path_to_cgroup command arguments
```

where:

- ▶ **subsystems** is a comma-separated list of subsystems, or ***** to launch the process in the hierarchies

associated with all available subsystems. Note that, as with **cgset** described in [Section 2.7, “Setting Parameters”](#), if cgroups of the same name exist in multiple hierarchies, the **-g** option creates processes in each of those groups. Ensure that the cgroup exists within each of the hierarchies whose subsystems you specify here.

- **path_to_cgroup** is the path to the cgroup relative to the hierarchy.
- **command** is the command to run.
- **arguments** are any arguments for the command.

You can also add the **--sticky** option before the **command** to keep any child processes in the same cgroup. If you do not set this option and the **cgred** daemon is running, child processes will be allocated to cgroups based on the settings found in **/etc/cgrules.conf**. The process itself, however, will remain in the cgroup in which you started it.

Alternative method

When you start a new process, it inherits the group of its parent process. Therefore, an alternative method for starting a process in a particular cgroup is to move your shell process to that group (refer to [Section 2.8, “Moving a Process to a Control Group”](#)), and then launch the process from that shell. For example:

```
~]# echo $$ > /cgroup/lab1/group1/tasks
~]# lynx
```

Note that after exiting **lynx**, your existing shell is still in the **group1** cgroup. Therefore, an even better way would be:

```
~]# sh -c "echo \$$ > /cgroup/lab1/group1/tasks && lynx"
```

2.9.1. Starting a Service in a Control Group

You can start certain services in a cgroup. Services that can be started in cgroups must:

- use a **/etc/sysconfig/service** file
- use the **daemon()** function from **/etc/init.d/functions** to start the service

To make an eligible service start in a cgroup, edit its file in the **/etc/sysconfig** directory to include an entry in the form **CGROUP_DAEMON="subsystem:control_group"** where **subsystem** is a subsystem associated with a particular hierarchy, and **control_group** is a cgroup in that hierarchy. For example:

```
CGROUP_DAEMON="cpuset:daemons/sql"
```

2.9.2. Process Behavior in the Root Control Group

Certain **blkio** and **cpu** configuration options affect processes (tasks) running in the root cgroup in a different way than those in a subgroup. Consider the following example:

1. Create two subgroups under one root group: **/rootgroup/red/** and **/rootgroup/blue/**
2. In each subgroup and in the root group, define the **cpu.shares** configuration option and set it to 1.

In the scenario configured above, one process placed in each group (that is, one task in **/rootgroup/tasks**, **/rootgroup/red/tasks** and **/rootgroup/blue/tasks**) ends up consuming 33.33% of the CPU:

```
/rootgroup/ process:      33.33%  
/rootgroup/blue/ process: 33.33%  
/rootgroup/red/  process: 33.33%
```

Any other processes placed in subgroups **blue** and **red** result in the 33.33% percent of the CPU assigned to that specific subgroup to be split among the multiple processes in that subgroup.

However, multiple processes placed in the root group cause the CPU resource to be split per process, rather than per group. For example, if **/rootgroup/** contains three processes, **/rootgroup/red/** contains one process and **/rootgroup/blue/** contains one process, and the **cpu.shares** option is set to **1** in all groups, the CPU resource is divided as follows:

```
/rootgroup/ processes:    20% + 20% + 20%  
/rootgroup/blue/ process: 20%  
/rootgroup/red/  process: 20%
```

Therefore, it is recommended to move all processes from the root group to a specific subgroup when using the **blkio** and **cpu** configuration options which divide an available resource based on a weight or a share (for example, **cpu.shares** or **blkio.weight**). To move all tasks from the root group into a specific subgroup, you can use the following commands:

```
rootgroup]# cat tasks >> red/tasks  
rootgroup]# echo > tasks
```

2.10. Generating the **/etc/cgconfig.conf** File

Configuration for the **/etc/cgconfig.conf** file can be generated from the current cgroup configuration using the **cgsnapshot** utility. This utility takes a snapshot of the current state of all subsystems and their cgroups and returns their configuration as it would appear in the **/etc/cgconfig.conf** file. [Example 2.6, “Using the cgsnapshot utility”](#) shows an example usage of the **cgsnapshot** utility.

Example 2.6. Using the cgsnapshot utility

Configure cgroups on the system using the following commands:

```
~]# mkdir /cgroup/cpu
~]# mount -t cgroup -o cpu cpu /cgroup/cpu
~]# mkdir /cgroup/cpu/lab1
~]# mkdir /cgroup/cpu/lab2
~]# echo 2 > /cgroup/cpu/lab1/cpu.shares
~]# echo 3 > /cgroup/cpu/lab2/cpu.shares
~]# echo 5000000 > /cgroup/cpu/lab1/cpu.rt_period_us
~]# echo 4000000 > /cgroup/cpu/lab1/cpu.rt_runtime_us
~]# mkdir /cgroup/cpuacct
~]# mount -t cgroup -o cpuacct cpuacct /cgroup/cpuacct
```

The above commands mounted two subsystems and created two cgroups, for the **cpu** subsystem, with specific values for some of their parameters. Executing the **cgsnapshot** command (with the **-s** option and an empty **/etc/cgsnapshot_blacklist.conf** file ^[4]) then produces the following output:

```
~]$ cgsnapshot -s
# Configuration file generated by cgsnapshot
mount {
    cpu = /cgroup/cpu;
    cpuacct = /cgroup/cpuacct;
}

group lab2 {
    cpu {
        cpu.rt_period_us="10000000";
        cpu.rt_runtime_us="0";
        cpu.shares="3";
    }
}

group lab1 {
    cpu {
        cpu.rt_period_us="50000000";
        cpu.rt_runtime_us="40000000";
        cpu.shares="2";
    }
}
```

The **-s** option used in the example above tells **cgsnapshot** to ignore all warnings in the output file caused by parameters not being defined in the blacklist or whitelist of the **cgsnapshot** utility. For more information on parameter blacklisting, refer to [Section 2.10.1, “Blacklisting Parameters”](#). For more information on parameter whitelisting, refer to [Section 2.10.2, “Whitelisting Parameters”](#).

When not specifying any options, the output generated by **cgsnapshot** is returned on the standard output. Use the **-f** to specify a file to which the output should be redirected. For example:

```
~]$ cgsnapshot -f ~/test/cgconfig_test.conf
```



The -f option overwrites the specified file

When using the **-f** option, note that it overwrites any content in the file you specify. Therefore, it is recommended not to direct the output straight to the **/etc/cgconfig.conf** file.

The **cgsnapshot** utility can also create configuration files per subsystem. By specifying the name of a subsystem, the output will consist of the corresponding configuration for that subsystem:

```
~]$ cgsnapshot cpuacct
# Configuration file generated by cgsnapshot
mount {
    cpuacct = /cgroup/cpuacct;
}
```

2.10.1. Blacklisting Parameters

The **cgsnapshot** utility allows parameter blacklisting. If a parameter is blacklisted, it does not appear in the output generated by **cgsnapshot**. By default, the **/etc/cgsnapshot_blacklist.conf** file is checked for blacklisted parameters. If a parameter is not present in the blacklist, the whitelist is checked. To specify a different blacklist, use the **-b** option. For example:

```
~]$ cgsnapshot -b ~/test/my_blacklist.conf
```

2.10.2. Whitelisting Parameters

The **cgsnapshot** utility also allows parameter whitelisting. If a parameter is whitelisted, it appears in the output generated by **cgsnapshot**. If a parameter is neither blacklisted or whitelisted, a warning appears informing of this:

```
~]$ cgsnapshot -f ~/test/cgconfig_test.conf
WARNING: variable cpu.rt_period_us is neither blacklisted nor whitelisted
WARNING: variable cpu.rt_runtime_us is neither blacklisted nor whitelisted
```

By default, there is no whitelist configuration file. To specify which file to use as a whitelist, use the **-w** option. For example:

```
~]$ cgsnapshot -w ~/test/my_whitelist.conf
```

Specifying the **-t** option tells **cgsnapshot** to generate a configuration with parameters from the whitelist only.

2.11. Obtaining Information About Control Groups

2.11.1. Finding a Process

To find the cgroup to which a process belongs, run:

```
~]$ ps -o cgroup
```

Or, if you know the PID for the process, run:


```
~]$ cat /proc/PID/cgroup
```

2.11.2. Finding a Subsystem

To find the subsystems that are available in your kernel and how are they mounted together to hierarchies, run:

```
~]$ cat /proc/cgroups
```

#subsys_name		hierarchy		num_cgroups	enabled
cpuset	2	1	1		
ns	0	1	1		
cpu	3	1	1		
cpuacct	4	1	1		
memory	5	1	1		
devices	6	1	1		
freezer	7	1	1		
net_cls	8	1	1		
blkio	9	3	1		
perf_event		0	1	1	
net_prio		0	1	1	

In the example output above, the **hierarchy** column lists IDs of the existing hierarchies on the system. Subsystems with the same hierarchy ID are attached to the same hierarchy. The **num_cgroup** column lists the number of existing cgroups in the hierarchy that uses a particular subsystem. The **enabled** column reports a value of **1** if a particular subsystem is enabled, or **0** if it is not.

Or, to find the mount points of particular subsystems, run:

```
~]$ lssubsys -m subsystems
```

where **subsystems** is a list of the subsystems in which you are interested. Note that the **lssubsys -m** command returns only the top-level mount point per each hierarchy.

2.11.3. Finding Hierarchies

It is recommended that you mount hierarchies under **/cgroup**. Assuming this is the case on your system, list or browse the contents of that directory to obtain a list of hierarchies. If **tree** is installed on your system, run it to obtain an overview of all hierarchies and the cgroups within them:

```
~]$ tree /cgroup
```

2.11.4. Finding Control Groups

To list the cgroups on a system, run:

```
~]$ lscgroup
```

You can restrict the output to a specific hierarchy by specifying a controller and path in the format **controller:path**. For example:

```
~]$ lscgroup cpuset:adminusers
```

lists only subgroups of the **adminusers** cgroup in the hierarchy to which the **cpuset** subsystem is attached.

2.11.5. Displaying Parameters of Control Groups

To display the parameters of specific cgroups, run:

```
~]$ cgget -r parameter list_of_cgroups
```

where ***parameter*** is a pseudo-file that contains values for a subsystem, and ***list_of_cgroups*** is a list of cgroups separated with spaces. For example:

```
~]$ cgget -r cpuset.cpus -r memory.limit_in_bytes lab1 lab2
```

displays the values of ***cpuset.cpus*** and ***memory.limit_in_bytes*** for cgroups ***lab1*** and ***lab2***.

If you do not know the names of the parameters themselves, use a command like:

```
~]$ cgget -g cpuset /
```

2.12. Unloading Control Groups



This command destroys all control groups

The ***cgclear*** command destroys all cgroups in all hierarchies. If you do not have these hierarchies stored in a configuration file, you will not be able to readily reconstruct them.

To clear an entire cgroup file system, use the ***cgclear*** command.

All tasks in the cgroup are reallocated to the root node of the hierarchies, all cgroups are removed, and the file system itself is unmounted from the system, destroying all previously mounted hierarchies. Finally, the directory where the cgroup file system was mounted is removed.



Accurate listing of all mounted cgroups

Using the ***mount*** command to create cgroups (as opposed to creating them using the ***cgconfig*** service) results in the creation of an entry in the ***/etc/mtab*** file (the mounted file systems table). This change is also reflected into the ***/proc/mounts*** file. However, the unloading of cgroups with the ***cgclear*** command, along with other ***cgconfig*** commands, uses a direct kernel interface which does not reflect its changes into the ***/etc/mtab*** file and only writes the new information into the ***/proc/mounts*** file. After unloading cgroups with the ***cgclear*** command, the unmounted cgroups may still be visible in the ***/etc/mtab*** file, and, consequently, displayed when the ***mount*** command is executed. Refer to the ***/proc/mounts*** file for an accurate listing of all mounted cgroups.

2.13. Using the Notification API

The cgroups notification API allows user space applications to receive notifications about the changing status of a cgroup. Currently, the notification API only supports monitoring of the Out of Memory (OOM) control file: ***memory.oom_control***. To create a notification handler, write a C program using the following instructions:

1. Using the **eventfd()** function, create a file descriptor for event notifications. For more information, refer to the **eventfd(2)** man page.
2. To monitor the **memory.oom_control** file, open it using the **open()** function. For more information, refer to the **open(2)** man page.
3. Use the **write()** function to write the following arguments to the **cgroup.event_control** file of the cgroup whose **memory.oom_control** file you are monitoring:

```
<event_file_descriptor> <OOM_control_file_descriptor>
```

where:

- **event_file_descriptor** is used to open the **cgroup.event_control** file,
- and **OOM_control_file_descriptor** is used to open the respective **memory.oom_control** file.

For more information on writing to a file, refer to the **write(1)** man page.

When the above program is started, it will be notified of any OOM situation in the cgroup it is monitoring. Note that OOM notifications only work in non-root cgroups.

For more information on the **memory.oom_control** tunable parameter, refer to [Section 3.7, “memory”](#). For more information on configuring notifications for OOM control, refer to [Example 3.3, “OOM Control and Notifications”](#).

2.14. Additional Resources

The definitive documentation for cgroup commands are the manual pages provided with the *libcgroup* package. The section numbers are specified in the list of man pages below.

The libcgroup Man Pages

- **man 1 cgclassify** — the **cgclassify** command is used to move running tasks to one or more cgroups.
- man 1 cgclear** — the **cgclear** command is used to delete all cgroups in a hierarchy.
- man 5 cgconfig.conf** — cgroups are defined in the **cgconfig.conf** file.
- man 8 cgconfigparser** — the **cgconfigparser** command parses the **cgconfig.conf** file and mounts hierarchies.
- man 1 cgcreate** — the **cgcreate** command creates new cgroups in hierarchies.
- man 1 cgdelete** — the **cgdelete** command removes specified cgroups.
- man 1 cgexec** — the **cgexec** command runs tasks in specified cgroups.
- man 1 cgget** — the **cgget** command displays cgroup parameters.
- man 1 cgsnapshot** — the **cgsnapshot** command generates a configuration file from existing subsystems.
- man 5 cgred.conf** — **cgred.conf** is the configuration file for the **cgred** service.
- man 5 cgrules.conf** — **cgrules.conf** contains the rules used for determining when tasks belong to certain cgroups.
- man 8 cgrulesengd** — the **cgrulesengd** service distributes tasks to cgroups.
- man 1 cgset** — the **cgset** command sets parameters for a cgroup.
- man 1 lscgroup** — the **lscgroup** command lists the cgroups in a hierarchy.
- man 1 lssubsys** — the **lssubsys** command lists the hierarchies containing the specified

subsystems.

[3] The **lssubsys** command is one of the utilities provided by the *libcgroup* package. You must install *libcgroup* to use it: refer to [Chapter 2, Using Control Groups](#) if you are unable to run **lssubsys**.

[4] The **cpu.shares** parameter is specified in the **/etc/cgnsnapshot_blacklist.conf** file by default, which would cause it to be omitted in the generated output in [Example 2.6, “Using the cgnsnapshot utility”](#). Thus, for the purposes of the example, an empty **/etc/cgnsnapshot_blacklist.conf** file is used.

Chapter 3. Subsystems and Tunable Parameters

Subsystems are kernel modules that are aware of cgroups. Typically, they are resource controllers that allocate varying levels of system resources to different cgroups. However, subsystems could be programmed for any other interaction with the kernel where the need exists to treat different groups of processes differently. The *application programming interface* (API) to develop new subsystems is documented in **cgroups.txt** in the kernel documentation, installed on your system at **/usr/share/doc/kernel-doc-kernel-version/Documentation/cgroups/** (provided by the *kernel-doc* package). The latest version of the cgroups documentation is also available on line at <http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>. Note, however, that the features in the latest documentation might not match those available in the kernel installed on your system.

State objects that contain the subsystem parameters for a cgroup are represented as *pseudofiles* within the cgroup virtual file system. These pseudo-files can be manipulated by shell commands or their equivalent system calls. For example, **cpuset.cpus** is a pseudo-file that specifies which CPUs a cgroup is permitted to access. If **/cgroup/cpuset/webserver** is a cgroup for the web server that runs on a system, and the following command is executed:

```
~]# echo 0,2 > /cgroup/cpuset/webserver/cpuset.cpus
```

The value **0,2** is written to the **cpuset.cpus** pseudofile and therefore limits any tasks whose PIDs are listed in **/cgroup/cpuset/webserver/tasks** to use only CPU 0 and CPU 2 on the system.

3.1. blkio

The Block I/O (**blkio**) subsystem controls and monitors access to I/O on block devices by tasks in cgroups. Writing values to some of these pseudofiles limits access or bandwidth, and reading values from some of these pseudofiles provides information on I/O operations.

The **blkio** subsystem offers two policies for controlling access to I/O:

- *Proportional weight division* — implemented in the Completely Fair Queuing I/O scheduler, this policy allows you to set weights to specific cgroups. This means that each cgroup has a set percentage (depending on the weight of the cgroup) of all I/O operations reserved. For more information, refer to [Section 3.1.1, “Proportional Weight Division Tunable Parameters”](#)
- *I/O throttling (Upper limit)* — this policy is used to set an upper limit for the number of I/O operations performed by a specific device. This means that a device can have a limited rate of *read* or *write* operations. For more information, refer to [Section 3.1.2, “I/O Throttling Tunable Parameters”](#)



Buffered write operations

Currently, the Block I/O subsystem does not work for buffered *write* operations. It is primarily targeted at direct I/O, although it works for buffered *read* operations.

3.1.1. Proportional Weight Division Tunable Parameters

blkio.weight

specifies the relative proportion (*weight*) of block I/O access available by default to a cgroup, in the range **100** to **1000**. This value is overridden for specific devices by the **blkio.weight_device** parameter. For example, to assign a default weight of **500** to a cgroup for access to block devices, run:

```
echo 500 > blkio.weight
```

blkio.weight_device

specifies the relative proportion (*weight*) of I/O access on specific devices available to a cgroup, in the range **100** to **1000**. The value of this parameter overrides the value of the **blkio.weight** parameter for the devices specified. Values take the format **major:minor weight**, where **major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, otherwise known as the *Linux Devices List* and available from <http://www.kernel.org/doc/Documentation/devices.txt>. For example, to assign a weight of **500** to a cgroup for access to **/dev/sda**, run:

```
echo 8:0 500 > blkio.weight_device
```

In the *Linux Allocated Devices* notation, **8:0** represents **/dev/sda**.

3.1.2. I/O Throttling Tunable Parameters

blkio.throttle.read_bps_device

specifies the upper limit on the number of *read* operations a device can perform. The rate of the *read* operations is specified in bytes per second. Entries have three fields: **major**, **minor**, and **bytes_per_second**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, and **bytes_per_second** is the upper limit rate at which *read* operations can be performed. For example, to allow the **/dev/sda** device to perform *read* operations at a maximum of 10 MBps, run:

```
~]# echo "8:0 10485760" > /cgroup/blkio/test/blkio.throttle.read_bps_device
```

blkio.throttle.read_iops_device

specifies the upper limit on the number of *read* operations a device can perform. The rate of the *read* operations is specified in operations per second. Entries have three fields: **major**, **minor**, and **operations_per_second**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, and **operations_per_second** is the upper limit rate at which *read* operations can be performed. For example, to allow the **/dev/sda** device to perform a maximum of 10 *read* operations per second, run:

```
~]# echo "8:0 10" > /cgroup/blkio/test/blkio.throttle.read_iops_device
```

blkio.throttle.write_bps_device

specifies the upper limit on the number of *write* operations a device can perform. The rate of the *write* operations is specified in bytes per second. Entries have three fields: **major**, **minor**, and **bytes_per_second**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, and **bytes_per_second** is the upper limit rate at which *write* operations can be performed. For example, to allow the **/dev/sda** device to perform *write* operations at a maximum of 10 MBps, run:

```
~]# echo "8:0 10485760" >
/cgroup/blkio/test/blkio.throttle.write_bps_device
```

blkio.throttle.write_iops_device

specifies the upper limit on the number of *write* operations a device can perform. The rate of the *write* operations is specified in operations per second. Entries have three fields: **major**, **minor**, and **operations_per_second**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, and **operations_per_second** is the upper limit rate at which *write* operations can be performed. For example, to allow the **/dev/sda** device to perform a maximum of 10 *write* operations per second, run:

```
~]# echo "8:0 10" >
/cgroup/blkio/test/blkio.throttle.write_iops_device
```

blkio.throttle.io_serviced

reports the number of I/O operations performed on specific devices by a cgroup as seen by the throttling policy. Entries have four fields: **major**, **minor**, **operation**, and **number**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, **operation** represents the type of operation (**read**, **write**, **sync**, or **async**) and **number** represents the number of operations.

blkio.throttle.io_service_bytes

reports the number of bytes transferred to or from specific devices by a cgroup. The only difference between **blkio.io_service_bytes** and **blkio.throttle.io_service_bytes** is that the former is not updated when the CFQ scheduler is operating on a request queue. Entries have four fields: **major**, **minor**, **operation**, and **bytes**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, **operation** represents the type of operation (**read**, **write**, **sync**, or **async**) and **bytes** is the number of bytes transferred.

3.1.3. blkio Common Tunable Parameters

The following parameters may be used for either of the policies listed in [Section 3.1, “blkio”](#).

blkio.reset_stats

resets the statistics recorded in the other pseudofiles. Write an integer to this file to reset the statistics for this cgroup.

blkio.time

reports the time that a cgroup had I/O access to specific devices. Entries have three fields: **major**, **minor**, and **time**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, and **time** is the length of time in milliseconds (ms).

blkio.sectors

reports the number of sectors transferred to or from specific devices by a cgroup. Entries have three fields: **major**, **minor**, and **sectors**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, and **sectors** is the number of disk sectors.

blkio.avg_queue_size

reports the average queue size for I/O operations by a cgroup, over the entire length of time of the group's existence. The queue size is sampled every time a queue for this cgroup gets a timeslice. Note that this report is available only if **CONFIG_DEBUG_BLK_CGROUP=y** is set on the system.

blkio.group_wait_time

reports the total time (in nanoseconds — ns) a cgroup spent waiting for a timeslice for one of its queues. The report is updated every time a queue for this cgroup gets a timeslice, so if you read this pseudofile while the cgroup is waiting for a timeslice, the report will not contain time spent waiting for the operation currently queued. Note that this report is available only if **CONFIG_DEBUG_BLK_CGROUP=y** is set on the system.

blkio.empty_time

reports the total time (in nanoseconds — ns) a cgroup spent without any pending requests. The report is updated every time a queue for this cgroup has a pending request, so if you read this pseudofile while the cgroup has no pending requests, the report will not contain time spent in the current empty state. Note that this report is available only if **CONFIG_DEBUG_BLK_CGROUP=y** is set on the system.

blkio.idle_time

reports the total time (in nanoseconds — ns) the scheduler spent idling for a cgroup in anticipation of a better request than those requests already in other queues or from other groups. The report is updated every time the group is no longer idling, so if you read this pseudofile while the cgroup is idling, the report will not contain time spent in the current idling state. Note that this report is available only if **CONFIG_DEBUG_BLK_CGROUP=y** is set on the system.

blkio.dequeue

reports the number of times requests for I/O operations by a cgroup were dequeued by specific devices. Entries have three fields: **major**, **minor**, and **number**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, and **number** is the number of requests the group was dequeued. Note that this report is available only if **CONFIG_DEBUG_BLK_CGROUP=y** is set on the system.

blkio.io_serviced

reports the number of I/O operations performed on specific devices by a cgroup as seen by the CFQ scheduler. Entries have four fields: **major**, **minor**, **operation**, and **number**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, **operation** represents the type of operation (**read**, **write**, **sync**, or **async**) and **number** represents the number of operations.

blkio.io_service_bytes

reports the number of bytes transferred to or from specific devices by a cgroup as seen by the CFQ scheduler. Entries have four fields: **major**, **minor**, **operation**, and **bytes**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, **operation** represents the type of operation (**read**, **write**, **sync**, or **async**) and **bytes** is the number of bytes transferred.

blkio.io_service_time

reports the total time between request dispatch and request completion for I/O operations on specific devices by a cgroup as seen by the CFQ scheduler. Entries have four fields: **major**, **minor**, **operation**, and **time**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, **operation** represents the type of operation (**read**, **write**, **sync**, or **async**) and **time** is the length of time in nanoseconds (ns). The time is reported in nanoseconds rather than a larger unit so that this report is meaningful even for solid-state devices.

blkio.io_wait_time

reports the total time I/O operations on specific devices by a cgroup spent waiting for service in the scheduler queues. When you interpret this report, note:

- ▶ the time reported can be greater than the total time elapsed, because the time reported is the cumulative total of all I/O operations for the cgroup rather than the time that the cgroup itself spent waiting for I/O operations. To find the time that the group as a whole has spent waiting, use the **blkio.group_wait_time** parameter.
- ▶ if the device has a **queue_depth** > 1, the time reported only includes the time until the request is dispatched to the device, not any time spent waiting for service while the device re-orders requests.

Entries have four fields: **major**, **minor**, **operation**, and **time**. **Major** and **minor** are device types and node numbers specified in *Linux Allocated Devices*, **operation** represents the type of operation (**read**, **write**, **sync**, or **async**) and **time** is the length of time in nanoseconds (ns). The time is reported in nanoseconds rather than a larger unit so that this report is meaningful even for solid-state devices.

blkio.io_merged

reports the number of BIOS requests merged into requests for I/O operations by a cgroup. Entries have two fields: **number** and **operation**. **Number** is the number of requests, and **operation** represents the type of operation (**read**, **write**, **sync**, or **async**).

blkio.io_queued

reports the number of requests queued for I/O operations by a cgroup. Entries have two fields: **number** and **operation**. **Number** is the number of requests, and **operation** represents the type of operation (**read**, **write**, **sync**, or **async**).

3.1.4. Example Usage

Refer to [Example 3.1, “blkio proportional weight division”](#) for a simple test of running two **dd** threads in two different cgroups with various **blkio.weight** values.

Example 3.1. blkio proportional weight division

1. Mount the **blkio** subsystem:

```
~]# mount -t cgroup -o blkio blkio /cgroup/blkio/
```

2. Create two cgroups for the **blkio** subsystem:

```
~]# mkdir /cgroup/blkio/test1/
~]# mkdir /cgroup/blkio/test2/
```

3. Set **blkio** weights in the previously-created cgroups:

```
~]# echo 1000 > /cgroup/blkio/test1/blkio.weight
~]# echo 500 > /cgroup/blkio/test2/blkio.weight
```

4. Create two large files:

```
~]# dd if=/dev/zero of=file_1 bs=1M count=4000
~]# dd if=/dev/zero of=file_2 bs=1M count=4000
```

The above commands create two files (**file_1** and **file_2**) of size 4 GB.

5. For each of the test cgroups, execute a **dd** command (which reads the contents of a file and outputs it to the null device) on one of the large files:

```
~]# cgexec -g blkio:test1 time dd if=file_1 of=/dev/null
~]# cgexec -g blkio:test2 time dd if=file_2 of=/dev/null
```

Both commands will output their completion time once they have finished.

6. Simultaneously with the two running **dd** threads, you can monitor the performance in real time by using the **iostat** utility. To install the **iostat** utility, execute, as root, the **yum install iostat** command. The following is an example of the output as seen in the **iostat** utility while running the previously-started **dd** threads:

```
Total DISK READ: 83.16 M/s | Total DISK WRITE: 0.00 B/s
  TIME  TID  PRIO  USER      DISK READ  DISK WRITE  SWAPIN      IO
COMMAND
15:18:04 15071 be/4 root        27.64 M/s    0.00 B/s   0.00 % 92.30 % dd
if=file_2 of=/dev/null
15:18:04 15069 be/4 root        55.52 M/s    0.00 B/s   0.00 % 88.48 % dd
if=file_1 of=/dev/null
```

In order to get the most accurate result in [Example 3.1. “blkio proportional weight division”](#), prior to the execution of the **dd** commands, flush all file system buffers and free pagecache, dentries and inodes using the following commands:

```
~]# sync
~]# echo 3 > /proc/sys/vm/drop_caches
```

Additionally, you can enable *group isolation* which provides stronger isolation between groups at the expense of throughput. When group isolation is disabled, fairness can be expected only for a sequential workload. By default, group isolation is enabled and fairness can be expected for random I/O workloads

as well. To enable group isolation, use the following command:

```
~]# echo 1 > /sys/block/<disk_device>/queue/iosched/group_isolation
```

where **<disk_device>** stands for the name of the desired device, for example **sda**.

3.2. cpu

The **cpu** subsystem schedules CPU access to cgroups. Access to CPU resources can be scheduled using two schedulers:

- ▶ *Completely Fair Scheduler (CFS)* — a proportional share scheduler which divides the CPU time (CPU bandwidth) proportionately between groups of tasks (cgroups) depending on the priority/weight of the task or shares assigned to cgroups. For more information about resource limiting using CFS, refer to [Section 3.2.1, “CFS Tunable Parameters”](#).
- ▶ *Real-Time scheduler (RT)* — a task scheduler that provides a way to specify the amount of CPU time that real-time tasks can use. For more information about resource limiting of real-time tasks, refer to [Section 3.2.2, “RT Tunable Parameters”](#).

3.2.1. CFS Tunable Parameters

In CFS, a cgroup can get more than its share of CPU if there are enough idle CPU cycles available in the system, due to the work conserving nature of the scheduler. This is usually the case for cgroups that consume CPU time based on relative shares. Ceiling enforcement can be used for cases when a hard limit on the amount of CPU that a cgroup can utilize is required (that is, tasks cannot use more than a set amount of CPU time).

The following options can be used to configure ceiling enforcement or relative sharing of CPU:

Ceiling Enforcement Tunable Parameters

cpu.cfs_period_us

specifies a period of time in microseconds (μ s, represented here as “**us**”) for how regularly a cgroup’s access to CPU resources should be reallocated. If tasks in a cgroup should be able to access a single CPU for 0.2 seconds out of every 1 second, set **cpu.cfs_quota_us** to **200000** and **cpu.cfs_period_us** to **1000000**. The upper limit of the **cpu.cfs_quota_us** parameter is 1 second and the lower limit is 1000 microseconds.

cpu.cfs_quota_us

specifies the total amount of time in microseconds (μ s, represented here as “**us**”) for which all tasks in a cgroup can run during one period (as defined by **cpu.cfs_period_us**). As soon as tasks in a cgroup use up all the time specified by the quota, they are throttled for the remainder of the time specified by the period and not allowed to run until the next period. If tasks in a cgroup should be able to access a single CPU for 0.2 seconds out of every 1 second, set **cpu.cfs_quota_us** to **200000** and **cpu.cfs_period_us** to **1000000**. Note that the quota and period parameters operate on a CPU basis. To allow a process to fully utilize two CPUs, for example, set **cpu.cfs_quota_us** to **200000** and **cpu.cfs_period_us** to **100000**.

Setting the value in **cpu.cfs_quota_us** to **-1** indicates that the cgroup does not adhere to any CPU time restrictions. This is also the default value for every cgroup (except the root cgroup).

cpu.stat

reports CPU time statistics using the following values:

- **nr_periods** — number of period intervals (as specified in **cpu.cfs_period_us**) that have elapsed.
- **nr_throttled** — number of times tasks in a cgroup have been throttled (that is, not allowed to run because they have exhausted all of the available time as specified by their quota).
- **throttled_time** — the total time duration (in nanoseconds) for which tasks in a cgroup have been throttled.

Relative Shares Tunable Parameters**cpu.shares**

contains an integer value that specifies a relative share of CPU time available to the tasks in a cgroup. For example, tasks in two cgroups that have **cpu.shares** set to **100** will receive equal CPU time, but tasks in a cgroup that has **cpu.shares** set to **200** receive twice the CPU time of tasks in a cgroup where **cpu.shares** is set to **100**. The value specified in the **cpu.shares** file must be **2** or higher.

Note that shares of CPU time are distributed per all CPU cores on multi-core systems. Even if a cgroup is limited to less than 100% of CPU on a multi-core system, it may use 100% of each individual CPU core. Consider the following example: if cgroup **A** is configured to use 25% and cgroup **B** 75% of the CPU, starting four CPU-intensive processes (one in **A** and three in **B**) on a system with four cores results in the following division of CPU shares:

Table 3.1. CPU share division

PID	cgroup	CPU	CPU share
100	A	0	100% of CPU0
101	B	1	100% of CPU1
102	B	2	100% of CPU2
103	B	3	100% of CPU3

Using relative shares to specify CPU access has two implications on resource management that should be considered:

- Because the CFS does not demand equal usage of CPU, it is hard to predict how much CPU time a cgroup will be allowed to utilize. When tasks in one cgroup are idle and are not using any CPU time, this left-over time is collected in a global pool of unused CPU cycles. Other cgroups are allowed to borrow CPU cycles from this pool.
- The actual amount of CPU time that is available to a cgroup can vary depending on the number of cgroups that exist on the system. If a cgroup has a relative share of **1000** and two other cgroups have a relative share of **500**, the first cgroup receives 50% of all CPU time in cases when processes in all cgroups attempt to use 100% of the CPU. However, if another cgroup is added with a relative share of **1000**, the first cgroup is only allowed 33% of the CPU (the rest of the cgroups receive 16.5%, 16.5%, and 33% of CPU).

3.2.2. RT Tunable Parameters

The RT scheduler works similar to the ceiling enforcement control of the CFS (for more information, refer to [Section 3.2.1, “CFS Tunable Parameters”](#)) but limits CPU access to real-time tasks only. The amount of time for which a real-time task can access the CPU is decided by allocating a run time and a period for each cgroup. All tasks in a cgroup are then allowed to access the CPU for the defined period of time for one run time (for example, tasks in a cgroup may be allowed to run 0.1 seconds in every 1 second).

cpu.rt_period_us

applicable to real-time scheduling tasks only, this parameter specifies a period of time in microseconds (μ s, represented here as "**us**") for how regularly a cgroup's access to CPU resources should be reallocated. If tasks in a cgroup should be able to access a single CPU for 0.2 seconds out of every 1 second, set **cpu.rt_runtime_us** to **200000** and **cpu.rt_period_us** to **1000000**.

cpu.rt_runtime_us

applicable to real-time scheduling tasks only, this parameter specifies a period of time in microseconds (μ s, represented here as "**us**") for the longest continuous period in which the tasks in a cgroup have access to CPU resources. Establishing this limit prevents tasks in one cgroup from monopolizing CPU time. If tasks in a cgroup should be able to access a single CPU for 0.2 seconds out of every 1 second, set **cpu.rt_runtime_us** to **200000** and **cpu.rt_period_us** to **1000000**. Note that the run time and period parameters operate on a CPU basis. To allow a real-time task to fully utilize two CPUs, for example, set **cpu.cfs_quota_us** to **200000** and **cpu.cfs_period_us** to **100000**.

3.2.3. Example Usage

Example 3.2. Limiting CPU access

The following examples assume you have an existing hierarchy of cgroups configured and the **cpu** subsystem mounted on your system:

- To allow one cgroup to use 25% of a single CPU and a different cgroup to use 75% of that same CPU, use the following commands:

```
~]# echo 250 > /cgroup/cpu/blue/cpu.shares
~]# echo 750 > /cgroup/cpu/red/cpu.shares
```

- To limit a cgroup to fully utilize a single CPU, use the following commands:

```
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_period_us
```

- To limit a cgroup to utilize 10% of a single CPU, use the following commands:

```
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

- On a multi-core system, to allow a cgroup to fully utilize two CPU cores, use the following commands:

```
~]# echo 200000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

3.3. cpuacct

The CPU Accounting (**cpuacct**) subsystem generates automatic reports on CPU resources used by the tasks in a cgroup, including tasks in child groups. Three reports are available:

cpuacct.usage

reports the total CPU time (in nanoseconds) consumed by all tasks in this cgroup (including tasks lower in the hierarchy).



Resetting cpuacct.usage

To reset the value in **cpuacct.usage**, execute the following command:

```
~]# echo 0 > /cgroup/cpuacct/cpuacct.usage
```

The above command also resets values in **cpuacct.usage_percpu**.

cpuacct.stat

reports the user and system CPU time consumed by all tasks in this cgroup (including tasks lower in the hierarchy) in the following way:

- **user** — CPU time consumed by tasks in user mode.

- **system** — CPU time consumed by tasks in system (kernel) mode.

CPU time is reported in the units defined by the **USER_HZ** variable.

cpuacct.usage_percpu

reports the CPU time (in nanoseconds) consumed on each CPU by all tasks in this cgroup (including tasks lower in the hierarchy).

3.4. cpuset

The **cpuset** subsystem assigns individual CPUs and memory nodes to cgroups. Each cpuset can be specified according to the following parameters, each one in a separate *pseudofile* within the cgroup virtual file system:



Mandatory parameters

Some subsystems have mandatory parameters that must be set before you can move a task into a cgroup which uses any of those subsystems. For example, before you move a task into a cgroup which uses the **cpuset** subsystem, the **cpuset.cpus** and **cpuset.mems** parameters must be defined for that cgroup.

cpuset.cpus (mandatory)

specifies the CPUs that tasks in this cgroup are permitted to access. This is a comma-separated list, with dashes ("-") to represent ranges. For example,

```
0-2,16
```

represents CPUs 0, 1, 2, and 16.

cpuset.mems (mandatory)

specifies the memory nodes that tasks in this cgroup are permitted to access. This is a comma-separated list in ASCII format, with dashes ("-") to represent ranges. For example,

```
0-2,16
```

represents memory nodes 0, 1, 2, and 16.

cpuset.memory_migrate

contains a flag (**0** or **1**) that specifies whether a page in memory should migrate to a new node if the values in **cpuset.mems** change. By default, memory migration is disabled (**0**) and pages stay on the node to which they were originally allocated, even if this node is no longer one of the nodes now specified in **cpuset.mems**. If enabled (**1**), the system will migrate pages to memory nodes within the new parameters specified by **cpuset.mems**, maintaining their relative placement if possible — for example, pages on the second node on the list originally specified by **cpuset.mems** will be allocated to the second node on the list now specified by **cpuset.mems**, if this place is available.

cpuset.cpu_exclusive

contains a flag (**0** or **1**) that specifies whether cpusets other than this one and its parents and children can share the CPUs specified for this cpuset. By default (**0**), CPUs are not allocated exclusively to one cpuset.

cpuset.mem_exclusive

contains a flag (**0** or **1**) that specifies whether other cpusets can share the memory nodes specified for this cpuset. By default (**0**), memory nodes are not allocated exclusively to one cpuset. Reserving memory nodes for the exclusive use of a cpuset (**1**) is functionally the same as enabling a memory hardwall with the **cpuset.mem_hardwall** parameter.

cpuset.mem_hardwall

contains a flag (**0** or **1**) that specifies whether kernel allocations of memory page and buffer data should be restricted to the memory nodes specified for this cpuset. By default (**0**), page and buffer data is shared across processes belonging to multiple users. With a hardwall enabled (**1**), each tasks' user allocation can be kept separate.

cpuset.memory_pressure

a read-only file that contains a running average of the *memory pressure* created by the processes in this cpuset. The value in this pseudofile is automatically updated when **cpuset.memory_pressure_enabled** is enabled, otherwise, the pseudofile contains the value **0**.

cpuset.memory_pressure_enabled

contains a flag (**0** or **1**) that specifies whether the system should compute the *memory pressure* created by the processes in this cgroup. Computed values are output to **cpuset.memory_pressure** and represent the rate at which processes attempt to free in-use memory, reported as an integer value of attempts to reclaim memory per second, multiplied by 1000.

cpuset.memory_spread_page

contains a flag (**0** or **1**) that specifies whether file system buffers should be spread evenly across the memory nodes allocated to this cpuset. By default (**0**), no attempt is made to spread memory pages for these buffers evenly, and buffers are placed on the same node on which the process that created them is running.

cpuset.memory_spread_slab

contains a flag (**0** or **1**) that specifies whether kernel slab caches for file input/output operations should be spread evenly across the cpuset. By default (**0**), no attempt is made to spread kernel slab caches evenly, and slab caches are placed on the same node on which the process that created them is running.

cpuset.sched_load_balance

contains a flag (**0** or **1**) that specifies whether the kernel will balance loads across the CPUs in this cpuset. By default (**1**), the kernel balances loads by moving processes from overloaded CPUs to less heavily used CPUs.

Note, however, that setting this flag in a cgroup has no effect if load balancing is enabled in any parent cgroup, as load balancing is already being carried out at a higher level. Therefore, to disable load balancing in a cgroup, disable load balancing also in each of its parents in the hierarchy. In this case, you should also consider whether load balancing should be enabled for any siblings of the cgroup in question.

cpuset.sched_relax_domain_level

contains an integer between **-1** and a small positive value, which represents the width of the range of CPUs across which the kernel should attempt to balance loads. This value is meaningless if **cpuset.sched_load_balance** is disabled.

The precise effect of this value varies according to system architecture, but the following values are typical:

Values of **cpuset.sched_relax_domain_level**

Value	Effect
-1	Use the system default value for load balancing
0	Do not perform immediate load balancing; balance loads only periodically
1	Immediately balance loads across threads on the same core
2	Immediately balance loads across cores in the same package
3	Immediately balance loads across CPUs on the same node or blade
4	Immediately balance loads across several CPUs on architectures with non-uniform memory access (NUMA)
5	Immediately balance loads across all CPUs on architectures with NUMA

3.5. devices

The **devices** subsystem allows or denies access to devices by tasks in a cgroup.



Technology preview

The Device Whitelist (**devices**) subsystem is considered to be a Technology Preview in Red Hat Enterprise Linux 6.

Technology preview features are currently not supported under Red Hat Enterprise Linux 6 subscription services, might not be functionally complete, and are generally not suitable for production use. However, Red Hat includes these features in the operating system as a customer convenience and to provide the feature with wider exposure. You might find these features useful in a non-production environment and are also free to provide feedback and functionality suggestions for a technology preview feature before it becomes fully supported.

devices.allow

specifies devices to which tasks in a cgroup have access. Each entry has four fields: **type**, **major**, **minor**, and **access**. The values used in the **type**, **major**, and **minor** fields correspond to device types and node numbers specified in *Linux Allocated Devices*, otherwise known as the *Linux Devices List* and available from <http://www.kernel.org/doc/Documentation/devices.txt>.

type

type can have one of the following three values:

- ▶ **a** — applies to all devices, both *character devices* and *block devices*
- ▶ **b** — specifies a block device
- ▶ **c** — specifies a character device

major, minor

major and **minor** are device node numbers specified by *Linux Allocated Devices*. The major and minor numbers are separated by a colon. For example, **8** is the major number that specifies SCSI disk drives, and the minor number **1** specifies the first partition on the first SCSI disk drive; therefore **8:1** fully specifies this partition, corresponding to a file system location of **/dev/sda1**.

* can stand for all major or all minor device nodes, for example **9:*** (all RAID devices) or ***:*** (all devices).

access

access is a sequence of one or more of the following letters:

- ▶ **r** — allows tasks to read from the specified device
- ▶ **w** — allows tasks to write to the specified device
- ▶ **m** — allows tasks to create device files that do not yet exist

For example, when **access** is specified as **r**, tasks can only read from the specified device, but when **access** is specified as **rw**, tasks can read from and write to the device.

devices.deny

specifies devices that tasks in a cgroup cannot access. The syntax of entries is identical with **devices.allow**.

devices.list

reports the devices for which access controls have been set for tasks in this cgroup.

3.6. freezer

The **freezer** subsystem suspends or resumes tasks in a cgroup.

freezer.state

freezer.state is only available in non-root cgroups, and has three possible values:

- ▶ **FROZEN** — tasks in the cgroup are suspended.
- ▶ **FREEZING** — the system is in the process of suspending tasks in the cgroup.
- ▶ **THAWED** — tasks in the cgroup have resumed.

To suspend a specific process:

1. Move that process to a cgroup in a hierarchy which has the **freezer** subsystem attached to it.
2. Freeze that particular cgroup to suspend the process contained in it.

It is not possible to move a process into a suspended (frozen) cgroup.

Note that while the **FROZEN** and **THAWED** values can be written to **freezer.state**, **FREEZING** cannot be written, only read.

3.7. memory

The **memory** subsystem generates automatic reports on memory resources used by the tasks in a cgroup, and sets limits on memory use by those tasks:

memory.stat

reports a wide range of memory statistics, as described in the following table:

Table 3.2. Values reported by memory.stat

Statistic	Description
cache	page cache, including tmpfs (shmem) , in bytes
rss	anonymous and swap cache, <i>not</i> including tmpfs (shmem) , in bytes
mapped_file	size of memory-mapped mapped files, including tmpfs (shmem) , in bytes
pgpgin	number of pages paged into memory
pgpgout	number of pages paged out of memory
swap	swap usage, in bytes
active_anon	anonymous and swap cache on active least-recently-used (LRU) list, including tmpfs (shmem) , in bytes
inactive_anon	anonymous and swap cache on inactive LRU list, including tmpfs (shmem) , in bytes
active_file	file-backed memory on active LRU list, in bytes
inactive_file	file-backed memory on inactive LRU list, in bytes
unevictable	memory that cannot be reclaimed, in bytes
hierarchical_memory_limit	memory limit for the hierarchy that contains the memory cgroup, in bytes
hierarchical_memsw_limit	memory plus swap limit for the hierarchy that contains the memory cgroup, in bytes

Additionally, each of these files other than **hierarchical_memory_limit** and **hierarchical_memsw_limit** has a counterpart prefixed **total_** that reports not only on the cgroup, but on all its children as well. For example, **swap** reports the swap usage by a cgroup and **total_swap** reports the total swap usage by the cgroup and all its child groups.

When you interpret the values reported by **memory.stat**, note how the various statistics inter-relate:

- ▶ **active_anon + inactive_anon** = anonymous memory + file cache for **tmpfs** + swap cache
Therefore, **active_anon + inactive_anon** \neq **rss**, because **rss** does not include **tmpfs**.
- ▶ **active_file + inactive_file** = cache - size of **tmpfs**

memory.usage_in_bytes

reports the total current memory usage by processes in the cgroup (in bytes).

memory.memsw.usage_in_bytes

reports the sum of current memory usage plus swap space used by processes in the cgroup (in bytes).

memory.max_usage_in_bytes

reports the maximum memory used by processes in the cgroup (in bytes).

memory.memsw.max_usage_in_bytes

reports the maximum amount of memory and swap space used by processes in the cgroup (in bytes).

memory.limit_in_bytes

sets the maximum amount of user memory (including file cache). If no units are specified, the value is interpreted as bytes. However, it is possible to use suffixes to represent larger units — **k** or **K** for kilobytes, **m** or **M** for Megabytes, and **g** or **G** for Gigabytes.

You cannot use **memory.limit_in_bytes** to limit the root cgroup; you can only apply values to groups lower in the hierarchy.

Write **-1** to **memory.limit_in_bytes** to remove any existing limits.

memory.memsw.limit_in_bytes

sets the maximum amount for the sum of memory and swap usage. If no units are specified, the value is interpreted as bytes. However, it is possible to use suffixes to represent larger units — **k** or **K** for kilobytes, **m** or **M** for Megabytes, and **g** or **G** for Gigabytes.

You cannot use **memory.memsw.limit_in_bytes** to limit the root cgroup; you can only apply values to groups lower in the hierarchy.

Write **-1** to **memory.memsw.limit_in_bytes** to remove any existing limits.



Setting the `memory.memsw.limit_in_bytes` and `memory.limit_in_bytes` parameters

It is important to set the `memory.limit_in_bytes` parameter *before* setting the `memory.memsw.limit_in_bytes` parameter: attempting to do so in the reverse order results in an error. This is because `memory.memsw.limit_in_bytes` becomes available only after all memory limitations (previously set in `memory.limit_in_bytes`) are exhausted.

Consider the following example: setting `memory.limit_in_bytes = 2G` and `memory.memsw.limit_in_bytes = 4G` for a certain cgroup will allow processes in that cgroup to allocate 2 GB of memory and, once exhausted, allocate another 2 GB of swap only. The `memory.memsw.limit_in_bytes` parameter represents the sum of memory and swap. Processes in a cgroup that does not have the `memory.memsw.limit_in_bytes` parameter set can potentially use up all the available swap (after exhausting the set memory limitation) and trigger an Out Of Memory situation caused by the lack of available swap.

The order in which the `memory.limit_in_bytes` and `memory.memsw.limit_in_bytes` parameters are set in the `/etc/cgconfig.conf` file is important as well. The following is a correct example of such a configuration:

```
memory {
    memory.limit_in_bytes = 1G;
    memory.memsw.limit_in_bytes = 1G;
}
```

`memory.failcnt`

reports the number of times that the memory limit has reached the value set in `memory.limit_in_bytes`.

`memory.memsw.failcnt`

reports the number of times that the memory plus swap space limit has reached the value set in `memory.memsw.limit_in_bytes`.

`memory.force_empty`

when set to `0`, empties memory of all pages used by tasks in this cgroup. This interface can only be used when the cgroup has no tasks. If memory cannot be freed, it is moved to a parent cgroup if possible. Use the `memory.force_empty` parameter before removing a cgroup to avoid moving out-of-use page caches to its parent cgroup.

`memory.swappiness`

sets the tendency of the kernel to swap out process memory used by tasks in this cgroup instead of reclaiming pages from the page cache. This is the same tendency, calculated the same way, as set in `/proc/sys/vm/swappiness` for the system as a whole. The default value is `60`. Values lower than `60` decrease the kernel's tendency to swap out process memory, values greater than `60` increase the kernel's tendency to swap out process memory, and values greater than `100` permit the kernel to swap out pages that are part of the address

space of the processes in this cgroup.

Note that a value of **0** does not prevent process memory being swapped out; swap out might still happen when there is a shortage of system memory because the global virtual memory management logic does not read the cgroup value. To lock pages completely, use **mlock()** instead of cgroups.

You cannot change the swappiness of the following groups:

- ▶ the root cgroup, which uses the swappiness set in **/proc/sys/vm/swappiness**.
- ▶ a cgroup that has child groups below it.

memory.use_hierarchy

contains a flag (**0** or **1**) that specifies whether memory usage should be accounted for throughout a hierarchy of cgroups. If enabled (**1**), the memory subsystem reclaims memory from the children of and process that exceeds its memory limit. By default (**0**), the subsystem does not reclaim memory from a task's children.

memory.oom_control

contains a flag (**0** or **1**) that enables or disables the Out of Memory killer for a cgroup. If enabled (**0**), tasks that attempt to consume more memory than they are allowed are immediately killed by the OOM killer. The OOM killer is enabled by default in every cgroup using the **memory** subsystem; to disable it, write **1** to the **memory.oom_control** file:

```
~]# echo 1 > /cgroup/memory/lab1/memory.oom_control
```

When the OOM killer is disabled, tasks that attempt to use more memory than they are allowed are paused until additional memory is freed.

The **memory.oom_control** file also reports the OOM status of the current cgroup under the **under_oom** entry. If the cgroup is out of memory and tasks in it are paused, the **under_oom** entry reports the value **1**.

The **memory.oom_control** file is capable of reporting an occurrence of an OOM situation using the notification API. For more information, refer to [Section 2.13, “Using the Notification API”](#) and [Example 3.3, “OOM Control and Notifications”](#).

3.7.1. Example Usage

Example 3.3. OOM Control and Notifications

The following example demonstrates how the OOM killer takes action when a task in a cgroup attempts to use more memory than allowed, and how a notification handler can report OOM situations:

1. Attach the **memory** subsystem to a hierarchy and create a cgroup:

```
~]# mount -t memory -o memory memory /cgroup/memory
~]# mkdir /cgroup/memory/blue
```

2. Set the amount of memory which tasks in the **blue** cgroup can use to 100MB:

```
~]# echo 104857600 > memory.limit_in_bytes
```

3. Change into the **blue** directory and make sure the OOM killer is enabled:

```
~]# cd /cgroup/memory/blue
blue]# cat memory.oom_control
oom_kill_disable 0
under_oom 0
```

4. Move the current shell process into the **tasks** file of the **blue** cgroup so that all other processes started in this shell are automatically moved to the **blue** cgroup:

```
blue]# echo $$ > tasks
```

5. Start a test program that attempts to allocate a large amount of memory exceeding the limit you set in [Step 2](#). As soon as the **blue** cgroup runs out of free memory, the OOM killer kills the test program and reports **Killed** to the standard output:

```
blue]# ~/mem-hog
Killed
```

The following is an example of such a test program ^[5]:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define KB (1024)
#define MB (1024 * KB)
#define GB (1024 * MB)

int main(int argc, char *argv[])
{
    char *p;

again:
    while ((p = (char *)malloc(GB)))
        memset(p, 0, GB);

    while ((p = (char *)malloc(MB)))
        memset(p, 0, MB);

    while ((p = (char *)malloc(KB)))
        memset(p, 0,
            KB);

    sleep(1);

    goto again;

    return 0;
}

```

6. Disable the OOM killer and re-run the test program. This time, the test program remains paused waiting for additional memory to be freed:

```

blue]# echo 1 > memory.oom_control
blue]# ~/mem-hog

```

7. While the test program is paused, note that the **under_oom** state of the cgroup has changed to indicate that the cgroup is out of available memory:

```

~]# cat /cgroup/memory/blue/memory.oom_control
oom_kill_disable 1
under_oom 1

```

Re-enabling the OOM killer immediately kills the test program.

8. To receive notifications about every OOM situation, create a program as specified in [Section 2.13, “Using the Notification API”](#). For example ^[6]:


```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/eventfd.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

static inline void die(const char *msg)
{
    fprintf(stderr, "error: %s: %s(%d)\n", msg, strerror(errno), errno);
    exit(EXIT_FAILURE);
}

static inline void usage(void)
{
    fprintf(stderr, "usage: oom_eventfd_test <cgroup.event_control>
<memory.oom_control>\n");
    exit(EXIT_FAILURE);
}

#define BUFSIZE 256

int main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    int efd, cfd, ofd, rb, wb;
    uint64_t u;

    if (argc != 3)
        usage();

    if ((efd = eventfd(0, 0)) == -1)
        die("eventfd");

    if ((cfd = open(argv[1], O_WRONLY)) == -1)
        die("cgroup.event_control");

    if ((ofd = open(argv[2], O_RDONLY)) == -1)
        die("memory.oom_control");

    if ((wb = snprintf(buf, BUFSIZE, "%d %d", efd, ofd)) >= BUFSIZE)
        die("buffer too small");

    if (write(cfd, buf, wb) == -1)
        die("write cgroup.event_control");

    if (close(cfd) == -1)
        die("close cgroup.event_control");

    for (;;) {
        if (read(efd, &u, sizeof(uint64_t)) != sizeof(uint64_t))
            die("read eventfd");

        printf("mem_cgroup oom event received\n");
    }

    return 0;
}

```

The above program detects OOM situations in a cgroup specified as an argument on the command line and reports them using the **mem_cgroup oom event received** string to the standard output.

9. Run the above notification handler program in a separate console, specifying the **blue** cgroup's control files as arguments:

```
~]$ ./oom_notification /cgroup/memory/blue/cgroup.event_control
/cgroup/memory/blue/memory.oom_control
```

10. In a different console, run the **mem_hog** test program to create an OOM situation to see the **oom_notification** program report it on the standard output:

```
blue]# ~/mem-hog
```

3.8. net_cls

The **net_cls** subsystem tags network packets with a class identifier (classid) that allows the Linux traffic controller (**tc**) to identify packets originating from a particular cgroup. The traffic controller can be configured to assign different priorities to packets from different cgroups.

net_cls.classid

net_cls.classid contains a single value that indicates a traffic control *handle*. The value of **classid** read from the **net_cls.classid** file is presented in the decimal format while the value to be written to the file is expected in the hexadecimal format. For example, **0x100001** represents the handle conventionally written as **10:1** in the format used by *iproute2*. In the **net_cls.classid** file, it would be represented by the number **1048577**.

The format for these handles is: **0xAAAABBBB**, where **AAAA** is the major number in hexadecimal and **BBBB** is the minor number in hexadecimal. You can omit any leading zeroes; **0x10001** is the same as **0x00010001**, and represents **1:1**. The following is an example of setting a **10:1** handle in the **net_cls.classid** file:

```
~]# echo 0x100001 > /cgroup/net_cls/red/net_cls.classid
~]# cat /cgroup/net_cls/red/net_cls.classid
1048577
```

Refer to the man page for **tc** to learn how to configure the traffic controller to use the handles that the **net_cls** adds to network packets.

3.9. net_prio

The Network Priority (**net_prio**) subsystem provides a way to dynamically set the priority of network traffic per each network interface for applications within various cgroups. A network's priority is a number assigned to network traffic and used internally by the system and network devices. Network priority is used to differentiate packets that are sent, queued, or dropped. The **tc** command may be used to set a network's priority (setting the network priority via the **tc** command is outside the scope of this guide; for more information, refer to the **tc** man page).

Typically, an application sets the priority of its traffic via the **SO_PRIORITY** socket option. However, applications are often not coded to set the priority value, or the application's traffic is site-specific and does not provide a defined priority.

Using the **net_prio** subsystem in a cgroup allows an administrator to assign a process to a specific cgroup which defines the priority of outgoing traffic on a given network interface.

net_prio.prioidx

a read-only file which contains a unique integer value that the kernel uses as an internal representation of this cgroup.

net_prio.ifpriomap

contains a map of priorities assigned to traffic originating from processes in this group and leaving the system on various interfaces. This map is represented by a list of pairs in the form **<network_interface> <priority>**:

```
~]# cat /cgroup/net_prio/iscsi/net_prio.ifpriomap
eth0 5
eth1 4
eth2 6
```

Contents of the **net_prio.ifpriomap** file can be modified by echoing a string into the file using the above format, for example:

```
~]# echo "eth0 5" > /cgroup/net_prio/iscsi/net_prio.ifpriomap
```

The above command forces any traffic originating from processes belonging to the **iscsi** **net_prio** cgroup, and with traffic outgoing on the **eth0** network interface, to have the priority set to the value **5**. The parent cgroup also has a writable **net_prio.ifpriomap** file that can be used to set a system default priority.

3.10. ns

The **ns** subsystem provides a way to group processes into separate *namespaces*. Within a particular namespace, processes can interact with each other but are isolated from processes running in other namespaces. These separate namespaces are sometimes referred to as *containers* when used for operating-system-level virtualization.

3.11. perf_event

When the **perf_event** subsystem is attached to a hierarchy, all cgroups in that hierarchy can be used to group processes and threads which can then be monitored with the **perf** tool, as opposed to monitoring each process or thread separately or per-CPU. Cgroups which use the **perf_event** subsystem do not contain any special tunable parameters other than the common parameters listed in [Section 3.12, “Common Tunable Parameters”](#).

For additional information on how tasks in a cgroup can be monitored using the **perf** tool, refer to the Red Hat Enterprise Linux *Developer Guide*, available from http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/.

3.12. Common Tunable Parameters

The following parameters are present in every created cgroup, regardless of the subsystem that the cgroup is using:

tasks

contains a list of processes, represented by their PIDs, that are running in a cgroup. The list of PIDs is not guaranteed to be ordered or unique (that is, it may contain duplicate entries). Writing a PID into the **tasks** file of a cgroup moves that process into that cgroup.

cgroup.procs

contains a list of thread groups, represented by their TGIDs, that are running in a cgroup. The list of TGIDs is not guaranteed to be ordered or unique (that is, it may contain duplicate entries). Writing a TGID into the **cgroup.procs** file of a cgroup moves that thread group into that cgroup.

cgroup.event_control

along with the cgroup notification API, allows notifications to be sent about a changing status of a cgroup.

notify_on_release

contains a Boolean value, **1** or **0**, that either enables or disables the execution of the release agent. If the **notify_on_release** is enabled, the kernel executes the contents of the **release_agent** file when a cgroup no longer contains any tasks (that is, the cgroup's **tasks** file contained some PIDs and those PIDs were removed, leaving the file empty). A path to the empty cgroup is provided as an argument to the release agent.

The default value of the **notify_on_release** parameter in the root cgroup is **0**. All non-root cgroups inherit the value in **notify_on_release** from their parent cgroup.

release_agent (present in the root cgroup only)

contains a command to be executed when a “notify on release” is triggered. Once a cgroup is emptied of all processes, and the **notify_on_release** flag is enabled, the kernel runs the command in the **release_agent** file and supplies it with a relative path (relative to the root cgroup) to the emptied cgroup as an argument. The release agent can be used, for example, to automatically remove empty cgroups; for more information, see [Example 3.4, “Automatically removing empty cgroups”](#).

Example 3.4. Automatically removing empty cgroups

Follow these steps to configure automatic removal of any emptied cgroup from the **cpu** cgroup:

1. Create a shell script that removes empty **cpu** cgroups, place it in, for example, **/usr/local/bin**, and make it executable.

```
~]# cat /usr/local/bin/remove-empty-cpu-cgroup.sh
#!/bin/sh
rmdir /cgroup/cpu/$1
~]# chmod +x /usr/local/bin/remove-empty-cpu-cgroup.sh
```

The **\$1** variable contains a relative path to the emptied cgroup.

2. In the **cpu** cgroup, enable the **notify_on_release** flag:

```
~]# echo 1 > /cgroup/cpu/notify_on_release
```

3. In the **cpu** cgroup, specify a release agent to be used:

```
~]# echo "/usr/local/bin/remove-empty-cpu-cgroup.sh" >
/cgroup/cpu/release_agent
```

4. Test your configuration to make sure emptied cgroups are properly removed:

```
cpu]# pwd; ls
/cgroup/cpu
cgroup.event_control cgroup.procs cpu.cfs_period_us
cpu.cfs_quota_us cpu.rt_period_us cpu.rt_runtime_us cpu.shares
cpu.stat libvirt notify_on_release release_agent tasks
cpu]# cat notify_on_release
1
cpu]# cat release_agent
/usr/local/bin/remove-empty-cpu-cgroup.sh
cpu]# mkdir blue; ls
blue cgroup.event_control cgroup.procs cpu.cfs_period_us
cpu.cfs_quota_us cpu.rt_period_us cpu.rt_runtime_us cpu.shares
cpu.stat libvirt notify_on_release release_agent tasks
cpu]# cat blue/notify_on_release
1
cpu]# cgexec -g cpu:blue dd if=/dev/zero of=/dev/null bs=1024k &
[1] 8623
cpu]# cat blue/tasks
8623
cpu]# kill -9 8623
cpu]# ls
cgroup.event_control cgroup.procs cpu.cfs_period_us
cpu.cfs_quota_us cpu.rt_period_us cpu.rt_runtime_us cpu.shares
cpu.stat libvirt notify_on_release release_agent tasks
```

3.13. Additional Resources

Subsystem-Specific Kernel Documentation

All of the following files are located under the `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups/` directory (provided by the `kernel-doc` package).

- » **blkio** subsystem — **blkio-controller.txt**
- » **cpuacct** subsystem — **cpuacct.txt**
- » **cpuset** subsystem — **cpuset.txt**
- » **devices** subsystem — **devices.txt**
- » **freezer** subsystem — **freezer-subsystem.txt**
- » **memory** subsystem — **memory.txt**
- » **net_prio** subsystem — **net_prio.txt**

Additionally, refer to the following files on further information about the **cpu** subsystem:

- » Real-Time scheduling — `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/scheduler/sched-rt-group.txt`
- » CFS scheduling — `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/scheduler/sched-bwc.txt`

[5] Source code provided by Red Hat Engineer František Hrbata.

[6] Source code provided by Red Hat Engineer František Hrbata.

Chapter 4. Control Group Application Examples

This chapter provides application examples that take advantage of the cgroup functionality.

4.1. Prioritizing Database I/O

Running each instance of a database server inside its own dedicated virtual guest allows you to allocate resources per database based on their priority. Consider the following example: a system is running two database servers inside two KVM guests. One of the databases is a high priority database and the other one a low priority database. When both database servers are run simultaneously, the I/O throughput is decreased to accommodate requests from both databases equally; [Figure 4.1, “I/O throughput without resource allocation”](#) indicates this scenario — once the low priority database is started (around time 45), I/O throughput is the same for both database servers.

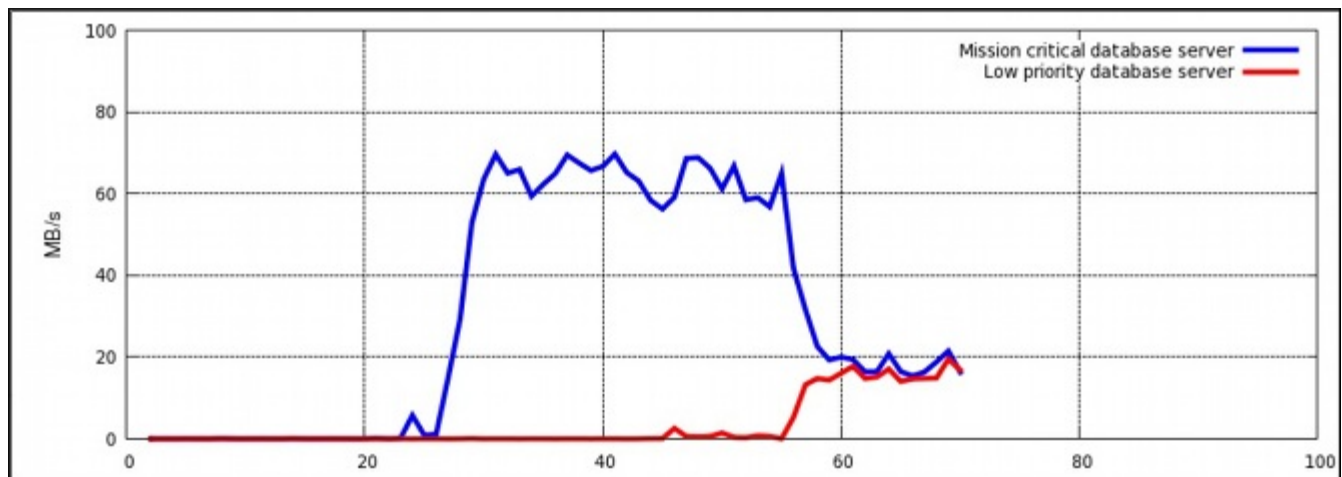


Figure 4.1. I/O throughput without resource allocation

To prioritize the high priority database server, it can be assigned to a cgroup with a high number of reserved I/O operations, whereas the low priority database server can be assigned to a cgroup with a low number of reserved I/O operations. To achieve this, follow the steps in [Procedure 4.1, “I/O throughput prioritization”](#), all of which are performed on the host system.

Procedure 4.1. I/O throughput prioritization

1. Attach the **blkio** subsystem to the **/cgroup/blkio** cgroup:

```
~]# mkdir /cgroup/blkio
~]# mount -t cgroup -o blkio blkio /cgroup/blkio
```

2. Create a high and low priority cgroup:

```
~]# mkdir /cgroup/blkio/high_prio
~]# mkdir /cgroup/blkio/low_prio
```

3. Acquire the PIDs of the processes that represent both virtual guests (in which the database servers are running) and move them to their specific cgroup. In our example, **VM_high** represents a virtual guest running a high priority database server, and **VM_low** represents a virtual guest running a low priority database server. For example:

```
~]# ps -eLf | grep qemu | grep VM_high | awk '{print $4}' | while read
pid; do echo $pid >> /cgroup/blkio/high_prio/tasks; done
~]# ps -eLf | grep qemu | grep VM_low | awk '{print $4}' | while read
pid; do echo $pid >> /cgroup/blkio/low_prio/tasks; done
```

- Set a ratio of 10:1 for the **high_prio** and **low_prio** cgroups. Processes in those cgroups (that is, processes running the virtual guests that have been added to those cgroups in the previous step) will immediately use only the resources made available to them.

```
~]# echo 1000 > /cgroup/blkio/high_prio/blkio.weight
~]# echo 100 > /cgroup/blkio/low_prio/blkio.weight
```

In our example, the low priority cgroup permits the low priority database server to use only about 10% of the I/O operations, whereas the high priority cgroup permits the high priority database server to use about 90% of the I/O operations.

[Figure 4.2, “I/O throughput with resource allocation”](#) illustrates the outcome of limiting the low priority database and prioritizing the high priority database. As soon as the database servers are moved to their appropriate cgroups (around time 75), I/O throughput is divided among both servers with the ratio of 10:1.

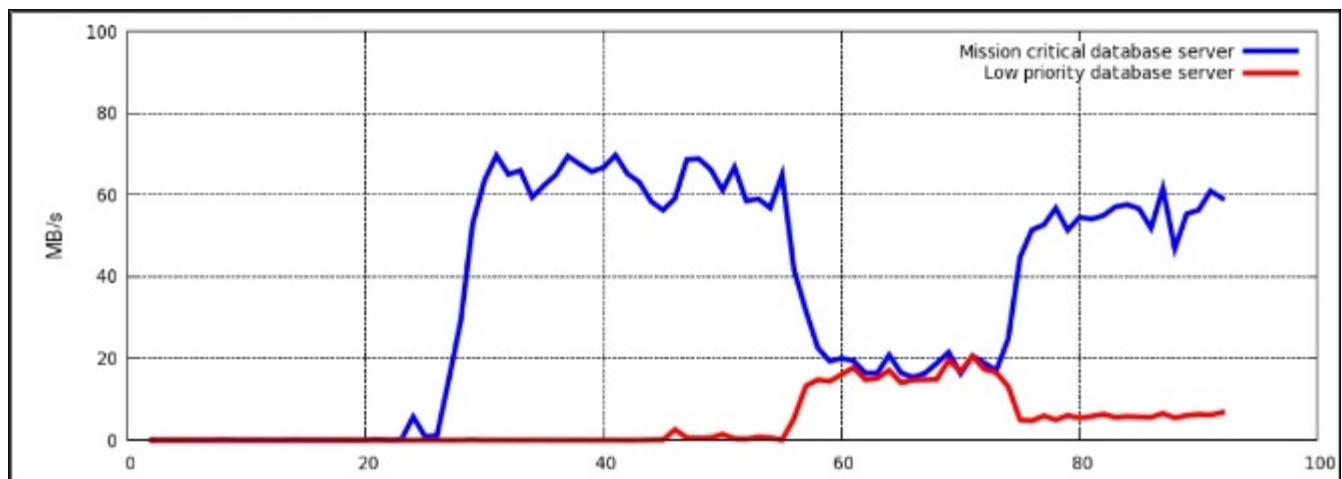


Figure 4.2. I/O throughput with resource allocation

Alternatively, block device I/O throttling can be used for the low priority database to limit its number of read and write operation. For more information on the **blkio** subsystem, refer to [Section 3.1, “blkio”](#).

4.2. Prioritizing Network Traffic

When running multiple network-related services on a single server system, it is important to define network priorities between these services. Defining these priorities ensures that packets originating from certain services have a higher priority than packets originating from other services. For example, such priorities are useful when a server system simultaneously functions as an NFS and Samba server. The NFS traffic must be of high priority as users expect high throughput. The Samba traffic can be deprioritized to allow better performance of the NFS server.

The **net_prio** subsystem can be used to set network priorities for processes in cgroups. These priorities are then translated into Type Of Service (TOS) bits and embedded into every packet. Follow the steps in [Procedure 4.2, “Setting Network Priorities for File Sharing Services”](#) to configure prioritization of two file sharing services (NFS and Samba).

Procedure 4.2. Setting Network Priorities for File Sharing Services

1. Attach the **net_prio** subsystem to the **/cgroup/net_prio** cgroup:

```
~]# mkdir /cgroup/net_prio
~]# mount -t cgroup -o net_prio net_prio /cgroup/net_prio
```

2. Create two cgroups, one for each service:

```
~]# mkdir /cgroup/net_prio/nfs_high
~]# mkdir /cgroup/net_prio/samba_low
```

3. To automatically move the **nfs** services to the **nfs_high** cgroup, add the following line to the **/etc/sysconfig/nfs** file:

```
CGROUP_DAEMON="net_prio:nfs_high"
```

This configuration ensures that **nfs** service processes are moved to the **nfs_high** cgroup when the **nfs** service is started or restarted. For more information about moving service processes to cgroups, refer to [Section 2.9.1, “Starting a Service in a Control Group”](#).

4. The **smbd** daemon does not have a configuration file in the **/etc/sysconfig** directory. To automatically move the **smbd** daemon to the **samba_low** cgroup, add the following line to the **/etc/cgrules.conf** file:

```
*:smbd                net_prio                samba_low
```

Note that this rule moves every **smbd** daemon, not only **/usr/sbin/smbd**, into the **samba_low** cgroup.

You can define rules for the **nmbd** and **winbindd** daemons to be moved to the **samba_low** cgroup in a similar way.

5. Start the **cgred** service to load the configuration from the previous step:

```
~]# service cgred start
Starting CGroup Rules Engine Daemon: [ OK ]
```

6. For the purposes of this example, let us assume both services use the **eth1** network interface. Define network priorities for each cgroup, where **1** denotes low priority and **10** denotes high priority:

```
~]# echo "eth1 1" > /cgroup/net_prio/samba_low
~]# echo "eth1 10" > /cgroup/net_prio/nfs_high
```

7. Start the **nfs** and **smb** services and check whether their processes have been moved into the correct cgroups:

```

~]# service smb start
Starting SMB services:                                [ OK ]
~]# cat /cgroup/net_prio/samba_low
16122
16124
~]# service nfs start
Starting NFS services:                                [ OK ]
Starting NFS quotas:                                  [ OK ]
Starting NFS mountd:                                  [ OK ]
Stopping RPC idmapd:                                  [ OK ]
Starting RPC idmapd:                                  [ OK ]
Starting NFS daemon:                                  [ OK ]
~]# cat /cgroup/net_prio/nfs_high
16321
16325
16376

```

Network traffic originating from NFS now has higher priority than traffic originating from Samba.

Similar to [Procedure 4.2, “Setting Network Priorities for File Sharing Services”](#), the **net_prio** subsystem can be used to set network priorities for client applications, for example, Firefox.

4.3. Per-group Division of CPU and Memory Resources

When a large amount of users use a single system, it is practical to provide certain users with more resources than others. Consider the following example: in a hypothetical company, there are three departments — finance, sales, and engineering. Because engineers use the system and its resources for more tasks than the other departments, it is logical that they have more resources available in case all departments are running CPU and memory intensive tasks.

Cgroups provide a way to limit the resources per each system group of users. For this example, assume that the following users have been created on the system:

```

~]$ grep home /etc/passwd
martin:x:500:500::/home/martin:/bin/bash
john:x:501:501::/home/john:/bin/bash
mark:x:502:502::/home/mark:/bin/bash
peter:x:503:503::/home/peter:/bin/bash
jenn:x:504:504::/home/jenn:/bin/bash
mike:x:505:505::/home/mike:/bin/bash

```

These users have been assigned to the following system groups:

```

~]$ grep -e "50[678]" /etc/group
finance:x:506:jenn,john
sales:x:507:mark,martin
engineering:x:508:peter,mike

```

For this example to work properly, you must have the *libcgroup* package installed. Using the **/etc/cgconfig.conf** and **/etc/cgrules.conf** files, you can create a hierarchy and a set of rules which determine the amount of resources for each user. To achieve this, follow the steps in [Procedure 4.3, “Per-group CPU and memory resource management”](#).

Procedure 4.3. Per-group CPU and memory resource management

1. In the **/etc/cgconfig.conf** file, configure the following subsystems to be mounted and

cgroups to be created:

```
mount {
    cpu      = /cgroup/cpu_and_mem;
    cpuacct  = /cgroup/cpu_and_mem;
    memory   = /cgroup/cpu_and_mem;
}

group finance {
    cpu {
        cpu.shares="250";
    }
    cpuacct {
        cpuacct.usage="0";
    }
    memory {
        memory.limit_in_bytes="2G";
        memory.memsw.limit_in_bytes="3G";
    }
}

group sales {
    cpu {
        cpu.shares="250";
    }
    cpuacct {
        cpuacct.usage="0";
    }
    memory {
        memory.limit_in_bytes="4G";
        memory.memsw.limit_in_bytes="6G";
    }
}

group engineering {
    cpu {
        cpu.shares="500";
    }
    cpuacct {
        cpuacct.usage="0";
    }
    memory {
        memory.limit_in_bytes="8G";
        memory.memsw.limit_in_bytes="16G";
    }
}
```

When loaded, the above configuration file mounts the **cpu**, **cpuacct**, and **memory** subsystems to a single **cpu_and_mem** cgroup. For more information on these subsystems, refer to [Chapter 3, Subsystems and Tunable Parameters](#). Next, it creates a hierarchy in **cpu_and_mem** which contains three cgroups: sales, finance, and engineering. In each of these cgroups, custom parameters are set for each subsystem:

- **cpu** — the **cpu.shares** parameter determines the share of CPU resources available to each process in all cgroups. Setting the parameter to **250**, **250**, and **500** in the finance, sales, and engineering cgroups respectively means that processes started in these groups will split the resources with a 1:1:2 ratio. Note that when a single process is running, it consumes as much CPU as necessary no matter which cgroup it is placed in. The CPU limitation only comes into effect when two or more processes compete for CPU resources.

- **cpuacct** — the **cpuacct.usage="0"** parameter is used to reset values stored in the **cpuacct.usage** and **cpuacct.usage_percpu** files. These files report total CPU time (in nanoseconds) consumed by all processes in a cgroup.
 - **memory** — the **memory.limit_in_bytes** parameter represents the amount of memory that is made available to all processes within a certain cgroup. In our example, processes started in the finance cgroup have 2 GB of memory available, processes in the sales group have 4 GB of memory available, and processes in the engineering group have 8 GB of memory available. The **memory.memsw.limit_in_bytes** parameter specifies the total amount of memory and swap space processes may use. Should a process in the finance cgroup hit the 2 GB memory limit, it is allowed to use another 1 GB of swap space, thus totaling the configured 3 GB.
2. To define the rules which the **cgrulesengd** daemon uses to move processes to specific cgroups, configure the **/etc/cgrules.conf** in the following way:

#<user/group>	<controller(s)>	<cgroup>
@finance	cpu,cpuacct,memory	finance
@sales	cpu,cpuacct,memory	sales
@engineering	cpu,cpuacct,memory	engineering

The above configuration creates rules that assign a specific system group (for example, **@finance**) the resource controllers it may use (for example, **cpu,cpuacct,memory**) and a cgroup (for example, **finance**) which contains all processes originating from that system group.

In our example, when the **cgrulesengd** daemon, started via the **service cgred start** command, detects a process that is started by a user that belongs to the finance system group (for example, **jenn**), that process is automatically moved to the **/cgroup/cpu_and_mem/finance/tasks** file and is subjected to the resource limitations set in the finance cgroup.

3. Start the **cgconfig** service to create the hierarchy of cgroups and set the needed parameters in all created cgroups:

```
~]# service cgconfig start
Starting cgconfig service: [ OK ]
```

Start the **cgred** service to let the **cgrulesengd** daemon detect any processes started in system groups configured in the **/etc/cgrules.conf** file:

```
~]# service cgred start
Starting CGroup Rules Engine Daemon: [ OK ]
```

Note that **cgred** is the name of the service that starts the **cgrulesengd** daemon.

4. To make all of the changes above persistent across reboots, configure the **cgconfig** and **cgred** services to be started by default:

```
~]# chkconfig cgconfig on
~]# chkconfig cgred on
```

To test whether this setup works, execute a CPU or memory intensive process and observe the results, for example, using the **top** utility. To test the CPU resource management, execute the following **dd** command under each user:

```
~]$ dd if=/dev/zero of=/dev/null bs=1024k
```

The above command reads the **/dev/zero** and outputs it to the **/dev/null** in chunks of 1024 KB. When the **top** utility is launched, you can see results similar to these:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
8201	peter	20	0	103m	1676	556	R	24.9	0.2	0:04.18	dd
8202	mike	20	0	103m	1672	556	R	24.9	0.2	0:03.47	dd
8199	jenn	20	0	103m	1676	556	R	12.6	0.2	0:02.87	dd
8200	john	20	0	103m	1676	556	R	12.6	0.2	0:02.20	dd
8197	martin	20	0	103m	1672	556	R	12.6	0.2	0:05.56	dd
8198	mark	20	0	103m	1672	556	R	12.3	0.2	0:04.28	dd
:											

All processes have been correctly assigned to their cgroups and are only allowed to consume CPU resource made available to them. If all but two processes, which belong to the finance and engineering cgroups, are stopped, the remaining resources are evenly split between both processes:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
8202	mike	20	0	103m	1676	556	R	66.4	0.2	0:06.35	dd
8200	john	20	0	103m	1672	556	R	33.2	0.2	0:05.08	dd
:											

Alternative method

Because the **cgrulesengd** daemon moves a process to a cgroup only after the appropriate conditions set by the rules in **/etc/cgrules.conf** have been fulfilled, that process may be running for a few milliseconds in an incorrect cgroup. An alternative way to move processes to their specified cgroups is to use the **pam_cgroup.so** PAM module. This module moves processes to available cgroups according to rules defined in the **/etc/cgrules.conf** file. Follow the steps in [Procedure 4.4, “Using a PAM module to move processes to cgroups”](#) to configure the **pam_cgroup.so** PAM module.

Procedure 4.4. Using a PAM module to move processes to cgroups

1. Install the *libcgroup-pam* package from the optional Red Hat Enterprise Linux Yum repository:

```
~]# yum install libcgroup-pam --enablerepo=rhel-6-server-optional-rpms
```

2. Ensure that the PAM module has been installed and exists:

```
~]# ls /lib64/security/pam_cgroup.so
/lib64/security/pam_cgroup.so
```

Note that on 32-bit systems, the module is placed in the **/lib/security** directory.

3. Add the following line to the **/etc/pam.d/su** file to use the **pam_cgroup.so** module each time the **su** command is executed:

```
session          optional          pam_cgroup.so
```

4. Configure the **/etc/cgconfig.conf** and **/etc/cgrules.conf** files as in [Procedure 4.4, “Using a PAM module to move processes to cgroups”](#).
5. Log out all users that are affected by the cgroup settings in the **/etc/cgrules.conf** file to apply the above configuration.

When using the **pam_cgroup.so** PAM module, you may disable the **cgred** service.

Revision History

Revision 1.0-22	Tue Nov 19 2013	Yoana Ruseva
Version for 6.5 GA release		
Revision 1.0-16	Thu Feb 21 2013	Martin Prpič
Red Hat Enterprise Linux 6.4 GA release of the <i>Resource Management Guide</i> . Includes various fixes and new content:		
<ul style="list-style-type: none">- Final use case scenarios — 584631- Documentation for the perf_event controller — 807326- Documentation for common cgroup files — 807329- Documentation for OOM control and the notification API — 822400, 822401- CPU ceiling enforcement documentation — 828991		
Revision 1.0-7	Wed Jun 20 2012	Martin Prpič
Red Hat Enterprise Linux 6.3 GA release of the <i>Resource Management Guide</i> .		
<ul style="list-style-type: none">- Added two use cases.- Added documentation for the net_prio subsystem.		
Revision 1.0-6	Tue Dec 6 2011	Martin Prpič
Red Hat Enterprise Linux 6.2 GA release of the <i>Resource Management Guide</i> .		
Revision 1.0-5	Thu May 19 2011	Martin Prpič
Red Hat Enterprise Linux 6.1 GA release of the <i>Resource Management Guide</i> .		
Revision 1.0-4	Tue Mar 1 2011	Martin Prpič
<ul style="list-style-type: none">- Fixed multiple examples — BZ#667623, BZ#667676, BZ#667699- Clarification of the cgclear command — BZ#577101- Clarification of the lssubsystem command — BZ#678517- Freezing a process — BZ#677548		
Revision 1.0-3	Wed Nov 17 2010	Rüdiger Landmann
Correct remount example — BZ#612805		
Revision 1.0-2	Thu Nov 11 2010	Rüdiger Landmann
Remove pre-release feedback instructions		
Revision 1.0-1	Wed Nov 10 2010	Rüdiger Landmann
Corrections from QE — BZ#581702 and BZ#612805		
Revision 1.0-0	Tue Nov 9 2010	Rüdiger Landmann
Feature-complete version for GA		