Red Hat Enterprise Linux 6

Resource Management Guide

Managing system resources on Red Hat Enterprise Linux 6

Rüdiger Landmann

Douglas Silas

Red Hat Enterprise Linux 6 Resource Management Guide
Managing system resources on Red Hat Enterprise Linux 6
Edition 1.0

Author	Rüdiger Landmann	r.landmann@redhat.com
Author	Douglas Silas	dhensley@redhat.com

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701

Managing system resources on Red Hat Enterprise Linux 6

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*[1] 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 includes 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 keycaps 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 keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set 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, click the **Left-handed mouse** check box and click

[1] https://fedorahosted.org/liberation-fonts/

> **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:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

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.

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 smartly divided up amongst 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 nine control group 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.).

[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.

- `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.
- `ns` — the *namespace* subsystem

Subsystems are also known as resource controllers

You may come across the term *resource controller* or simply *controller* in control group 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

Any single subsystem (such as `cpu`) can be attached to at most one hierarchy.

As a consequence, the `cpu` subsystem can never be attached to two different hierarchies.

Rule 2

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.

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 is made 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_and_mem`, 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_and_mem`, and any one cgroup in `net`.

The cgroup in `cpu_and_mem` that the `http` 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 control groups, therefore, every system task is always in at least one cgroup.

Rule 4

Any process (task) on the system which forks itself creates a child process (task). The child task automatically becomes members of all of the cgroups its parent is members of. The child task can then be moved to different cgroups as needed, but initially, it always inherits the cgroups (the "environment" in process terminology) of its parent task.

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.

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 control group 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.

Using Control Groups

The easiest way to work with control groups 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.

Note: 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. We recommend 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 control group configuration file — **/etc/cgconfig.conf**. Control groups are therefore recreated from session to session and become persistent. Depending on the contents of the configuration file, **cgconfig** can create hierarchies, mount necessary file systems, create control groups, and set subsystem parameters for each group.

The default **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 **service cgconfig stop**), it unmounts all the hierarchies that it mounted.

2.1.1. The cgconfig.conf File

The **cgconfig.conf** file contains two major types of entry — *mount* and *group*. Mount entries create and mount hierarchies as virtual filesystems, and attach subsystems to those hierarchies. For example:

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

creates a hierarchy for the `cpuset` subsystem, the equivalent of the shell commands:

```
mkdir /cgroup/cpuset
mount -t cgroup -o cpuset cpuset /cgroup/cpuset
```

Group entries create control groups and set subsystem parameters. For example:

```
group daemons/sql {
    perm {
        task {
            uid = root;
            gid = sqladmin;
        } admin {
            uid = root;
            gid = root;
        }
    } cpuset {
        cpuset.cpus = 0-3;
    }
}
```

creates a control group for sql daemons, with permissions for users in the **sqladmin** group to add tasks to the control group and the **root** user to modify subsystem parameters. When combined with the example of the mount entry above, the equivalent shell commands are:

```
mkdir -p /cgroup/cpu/daemons/sql
chown root:root /cgroup/cpu/daemons/sql/*
chown root:sqladmin /cgroup/cpu/daemons/sql/tasks
echo 0-3 > /cgroup/cpu/daemons/sql/cpuset.cpus
```

When you install *cgroups*, a sample config file is written to **/etc/cgconfig.conf**. The # 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

Warning — Effects on running systems

The following instructions, which cover creating a new hierarchy and attaching subsystems to it, assume that control groups 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 control groups 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 control group 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 2.1. 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 we create. We will attach the cpu, cpuset and memory subsystems to a hierarchy we name cpu_and_mem, and **mount** the cpu_and_mem hierarchy on **/cgroup/cpu_and_mem**:
>
> ```
> ~]# mount -t cgroup -o cpu,cpuset,memory cpu_and_mem /cgroup/cpu_and_mem
> ```
>
> You can list all available subsystems along with their current mount points (i.e. where the hierarchy they are attached to is mounted) with the **lssubsys** [1] command:
>
> ```
> ~]# lssubsys -am
> cpu,cpuset,memory /cgroup/cpu_and_mem
> net_cls
> ns
> cpu
> 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

[1] The **lssubsys** command is one of the utilties 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**.

- the `net_cls`, `ns`, `cpu`, `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 2.2. 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
cpu
cpuacct
devices
freezer
blkio
```

We remount the `cpu_and_mem` hierarchy, using the **`remount`** option, and including `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 2.11, "Unloading Groups"*.

2.5. Creating Cgroups

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** pseudofile for this control group. This user can add tasks to the control group.

Note — Removing tasks

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

- **-a** (optional) — specifies a user (by user ID, uid) and a group (by group ID, gid) to own all pseudofiles other than **tasks** for this control group. This user can modify the access that the tasks in this control group 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 control group 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 2.3, "cgcreate usage"*.

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

> Example 2.3. 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`. We now run:

```
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 we did not specify it in the **cgcreate** command.

Alternative method

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

```
mkdir /cgroup/hierarchy/name/child_name
```

For example:

```
mkdir /cgroup/cpuset/lab1/group1
```

2.6. Removing Cgroups

Remove cgroups with the **cgdelete**, which has a syntax similar to that of **cgcreate**. Run: **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 control group, 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 control group. For example, if **/cgroup/cpuset/group1** exists, specify the CPUs to which this group has access with the following command:

```
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 control group *relative to the root of the hierarchy*. For example, to set the parameter of the root group, run:

```
$ cgset -r cpuset.cpus=1 /
```

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

```
$ cgset -r cpuset.cpus=1 .
```

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

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

```
$ cgset -r cpuset.cpus=1 group1
```

A trailing slash on the name of the group (for example, **`cpuset.cpus=1 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 control group whose parameters are to be copied, relative to the root group of the hierarchy
- *`path_to_target_cgroup`* is the path to the destination control group, 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.

Alternative method

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

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

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

2.8. Moving a Process to a Control Group

Move a process into a control group by running the **`cgclassify`** command:

```
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 control groups 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 control group 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 control group. If you do not set this option and the **cgred** daemon is running, child processes will be allocated to control groups based on the settings found in **/etc/cgrules.conf**. The process itself, however, will remain in the control group 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 control group **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 control group directly, write its PID to the **tasks** file of the control group. For example, to move a process with the PID **1701** into a control group at **/cgroup/lab1/group1/**:

```
echo 1701 > /cgroup/lab1/group1/tasks
```

2.8.1. The cgred Daemon

Cgred is a daemon that moves tasks into control groups 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 hierarchies control_group*
- *user*:*command hierarchies 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** control group. To associate particular commands with particular control groups, 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` control group in the hierarchy that contains the `devices` subsystem. Note, however, that the daemon moves the process to the control group 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/cgrules.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

Important — Mandatory parameters

Some controllers have mandatory parameters that you must set before you run a task in a hierarchy that includes those controllers. For example, before you use the `cpuset` controller, *`cpuset.cpus`* and *`cpuset.mems`* must be defined.

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 *Section 3.10, "Additional Resources"* for a description of which parameters are mandatory for given subsystems.

Launch processes in a control group by running the **cgexec** command. For example, this command launches the **lynx** web browser within the `group1` control group, 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 arguements*** , 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 control groups 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 control group relative to the hierarchy.
- *`command`* is the command to run
- *`arguements`* are any arguements for the command

You can also add the **--sticky** option before the *command* to keep any child processes in the same control group. If you do not set this option and the **cgred** daemon is running, child processes will be allocated to control groups based on the settings found in **/etc/cgrules.conf**. The process itself, however, will remain in the control group 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 control group 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` control group. 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 some services in a control group. Services that can be started in control groups must:

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

To make an eligible service start in a control group, 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 control group in that hierarchy. For example:

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

2.10. Obtaining Information About Control Groups

2.10.1. Finding a Process

To find the control group to which a process belongs, run:

```
ps -O cgroup
```

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

```
cat /proc/PID/cgroup
```

2.10.2. Finding a Subsystem

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

```
cat /proc/cgroups
```

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.

2.10.3. Finding Hierarchies

We recommend 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 control groups within them:

```
tree /cgroup
```

2.10.4. Finding Control Groups

To list the control groups 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** control group in the hierarchy to which the `cpuset` subsystem is attached.

2.10.5. Displaying Parameters of Control Groups

To display the parameters of specific control groups, run:

```
cgget -r parameter list_of_cgroups
```

where *parameter* is a pseudofile that contains values for a subsystem, and *list_of_cgroups* is a list of control groups 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 control groups `lab1` and `lab2`.

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

```
cgget -g cpuset /
```

2.11. Unloading Groups

Warning — This Command Destroys all Control Groups

The **cgclear** command destroys all control groups 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 control group file system, use the **cgclear** command.

All tasks in the control group are reallocated to the root node of the hierarchies, all control groups are removed, and the filesystem itself is unmounted from the system, thus destroying all previously mounted hierarchies. Finally, the directory where the cgroup filesystem was mounted is actually deleted.

2.12. Additional Resources

The definitive documentation for control group 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 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.

Subsystems and Tunable Parameters

Subsystems are kernel modules that are aware of control groups. Typically, they are resource controllers that allocate varying levels of system resources to different control groups. 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/**. 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 control group are represented as *pseudofiles* within the control group's virtual file system. These pseudofiles can be manipulated by shell commands or their equivalent system calls. For example, **cpuset.cpus** is a pseudofile that specifies which CPUs a control group is permitted to access. If `/cgroup/cpuset/webserver` is a control group for the web server that runs on a system, and we run the following command:

```
~]# 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 `blkio` subsystem controls and monitors access to I/O on block devices by tasks in control groups. 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.

blkio.weight
: specifies the relative proportion (*weight*) of block I/O access available by default to a control group, in the range **100** to **1000**. This value is overriden for specific devices by the *blkio.weight_device* parameter.For example, to assign a default weight of **500** to a control group 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 control group, in the range **100** to **1000**. The value of this parameter overrides the value of *blkio.weight* 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 control group for access to `/dev/sda`, run:

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

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

blkio.time
: reports the time that a control group 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 control group. 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.io_service_bytes
: reports the number of bytes transferred to or from specific devices by a control group. 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_serviced
: reports the number of I/O operations performed on specific devices by a control group. 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 *number* represents the number of operations.

blkio.io_service_time
: reports the total time between request dispatch and request completion for I/O operations on specific devices by a control group. 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 *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 control group spent waiting for service in the scheduler queues. When you interpret this report, note:

  - the time reported can be greated than the total time elapsed, because the time reported is the cumulative total of all I/O operations for the control group rather than the time that the control group itself spent waiting for I/O operations. To find the time that the group as a whole has spent waiting, use **blkio.group_wait_time**.
  - 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 *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 *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 control group. 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 control group. 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.avg_queue_size
: reports the average queue size for I/O operations by a control group, over the entire length of time of the group's existence. The queue size is sampled every time a queue for this control group 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 control group spent waiting for a timeslice for one of its queues. The report is updated every time a queue for this control group gets a timeslice, so if you read this pseudofile while the control group 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 control group spent without any pending requests. The report is updated every time a queue for this control group has a pending request, so if you read this pseudofile while the control group 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 control group 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 control group 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 control group 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.reset_stats
: resets the statistics recorded in the other pseudofiles. Write an integer to this file to reset the statistics for this cgroup.

3.2. cpu

The cpu subsystem schedules CPU access to control groups. Access to CPU resources can be scheduled according to the following parameters, each one in a separate *pseudofile* within the control group virtual file system:

cpu.shares
: contains an integer value that specifies a relative share of CPU time available to the tasks in a control group. For example, tasks in two control groups that have **cpu.shares** set to **1** will receive equal CPU time, but tasks in a control group that has **cpu.shares** set to **2** receive twice the CPU time of tasks in a control group where **cpu.shares** is set to **1**.

cpu.rt_runtime_us
: specifies a period of time in microseconds (µs, represented here as "us") for the longest continuous period in which the tasks in a control group have access to CPU resources. Establishing this limit prevents tasks in one control group from monopolizing CPU time. If the tasks in a control group should be able to access CPU resources for 4 seconds out of every 5 seconds, set **`cpu.rt_runtime_us`** to **`4000000`** and **`cpu.rt_period_us`** to **`5000000`**.

cpu.rt_period_us
: specifies a period of time in microseconds (µs, represented here as "us") for how regularly a control group's access to CPU resource should be reallocated. If the tasks in a control group should be able to access CPU resources for 4 seconds out of every 5 seconds, set **`cpu.rt_runtime_us`** to **`4000000`** and **`cpu.rt_period_us`** to **`5000000`**.

3.3. cpuacct

The `cpuacct` subsystem generates automatic reports on CPU resources used by the tasks in a control group, including tasks in child groups. Three reports are available:

cpuacct.stat
: reports the number of CPU cycles (in the units defined by USER_HZ on the system) consumed by tasks in this control group and its children in both user mode and system (kernel) mode.

cpuacct.usage
: reports the total CPU time (in nanoseconds) consumed by all tasks in this control group (including tasks lower in the hierarchy).

cpuacct.usage_percpu
: reports the CPU time (in nanoseconds) consumed on each CPU by all tasks in this control group (including tasks lower in the hierarchy).

3.4. cpuset

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

cpuset.cpus (mandatory)
: specifies the CPUs that tasks in this control group are permitted to access. This is a comma-separated list in ASCII format, 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 control group 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 **cpuset.mem_hardwall**.

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's 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 control group. 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 control group has no effect if load balancing is enabled in any parent control group, as load balancing is already being carried out at a higher level. Therefore, to disable load balancing in a control group, 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 control group 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 control group.

Technology Preview

The `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 control group 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 control group 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 control group.

3.6. freezer

The `freezer` subsystem suspends or resumes tasks in a control group.

freezer.state
: `freezer.state` has three possible values:

  - **`FROZEN`** — tasks in the control group are suspended.
  - **`FREEZING`** — the system is in the process of suspending tasks in the control group.
  - **`THAWED`** — tasks in the control group have resumed.

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 control group, 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 3.1. Values reported by memory.stat

Statistic	Description
cache	page cache, including tmpfs (shmem), in bytes
rss	anonymous and swap cache, *not* 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 control group, but on all its children as well. For example, swap reports the swap usage by a control group and total_swap reports the total swap usage by the control group 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 ≠ 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 control group (in bytes).

memory.memsw.usage_in_bytes
: reports the sum of current memory usage plus swap space used by processes in the control group (in bytes).

memory.max_usage_in_bytes
: reports the maximum memory used by processes in the control group (in bytes).

memory.memsw.max_usage_in_bytes
: reports the maximum amount of memory and swap space used by processes in the control group (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 control group; 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 control group; you can only apply values to groups lower in the hierarchy.

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

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 control group. This interface can only be used when the control group has no tasks. If memory cannot be freed, it is moved to a parent control group if possible. Use `memory.force_empty` before removing a control group to avoid moving out-of-use page caches to its parent control group.

memory.swappiness
: sets the tendency of the kernel to swap out process memory used by tasks in this control group 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 control group.

  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 control group, which uses the swappiness set in **`/proc/sys/vm/swappiness`**.
- a control group 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 control groups. If enabled (**`1`**), the memory controller reclaims memory from the children of and process that exceeds its memory limit. By default (**`0`**), the controller does not reclaim memory from a task's children.

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 control group. The traffic controller can be configured to assign different priorities to packets from different control groups.

net_cls.classid
: `net_cls.classid` contains a single value in hexadecimal format that indicates a traffic control *handle*. For example, **`0x100001`** represents the handle conventionally written as **`10:1`** in the format used by *iproute2*.

  The format for these handles is: **`0x`*`AAAABBBB`***, 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`**.

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. 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.

3.10. 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.

- `blkio` subsystem — **`blkio-controller.txt`**
- `cpuacct` subsystem — **`cpuacct.txt`**
- `cpuset` subsystem — **`cpusets.txt`**
- `devices` subsystem — **`devices.txt`**
- `freezer` subsystem — **`freezer-subsystem.txt`**

- `memory` subsystem — **`memory.txt`**

Appendix A. Revision History

Revision 1.0-3 Wed Nov 17 2010 **Rüdiger Landmann** *r.landmann@redhat.com*

Correct remount example — *BZ#612805*[1]

Revision 1.0-2 Thu Nov 11 2010 **Rüdiger Landmann** *r.landmann@redhat.com*

Remove pre-release feedback instructions

Revision 1.0-1 Wed Nov 10 2010 **Rüdiger Landmann** *r.landmann@redhat.com*

Corrections from QE — *BZ#581702*[2] and *BZ#612805*[3]

Revision 1.0-0 Tue Nov 9 2010 **Rüdiger Landmann** *r.landmann@redhat.com*

Feature-complete version for GA