NAME

cgroups − Linux control groups

DESCRIPTION

Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, and so on).

Terminology

A *cgroup* is a collection of processes that are bound to a set of limits or parameters defined via the cgroup filesystem.

A *subsystem* is a kernel component that modifies the behavior of the processes in a cgroup. Various subsystems have been implemented, making it possible to do things such as limiting the amount of CPU time and memory available to a cgroup, accounting for the CPU time used by a cgroup, and freezing and resuming execution of the processes in a cgroup. Subsystems are sometimes also known as *resource controllers* (or simply, controllers).

The cgroups for a controller are arranged in a *hierarchy*. This hierarchy is defined by creating, removing, and renaming subdirectories within the cgroup filesystem. At each level of the hierarchy, attributes (e.g., limits) can be defined. The limits, control, and accounting provided by cgroups generally have effect throughout the subhierarchy underneath the cgroup where the attributes are defined. Thus, for example, the limits placed on a cgroup at a higher level in the hierarchy cannot be exceeded by descendant cgroups.

Cgroups version 1 and version 2

The initial release of the cgroups implementation was in Linux 2.6.24. Over time, various cgroup controllers have been added to allow the management of various types of resources. However, the development of these controllers was largely uncoordinated, with the result that many inconsistencies arose between controllers and management of the cgroup hierarchies became rather complex. (A longer description of these problems can be found in the kernel source file *Documentation/cgroup−v2.txt*.)

Because of the problems with the initial cgroups implementation (cgroups version 1), starting in Linux 3.10, work began on a new, orthogonal implementation to remedy these problems. Initially marked experimental, and hidden behind the *−o __DEVEL__sane_behavior* mount option, the new version (cgroups version 2) was eventually made official with the release of Linux 4.5. Differences between the two versions are described in the text below.

Although cgroups v2 is intended as a replacement for cgroups v1, the older system continues to exist (and for compatibility reasons is unlikely to be removed). Currently, cgroups v2 implements only a subset of the controllers available in cgroups v1. The two systems are implemented so that both v1 controllers and v2 controllers can be mounted on the same system. Thus, for example, it is possible to use those controllers that are supported under version 2, while also using version 1 controllers where version 2 does not yet support those controllers. The only restriction here is that a controller can't be simultaneously employed in both a cgroups v1 hierarchy and in the cgroups v2 hierarchy.

CGROUPS VERSION 1

Under cgroups v1, each controller may be mounted against a separate cgroup filesystem that provides its own hierarchical organization of the processes on the system. It is also possible to comount multiple (or even all) cgroups v1 controllers against the same cgroup filesystem, meaning that the comounted controllers manage the same hierarchical organization of processes.

For each mounted hierarchy, the directory tree mirrors the control group hierarchy. Each control group is represented by a directory, with each of its child control cgroups represented as a child directory. For instance, */user/joe/1.session* represents control group *1.session*, which is a child of cgroup *joe*, which is a child of */user*. Under each cgroup directory is a set of files which can be read or written to, reflecting resource limits and a few general cgroup properties.

Tasks (threads) versus processes

In cgroups v1, a distinction is drawn between *processes* and *tasks*. In this view, a process can consist of multiple tasks (more commonly called threads, from a user-space perspective, and called such in the remainder of this man page). In cgroups v1, it is possible to independently manipulate the cgroup memberships of the threads in a process.

The cgroups v1 ability to split threads across different cgroups caused problems in some cases. For example, it made no sense for the *memory* controller, since all of the threads of a process share a single address space. Because of these problems, the ability to independently manipulate the cgroup memberships of the threads in a process was removed in the initial cgroups v2 implementation, and subsequently restored in a more limited form (see the discussion of "thread mode" below).

Mounting v1 controllers

The use of cgroups requires a kernel built with the **CONFIG_CGROUP** option. In addition, each of the v1 controllers has an associated configuration option that must be set in order to employ that controller.

In order to use a v1 controller, it must be mounted against a cgroup filesystem. The usual place for such mounts is under a **tmpfs**(5) filesystem mounted at */sys/fs/cgroup*. Thus, one might mount the *cpu* controller as follows:

```
mount -t cgroup -o cpu none /sys/fs/cgroup/cpu
```

It is possible to comount multiple controllers against the same hierarchy. For example, here the *cpu* and *cpuacct* controllers are comounted against a single hierarchy:

```
mount -t cgroup -o cpu,cpuacct none /sys/fs/cgroup/cpu,cpuacct
```

Comounting controllers has the effect that a process is in the same cgroup for all of the comounted controllers. Separately mounting controllers allows a process to be in cgroup */foo1* for one controller while being in */foo2/foo3* for another.

It is possible to comount all v1 controllers against the same hierarchy:

```
mount -t cgroup -o all cgroup /sys/fs/cgroup
```

(One can achieve the same result by omitting *−o all*, since it is the default if no controllers are explicitly specified.)

It is not possible to mount the same controller against multiple cgroup hierarchies. For example, it is not possible to mount both the *cpu* and *cpuacct* controllers against one hierarchy, and to mount the *cpu* controller alone against another hierarchy. It is possible to create multiple mount points with exactly the same set of comounted controllers. However, in this case all that results is multiple mount points providing a view of the same hierarchy.

Note that on many systems, the v1 controllers are automatically mounted under */sys/fs/cgroup*; in particular, **systemd**(1) automatically creates such mount points.

Unmounting v1 controllers

A mounted cgroup filesystem can be unmounted using the **umount**(8) command, as in the following example:

```
umount /sys/fs/cgroup/pids
```

But note well: a cgroup filesystem is unmounted only if it is not busy, that is, it has no child cgroups. If this is not the case, then the only effect of the **umount**(8) is to make the mount invisible. Thus, to ensure that the mount point is really removed, one must first remove all child cgroups, which in turn can be done only after all member processes have been moved from those cgroups to the root cgroup.

Cgroups version 1 controllers

Each of the cgroups version 1 controllers is governed by a kernel configuration option (listed below). Additionally, the availability of the cgroups feature is governed by the **CONFIG_CGROUPS** kernel configuration option.

cpu (since Linux 2.6.24; **CONFIG_CGROUP_SCHED**)

Cgroups can be guaranteed a minimum number of "CPU shares" when a system is busy. This does not limit a cgroup's CPU usage if the CPUs are not busy. For further information, see *Documentation/scheduler/sched-design-CFS.txt*.

In Linux 3.2, this controller was extended to provide CPU "bandwidth" control. If the kernel is configured with **CONFIG_CFS_BANDWIDTH**, then within each scheduling period (defined via a file in the cgroup directory), it is possible to define an upper limit on the CPU time allocated to the processes in a cgroup. This upper limit applies even if there is no other competition for the CPU. Further information can be found in the kernel source file *Documentation/scheduler/sched−bwc.txt*.

cpuacct (since Linux 2.6.24; **CONFIG_CGROUP_CPUACCT**)

This provides accounting for CPU usage by groups of processes.

Further information can be found in the kernel source file *Documentation/cgroup−v1/cpuacct.txt*.

cpuset (since Linux 2.6.24; **CONFIG_CPUSETS**)

This cgroup can be used to bind the processes in a cgroup to a specified set of CPUs and NUMA nodes.

Further information can be found in the kernel source file *Documentation/cgroup−v1/cpusets.txt*.

memory (since Linux 2.6.25; **CONFIG_MEMCG**)

The memory controller supports reporting and limiting of process memory, kernel memory, and swap used by cgroups.

Further information can be found in the kernel source file *Documentation/cgroup−v1/memory.txt*.

devices (since Linux 2.6.26; **CONFIG_CGROUP_DEVICE**)

This supports controlling which processes may create (mknod) devices as well as open them for reading or writing. The policies may be specified as whitelists and blacklists. Hierarchy is enforced, so new rules must not violate existing rules for the target or ancestor cgroups.

Further information can be found in the kernel source file *Documentation/cgroup-v1/devices.txt*.

freezer (since Linux 2.6.28; **CONFIG_CGROUP_FREEZER**)

The *freezer* cgroup can suspend and restore (resume) all processes in a cgroup. Freezing a cgroup */A* also causes its children, for example, processes in */A/B*, to be frozen.

Further information can be found in the kernel source file *Documentation/cgroup-v1/freezer-subsystem.txt*.

net_cls (since Linux 2.6.29; **CONFIG_CGROUP_NET_CLASSID**)

This places a classid, specified for the cgroup, on network packets created by a cgroup. These classids can then be used in firewall rules, as well as used to shape traffic using **tc**(8). This applies only to packets leaving the cgroup, not to traffic arriving at the cgroup.

Further information can be found in the kernel source file *Documentation/cgroup-v1/net_cls.txt*.

blkio (since Linux 2.6.33; **CONFIG_BLK_CGROUP**)

The *blkio* cgroup controls and limits access to specified block devices by applying IO control in the form of throttling and upper limits against leaf nodes and intermediate nodes in the storage hierarchy.

Two policies are available. The first is a proportional-weight time-based division of disk implemented with CFQ. This is in effect for leaf nodes using CFQ. The second is a throttling policy which specifies upper I/O rate limits on a device.

Further information can be found in the kernel source file *Documentation/cgroup-v1/blkio-controller.txt*.

perf_event (since Linux 2.6.39; **CONFIG_CGROUP_PERF**)

This controller allows *perf* monitoring of the set of processes grouped in a cgroup.

Further information can be found in the kernel source file *tools/perf/Documentation/perf-record.txt*.

net_prio (since Linux 3.3; **CONFIG_CGROUP_NET_PRIO**)
This allows priorities to be specified, per network interface, for cgroups.

Further information can be found in the kernel source file *Documentation/cgroup-v1/net_prio.txt*.

hugetlb (since Linux 3.5; **CONFIG_CGROUP_HUGETLB**)
This supports limiting the use of huge pages by cgroups.

Further information can be found in the kernel source file *Documentation/cgroup-v1/hugetlb.txt*.

pids (since Linux 4.3; **CONFIG_CGROUP_PIDS**)
This controller permits limiting the number of process that may be created in a cgroup (and its descendants).

Further information can be found in the kernel source file *Documentation/cgroup-v1/pids.txt*.

rdma (since Linux 4.11; **CONFIG_CGROUP_RDMA**)
The RDMA controller permits limiting the use of RDMA/IB-specific resources per cgroup.

Further information can be found in the kernel source file *Documentation/cgroup-v1/rdma.txt*.

Creating cgroups and moving processes

A cgroup filesystem initially contains a single root cgroup, '/', which all processes belong to. A new cgroup is created by creating a directory in the cgroup filesystem:

```
mkdir /sys/fs/cgroup/cpu/cg1
```

This creates a new empty cgroup.

A process may be moved to this cgroup by writing its PID into the cgroup's *cgroup.procs* file:

```
echo $$ > /sys/fs/cgroup/cpu/cg1/cgroup.procs
```

Only one PID at a time should be written to this file.

Writing the value 0 to a *cgroup.procs* file causes the writing process to be moved to the corresponding cgroup.

When writing a PID into the *cgroup.procs*, all threads in the process are moved into the new cgroup at once.

Within a hierarchy, a process can be a member of exactly one cgroup. Writing a process's PID to a *cgroup.procs* file automatically removes it from the cgroup of which it was previously a member.

The *cgroup.procs* file can be read to obtain a list of the processes that are members of a cgroup. The returned list of PIDs is not guaranteed to be in order. Nor is it guaranteed to be free of duplicates. (For example, a PID may be recycled while reading from the list.)

In cgroups v1, an individual thread can be moved to another cgroup by writing its thread ID (i.e., the kernel thread ID returned by **clone**(2) and **gettid**(2)) to the *tasks* file in a cgroup directory. This file can be read to discover the set of threads that are members of the cgroup.

Removing cgroups

To remove a cgroup, it must first have no child cgroups and contain no (nonzombie) processes. So long as that is the case, one can simply remove the corresponding directory pathname. Note that files in a cgroup directory cannot and need not be removed.

Cgroups v1 release notification

Two files can be used to determine whether the kernel provides notifications when a cgroup becomes empty. A cgroup is considered to be empty when it contains no child cgroups and no member processes.

A special file in the root directory of each cgroup hierarchy, *release_agent*, can be used to register the pathname of a program that may be invoked when a cgroup in the hierarchy becomes empty. The pathname of the newly empty cgroup (relative to the cgroup mount point) is provided as the sole command-line argument when the *release_agent* program is invoked. The *release_agent* program might remove the cgroup directory, or perhaps repopulate it with a process.

The default value of the *release_agent* file is empty, meaning that no release agent is invoked.

The content of the *release_agent* file can also be specified via a mount option when the cgroup filesystem is mounted:

```
mount -o release_agent=pathname ...
```

Whether or not the *release_agent* program is invoked when a particular cgroup becomes empty is determined by the value in the *notify_on_release* file in the corresponding cgroup directory. If this file contains the value 0, then the *release_agent* program is not invoked. If it contains the value 1, the *release_agent* program is invoked. The default value for this file in the root cgroup is 0. At the time when a new cgroup is created, the value in this file is inherited from the corresponding file in the parent cgroup.

Cgroup v1 named hierarchies

In cgroups v1, it is possible to mount a cgroup hierarchy that has no attached controllers:

```
mount -t cgroup -o none,name=somename none /some/mount/point
```

Multiple instances of such hierarchies can be mounted; each hierarchy must have a unique name. The only purpose of such hierarchies is to track processes. (See the discussion of release notification below.) An example of this is the *name=systemd* cgroup hierarchy that is used by **systemd**(1) to track services and user sessions.

Since Linux 5.0, the *cgroup_no_v1* kernel boot option (described below) can be used to disable cgroup v1 named hierarchies, by specifying *cgroup_no_v1=named*.

CGROUPS VERSION 2

In cgroups v2, all mounted controllers reside in a single unified hierarchy. While (different) controllers may be simultaneously mounted under the v1 and v2 hierarchies, it is not possible to mount the same controller simultaneously under both the v1 and the v2 hierarchies.

The new behaviors in cgroups v2 are summarized here, and in some cases elaborated in the following subsections.

1. Cgroups v2 provides a unified hierarchy against which all controllers are mounted.
2. "Internal" processes are not permitted. With the exception of the root cgroup, processes may reside only in leaf nodes (cgroups that do not themselves contain child cgroups). The details are somewhat more subtle than this, and are described below.
3. Active cgroups must be specified via the files *cgroup.controllers* and *cgroup.subtree_control*.
4. The *tasks* file has been removed. In addition, the *cgroup.clone_children* file that is employed by the *cpuset* controller has been removed.
5. An improved mechanism for notification of empty cgroups is provided by the *cgroup.events* file.

For more changes, see the *Documentation/cgroup-v2.txt* file in the kernel source.

Some of the new behaviors listed above saw subsequent modification with the addition in Linux 4.14 of "thread mode" (described below).

Cgroups v2 unified hierarchy

In cgroups v1, the ability to mount different controllers against different hierarchies was intended to allow great flexibility for application design. In practice, though, the flexibility turned out to be less useful than expected, and in many cases added complexity. Therefore, in cgroups v2, all available controllers are mounted against a single hierarchy. The available controllers are automatically mounted, meaning that it is not necessary (or possible) to specify the controllers when mounting the cgroup v2 filesystem using a command such as the following:

```
mount -t cgroup2 none /mnt/cgroup2
```

A cgroup v2 controller is available only if it is not currently in use via a mount against a cgroup v1 hierarchy. Or, to put things another way, it is not possible to employ the same controller against both a v1 hierarchy and the unified v2 hierarchy. This means that it may be necessary first to unmount a v1 controller (as described above) before that controller is available in v2. Since **systemd**(1) makes heavy use of some v1 controllers by default, it can in some cases be simpler to boot the system with selected v1 controllers disabled. To do this, specify the *cgroup_no_v1=list* option on the kernel boot command line; *list* is a comma-separated list of the names of the controllers to disable, or the word *all* to disable all v1 controllers. (This situation is correctly handled by **systemd**(1), which falls back to operating without the specified controllers.)

Note that on many modern systems, **systemd**(1) automatically mounts the *cgroup2* filesystem at */sys/fs/cgroup/unified* during the boot process.

Cgroups v2 controllers

The following controllers, documented in the kernel source file *Documentation/cgroup-v2.txt*, are supported in cgroups version 2:

io (since Linux 4.5)
This is the successor of the version 1 *blkio* controller.

memory (since Linux 4.5)
This is the successor of the version 1 *memory* controller.

pids (since Linux 4.5)
This is the same as the version 1 *pids* controller.

perf_event (since Linux 4.11)
This is the same as the version 1 *perf_event* controller.

rdma (since Linux 4.11)
This is the same as the version 1 *rdma* controller.

cpu (since Linux 4.15)
This is the successor to the version 1 *cpu* and *cpuacct* controllers.

Cgroups v2 subtree control

Each cgroup in the v2 hierarchy contains the following two files:

cgroup.controllers
This read-only file exposes a list of the controllers that are *available* in this cgroup. The contents of this file match the contents of the *cgroup.subtree_control* file in the parent cgroup.

cgroup.subtree_control
This is a list of controllers that are *active* (*enabled*) in the cgroup. The set of controllers in this file is a subset of the set in the *cgroup.controllers* of this cgroup. The set of active controllers is modified by writing strings to this file containing space-delimited controller names, each preceded by '+' (to enable a controller) or '−' (to disable a controller), as in the following example:

```
echo '+pids -memory' > x/y/cgroup.subtree_control
```

An attempt to enable a controller that is not present in *cgroup.controllers* leads to an **ENOENT** error when writing to the *cgroup.subtree_control* file.

Because the list of controllers in *cgroup.subtree_control* is a subset of those *cgroup.controllers*, a controller that has been disabled in one cgroup in the hierarchy can never be re-enabled in the subtree below that cgroup.

A cgroup's *cgroup.subtree_control* file determines the set of controllers that are exercised in the *child* cgroups. When a controller (e.g., *pids*) is present in the *cgroup.subtree_control* file of a parent cgroup, then the corresponding controller-interface files (e.g., *pids.max*) are automatically created in the children of that cgroup and can be used to exert resource control in the child cgroups.

Cgroups v2 "no internal processes" rule

Cgroups v2 enforces a so-called "no internal processes" rule. Roughly speaking, this rule means that, with the exception of the root cgroup, processes may reside only in leaf nodes (cgroups that do not themselves contain child cgroups). This avoids the need to decide how to partition resources between processes which are members of cgroup A and processes in child cgroups of A.

For instance, if cgroup */cg1/cg2* exists, then a process may reside in */cg1/cg2*, but not in */cg1*. This is to avoid an ambiguity in cgroups v1 with respect to the delegation of resources between processes in */cg1* and its child cgroups. The recommended approach in cgroups v2 is to create a subdirectory called *leaf* for any nonleaf cgroup which should contain processes, but no child cgroups. Thus, processes which previously would have gone into */cg1* would now go into */cg1/leaf*. This has the advantage of making explicit the relationship between processes in */cg1/leaf* and */cg1*'s other children.

The "no internal processes" rule is in fact more subtle than stated above. More precisely, the rule is that a (nonroot) cgroup can't both (1) have member processes, and (2) distribute resources into child cgroups—that is, have a nonempty *cgroup.subtree_control* file. Thus, it *is* possible for a cgroup to have both member processes and child cgroups, but before controllers can be enabled for that cgroup, the member processes must be moved out of the cgroup (e.g., perhaps into the child cgroups).

With the Linux 4.14 addition of "thread mode" (described below), the "no internal processes" rule has been relaxed in some cases.

Cgroups v2 cgroup.events file

With cgroups v2, a new mechanism is provided to obtain notification about when a cgroup becomes empty. The cgroups v1 *release_agent* and *notify_on_release* files are removed, and replaced by a new, more general-purpose file, *cgroup.events*. This read-only file contains key-value pairs (delimited by newline characters, with the key and value separated by spaces) that identify events or state for a cgroup. Currently, only one key appears in this file, *populated*, which has either the value 0, meaning that the cgroup (and its descendants) contain no (nonzombie) processes, or 1, meaning that the cgroup contains member processes.

The *cgroup.events* file can be monitored, in order to receive notification when a cgroup transitions between the populated and unpopulated states (or vice versa). When monitoring this file using **inotify**(7), transitions generate **IN_MODIFY** events, and when monitoring the file using **poll**(2), transitions cause the bits **POLLPRI** and **POLLERR** to be returned in the *revents* field.

The cgroups v2 release-notification mechanism provided by the *populated* field of the *cgroup.events* file offers at least two advantages over the cgroups v1 *release_agent* mechanism. First, it allows for cheaper notification, since a single process can monitor multiple *cgroup.events* files. By contrast, the cgroups v1 mechanism requires the creation of a process for each notification. Second, notification can be delegated to a process that lives inside a container associated with the newly empty cgroup.

Cgroups v2 cgroup.stat file

Each cgroup in the v2 hierarchy contains a read-only *cgroup.stat* file (first introduced in Linux 4.14) that consists of lines containing key-value pairs. The following keys currently appear in this file:

nr_descendants

This is the total number of visible (i.e., living) descendant cgroups underneath this cgroup.

nr_dying_descendants

This is the total number of dying descendant cgroups underneath this cgroup. A cgroup enters the dying state after being deleted. It remains in that state for an undefined period (which will depend on system load) while resources are freed before the cgroup is destroyed. Note that the presence of some cgroups in the dying state is normal, and is not indicative of any problem.

A process can't be made a member of a dying cgroup, and a dying cgroup can't be brought back to life.

Limiting the number of descendant cgroups

Each cgroup in the v2 hierarchy contains the following files, which can be used to view and set limits on the number of descendant cgroups under that cgroup:

cgroup.max.depth (since Linux 4.14)

This file defines a limit on the depth of nesting of descendant cgroups. A value of 0 in this file means that no descendant cgroups can be created. An attempt to create a descendant whose nesting level exceeds the limit fails (*mkdir*(2) fails with the error **EAGAIN**).

Writing the string *"max"* to this file means that no limit is imposed. The default value in this file is *"max"*.

cgroup.max.descendants (since Linux 4.14)

This file defines a limit on the number of live descendant cgroups that this cgroup may have. An attempt to create more descendants than allowed by the limit fails (*mkdir*(2) fails with the error **EAGAIN**).

Writing the string *"max"* to this file means that no limit is imposed. The default value in this file is *"max"*.

CGROUPS DELEGATION: DELEGATING A HIERARCHY TO A LESS PRIVILEGED USER

In the context of cgroups, delegation means passing management of some subtree of the cgroup hierarchy to a nonprivileged user. Cgroups v1 provides support for delegation based on file permissions in the cgroup hierarchy but with less strict containment rules than v2 (as noted below). Cgroups v2 supports delegation with containment by explicit design. The focus of the discussion in this section is on delegation in cgroups v2, with some differences for cgroups v1 noted along the way.

Some terminology is required in order to describe delegation. A *delegater* is a privileged user (i.e., root) who owns a parent cgroup. A *delegatee* is a nonprivileged user who will be granted the permissions needed to manage some subhierarchy under that parent cgroup, known as the *delegated subtree*.

To perform delegation, the delegater makes certain directories and files writable by the delegatee, typically by changing the ownership of the objects to be the user ID of the delegatee. Assuming that we want to delegate the hierarchy rooted at (say) */dlgt_grp* and that there are not yet any child cgroups under that cgroup, the ownership of the following is changed to the user ID of the delegatee:

/dlgt_grp

Changing the ownership of the root of the subtree means that any new cgroups created under the subtree (and the files they contain) will also be owned by the delegatee.

/dlgt_grp/cgroup.procs

Changing the ownership of this file means that the delegatee can move processes into the root of the delegated subtree.

/dlgt_grp/cgroup.subtree_control (cgroups v2 only)

Changing the ownership of this file means that that the delegatee can enable controllers (that are present in */dlgt_grp/cgroup.controllers*) in order to further redistribute resources at lower levels in the subtree. (As an alternative to changing the ownership of this file, the delegater might instead add selected controllers to this file.)

/dlgt_grp/cgroup.threads (cgroups v2 only)

Changing the ownership of this file is necessary if a threaded subtree is being delegated (see the description of "thread mode", below). This permits the delegatee to write thread IDs to the file. (The ownership of this file can also be changed when delegating a domain subtree, but currently this serves no purpose, since, as described below, it is not possible to move a thread between domain cgroups by writing its thread ID to the *cgroup.threads* file.)

In cgroups v1, the corresponding file that should instead be delegated is the *tasks* file.

The delegater should *not* change the ownership of any of the controller interfaces files (e.g., *pids.max*, *memory.high*) in *dlgt_grp*. Those files are used from the next level above the delegated subtree in order to distribute resources into the subtree, and the delegatee should not have permission to change the resources that are distributed into the delegated subtree.

See also the discussion of the */sys/kernel/cgroup/delegate* file in NOTES for information about further delegatable files in cgroups v2.

After the aforementioned steps have been performed, the delegatee can create child cgroups within the delegated subtree (the cgroup subdirectories and the files they contain will be owned by the delegatee) and move processes between cgroups in the subtree. If some controllers are present in *dlgt_grp/cgroup.subtree_control*, or the ownership of that file was passed to the delegatee, the delegatee can also control the further redistribution of the corresponding resources into the delegated subtree.

Cgroups v2 delegation: nsdelegate and cgroup namespaces

Starting with Linux 4.13, there is a second way to perform cgroup delegation in the cgroups v2 hierarchy. This is done by mounting or remounting the cgroup v2 filesystem with the *nsdelegate* mount option. For example, if the cgroup v2 filesystem has already been mounted, we can remount it with the *nsdelegate* option as follows:

```
mount -t cgroup2 -o remount,nsdelegate \
                 none /sys/fs/cgroup/unified
```

The effect of this mount option is to cause cgroup namespaces to automatically become delegation boundaries. More specifically, the following restrictions apply for processes inside the cgroup namespace:

- Writes to controller interface files in the root directory of the namespace will fail with the error **EPERM**. Processes inside the cgroup namespace can still write to delegatable files in the root directory of the cgroup namespace such as *cgroup.procs* and *cgroup.subtree_control*, and can create subhierarchy underneath the root directory.

- Attempts to migrate processes across the namespace boundary are denied (with the error **ENOENT**). Processes inside the cgroup namespace can still (subject to the containment rules described below) move processes between cgroups *within* the subhierarchy under the namespace root.

The ability to define cgroup namespaces as delegation boundaries makes cgroup namespaces more useful. To understand why, suppose that we already have one cgroup hierarchy that has been delegated to a nonprivileged user, *cecilia*, using the older delegation technique described above. Suppose further that *cecilia* wanted to further delegate a subhierarchy under the existing delegated hierarchy. (For example, the delegated hierarchy might be associated with an unprivileged container run by *cecilia*.) Even if a cgroup namespace was employed, because both hierarchies are owned by the unprivileged user *cecilia*, the following illegitimate actions could be performed:

- A process in the inferior hierarchy could change the resource controller settings in the root directory of that hierarchy. (These resource controller settings are intended to allow control to be exercised from the *parent* cgroup; a process inside the child cgroup should not be allowed to modify them.)

- A process inside the inferior hierarchy could move processes into and out of the inferior hierarchy if the cgroups in the superior hierarchy were somehow visible.

Employing the *nsdelegate* mount option prevents both of these possibilities.

The *nsdelegate* mount option only has an effect when performed in the initial mount namespace; in other mount namespaces, the option is silently ignored.

Note: On some systems, **systemd**(1) automatically mounts the cgroup v2 filesystem. In order to experiment with the *nsdelegate* operation, it may be useful to boot the kernel with the following command-line options:

```
cgroup_no_v1=all systemd.legacy_systemd_cgroup_controller
```

These options cause the kernel to boot with the cgroups v1 controllers disabled (meaning that the controllers are available in the v2 hierarchy), and tells **systemd**(1) not to mount and use the cgroup v2 hierarchy, so that the v2 hierarchy can be manually mounted with the desired options after boot-up.

Cgroup delegation containment rules

Some delegation *containment rules* ensure that the delegatee can move processes between cgroups within the delegated subtree, but can't move processes from outside the delegated subtree into the subtree or vice versa. A nonprivileged process (i.e., the delegatee) can write the PID of a "target" process into a *cgroup.procs* file only if all of the following are true:

- The writer has write permission on the *cgroup.procs* file in the destination cgroup.

- The writer has write permission on the *cgroup.procs* file in the nearest common ancestor of the source and destination cgroups. Note that in some cases, the nearest common ancestor may be the source or destination cgroup itself. This requirement is not enforced for cgroups v1 hierarchies, with the consequence that containment in v1 is less strict than in v2. (For example, in cgroups v1 the user that owns two distinct delegated subhierarchies can move a process between the hierarchies.)

- If the cgroup v2 filesystem was mounted with the *nsdelegate* option, the writer must be able to see the source and destination cgroups from its cgroup namespace.
- In cgroups v1: the effective UID of the writer (i.e., the delegatee) matches the real user ID or the saved set-user-ID of the target process. Before Linux 4.11, this requirement also applied in cgroups v2 (This was a historical requirement inherited from cgroups v1 that was later deemed unnecessary, since the other rules suffice for containment in cgroups v2.)

Note: one consequence of these delegation containment rules is that the unprivileged delegatee can't place the first process into the delegated subtree; instead, the delegater must place the first process (a process owned by the delegatee) into the delegated subtree.

CGROUPS VERSION 2 THREAD MODE

Among the restrictions imposed by cgroups v2 that were not present in cgroups v1 are the following:

- *No thread-granularity control*: all of the threads of a process must be in the same cgroup.
- *No internal processes*: a cgroup can't both have member processes and exercise controllers on child cgroups.

Both of these restrictions were added because the lack of these restrictions had caused problems in cgroups v1. In particular, the cgroups v1 ability to allow thread-level granularity for cgroup membership made no sense for some controllers. (A notable example was the *memory* controller: since threads share an address space, it made no sense to split threads across different *memory* cgroups.)

Notwithstanding the initial design decision in cgroups v2, there were use cases for certain controllers, notably the *cpu* controller, for which thread-level granularity of control was meaningful and useful. To accommodate such use cases, Linux 4.14 added *thread mode* for cgroups v2.

Thread mode allows the following:

- The creation of *threaded subtrees* in which the threads of a process may be spread across cgroups inside the tree. (A threaded subtree may contain multiple multithreaded processes.)
- The concept of *threaded controllers*, which can distribute resources across the cgroups in a threaded subtree.
- A relaxation of the "no internal processes rule", so that, within a threaded subtree, a cgroup can both contain member threads and exercise resource control over child cgroups.

With the addition of thread mode, each nonroot cgroup now contains a new file, *cgroup.type*, that exposes, and in some circumstances can be used to change, the "type" of a cgroup. This file contains one of the following type values:

domain This is a normal v2 cgroup that provides process-granularity control. If a process is a member of this cgroup, then all threads of the process are (by definition) in the same cgroup. This is the default cgroup type, and provides the same behavior that was provided for cgroups in the initial cgroups v2 implementation.

threaded
This cgroup is a member of a threaded subtree. Threads can be added to this cgroup, and controllers can be enabled for the cgroup.

domain threaded
This is a domain cgroup that serves as the root of a threaded subtree. This cgroup type is also known as "threaded root".

domain invalid
This is a cgroup inside a threaded subtree that is in an "invalid" state. Processes can't be added to the cgroup, and controllers can't be enabled for the cgroup. The only thing that can be done with this cgroup (other than deleting it) is to convert it to a *threaded* cgroup by writing the string *"threaded"* to the *cgroup.type* file.

The rationale for the existence of this "interim" type during the creation of a threaded subtree (rather than the kernel simply immediately converting all cgroups under the threaded root to the type *threaded*) is to allow for possible future extensions to the thread mode model

Threaded versus domain controllers

With the addition of threads mode, cgroups v2 now distinguishes two types of resource controllers:

- *Threaded* controllers: these controllers support thread-granularity for resource control and can be enabled inside threaded subtrees, with the result that the corresponding controller-interface files appear inside the cgroups in the threaded subtree. As at Linux 4.19, the following controllers are threaded: *cpu*, *perf_event*, and *pids*.
- *Domain* controllers: these controllers support only process granularity for resource control. From the perspective of a domain controller, all threads of a process are always in the same cgroup. Domain controllers can't be enabled inside a threaded subtree.

Creating a threaded subtree

There are two pathways that lead to the creation of a threaded subtree. The first pathway proceeds as follows:

1. We write the string *"threaded"* to the *cgroup.type* file of a cgroup *y/z* that currently has the type *domain*. This has the following effects:
   - The type of the cgroup *y/z* becomes *threaded*.
   - The type of the parent cgroup, *y*, becomes *domain threaded*. The parent cgroup is the root of a threaded subtree (also known as the "threaded root").
   - All other cgroups under *y* that were not already of type *threaded* (because they were inside already existing threaded subtrees under the new threaded root) are converted to type *domain invalid*. Any subsequently created cgroups under *y* will also have the type *domain invalid*.
2. We write the string *"threaded"* to each of the *domain invalid* cgroups under *y*, in order to convert them to the type *threaded*. As a consequence of this step, all threads under the threaded root now have the type *threaded* and the threaded subtree is now fully usable. The requirement to write *"threaded"* to each of these cgroups is somewhat cumbersome, but allows for possible future extensions to the thread-mode model.

The second way of creating a threaded subtree is as follows:

1. In an existing cgroup, *z*, that currently has the type *domain*, we (1) enable one or more threaded controllers and (2) make a process a member of *z*. (These two steps can be done in either order.) This has the following consequences:
   - The type of *z* becomes *domain threaded*.
   - All of the descendant cgroups of *x* that were not already of type *threaded* are converted to type *domain invalid*.
2. As before, we make the threaded subtree usable by writing the string *"threaded"* to each of the *domain invalid* cgroups under *y*, in order to convert them to the type *threaded*.

One of the consequences of the above pathways to creating a threaded subtree is that the threaded root cgroup can be a parent only to *threaded* (and *domain invalid*) cgroups. The threaded root cgroup can't be a parent of a *domain* cgroups, and a *threaded* cgroup can't have a sibling that is a *domain* cgroup.

Using a threaded subtree

Within a threaded subtree, threaded controllers can be enabled in each subgroup whose type has been changed to *threaded*; upon doing so, the corresponding controller interface files appear in the children of that cgroup.

A process can be moved into a threaded subtree by writing its PID to the *cgroup.procs* file in one of the cgroups inside the tree. This has the effect of making all of the threads in the process members of the corresponding cgroup and makes the process a member of the threaded subtree. The threads of the process can then be spread across the threaded subtree by writing their thread IDs (see **gettid**(2)) to the *cgroup.threads* files in different cgroups inside the subtree. The threads of a process must all reside in the same threaded subtree.

As with writing to *cgroup.procs*, some containment rules apply when writing to the *cgroup.threads* file:

- The writer must have write permission on the cgroup.threads file in the destination cgroup.
- The writer must have write permission on the *cgroup.procs* file in the common ancestor of the source and destination cgroups. (In some cases, the common ancestor may be the source or destination cgroup itself.)
- The source and destination cgroups must be in the same threaded subtree. (Outside a threaded subtree, an attempt to move a thread by writing its thread ID to the *cgroup.threads* file in a different *domain* cgroup fails with the error **EOPNOTSUPP**.)

The *cgroup.threads* file is present in each cgroup (including *domain* cgroups) and can be read in order to discover the set of threads that is present in the cgroup. The set of thread IDs obtained when reading this file is not guaranteed to be ordered or free of duplicates.

The *cgroup.procs* file in the threaded root shows the PIDs of all processes that are members of the threaded subtree. The *cgroup.procs* files in the other cgroups in the subtree are not readable.

Domain controllers can't be enabled in a threaded subtree; no controller-interface files appear inside the cgroups underneath the threaded root. From the point of view of a domain controller, threaded subtrees are invisible: a multithreaded process inside a threaded subtree appears to a domain controller as a process that resides in the threaded root cgroup.

Within a threaded subtree, the "no internal processes" rule does not apply: a cgroup can both contain member processes (or thread) and exercise controllers on child cgroups.

Rules for writing to cgroup.type and creating threaded subtrees

A number of rules apply when writing to the *cgroup.type* file:

- Only the string *"threaded"* may be written. In other words, the only explicit transition that is possible is to convert a *domain* cgroup to type *threaded*.
- The effect of writing *"threaded"* depends on the current value in *cgroup.type*, as follows:
  - *domain* or *domain threaded*: start the creation of a threaded subtree (whose root is the parent of this cgroup) via the first of the pathways described above;
  - *domain invalid*: convert this cgroup (which is inside a threaded subtree) to a usable (i.e., *threaded*) state;
  - *threaded*: no effect (a "no-op").
- We can't write to a *cgroup.type* file if the parent's type is *domain invalid*. In other words, the cgroups of a threaded subtree must be converted to the *threaded* state in a top-down manner.

There are also some constraints that must be satisfied in order to create a threaded subtree rooted at the cgroup *x*:

- There can be no member processes in the descendant cgroups of *x*. (The cgroup *x* can itself have member processes.)
- No domain controllers may be enabled in *x*'s *cgroup.subtree_control* file.

If any of the above constraints is violated, then an attempt to write *"threaded"* to a *cgroup.type* file fails with the error **ENOTSUP**.

The "domain threaded" cgroup type

According to the pathways described above, the type of a cgroup can change to *domain threaded* in either of the following cases:

- The string *"threaded"* is written to a child cgroup.
- A threaded controller is enabled inside the cgroup and a process is made a member of the cgroup.

A *domain threaded* cgroup, *x*, can revert to the type *domain* if the above conditions no longer hold true—that is, if all *threaded* child cgroups of *x* are removed and either *x* no longer has threaded controllers enabled or no longer has member processes.

When a *domain threaded* cgroup *x* reverts to the type *domain*:

* All *domain invalid* descendants of *x* that are not in lower-level threaded subtrees revert to the type *domain*.

* The root cgroups in any lower-level threaded subtrees revert to the type *domain threaded*.

Exceptions for the root cgroup

The root cgroup of the v2 hierarchy is treated exceptionally: it can be the parent of both *domain* and *threaded* cgroups. If the string *"threaded"* is written to the *cgroup.type* file of one of the children of the root cgroup, then

* The type of that cgroup becomes *threaded*.

* The type of any descendants of that cgroup that are not part of lower-level threaded subtrees changes to *domain invalid*.

Note that in this case, there is no cgroup whose type becomes *domain threaded*. (Notionally, the root cgroup can be considered as the threaded root for the cgroup whose type was changed to *threaded*.)

The aim of this exceptional treatment for the root cgroup is to allow a threaded cgroup that employs the *cpu* controller to be placed as high as possible in the hierarchy, so as to minimize the (small) cost of traversing the cgroup hierarchy.

The cgroups v2 "cpu" controller and realtime threads

As at Linux 4.19, the cgroups v2 *cpu* controller does not support control of realtime threads (specifically threads scheduled under any of the policies **SCHED_FIFO**, **SCHED_RR**, described **SCHED_DEADLINE**; see **sched**(7)). Therefore, the *cpu* controller can be enabled in the root cgroup only if all realtime threads are in the root cgroup. (If there are realtime threads in nonroot cgroups, then a **write**(2) of the string *"+cpu"* to the *cgroup.subtree_control* file fails with the error **EINVAL**.)

On some systems, **systemd**(1) places certain realtime threads in nonroot cgroups in the v2 hierarchy. On such systems, these threads must first be moved to the root cgroup before the *cpu* controller can be enabled.

ERRORS

The following errors can occur for **mount**(2):

EBUSY

An attempt to mount a cgroup version 1 filesystem specified neither the *name=* option (to mount a named hierarchy) nor a controller name (or *all*).

NOTES

A child process created via **fork**(2) inherits its parent's cgroup memberships. A process's cgroup memberships are preserved across **execve**(2).

/proc files

/proc/cgroups (since Linux 2.6.24)

This file contains information about the controllers that are compiled into the kernel. An example of the contents of this file (reformatted for readability) is the following:

```
#subsys_name    hierarchy      num_cgroups    enabled
cpuset          4              1              1
cpu             8              1              1
cpuacct         8              1              1
blkio           6              1              1
memory          3              1              1
devices         10             84             1
freezer         7              1              1
net_cls         9              1              1
```

```
perf_event       5              1              1
net_prio         9              1              1
hugetlb          0              1              0
pids             2              1              1
```

The fields in this file are, from left to right:

1. The name of the controller.

2. The unique ID of the cgroup hierarchy on which this controller is mounted. If multiple cgroups v1 controllers are bound to the same hierarchy, then each will show the same hierarchy ID in this field. The value in this field will be 0 if:

   a) the controller is not mounted on a cgroups v1 hierarchy;

   b) the controller is bound to the cgroups v2 single unified hierarchy; or

   c) the controller is disabled (see below).

3. The number of control groups in this hierarchy using this controller.

4. This field contains the value 1 if this controller is enabled, or 0 if it has been disabled (via the *cgroup_disable* kernel command-line boot parameter).

/proc/[pid]/cgroup (since Linux 2.6.24)

This file describes control groups to which the process with the corresponding PID belongs. The displayed information differs for cgroups version 1 and version 2 hierarchies.

For each cgroup hierarchy of which the process is a member, there is one entry containing three colon-separated fields:

```
hierarchy-ID:controller-list:cgroup-path
```

For example:

```
5:cpuacct,cpu,cpuset:/daemons
```

The colon-separated fields are, from left to right:

1. For cgroups version 1 hierarchies, this field contains a unique hierarchy ID number that can be matched to a hierarchy ID in */proc/cgroups*. For the cgroups version 2 hierarchy, this field contains the value 0.

2. For cgroups version 1 hierarchies, this field contains a comma-separated list of the controllers bound to the hierarchy. For the cgroups version 2 hierarchy, this field is empty.

3. This field contains the pathname of the control group in the hierarchy to which the process belongs. This pathname is relative to the mount point of the hierarchy.

/sys/kernel/cgroup files

/sys/kernel/cgroup/delegate (since Linux 4.15)

This file exports a list of the cgroups v2 files (one per line) that are delegatable (i.e., whose ownership should be changed to the user ID of the delegatee). In the future, the set of delegatable files may change or grow, and this file provides a way for the kernel to inform user-space applications of which files must be delegated. As at Linux 4.15, one sees the following when inspecting this file:

```
$ cat /sys/kernel/cgroup/delegate
cgroup.procs
cgroup.subtree_control
cgroup.threads
```

/sys/kernel/cgroup/features (since Linux 4.15)

Over time, the set of cgroups v2 features that are provided by the kernel may change or grow, or some features may not be enabled by default. This file provides a way for user-space applications to discover what features the running kernel supports and has enabled. Features are listed one per line:

```
$ cat /sys/kernel/cgroup/features
nsdelegate
```

The entries that can appear in this file are:

nsdelegate (since Linux 4.15)
The kernel supports the *nsdelegate* mount option.

SEE ALSO

prlimit(1), **systemd**(1), **systemd-cgls**(1), **systemd-cgtop**(1), **clone**(2), **ioprio_set**(2), **perf_event_open**(2), **setrlimit**(2), **cgroup_namespaces**(7), **cpuset**(7), **namespaces**(7), **sched**(7), **user_namespaces**(7)

COLOPHON

This page is part of release 5.02 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.