Control Groups

Written by Paul Menage <menage@google.com> based on Documentation/admin-guide/cgroup-v1/cpusets.rst

Modified by Paul Jackson <pj@sgi.com>

Modified by Christoph Lameter <cl@linux.com>

1. Control Groups

1.1 What are cgroups ?

Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.

Definitions:

A *cgroup* associates a set of tasks with a set of parameters for one or more subsystems.

A *subsystem* is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem.

A *hierarchy* is a set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the cgroup virtual filesystem associated with it.

At any one time there may be multiple active hierarchies of task cgroups. Each hierarchy is a partition of all tasks in the system.

User-level code may create and destroy cgroups by name in an instance of the cgroup virtual file system, specify and query to which cgroup a task is assigned, and list the task PIDs assigned to a cgroup. Those creations and assignments only affect the hierarchy associated with that instance of the cgroup file system.

On their own, the only use for cgroups is for simple job tracking. The intention is that other subsystems hook into the generic cgroup support to provide new attributes for cgroups, such as accounting/limiting the resources which processes in a cgroup can access. For example, cpusets (see Documentation/admin-guide/cgroup-v1/cpusets.rst) allow you to associate a set of CPUs and a set of memory nodes with the tasks in each cgroup.

1.2 Why are cgroups needed ?

There are multiple efforts to provide process aggregations in the Linux kernel, mainly for resource-tracking purposes. Such efforts include cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server namespaces. These all require the basic notion of a grouping/partitioning of processes, with newly forked processes ending up in the same group (cgroup) as their parent process.

The kernel cgroup patch provides the minimum essential kernel mechanisms required to efficiently implement such groups. It has minimal impact on the system fast paths, and provides hooks for specific subsystems such as cpusets to provide additional behaviour as desired.

Multiple hierarchy support is provided to allow for situations where the division of tasks into cgroups is distinctly different for different subsystems - having parallel hierarchies allows each hierarchy to be a natural division of tasks, without having to handle complex combinations of tasks that would be present if several unrelated subsystems needed to be forced into the same tree of cgroups.

At one extreme, each resource controller or subsystem could be in a separate hierarchy; at the other extreme, all subsystems would be attached to the same hierarchy.

As an example of a scenario (originally proposed by vatsa@in.ibm.com) that can benefit from multiple hierarchies, consider a large university server with various users - students, professors, system tasks etc. The resource planning for this server could be along the following lines:

```
CPU :          "Top cpuset"
                 /       \
         CPUSet1         CPUSet2
            |               |
         (Professors)    (Students)

         In addition (system tasks) are attached to topcpuset (so
         that they can run anywhere) with a limit of 20%

Memory : Professors (50%), Students (30%), system (20%)
```

```
Disk : Professors (50%), Students (30%), system (20%)

Network : WWW browsing (20%), Network File System (60%), others (20%)
                          / \
         Professors (15%)  students (5%)
```

Browsers like Firefox/Lynx go into the WWW network class, while (k)nfsd goes into the NFS network class.

At the same time Firefox/Lynx will share an appropriate CPU/Memory class depending on who launched it (prof/student).

With the ability to classify tasks differently for different resources (by putting those resource subsystems in different hierarchies), the admin can easily set up a script which receives exec notifications and depending on who is launching the browser he can:

```
# echo browser_pid > /sys/fs/cgroup/<restype>/<userclass>/tasks
```

With only a single hierarchy, he now would potentially have to create a separate cgroup for every browser launched and associate it with appropriate network and other resource class. This may lead to proliferation of such cgroups.

Also let's say that the administrator would like to give enhanced network access temporarily to a student's browser (since it is night and the user wants to do online gaming :)) OR give one of the student's simulation apps enhanced CPU power.

With ability to write PIDs directly to resource classes, it's just a matter of:

```
# echo pid > /sys/fs/cgroup/network/<new_class>/tasks
(after some time)
# echo pid > /sys/fs/cgroup/network/<orig_class>/tasks
```

Without this ability, the administrator would have to split the cgroup into multiple separate ones and then associate the new cgroups with the new resource classes.

1.3 How are cgroups implemented ?

Control Groups extends the kernel as follows:

- Each task in the system has a reference-counted pointer to a css_set.
- A css_set contains a set of reference-counted pointers to cgroup_subsys_state objects, one for each cgroup subsystem registered in the system. There is no direct link from a task to the cgroup of which it's a member in each hierarchy, but this can be determined by following pointers through the cgroup_subsys_state objects. This is because accessing the subsystem state is something that's expected to happen frequently and in performance-critical code, whereas operations that require a task's actual cgroup assignments (in particular, moving between cgroups) are less common. A linked list runs through the cg_list field of each task_struct using the css_set, anchored at css_set->tasks.
- A cgroup hierarchy filesystem can be mounted for browsing and manipulation from user space.
- You can list all the tasks (by PID) attached to any cgroup.

The implementation of cgroups requires a few, simple hooks into the rest of the kernel, none in performance-critical paths:

- in init/main.c, to initialize the root cgroups and initial css_set at system boot.
- in fork and exit, to attach and detach a task from its css_set.

In addition, a new file system of type "cgroup" may be mounted, to enable browsing and modifying the cgroups presently known to the kernel. When mounting a cgroup hierarchy, you may specify a comma-separated list of subsystems to mount as the filesystem mount options. By default, mounting the cgroup filesystem attempts to mount a hierarchy containing all registered subsystems.

If an active hierarchy with exactly the same set of subsystems already exists, it will be reused for the new mount. If no existing hierarchy matches, and any of the requested subsystems are in use in an existing hierarchy, the mount will fail with -EBUSY. Otherwise, a new hierarchy is activated, associated with the requested subsystems.

It's not currently possible to bind a new subsystem to an active cgroup hierarchy, or to unbind a subsystem from an active cgroup hierarchy. This may be possible in future, but is fraught with nasty error-recovery issues.

When a cgroup filesystem is unmounted, if there are any child cgroups created below the top-level cgroup, that hierarchy will remain active even though unmounted; if there are no child cgroups then the hierarchy will be deactivated.

No new system calls are added for cgroups - all support for querying and modifying cgroups is via this cgroup file system.

Each task under /proc has an added file named 'cgroup' displaying, for each active hierarchy, the subsystem names and the cgroup name as the path relative to the root of the cgroup file system.

Each cgroup is represented by a directory in the cgroup file system containing the following files describing that cgroup:

- tasks: list of tasks (by PID) attached to that cgroup. This list is not guaranteed to be sorted. Writing a thread ID into this file moves the thread into this cgroup.
- cgroup.procs: list of thread group IDs in the cgroup. This list is not guaranteed to be sorted or free of duplicate TGIDs, and userspace should sort/uniquify the list if this property is required. Writing a thread group ID into this file moves all threads in that group into this cgroup.

- notify_on_release flag: run the release agent on exit?
- release_agent: the path to use for release notifications (this file exists in the top cgroup only)

Other subsystems such as cpusets may add additional files in each cgroup dir.

New cgroups are created using the mkdir system call or shell command. The properties of a cgroup, such as its flags, are modified by writing to the appropriate file in that cgroups directory, as listed above.

The named hierarchical structure of nested cgroups allows partitioning a large system into nested, dynamically changeable, "soft-partitions".

The attachment of each task, automatically inherited at fork by any children of that task, to a cgroup allows organizing the work load on a system into related sets of tasks. A task may be re-attached to any other cgroup, if allowed by the permissions on the necessary cgroup file system directories.

When a task is moved from one cgroup to another, it gets a new css_set pointer - if there's an already existing css_set with the desired collection of cgroups then that group is reused, otherwise a new css_set is allocated. The appropriate existing css_set is located by looking into a hash table.

To allow access from a cgroup to the css_sets (and hence tasks) that comprise it, a set of cg_cgroup_link objects form a lattice; each cg_cgroup_link is linked into a list of cg_cgroup_links for a single cgroup on its cgrp_link_list field, and a list of cg_cgroup_links for a single css_set on its cg_link_list.

Thus the set of tasks in a cgroup can be listed by iterating over each css_set that references the cgroup, and sub-iterating over each css_set's task set.

The use of a Linux virtual file system (vfs) to represent the cgroup hierarchy provides for a familiar permission and name space for cgroups, with a minimum of additional kernel code.

1.4 What does notify_on_release do ?

If the notify_on_release flag is enabled (1) in a cgroup, then whenever the last task in the cgroup leaves (exits or attaches to some other cgroup) and the last child cgroup of that cgroup is removed, then the kernel runs the command specified by the contents of the "release_agent" file in that hierarchy's root directory, supplying the pathname (relative to the mount point of the cgroup file system) of the abandoned cgroup. This enables automatic removal of abandoned cgroups. The default value of notify_on_release in the root cgroup at system boot is disabled (0). The default value of other cgroups at creation is the current value of their parents' notify_on_release settings. The default value of a cgroup hierarchy's release_agent path is empty.

1.5 What does clone_children do ?

This flag only affects the cpuset controller. If the clone_children flag is enabled (1) in a cgroup, a new cpuset cgroup will copy its configuration from the parent during initialization.

1.6 How do I use cgroups ?

To start a new job that is to be contained within a cgroup, using the "cpuset" cgroup subsystem, the steps are something like:

```
1) mount -t tmpfs cgroup_root /sys/fs/cgroup
2) mkdir /sys/fs/cgroup/cpuset
3) mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset
4) Create the new cgroup by doing mkdir's and write's (or echo's) in
   the /sys/fs/cgroup/cpuset virtual file system.
5) Start a task that will be the "founding father" of the new job.
6) Attach that task to the new cgroup by writing its PID to the
   /sys/fs/cgroup/cpuset tasks file for that cgroup.
7) fork, exec or clone the job tasks from this founding father task.
```

For example, the following sequence of commands will setup a cgroup named "Charlie", containing just CPUs 2 and 3, and Memory Node 1, and then start a subshell 'sh' in that cgroup:

```
mount -t tmpfs cgroup_root /sys/fs/cgroup
mkdir /sys/fs/cgroup/cpuset
mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
cd /sys/fs/cgroup/cpuset
mkdir Charlie
cd Charlie
/bin/echo 2-3 > cpuset.cpus
/bin/echo 1 > cpuset.mems
/bin/echo $$ > tasks
sh
# The subshell 'sh' is now running in cgroup Charlie
# The next line should display '/Charlie'
cat /proc/self/cgroup
```

2. Usage Examples and Syntax

2.1 Basic Usage

Creating, modifying, using cgroups can be done through the cgroup virtual filesystem.

To mount a cgroup hierarchy with all available subsystems, type:

```
# mount -t cgroup xxx /sys/fs/cgroup
```

The "xxx" is not interpreted by the cgroup code, but will appear in /proc/mounts so may be any useful identifying string that you like.

Note: Some subsystems do not work without some user input first. For instance, if cpusets are enabled the user will have to populate the cpus and mems files for each new cgroup created before that group can be used.

As explained in section *1.2 Why are cgroups needed?* you should create different hierarchies of cgroups for each single resource or group of resources you want to control. Therefore, you should mount a tmpfs on /sys/fs/cgroup and create directories for each cgroup resource or resource group:

```
# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/rg1
```

To mount a cgroup hierarchy with just the cpuset and memory subsystems, type:

```
# mount -t cgroup -o cpuset,memory hier1 /sys/fs/cgroup/rg1
```

While remounting cgroups is currently supported, it is not recommend to use it. Remounting allows changing bound subsystems and release_agent. Rebinding is hardly useful as it only works when the hierarchy is empty and release_agent itself should be replaced with conventional fsnotify. The support for remounting will be removed in the future.

To Specify a hierarchy's release_agent:

```
# mount -t cgroup -o cpuset,release_agent="/sbin/cpuset_release_agent" \
  xxx /sys/fs/cgroup/rg1
```

Note that specifying 'release_agent' more than once will return failure.

Note that changing the set of subsystems is currently only supported when the hierarchy consists of a single (root) cgroup. Supporting the ability to arbitrarily bind/unbind subsystems from an existing cgroup hierarchy is intended to be implemented in the future.

Then under /sys/fs/cgroup/rg1 you can find a tree that corresponds to the tree of the cgroups in the system. For instance, /sys/fs/cgroup/rg1 is the cgroup that holds the whole system.

If you want to change the value of release_agent:

```
# echo "/sbin/new_release_agent" > /sys/fs/cgroup/rg1/release_agent
```

It can also be changed via remount.

If you want to create a new cgroup under /sys/fs/cgroup/rg1:

```
# cd /sys/fs/cgroup/rg1
# mkdir my_cgroup
```

Now you want to do something with this cgroup:

```
 # cd my_cgroup
```

In this directory you can find several files:

```
# ls
cgroup.procs notify_on_release tasks
(plus whatever files added by the attached subsystems)
```

Now attach your shell to this cgroup:

```
# /bin/echo $$ > tasks
```

You can also create cgroups inside your cgroup by using mkdir in this directory:

```
# mkdir my_sub_cs
```

To remove a cgroup, just use rmdir:

```
# rmdir my_sub_cs
```

This will fail if the cgroup is in use (has cgroups inside, or has processes attached, or is held alive by other subsystem-specific reference).

2.2 Attaching processes

```
# /bin/echo PID > tasks
```

Note that it is PID, not PIDs. You can only attach ONE task at a time. If you have several tasks to attach, you have to do it one after another:

```
# /bin/echo PID1 > tasks
# /bin/echo PID2 > tasks
        ...
# /bin/echo PIDn > tasks
```

You can attach the current shell task by echoing 0:

```
# echo 0 > tasks
```

You can use the cgroup.procs file instead of the tasks file to move all threads in a threadgroup at once. Echoing the PID of any task in a threadgroup to cgroup.procs causes all tasks in that threadgroup to be attached to the cgroup. Writing 0 to cgroup.procs moves all tasks in the writing task's threadgroup.

Note: Since every task is always a member of exactly one cgroup in each mounted hierarchy, to remove a task from its current cgroup you must move it into a new cgroup (possibly the root cgroup) by writing to the new cgroup's tasks file.

Note: Due to some restrictions enforced by some cgroup subsystems, moving a process to another cgroup can fail.

2.3 Mounting hierarchies by name

Passing the name=<x> option when mounting a cgroups hierarchy associates the given name with the hierarchy. This can be used when mounting a pre-existing hierarchy, in order to refer to it by name rather than by its set of active subsystems. Each hierarchy is either nameless, or has a unique name.

The name should match [w.-]+

When passing a name=<x> option for a new hierarchy, you need to specify subsystems manually; the legacy behaviour of mounting all subsystems when none are explicitly specified is not supported when you give a subsystem a name.

The name of the subsystem appears as part of the hierarchy description in /proc/mounts and /proc/<pid>/cgroups.

3. Kernel API

3.1 Overview

Each kernel subsystem that wants to hook into the generic cgroup system needs to create a cgroup_subsys object. This contains various methods, which are callbacks from the cgroup system, along with a subsystem ID which will be assigned by the cgroup system.

Other fields in the cgroup_subsys object include:

- subsys_id: a unique array index for the subsystem, indicating which entry in cgroup->subsys[] this subsystem should be managing.
- name: should be initialized to a unique subsystem name. Should be no longer than MAX_CGROUP_TYPE_NAMELEN.
- early_init: indicate if the subsystem needs early initialization at system boot.

Each cgroup object created by the system has an array of pointers, indexed by subsystem ID; this pointer is entirely managed by the subsystem; the generic cgroup code will never touch this pointer.

3.2 Synchronization

There is a global mutex, cgroup_mutex, used by the cgroup system. This should be taken by anything that wants to modify a cgroup. It may also be taken to prevent cgroups from being modified, but more specific locks may be more appropriate in that situation.

See kernel/cgroup.c for more details.

Subsystems can take/release the cgroup_mutex via the functions cgroup_lock()/cgroup_unlock().

Accessing a task's cgroup pointer may be done in the following ways: - while holding cgroup_mutex - while holding the task's alloc_lock (via task_lock()) - inside an rcu_read_lock() section via rcu_dereference()

3.3 Subsystem API

Each subsystem should:

- add an entry in linux/cgroup_subsys.h
- define a cgroup_subsys object called <name>_cgrp_subsys

Each subsystem may export the following methods. The only mandatory methods are css_alloc/free. Any others that are null are presumed to be successful no-ops.

`struct cgroup_subsys_state *css_alloc(struct cgroup *cgrp)` (cgroup_mutex held by caller)

Called to allocate a subsystem state object for a cgroup. The subsystem should allocate its subsystem state object for the passed cgroup, returning a pointer to the new object on success or a ERR_PTR() value. On success, the subsystem pointer should point to a

structure of type cgroup_subsys_state (typically embedded in a larger subsystem-specific object), which will be initialized by the cgroup system. Note that this will be called at initialization to create the root subsystem state for this subsystem; this case can be identified by the passed cgroup object having a NULL parent (since it's the root of the hierarchy) and may be an appropriate place for initialization code.

`int css_online(struct cgroup *cgrp)` (cgroup_mutex held by caller)

Called after @cgrp successfully completed all allocations and made visible to cgroup_for_each_child/descendant_*() iterators. The subsystem may choose to fail creation by returning -errno. This callback can be used to implement reliable state sharing and propagation along the hierarchy. See the comment on cgroup_for_each_descendant_pre() for details.

`void css_offline(struct cgroup *cgrp);` (cgroup_mutex held by caller)

This is the counterpart of css_online() and called iff css_online() has succeeded on @cgrp. This signifies the beginning of the end of @cgrp. @cgrp is being removed and the subsystem should start dropping all references it's holding on @cgrp. When all references are dropped, cgroup removal will proceed to the next step - css_free(). After this callback, @cgrp should be considered dead to the subsystem.

`void css_free(struct cgroup *cgrp)` (cgroup_mutex held by caller)

The cgroup system is about to free @cgrp; the subsystem should free its subsystem state object. By the time this method is called, @cgrp is completely unused; @cgrp->parent is still valid. (Note - can also be called for a newly-created cgroup if an error occurs after this subsystem's create() method has been called for the new cgroup).

`int can_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)` (cgroup_mutex held by caller)

Called prior to moving one or more tasks into a cgroup; if the subsystem returns an error, this will abort the attach operation. @tset contains the tasks to be attached and is guaranteed to have at least one task in it.

If there are multiple tasks in the taskset, then:

- it's guaranteed that all are from the same thread group
- @tset contains all tasks from the thread group whether or not they're switching cgroups
- the first task is the leader

Each @tset entry also contains the task's old cgroup and tasks which aren't switching cgroup can be skipped easily using the cgroup_taskset_for_each() iterator. Note that this isn't called on a fork. If this method returns 0 (success) then this should remain valid while the caller holds cgroup_mutex and it is ensured that either attach() or cancel_attach() will be called in future.

`void css_reset(struct cgroup_subsys_state *css)` (cgroup_mutex held by caller)

An optional operation which should restore @css's configuration to the initial state. This is currently only used on the unified hierarchy when a subsystem is disabled on a cgroup through "cgroup.subtree_control" but should remain enabled because other subsystems depend on it. cgroup core makes such a css invisible by removing the associated interface files and invokes this callback so that the hidden subsystem can return to the initial neutral state. This prevents unexpected resource control from a hidden css and ensures that the configuration is in the initial state when it is made visible again later.

`void cancel_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)` (cgroup_mutex held by caller)

Called when a task attach operation has failed after can_attach() has succeeded. A subsystem whose can_attach() has some side-effects should provide this function, so that the subsystem can implement a rollback. If not, not necessary. This will be called only about subsystems whose can_attach() operation have succeeded. The parameters are identical to can_attach().

`void attach(struct cgroup *cgrp, struct cgroup_taskset *tset)` (cgroup_mutex held by caller)

Called after the task has been attached to the cgroup, to allow any post-attachment activity that requires memory allocations or blocking. The parameters are identical to can_attach().

`void fork(struct task_struct *task)`

Called when a task is forked into a cgroup.

`void exit(struct task_struct *task)`

Called during task exit.

`void free(struct task_struct *task)`

Called when the task_struct is freed.

`void bind(struct cgroup *root)` (cgroup_mutex held by caller)

Called when a cgroup subsystem is rebound to a different hierarchy and root cgroup. Currently this will only involve movement between the default hierarchy (which never has sub-cgroups) and a hierarchy that is being created/destroyed (and hence has no sub-cgroups).

4. Extended attribute usage

cgroup filesystem supports certain types of extended attributes in its directories and files. The current supported types are:

- Trusted (XATTR_TRUSTED)
- Security (XATTR_SECURITY)

Both require CAP_SYS_ADMIN capability to set.

Like in tmpfs, the extended attributes in cgroup filesystem are stored using kernel memory and it's advised to keep the usage at minimum. This is the reason why user defined extended attributes are not supported, since any user can do it and there's no limit in the value size.

The current known users for this feature are SELinux to limit cgroup usage in containers and systemd for assorted meta data like main PID in a cgroup (systemd creates a cgroup per service).

5. Questions

```
Q: what's up with this '/bin/echo' ?
A: bash's builtin 'echo' command does not check calls to write() against
   errors. If you use it in the cgroup file system, you won't be
   able to tell whether a command succeeded or failed.

Q: When I attach processes, only the first of the line gets really attached !
A: We can only return one error code per call to write(). So you should also
   put only ONE PID.
```