CGROUPS

Written by Paul Menage <menage@google.com> based on
Documentation/cgroups/cpusets.txt

Modified by Paul Jackson <pj@sgi.com>
Modified by Christoph Lameter <clameter@sgi.com>

CONTENTS:

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/cgroups/cpusets.txt) allows 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 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
         |                  |
      (Profs)            (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 : Prof (50%), students (30%), system (20%)

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

Browsers like Firefox/Lynx go into the WWW network class, while (k)nfsd go into 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) then 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 > /mnt/<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 approp network and other resource class. This may lead to proliferation of such cgroups.

Also lets 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 students simulation apps enhanced CPU power,

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

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

Without this ability, he 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 tgids 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. This is a read-only file, for now.
- 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, else 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 setting. The default value of
a cgroup hierarchy's release_agent path is empty.

1.5 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) mkdir /dev/cgroup
2) mount -t cgroup -ocpuset cpuset /dev/cgroup
3) Create the new cgroup by doing mkdir's and write's (or echo's) in
   the /dev/cgroup virtual file system.
4) Start a task that will be the "founding father" of the new job.
5) Attach that task to the new cgroup by writing its pid to the
   /dev/cgroup tasks file for that cgroup.
6) 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 cgroup cpuset -ocpuset /dev/cgroup
cd /dev/cgroup
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 the cgroups can be done through the cgroup
virtual filesystem.

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

```
# mount -t cgroup xxx /dev/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.

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

```
# mount -t cgroup -o cpuset,memory hier1 /dev/cgroup
```

To change the set of subsystems bound to a mounted hierarchy, just remount with different options:

```
# mount -o remount,cpuset,ns hier1 /dev/cgroup
```

Now memory is removed from the hierarchy and ns is added.

Note this will add ns to the hierarchy but won't remove memory or cpuset, because the new options are appended to the old ones:

```
# mount -o remount,ns /dev/cgroup
```

To Specify a hierarchy's release_agent:

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

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 /dev/cgroup you can find a tree that corresponds to the tree of the cgroups in the system. For instance, /dev/cgroup is the cgroup that holds the whole system.

If you want to change the value of release_agent:

```
# echo "/sbin/new_release_agent" > /dev/cgroup/release_agent
```

It can also be changed via remount.

If you want to create a new cgroup under /dev/cgroup:

```
# cd /dev/cgroup
# 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
```

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.

2.4 Notification API

There is mechanism which allows to get notifications about changing
status of a cgroup.

To register new notification handler you need:
 - create a file descriptor for event notification using eventfd(2);
 - open a control file to be monitored (e.g. memory.usage_in_bytes);
 - write "<event_fd> <control_fd> <args>" to cgroup.event_control.
   Interpretation of args is defined by control file implementation;

eventfd will be woken up by control file implementation or when the cgroup is removed.

To unregister notification handler just close eventfd.

NOTE: Support of notifications should be implemented for the control file. See documentation for the subsystem.

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>_subsys

If a subsystem can be compiled as a module, it should also have in its module initcall a call to cgroup_load_subsys(), and in its exitcall a call to cgroup_unload_subsys(). It should also set its_subsys.module = THIS_MODULE in its .c file.

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

```
struct cgroup_subsys_state *create(struct cgroup_subsys *ss,
                                   struct cgroup *cgrp)
(cgroup_mutex held by caller)
```

Called to create 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 negative error code. 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.

```
void destroy(struct cgroup_subsys *ss, struct cgroup *cgrp)
(cgroup_mutex held by caller)
```

The cgroup system is about to destroy the passed cgroup; the subsystem should do any necessary cleanup and free its subsystem state object. By the time this method is called, the cgroup has already been unlinked from the file system and from the child list of its parent; cgroup->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 pre_destroy(struct cgroup_subsys *ss, struct cgroup *cgrp);
```

Called before checking the reference count on each subsystem. This may be useful for subsystems which have some extra references even if there are not tasks in the cgroup. If pre_destroy() returns error code, rmdir() will fail with it. From this behavior, pre_destroy() can be called multiple times against a cgroup.

```
int can_attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
               struct task_struct *task, bool threadgroup)
(cgroup_mutex held by caller)
```

Called prior to moving a task into a cgroup; if the subsystem returns an error, this will abort the attach operation. If a NULL

task is passed, then a successful result indicates that *any*
unspecified task can be moved into the cgroup. 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. If threadgroup is
true, then a successful result indicates that all threads in the given
thread's threadgroup can be moved together.

```
void cancel_attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
               struct task_struct *task, bool threadgroup)
(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.

```
void attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
            struct cgroup *old_cgrp, struct task_struct *task,
            bool threadgroup)
(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.
If threadgroup is true, the subsystem should take care of all threads
in the specified thread's threadgroup. Currently does not support any
subsystem that might need the old_cgrp for every thread in the group.

```
void fork(struct cgroup_subsy *ss, struct task_struct *task)
```

Called when a task is forked into a cgroup.

```
void exit(struct cgroup_subsys *ss, struct task_struct *task)
```

Called during task exit.

```
int populate(struct cgroup_subsys *ss, struct cgroup *cgrp)
(cgroup_mutex held by caller)
```

Called after creation of a cgroup to allow a subsystem to populate
the cgroup directory with file entries. The subsystem should make
calls to cgroup_add_file() with objects of type cftype (see
include/linux/cgroup.h for details). Note that although this
method can return an error code, the error code is currently not
always handled well.

```
void post_clone(struct cgroup_subsys *ss, struct cgroup *cgrp)
(cgroup_mutex held by caller)
```

Called at the end of cgroup_clone() to do any parameter
initialization which might be required before a task could attach. For
example in cpusets, no task may attach before 'cpus' and 'mems' are set
up.

```
void bind(struct cgroup_subsys *ss, struct cgroup *root)
```

```
(cgroup_mutex and ss->hierarchy_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. 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.
```