Subject: [RFC] Control Groups Roadmap ideas
Posted by Paul Menage on Tue, 08 Apr 2008 21:14:08 GMT
View Forum Message <> Reply to Message

This is a list of some of the sub-projects that I'm planning for
Control Groups, or that I know others are planning on or working on.
Any comments or suggestions are welcome.

1) Stateless subsystems

This was motivated by the recent "freezer" subsystem proposal, which
included a facility for sending signals to all members of a cgroup.
This wasn't specifically freezer-related, and wasn't even something
that needed particular per-cgroup state - its only state is that set
of processes, which is already tracked by crgoups. So it could
theoretically be mounted on multiple hierarchies at once, and wouldn't
need an entry in the css_set array.

This would require a few internal plumbing changes in cgroups, in particular:

- hashing css_set objects based on their cgroups rather than their css pointers
- allowing stateless subsystems to be in multiple hierarchies
- changing the way hierarchy ids are calculated - simply ORing
together the subsystem would no longer work since that could result in
duplicates

2) More flexible binding/unbinding/rebinding

Currently you can only add/remove subsystems to a hierarchy when it
has just a single (root) cgroup. This is a bit inflexible, so I'm
planning to support:

- adding a subsystem to an existing hierarchy by automatically
creating a subsys state object for the new subsystem for each existing
cgroup in the hierarchy and doing the appropriate
can_attach()/attach_tasks() callbacks for all tasks in the system

- removing a subsystem from an existing hierarchy by moving all tasks
to that subsystem's root cgroup and destroying the child subsystem
state objects

- merging two existing hierarchies that have identical cgroup trees

- (maybe) splitting one hierarchy into two separate hierarchies

Whether all these operations should be forced through the mount()
system call, or whether they should be done via operations on cgroup
control files, is something I've not figured out yet.

3) Subsystem dependencies

This would be a fairly simple change, essentially allowing one
subsystem to require that it only be mounted on a hierarchy when some
other subsystem was also present. The implementation would probably be
a callback that allows a subsystem to confirm whether it's prepared to
be included in a proposed hierarchy containing a specified subsystem
bitmask; it would be able to prevent the hierarchy from being created
by giving an error return. An example of a use for this would be a
swap subsystem that is mostly independent of the memory controller,
but uses the page-ownership tracking of the memory controller to
determine which cgroup to charge swap pages to. Hence it would require
that it only be mounted on a hierarchy that also included a memory
controller. The memory controller would make no such requirement by
itself, so could be used on its own without the swap controller.

4) Subsystem Inheritance

This is an idea that I've been kicking around for a while trying to
figure out whether its usefulness is worth the in-kernel complexity,
versus doing it in userspace. It comes from the idea that although
cgroups supports multiple hierarchies so that different subsystems can
see different task groupings, one of the more common uses of this is
(I believe) to support a setup where say we have separate groups A, B
and C for one resource X, but for resource Y we want a group
consisting of A+B+C. E.g. we want individual CPU limits for A, B and
C, but for disk I/O we want them all to share a common limit. This can
be done from userspace by mounting two hierarchies, one for CPU and
one for disk I/O, and creating appropriate groupings, but it could
also be done in the kernel as follows:

- each subsystem "foo" would have a "foo.inherit" file provided by
(and handled by) cgroups in each group directory

- setting the foo.inherit flag (i.e. writing 1 to it) would cause
tasks in that cgroup to share the "foo" subsystem state with the
parent cgroup

- from the subsystem's point of view, it would only need to worry

about its own foo_cgroup objects and which task was associated with each object; the subsystem wouldn't need to care about which tasks were part of each cgroup, and which cgroups were sharing state; that would all be taken care of by the cgroup framework

I've mentioned this a couple of times on the containers list as part of other random discussions; at one point Serge Hallyn expressed some interest but there's not been much noise about it either way. I figured I'd include it on this list anyway to see what people think of it.

5) "procs" control file

This would be the equivalent of the "tasks" file, but acting/reporting on entire thread groups. Not sure exactly what the read semantics should be if a sub-thread of a process is in the cgroup, but not its thread group leader.

6) Statistics / binary API

Balaji Rao is working on a generic way to gather per-subsystem statistics; it would also be interesting to construct an extensible binary API via taskstats. One possible way to do this (taken from my email earlier today) would be:

With the taskstats interface, we could have operations to:

- describe the API exported by a given subsystem (automatically generated, based on its registered control files and their access methods)

- retrieve a specified set of stats in a binary format

So as a concrete example, with the memory, cpuacct and cpu subsystems configured, the reported API might look something like (in pseudo-code form)

0 : memory.usage_in_bytes : u64
1 : memory.limit_in_bytes : u64
2 : memory.failcnt : u64
3 : memory.stat : map
4 : cpuacct.usage : u64
5 : cpu.shares : u64
6 : cpu.rt_runtime_ms : s64

7 : cpu.stat : map

This list would be auto-generated by cgroups based on inspection of
the control files.

The user could then request stats 0, 3 and 7 for a cgroup to get the
memory.usage_in_bytes, memory.stat and cpu.stat statistics.

The stats could be returned in a binary format; the format for each
individual stat would depend on the type of that stat, and these could
be simply concatenated together.

A u64 or s64 stat would simply be a 64-bit value in the data stream

A map stat would be represented as a sequence of 64-bit values,
representing the values in the map. There would be no need to include
the size of the map or the key ordering in the binary format, since
userspace could determine that by reading the ASCII version of the map
control file once at startup.

So in the case of the request above for stats 0, 3 & 7, the binary
stats stream would be a sequence of 64-bit values consisting of:

<memory.usage>
<memory.stat.cache>
<memory.stat.rss>
<memory.stat.active>
<memory.stat.inactive>
<cpu.stat.utime>
<cpu.stat.stime>

If more stats were added to memory.stat or cpu.stat by a future
version of the code, then they would automatically appear; any that
userspace didn't understand it could ignore.

The userspace side of this could be handled by libcg.

8) Subsystems from modules

Having completely unknown subsystems registered at run time would
involve adding a bunch of complexity and additional locking to cgroups
- but allowing a subsystem to be known at compile time but just
stubbed until first mounted (at which time its module would be loaded)
should increase the flexibility of cgroups without hurting its
complexity or performance.

7) New subsystems

- Swap, disk I/O - already being worked on by others

- OOM handler. Exactly what semantics this should provide aren't 100% clear. At Google we have a useful OOM handler that allows root to intercept OOMs as they're about to happen, and take appropriate action such as killing some other lower-priority job to free up memory, etc. Another useful feature of this subsystem might be to allow a process in that cgroup to get an early notification that its cgroup is getting close to OOM. This needs to be a separate subsystem since it could be used to provide OOM notification/handling for localized OOMs caused either by cpusets or the memory controller.

- network tx/rx isolation. The cleanest way that we've found to do this is to provide a per-cgroup id which can be exposed as a traffic filter for regular Linux traffic control - then you can construct arbitrary network queueing structures without requiring any new APIs, and tie flows from particular cgroups into the appropriate queues.

8) per-mm owner field

To remove the need for per-subsystem counted references from the mm.
Being developed by Balbir Singh

_____