Subject: [PATCH 0/6] containers: Generic Process Containers (V6)
Posted by Paul Menage on Fri, 22 Dec 2006 14:14:42 GMT
View Forum Message <> Reply to Message

--

This is an update to my multi-hierarchy multi-subsystem generic process containers patch. Changes include:

- updated to 2.6.20-rc1

- incorporating some fixes from Srivatsa Vaddagiri

- release agent path is per-hierarchy, and defaults to empty

- dropped the patch splitting cpusets and memsets for now, since it's a fairly mechanical patch, but a bit painful to maintain in the presence of any other cpusets changes in mainline.

A couple of important issues have arisen on which feedback would be appreciated:

1) The need (or otherwise) for multi-subsystem hierarchies

(Based on discussions with Paul Jackson)

The patch as it stands allows you to mount a container hierarchy as a filesystem, and at that point select a set of subsystems to be bound to that hierarchy. Subsystems can't be bound or unbound while the hierarchy is active. Dynamic binding/unbinding in this way is theoretically possible, but runs into various nasty conditions when you have to attach or detach a subsystem to a potentially large tree of containers and tasks. E.g. what do you do if halfway through unbinding the subsystem state from the containers in a hierarchy, you run into a container whose subsystem state is busy and hence can't be freed? Either we'd need to put fairly strict limits on the properties of subsystems that can be dynamically bound/unbound (i.e. can't refuse to transfer a task from one container to another, can't use css refcounting to keep subsystem state alive, etc) or we'd need to be prepared to do some very nasty error-handling and rollback.

PaulJ pointed out that there's a continuum between:
 1) just one controller per container, period, or
 2) full dynamic binding and unbinding of any controller from any
   one or more containers, with no limitations due to what else
   is or ever was bound to what when.

My current patch falls somewhere in the middle. PaulJ wondered whether
it would be cleaner just to aim for case 1 - i.e. rather than having
the concept of hierarchies to tie multiple subsystems together, have
each subsystem be its own hierarchy.

My own feeling is that having each subsystem be its own hierarchy is
certainly possible, but results in a lot more effort in userspace to
manage - you have to manage N hierarchies of containers for N
subsystems, rather than just one.

You also have the issue that if a task is forking just as you start
moving it from one container to another (in each of the N hierarchies)
you could end up with the child in the original container in some
hierarchies, and in the new container in others, which isn't ideal.

>From a performance point of view, one-controller-per-hierarchy has a
little more overhead at fork()/exit() time (since there are more
reference counts to atomically update) but a little less overhead for
accessing the subsystem state for a particular task, since there's one
less level of indirection involved.

2) Dynamic creation/destruction of containers from inside the kernel

-- --------

A recent patch from Serge Hallyn on containers@lists.osdl.org proposed
a container filesystem somewhat similar to the one in this patch,
designed to expose the hierarchy of namespaces (specifically, nsproxy
objects). A major difference was that a child container could be
created dynamically from within sys_unshare(), and automatically freed
once it was no longer being referenced.

This could be fitted into my container model with a couple of small changes:

- a container_clone() function that essentially creates a new child
container of the current container (in the specified subsystem's
hierarchy) and moves the current process into the new container - the
equivalent of doing

```
mkdir $current_container_dir/$some_unique_name
echo $$ > $current_container_dir/$some_unique_name
```

- an auto_delete option for containers - similar to notify_on_exit,
but rather than invoking a userspace program, simply deletes the
container from within the kernel.

Then the nsproxy container could be implemented easily as a container

subsystem - rather than having a direct nsproxy field in struct task,
it would use the generic container pointer associated with the nsproxy
subsystem; sys_unshare() would call container_clone() to create the
new container.

== =====

Generic Process Containers

There have recently been various proposals floating around for
resource management/accounting subsystems in the kernel, including
ResGroups, User BeanCounters, NSProxy containers, and others. These
all need the basic abstraction of being able to group together
multiple processes in an aggregate, in order to track/limit the
resources permitted to those processes, and all implement this
grouping in different ways.

Already existing in the kernel is the cpuset subsystem; this has a
process grouping mechanism that is mature, tested, and well documented
(particularly with regards to synchronization rules).

This patchset extracts the process grouping code from cpusets into a
generic container system, and makes the cpusets code a client of
the container system.

It also provides several example clients of the container system,
including ResGroups and BeanCounters

The change is implemented in three stages, plus three example
subsystems that aren't necessarily intended to be merged as part of
this patch set, but demonstrate the applicability of the framework.

1) extract the process grouping code from cpusets into a standalone system

2) remove the process grouping code from cpusets and hook into the
  container system

3) convert the container system to present a generic multi-hierarchy
  API, and make cpusets a client of that API

4) example of a simple CPU accounting container subsystem

5) example of implementing ResGroups and its numtasks controller over
  generic containers

6) example of implementing BeanCounters and its numfiles counter over

generic containers

The intention is that the various resource management efforts can also become container clients, with the result that:

- the userspace APIs are (somewhat) normalised

- it's easier to test out e.g. the ResGroups CPU controller in conjunction with the BeanCounters memory controller

- the additional kernel footprint of any of the competing resource management systems is substantially reduced, since it doesn't need to provide process grouping/containment, hence improving their chances of getting into the kernel

Signed-off-by: Paul Menage <menage@google.com>

--