Subject: Re: [PATCH 1/6] containers: Generic container system abstracted from cpusets code
Posted by Paul Menage on Tue, 02 Jan 2007 22:15:03 GMT
View Forum Message <> Reply to Message

On 12/30/06, Eric W. Biederman <ebiederm@xmission.com> wrote:
> Paul Menage <menage@google.com> writes:
>
> > This patch creates a generic process container system based on (and
> > parallel top) the cpusets code. At a coarse level it was created by
> > copying kernel/cpuset.c, doing s/cpuset/container/g, and stripping out any
> > code that was cpuset-specific rather than applicable to any process
> > container subsystem.
>
> First thank you for bring the conversation here. Given what
> you are implementing I rather object to the term containers as
> that is what we have been using to refer to the aggregate whole
> and not the individual pieces.

I guess I'm agnostic on the exact term used - but one point is that this isn't intended just for the virtual server support that you're working on, but rather any kernel facility that wants to be able to associate data and/or behaviour with a group of processes (not necessarily related by process inheritance, but where the grouping is inherited at fork time). This includes Cpusets, and general resource isolation/management, without virtualization.

Terms like "process group", "session", etc, are already used up. A "container" seems like a reasonable term for a generic process grouping from which the processes can't escape without root privileges. Since you're not only "containing" processes but also "virtualizing" them, the term "virtual server" would seem better for your work, unless you were wanting to keep the same name as Solaris Containers.

>
> I'm still digesting this but do you think you could make the code
> pid namespace safe before moving it all over creation.
>
> i.e. pid_nr(task_pid(task)) not task->pid.
>
> I hadn't realized we had any users like the one below left.

OK, I'll take a look at that.

>
> The whole interface that reads out the processes in your task
> grouping looks scary. It takes the tasklist_lock and holds

> it for an indefinite duration. All it currently needs is
> the rcu_read_lock. Holding the tasklist_lock looks like a good
> way to kill performance on a big box. Even hold the cpu for
> an indefinite duration I find a little worrying but no where
> near as bad as taking a global lock for an indefinite period
> of time. Although I am curious why this is even needed when
> we have /proc/<pid>/cpuset which gets us the information
> in another way.

As PaulJ mentioned, it's much more efficient to read this once for a given container, rather than having to iterate over the whole of /proc. If it's possible to use RCU for this, I'd be very happy to change it to do that.

>
> I hate attach_task. Allowing movement of a process from
> one set to another by another process looks like a great way
> to create subtle races. The very long and exhaustive locking
> comments seem to verify this. For most of the unix API
> we have avoided things for precisely this reason. Leaving that
> set of races to the debugging commands in sys_ptrace.

If the only way to get a process into a new container is to clone the current container and shift the current process into it, that's a very restrictive model, and too restrictive for some of the things that I and others want to do. (E.g. moving a process between different resource containers based on which client it's currently doing work; adding a new process to an existing running job).

I'd be interested in supporting the clone-based model as well, though. With the addition of that, it would always be possible for a subsystem that wants to just support the clone model, to always fail its can_attach_task() call to prevent the container system from moving external processes into the container.

>
> You are putting a pointer into the task_struct for each class
> of resource you want to count. Ouch.

No, I'm putting a pointer for each independent hierarchy that you want to maintain - multiple classes of resources can be tracked in the same hierarchy. The max number of hierarchies is a number that's configurable at compile time.

> Andi Kleen was sufficiently
> paranoid about the space bloat that we were obliged to introduce
> struct nsproxy.

I think that nsproxy would be a good example of something that could be attached as a generic container subsystem. I'd need to make a couple of additions - a way to dynamically create a new container at fork/unshare time and move the newly unshared process into it, and a way to auto-delete a container (see below), so I'm not suggesting it quite yet.

> Process resource control that looks like a good reason to add some
> more unshare flags or some separate syscalls whichever is simpler.
> At least that has a simple user interface that is easy to audit.
>

There are too many different resources and competing views on resource control to be able to handle this via a few extra flags, I think.

>
> Why does any of this code need a user mode helper? I guess
> because of the complicated semantics this doesn't do proper
> reference counting so you can't implicitly free these things
> on the exit of the last task that uses them. That isn't the
> unix way and I don't like it. Way over complicated.

That's there for compatibility with cpusets. I was thinking of adding an auto-delete option that does queue_work() to trigger a vfs_rmdir() from the work queue, which would avoid the races that PaulJ was concerned about. But I can also envisage more exotic cases where userspace wants to do something more complex (e.g. read some final accounting values) before deleting the container.

Paul

_____

Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers