Subject: Thoughts on Namespace / Subsystem unification
Posted by Paul Menage on Mon, 03 Sep 2007 12:22:07 GMT
View Forum Message <> Reply to Message

Today at the mini-summit I think that the fact that I was only connected via Skype made it way too difficult for me to get across the idea of my proposals for exploring the potential benefits to be gained from unifying namespaces and "task container subsystems", hereafter just referred to mainly as "subsystems" to avoid confusion over the term container. (Yes, the name may well be changing to something like "task sets" ...) So I'll flesh them out a bit in an email instead.
This should be regarded more as a partially-formed concept/vision than a complete design proposal.

The idea is based on that fact that subsystems and namespaces have a bunch of similarities:

- associate each process with a piece of state (where that state may be resource limits/usage, object translation table, etc)

- allow multiple processes to share the same piece of state in aggregate (e.g. multiple processes allocate resources from the same limit, or use the same ipc lookup table)

- aren't generally changeable/escapable (except by users with root or delegated privileges)

- have a shared aggregator object (nsproxy or css_group) that allows multiple tasks that share the same namespaces/subsystems to cheaply add/remove refcounts from a whole bunch of objects at once.

- are used as state for code that may have hooks scattered throughout the kernel code (e.g. namespace indirection, resource checking).

And they also have a few differences:

1) "subsystems" have a generic and flexible control/monitoring API via the "containerfs" filesystem. Namespaces are viewable internally via existing Unix/Linux APIs, and may potentially have additional custom control/monitoring set up as special-purpose code. (But I believe most don't).

I think that it could be very useful for namespaces to have the same support for control/monitoring. For example, consider the IPC namespace. This has a shm_ctlmni field that controls how many shm ids can be created in total in that namespace. Currently only the root IPC namespace can have its shm_ctlmni updated via sysctl; child namespaces

aren't configurable in the same way. It could be plausible to have the shm_ctlmni in other namespaces be updateable too, assuming that the relevant /proc file was virtualized. But then there are issues such as:

-how does a process in the parent namespace read/write the shmmni value in the child namespace? Having to fork a child into the namespace via something like sys_hijack() seems overly expensive.

- should a namespace' shmmni value be writeable only by its parent, or writeable by the child too (in which case, how does the parent limit the child's IPC id creation?)

If the IPC namespace had the concept of an "internal" view (the shmmni value seen and writeable by the child via normal IPC interfaces) and an "external" view (the shmmni value seen and writeable by the parent, via a control file in containerfs) these problems could be resolved. The child could control its own shmmni value, and the parent could impose an additional limit to control the child's resources. (If it turns out that I've misunderstood the IPC namespace and this was actually a bad example, I hope that you can still appreciate the generic argument that I'm trying to make here).

2) entering the "container" associated with a subsystem is well supported since subsystems are expecting the relevant state pointers to be somewhat volatile; entering namespaces is tricky since lots of existing code doesn't expect the namespace pointer to be volatile, and can't necessarily be updated to allow such volatility since they're performance-critical structures.

But the fact that this is a distinction between namespaces and subsystems is a bit artificial. I think it's quite possible to imagine some namespaces whose implementation can quite easily handle tasks changing their namespace pointer unexpectedly, since they're written to handle the tricky issues this introduces, and aren't so performance critical that they can't do locking when necessary.

3) "subsystems" have new instances created via a mkdir in "containerfs", namespaces have new instances created via clone() or unshare(). But this could just be considered two different ways of creating the same kind of object. The container_clone() call already exists to support the clone/unshare approach used by namespaces. The choice of which was appropriate (or even both?) could be made by the kernel code for the subsystem/namespace in question.

4) "namespaces" expire as soon as no tasks are using them; "subsystems" persist until explicitly deleted. But containerfs already

has "notify on release" support; extending this to include "delete on release" wouldn't be hard for people who wanted their resource controllers and other subsystems cleaned up as soon as they weren't in use, and the same code could support the expected behaviour for namespaces. And in the opposite direction, some users might want to be able to set up some kind of namespace environment and have it persist even when there were no active processes in the nsproxy. (Perhaps pre-allocating environments, or reusing them across multiple operations).

5) There's no straightforward way to view/list namespaces from userspace, since the nsproxy is regarded as purely an in-kernel convenience/performance feature, whereas "subsystems" can be easily viewed and listed via containerfs directories. But this seems like it would be useful behaviour for namespaces too.

I hope this demonstrates that the distinction between namespaces and "subsystems" is at least partially arbitrary, and that namespaces could benefit from a lot of the support that subsystems get automatically from the "task containers" framework.

The ns_container subsystem is a first step towards linking subsystems and namespaces - it associates an entire set of namespaces (via an nsproxy) with a "task container", so the nsproxy is on the same level with other subsystems. But based on the similarities/differences explored above, my argument is that we should explore the idea that subsystems and namespaces should be considered on the same level, rather than subsystems be considered as being on the same level as the nsproxy aggregate. If we could come up with a single abstraction that captures the similarities and differences between namespaces and subsystems, this could give the benefits of both.

I'll call the aggregation of multiple such abstractions a "container" for brevity, although in practice it's somewhere between the concept of my "task container" and the full vision of containers as self-contained virtualised environments.

The abstraction (I'm not sure I have an elegant name for it yet) would have the properties listed as the similarities above; it would be tied to some kind of aggregator that would be similar to an nsproxy or a "task container". It would have a generic filesystem-base control/monitoring API. It would be parameterizable with options such as:

- should a process be allowed to enter this "container" (a property specified by the code itself)

- whether it can be created via mkdir and/or clone/unshare (specified by the code itself)

- what action should be taken if this "container" becomes empty (probably user-specifiable, with options such as "ignore", "notify", "delete")

(I think these three options capture the essential differences between "subsystems" and namespaces as they exist currently).

It's a bit different from the arguments of "everything's a namespace" that have been made in the past, since the new abstraction resembles more a "task container subsystem" than it does the existing definition of a namespace.

In a way it would incorporate some of the ideas of the "rcfs" subsystem that Vatsa proposed a while ago, but with differences such as not having separate arrays for subsystems and namespaces, and having the "container" be a much more first-class object, both in terms of kernel support and in terms of visibility from userspace (compared to the current situation where an nsproxy is purely an in-kernel convenience that's not visible from userspace). There would also be more focus on adding control/monitoring APIs to namespaces.

Paul

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: Thoughts on Namespace / Subsystem unification
Posted by ebiederm on Mon, 03 Sep 2007 14:14:33 GMT
View Forum Message <> Reply to Message

"Paul Menage" <menage@google.com> writes:

> Today at the mini-summit I think that the fact that I was only
> connected via Skype made it way too difficult for me to get across the
> idea of my proposals for exploring the potential benefits to be gained
> from unifying namespaces and "task container subsystems", hereafter
> just referred to mainly as "subsystems" to avoid confusion over the
> term container. (Yes, the name may well be changing to something like
> "task sets" ...) So I'll flesh them out a bit in an email instead.
> This should be regarded more as a partially-formed concept/vision than
> a complete design proposal.
>
>

> The idea is based on that fact that subsystems and namespaces have a
> bunch of similarities:
>
> - associate each process with a piece of state (where that state may
> be resource limits/usage, object translation table, etc)
>
> - allow multiple processes to share the same piece of state in
> aggregate (e.g. multiple processes allocate resources from the same
> limit, or use the same ipc lookup table)
>
> - aren't generally changeable/escapable (except by users with root or
> delegated privileges)
>
> - have a shared aggregator object (nsproxy or css_group) that allows
> multiple tasks that share the same namespaces/subsystems to cheaply
> add/remove refcounts from a whole bunch of objects at once.
>
> - are used as state for code that may have hooks scattered throughout
> the kernel code (e.g. namespace indirection, resource checking).
>
> And they also have a few differences:
>
> 1) "subsystems" have a generic and flexible control/monitoring API via
> the "containerfs" filesystem. Namespaces are viewable internally via
> existing Unix/Linux APIs, and may potentially have additional custom
> control/monitoring set up as special-purpose code. (But I believe most
> don't).
>
> I think that it could be very useful for namespaces to have the same
> support for control/monitoring. For example, consider the IPC
> namespace. This has a shm_ctlmni field that controls how many shm ids
> can be created in total in that namespace. Currently only the root IPC
> namespace can have its shm_ctlmni updated via sysctl; child namespaces
> aren't configurable in the same way. It could be plausible to have the
> shm_ctlmni in other namespaces be updateable too, assuming that the
> relevant /proc file was virtualized. But then there are issues such
> as:
>
> -how does a process in the parent namespace read/write the shmmni
> value in the child namespace? Having to fork a child into the
> namespace via something like sys_hijack() seems overly expensive.

When complete we should be able to see the appropriate /proc file
from if we can see the process. This is just a matter of sorting
out the implementation in /proc.

The plan from my side is to be able to mount /proc /sys etc filesystems
and see what a processes inside a namespace will see from the outside.

The initial mount may need to be done from the inside but after that
all should be good.

> - should a namespace' shmmni value be writeable only by its parent, or
> writeable by the child too (in which case, how does the parent limit
> the child's IPC id creation?)

> If the IPC namespace had the concept of an "internal" view (the shmmni
> value seen and writeable by the child via normal IPC interfaces) and
> an "external" view (the shmmni value seen and writeable by the parent,
> via a control file in containerfs) these problems could be resolved.
> The child could control its own shmmni value, and the parent could
> impose an additional limit to control the child's resources. (If it
> turns out that I've misunderstood the IPC namespace and this was
> actually a bad example, I hope that you can still appreciate the
> generic argument that I'm trying to make here).

It is a mixed example.

> 2) entering the "container" associated with a subsystem is well
> supported since subsystems are expecting the relevant state pointers
> to be somewhat volatile; entering namespaces is tricky since lots of
> existing code doesn't expect the namespace pointer to be volatile, and
> can't necessarily be updated to allow such volatility since they're
> performance-critical structures.
>
> But the fact that this is a distinction between namespaces and
> subsystems is a bit artificial. I think it's quite possible to imagine
> some namespaces whose implementation can quite easily handle tasks
> changing their namespace pointer unexpectedly, since they're written
> to handle the tricky issues this introduces, and aren't so performance
> critical that they can't do locking when necessary.

So far I think the extra volatility of subsystems is a misfeature.
I think with a little care you could get the cheapness of the
current namespaces with the flexibility of containers. Although
this is something that needs great care.

> 3) "subsystems" have new instances created via a mkdir in
> "containerfs", namespaces have new instances created via clone() or
> unshare(). But this could just be considered two different ways of
> creating the same kind of object. The container_clone() call already
> exists to support the clone/unshare approach used by namespaces. The
> choice of which was appropriate (or even both?) could be made by the
> kernel code for the subsystem/namespace in question.

Yes. Although currently I think the filesystem interface is the most
questionable part of the resource controlling subsystems. No

offense, but we keep seeming to run into weird limitations and
I have a hard time wrapping my head around the whys and wherefores
of that model.

> 4) "namespaces" expire as soon as no tasks are using them;
> "subsystems" persist until explicitly deleted. But containerfs already
> has "notify on release" support; extending this to include "delete on
> release" wouldn't be hard for people who wanted their resource
> controllers and other subsystems cleaned up as soon as they weren't in
> use, and the same code could support the expected behaviour for
> namespaces. And in the opposite direction, some users might want to be
> able to set up some kind of namespace environment and have it persist
> even when there were no active processes in the nsproxy. (Perhaps
> pre-allocating environments, or reusing them across multiple
> operations).
>
> 5) There's no straightforward way to view/list namespaces from
> userspace, since the nsproxy is regarded as purely an in-kernel
> convenience/performance feature, whereas "subsystems" can be easily
> viewed and listed via containerfs directories. But this seems like it
> would be useful behaviour for namespaces too.

Maybe. So far the subsystems interfaces to user space seem
overdesigned and inflexible in really weird ways to me.

With namespaces we can certainly add more. Currently we have enough
to make progress.

> I hope this demonstrates that the distinction between namespaces and
> "subsystems" is at least partially arbitrary, and that namespaces
> could benefit from a lot of the support that subsystems get
> automatically from the "task containers" framework.

I definitely agree that the distinction is arbitrary, and it something
I have been pointing out for a while. Which is how we got as
far as css_group etc.

> The ns_container subsystem is a first step towards linking subsystems
> and namespaces - it associates an entire set of namespaces (via an
> nsproxy) with a "task container", so the nsproxy is on the same level
> with other subsystems.

And I never understood why anyone did it that way.

> But based on the similarities/differences
> explored above, my argument is that we should explore the idea that
> subsystems and namespaces should be considered on the same level,
> rather than subsystems be considered as being on the same level as the

> nsproxy aggregate. If we could come up with a single abstraction that
> captures the similarities and differences between namespaces and
> subsystems, this could give the benefits of both.

Sure.

> I'll call the aggregation of multiple such abstractions a "container"
> for brevity, although in practice it's somewhere between the concept
> of my "task container" and the full vision of containers as
> self-contained virtualised environments.
>
> The abstraction (I'm not sure I have an elegant name for it yet) would
> have the properties listed as the similarities above; it would be tied
> to some kind of aggregator that would be similar to an nsproxy or a
> "task container". It would have a generic filesystem-base
> control/monitoring API. It would be parameterizable with options such
> as:
>
> - should a process be allowed to enter this "container" (a property
> specified by the code itself)

There are weird security aspects to enter which is why Serge's
sys_hijack thing may make more sense. Frankly I'm not convinced
that there is a better way to do things.

> - whether it can be created via mkdir and/or clone/unshare (specified
> by the code itself)
>
> - what action should be taken if this "container" becomes empty
> (probably user-specifiable, with options such as "ignore", "notify",
> "delete")
>
> (I think these three options capture the essential differences between
> "subsystems" and namespaces as they exist currently).
>
> It's a bit different from the arguments of "everything's a namespace"
> that have been made in the past, since the new abstraction resembles
> more a "task container subsystem" than it does the existing definition
> of a namespace.

A bit.

> In a way it would incorporate some of the ideas of the "rcfs"
> subsystem that Vatsa proposed a while ago, but with differences such
> as not having separate arrays for subsystems and namespaces, and
> having the "container" be a much more first-class object, both in
> terms of kernel support and in terms of visibility from userspace
> (compared to the current situation where an nsproxy is purely an

> in-kernel convenience that's not visible from userspace). There would
> also be more focus on adding control/monitoring APIs to namespaces.

Currently I am not convinced that we want a first class container
object in the kernel. There is all kinds of weirdness that results.
But I am a minimalist and like to start with the simplest thing that
we can possibly start with.

I do think having common idioms and common infrastructure is useful.

Eric

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: Thoughts on Namespace / Subsystem unification
Posted by Paul Menage on Mon, 03 Sep 2007 16:27:14 GMT
View Forum Message <> Reply to Message

On 9/3/07, Eric W. Biederman <ebiederm@xmission.com> wrote:
> >
> > -how does a process in the parent namespace read/write the shmmni
> > value in the child namespace? Having to fork a child into the
> > namespace via something like sys_hijack() seems overly expensive.
>
> When complete we should be able to see the appropriate /proc file
> from if we can see the process. This is just a matter of sorting
> out the implementation in /proc.

So inside a child namespace we have a "limit" values such as shmmni,
which normally are system-wide value that root can set to limit total
resource usage in the OS; if the only way to access this value is via
a namespace view, then who "owns" that value? - is it something that
the parent namespace can set to limit the child's total resource
usage, or is it something that the child can set to limit its own
total resource usage. If the former, then we break virtualization a
bit since in the child the value will be read-only; if the latter then
the parent has less control over the child's resource usage that it
would like. A separate monitoring/control API that the parent (who
knows that virtualization is involved) can access lets you set an
external limit as well as an internal limit.

> >
> > But the fact that this is a distinction between namespaces and
> > subsystems is a bit artificial. I think it's quite possible to imagine
> > some namespaces whose implementation can quite easily handle tasks

> > changing their namespace pointer unexpectedly, since they're written
> > to handle the tricky issues this introduces, and aren't so performance
> > critical that they can't do locking when necessary.
>
> So far I think the extra volatility of subsystems is a misfeature.
> I think with a little care you could get the cheapness of the
> current namespaces with the flexibility of containers. Although
> this is something that needs great care.

For read-only access to a subsystem state object, an rcu_read_lock()
is sufficient - once in the RCU section, access to a subsystem state
is as cheap as accessing a namespace - they're both constant indexed
offsets from a pointer in task_struct. Similarly if you're updating a
value but aren't too worried if you update the state that the task
just moved away from (e.g. in a rate-based scheduler, you probably
don't care too much if you charge the task's old container for, say,
CPU cycles, rather than its new container).

>
> Maybe. So far the subsystems interfaces to user space seem
> overdesigned and inflexible in really weird ways to me.

Can you elaborate on that? I'm always interested in trying to make my
interfaces less weird where possible ...

> > The ns_container subsystem is a first step towards linking subsystems
> > and namespaces - it associates an entire set of namespaces (via an
> > nsproxy) with a "task container", so the nsproxy is on the same level
> > with other subsystems.
>
> And I never understood why anyone did it that way.

The ns_container subsystem lets you name a collection of namespaces,
and associate them with a bunch of resource controllers, so in that
sense it's definitely useful. The question in my mind is whether it
goes far enough.

> > - should a process be allowed to enter this "container" (a property
> > specified by the code itself)
>
> There are weird security aspects to enter which is why Serge's
> sys_hijack thing may make more sense. Frankly I'm not convinced
> that there is a better way to do things.

Agreed for some kinds of namespaces, which is why certain
namespaces/subsystems might want to declare that they can't be entered
without a hijack-like interface. But I think it's too strong a
restriction to enforce for all namespaces/subsystems.

For instance:

- a webserver that wants to charge resource usage based on which client it's currently doing work for, needs to be able to shuffle its threads between different resource controller instances.

- we're experimenting with using cpusets for memory isolation; part of this involves an early initscript that creates a "sys" container and isolates all the running system daemons in that container. To do this with no support for migrating tasks between cpusets, and only relying on sys_hijack, would involve a custom init, I think, rather than a simple initscript.

>
> Currently I am not convinced that we want a first class container
> object in the kernel. There is all kinds of weirdness that results.

Such as? The "first class container" object doesn't have to be called a container, or even directly analogous to a "virtual server container". It's just a way to name an association between a collection of namespaces/subsystems and a collection of tasks.

Paul

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: Thoughts on Namespace / Subsystem unification
Posted by serue on Thu, 06 Sep 2007 15:52:13 GMT
View Forum Message <> Reply to Message

Quoting Paul Menage (menage@google.com):
> On 9/3/07, Eric W. Biederman <ebiederm@xmission.com> wrote:
> > >
> > > -how does a process in the parent namespace read/write the shmmni
> > > value in the child namespace? Having to fork a child into the
> > > namespace via something like sys_hijack() seems overly expensive.
> > >
> > When complete we should be able to see the appropriate /proc file
> > from if we can see the process. This is just a matter of sorting
> > out the implementation in /proc.
> >
> So inside a child namespace we have a "limit" values such as shmmni,
> which normally are system-wide value that root can set to limit total
> resource usage in the OS; if the only way to access this value is via

> a namespace view, then who "owns" that value? - is it something that
> the parent namespace can set to limit the child's total resource
> usage, or is it something that the child can set to limit its own
> total resource usage. If the former, then we break virtualization a
> bit since in the child the value will be read-only; if the latter then
> the parent has less control over the child's resource usage that it
> would like. A separate monitoring/control API that the parent (who
> *knows* that virtualization is involved) can access lets you set an
> external limit as well as an internal limit.

Hopefully a container would only be able to lower it's value, not raise
it. The value is presumably inherited from the system, and the binary
which starts the vm, which is owned by the host admin, can further lower
the value before relinguishing control to the vm admin. Then the host
admin can hopefully do a container enter to investigate or further lower
the value if needed.

That's how I would see it...

>
> > >
> > > But the fact that this is a distinction between namespaces and
> > > subsystems is a bit artificial. I think it's quite possible to imagine
> > > some namespaces whose implementation can quite easily handle tasks
> > > changing their namespace pointer unexpectedly, since they're written
> > > to handle the tricky issues this introduces, and aren't so performance
> > > critical that they can't do locking when necessary.
> >
> > So far I think the extra volatility of subsystems is a misfeature.
> > I think with a little care you could get the cheapness of the
> > current namespaces with the flexibility of containers. Although
> > this is something that needs great care.
>
> For read-only access to a subsystem state object, an rcu_read_lock()
> is sufficient - once in the RCU section, access to a subsystem state
> is as cheap as accessing a namespace - they're both constant indexed
> offsets from a pointer in task_struct. Similarly if you're updating a
> value but aren't too worried if you update the state that the task
> just moved away from (e.g. in a rate-based scheduler, you probably
> don't care too much if you charge the task's old container for, say,
> CPU cycles, rather than its new container).
>
> >
> > Maybe. So far the subsystems interfaces to user space seem
> > overdesigned and inflexible in really weird ways to me.
>
> Can you elaborate on that? I'm always interested in trying to make my
> interfaces less weird where possible ...

>
> > > The ns_container subsystem is a first step towards linking subsystems
> > > and namespaces - it associates an entire set of namespaces (via an
> > > nsproxy) with a "task container", so the nsproxy is on the same level
> > > with other subsystems.
> > >
> > And I never understood why anyone did it that way.

Here's what you gain, Eric: you can easily compose a ns_container and
cpuset subsystem onto one hierarchy, and lock a container/vm/whatever to
a cpuset. Ditto for any other resource mgmt containers. So with no
additional namespace related effort, we can leverage all the resource
management implemented through containers, as virtual server resource
mgmt.

> The ns_container subsystem lets you name a collection of namespaces,
> and associate them with a bunch of resource controllers, so in that
> sense it's definitely useful. The question in my mind is whether it
> goes far enough.
>
> > > - should a process be allowed to enter this "container" (a property
> > > specified by the code itself)
> > >
> > There are weird security aspects to enter which is why Serge's
> > sys_hijack thing may make more sense. Frankly I'm not convinced
> > that there is a better way to do things.
>
> Agreed for some kinds of namespaces, which is why certain
> namespaces/subsystems might want to declare that they can't be entered
> without a hijack-like interface. But I think it's too strong a
> restriction to enforce for all namespaces/subsystems.

I'm pretty sure Eric was talking just about namespaces here.
Arbitrarily changing your resource mgmt limits by entering a new
container is presumably no big deal.

(Eric do correct me if I'm wrong :)

> For instance:
>
> - a webserver that wants to charge resource usage based on which
> client it's currently doing work for, needs to be able to shuffle its
> threads between different resource controller instances.
>
> - we're experimenting with using cpusets for memory isolation; part of
> this involves an early initscript that creates a "sys" container and
> isolates all the running system daemons in that container. To do this
> with no support for migrating tasks between cpusets, and only relying

> on sys_hijack, would involve a custom init, I think, rather than a
> simple initscript.
>
> >
> > Currently I am not convinced that we want a first class container
> > object in the kernel. There is all kinds of weirdness that results.
>
> Such as? The "first class container" object doesn't have to be called
> a container, or even directly analogous to a "virtual server
> container". It's just a way to name an association between a
> collection of namespaces/subsystems and a collection of tasks.

-serge

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers