Subject: Re: [ckrm-tech] [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Jackson on Sun, 11 Mar 2007 21:15:55 GMT
View Forum Message <> Reply to Message

Sam, responding to Herbert:

```
> > from my personal PoV the following would be fine:
> >
> >  spaces (for the various 'spaces')
> >    ...
> >  container (for resource accounting/limits)
> >    ...
>
> I like these a lot ...
```

Hmmm ... ok ...

Let me see if I understand this.

We have actors, known as threads, tasks or processes, which use things, which are instances of such classes of things as disk partitions, file systems, memory, cpus, and semaphores.

We assign names to these things, such as SysV id's to the semaphores, mount points to the file systems, pathnames to files and file descriptors to open files. These names provide handles that are typically more convenient and efficient to use, but alas less persistent, less ubiquitous, and needing of some dereferencing when used, to identify the underlying thing.

Any particular assignment of names to some of the things in particular class forms one namespace (aka 'space', above). For each class of things, a given task is assigned one such namespace. Typically many related tasks (such as all those of a login session or a job) will be assigned the same set of namespaces, leading to various opportunities for optimizing the management of namespaces in the kernel.

This assignment of names to things is neither injective nor surjective nor even a complete map.

For example, not all file systems are mounted, certainly not all possible mount points (all directories) serve as mount points, sometimes the same file system is mounted in multiple places, and sometimes more than one file system is mounted on the same mount point, one hiding the other.

In so far as the code managing this naming is concerned, the names are usually fairly arbitrary, except that there seems to be a tendency toward properly virtualizing these namespaces, presenting to a task the namespaces assigned it as if that was all there was, hiding the presence of alternative namespaces, and intentionally not providing a 'global view' that encompasses all namespaces of a given class.

This tendency culminates in the full blown virtual machines, such as Xen and KVM, which virtualize more or less all namespaces.

Because the essential semantics relating one namespace to another are rather weak (the namespaces for any given class of things are or can be pretty much independent of each other), there is a preference and a tradition to keep such sets of namespaces a simple flat space.

Conclusions regarding namespaces, aka spaces:

> A namespace provide a set of convenient handles for things of a particular class.
>
> For each class of things, every task gets one namespace (perhaps a Null or Default one.)
>
> Namespaces are partial virtualizations, the 'space of namespaces' is pretty flat, and the assignment of names in one namespace is pretty independent of the next.

===

That much covers what I understand (perhaps in error) of namespaces.

So what's this resource accounting/limits stuff?

I think this depends on adding one more category to our universe.

For the purposes of introducing yet more terms, I will call this new category a "metered class."

Each time we set about to manage some resource, we tend to construct some more elaborate "metered classes" out of the elemental classes of things (partitions, cpus, ...) listed above.

Examples of these more elaborate metered classes include percentages of a networks bandwidth, fractions of a nodes memory (the fake numa patch), subsets of the systems cpus and nodes (cpusets), ...

These more elaborate metered classes each have fairly 'interesting' and specialized forms. Their semantics are closely adapted to the underlying class of things from which they are formed, and to the usually challenging, often conflicting, constraints on managing the usage of such a resource.

For example, the rules that apply to percentages of a networks bandwidth have little in common with the rules that apply to sets of subsets of a systems cpus and nodes.

We then attach tasks to these metered classes. Each task is assigned one metered instance from each metered class. For example, each task is assigned to a cpuset.

For metered classes that are visible across the system, we tend to name these classes, and then use those names when attaching tasks to them. See for example cpusets.

For metered classes that are only privately visible within the current context of a task, such as setrlimit, set_mempolicy, mbind and set_mempolicy, we tend to implicitly attach each task to its current metered class and provide it explicit means to manipulate the individual attributes of that metered class by direct system calls.

Conclusions regarding metered classes, aka containers:

> Unlike namespaces, metered classes have rich and varied semantics, sometimes elaborate inheritance and transfer rules, and frequently non-flat topologies.
>
> Depending on the scope of visibility of a metered class, it may or may not have much of a formal name space.

===

My current understanding of Paul Menage's container patch is that it is a useful improvement for some of the metered classes - those that could make good use of a file system like hierarchy for their interface.
It probably doesn't benefit all metered classes, as they won't all benefit from a file system like hierarchy, or even have a formal name space, and it doesn't seem to benefit the name space implementation, which happily remains flat.

I could easily be wrong in this understanding.

===

For those metered classes which have system wide names, it may -seem- that attaching tasks to selected instances of those classes is much the same operation as is attaching a task to a namespace. Perhaps the sense that this was so has been the driving force behind trying to unite namespaces and containers.

However, as I've described above, these seem rather different to me.
The underlying semantics, topology, and variety are different, and
some of these differences are necessarily exposed at the point that
we attach tasks to namespaces or containers.

Moreover, each of these namespaces and each of these metered classes
typically has its own dedicated API across the kernel-user boundary,
so sharing of kernel implementation internals is mostly just a
private matter for the kernel.

Conclusions:

We're discussing kernel internal optimizations here, not
user visible API issues. As Serge keeps reminding us, this is
just an optimization.

I tend to favor keeping spaces and containers relatively separate,
minimizing the entwining of them to where there is a clear
performance (space, time, or scaling) win, and to where that
optimization doesn't depend on trying to force fit either spaces
or containers into the mold of the other.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401