Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Srivatsa Vaddagiri on Thu, 08 Mar 2007 10:13:47 GMT
View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 08:12:00PM -0700, Eric W. Biederman wrote:
> The review is still largely happening at the why level but no
> one is addressing that yet. So please can we have a why.

Here's a brief summary of what's happening and why. If its not clear, pls get back to us with specific questions.

There have been various projects attempting to provide resource management support in Linux, including CKRM/Resource Groups and UBC. Each had its own task-grouping mechanism.

Paul Menage observed [1] that cpusets in the kernel already has a grouping mechanism which was working well for cpusets. He went ahead and generalized the grouping code in cpusets so that it could be used for overall resource management purpose. With his patches, it is possible to even create multiple hierarchies of groups (see [2] on why multiple hierarchies) as follows:

```
mount -t container -o cpuset none /dev/cpuset <- cpuset hierarchy
mount -t container -o mem,cpu none /dev/mem <- memory/cpu hierarchy
mount -t container -o disk none /dev/disk <- disk hierarchy
```

In each hierarchy, you can create task groups and manipulate the resource parameters of each group. You can also move tasks between groups at run-time (see [3] on why this is required). Each hierarchy is also manipulated independent of the other.

Paul's patches also introduced a 'struct container' in the kernel, which serves these key purposes:

- Task-grouping
 'struct container' represents a task-group created in each hierarchy.
 So every directory created under /dev/cpuset or /dev/mem above will
 have a corresponding 'struct container' inside the kernel.
 All tasks pointing to the same 'struct container' are considered
 to be part of a group

 The 'struct container' in turn has pointers to resource objects
 which store actual resource parameters for that group. In above
 example, 'struct container' created under /dev/cpuset will have a
 pointer to 'struct cpuset' while 'struct container' created
 under /dev/disk will have pointer to 'struct disk_quota_or_whatever'.

- Maintain hierarchical information

The 'struct container' also keeps track of hierarchical relationship
between groups.

The filesystem interface in the patches essentially serves these
purposes:

- Provide an interface to manipulate task-groups. This includes
  creating/deleting groups, listing tasks present in a group and
  moving tasks across groups

- Provdes an interface to manipulate the resource objects
  (limits etc) pointed to by 'struct container'.

As you know, the introduction of 'struct container' was objected to and
was felt redundant as a means to group tasks. Thats where I took a shot
at converting over Paul Menage's patch to avoid 'struct container'
abstraction and insead work with 'struct nsproxy'. In the rcfs patch,
each directory (in /dev/cpuset or /dev/disk) is associated with a
'struct nsproxy' instead. The most important need of the filesystem
interface is not to manipulate the nsproxy objects directly, but to manipulate
the resource objects (nsproxy->ctlr_data[] in the patches) which store
information like limit etc.

> I have a question? What does rcfs look like if we start with
> the code that is in the kernel? That is start with namespaces
> and nsproxy and just build a filesystem to display/manipulate them?
> With the code built so it will support adding resource controllers
> when they are ready?

If I am not mistaken, Serge did attempt something in that direction,
only that it was based on Paul's container patches. rcfs can no doubt
support the same feature.

```
> >   struct ipc_namespace *ipc_ns;
> >   struct mnt_namespace *mnt_ns;
> >   struct pid_namespace *pid_ns;
> > +#ifdef CONFIG_RCFS
> > + struct list_head list;
>
```
> This extra list of nsproxy's is unneeded and a performance problem the
> way it is used. In general we want to talk about the individual resource
> controllers not the nsproxy.

I think if you consider the multiple hierarchy picture, the need becomes
obvious.

Lets say that you had these hierarchies : /dev/cpuset, /dev/mem, /dev/disk
and the various resource classes (task-groups) under them as below:

 /dev/cpuset/C1, /dev/cpuset/C1/C11, /dev/cpuset/C2
 /dev/mem/M1, /dev/mem/M2, /dev/mem/M3
 /dev/disk/D1, /dev/disk/D2, /dev/disk/D3

The nsproxy structure basically has pointers to a resource objects in
each of these hierarchies.

 nsproxy { ..., C1, M1, D1} could be one nsproxy
 nsproxy { ..., C1, M2, D3} could be another nsproxy and so on

So you see, because of multi-hierachies, we can have different
combinations of resource classes.

When we support task movement across resource classes, we need to find a
nsproxy which has the right combination of resource classes that the
task's nsproxy can be hooked to.

That's where we need the nsproxy list. Hope this makes it clear.

> > + void *ctlr_data[CONFIG_MAX_RC_SUBSYS];
>
> I still don't understand why these pointers are so abstract,
> and why we need an array lookup into them?

we can avoid these abstract pointers and instead have a set of pointers
like this:

```
 struct nsproxy {
  ...
  struct cpu_limit *cpu; /* cpu control namespace */
  struct rss_limit *rss; /* rss control namespace */
  struct cpuset *cs; /* cpuset namespace */

 }
```

But that will make some code (like searching for a right nsproxy when a
task moves across classes/groups) very awkward.

> I'm still inclined to think this should be part of /proc, instead of a purely
> separate fs. But I might be missing something.

A separate filesystem would give us more flexibility like the
implementing multi-hierarchy support described above.

--
Regards,
vatsa

References:

1. http://lkml.org/lkml/2006/09/20/200
2. http://lkml.org/lkml/2006/11/6/95
3. http://lkml.org/lkml/2006/09/5/178