Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Paul Menage on Tue, 13 May 2008 21:17:29 GMT
View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 1:01 PM, Andrew Morton
<akpm@linux-foundation.org> wrote:
>
> This, umm, doesn't seem to do much to make the kernel a simpler place.
>
> Do we expect to gain much from this? Hope so... What?
>

The goal is to prevent cgroup_mutex becoming a BKL for cgroups, to make it easier for subsystems to lock just the bits that they need to remain stable rather than everything.
>
> Vague handwaving: lockdep doesn't know anything about any of this.
> Whereas if we were more conventional in using separate locks and
> suitable lock types for each application, we would retain full lockdep
> coverage.

That's a good point - I'd not thought about lockdep. That's a good argument in favour of not having the locking done in the framework.

Stepping back a bit, the idea is definitely that where appropriate subsystems will use their own fine-grained locking. E.g. the res_counter abstraction does this already with a spinlock in each res_counter, and cpusets has the global callback_mutex that just synchronizes cpuset operations. But there are some cases where they need a bit of help from cgroups, such as when doing operations that require stable hierarchies, task membership of cgroups, etc.

Right now the default behaviour is to call cgroup_lock(), which I'd like to get away from. Having the framework do the locking results in less need for cleanup code in the subsystem handlers themselves, but that's not an unassailable argument for doing it that way.

This is one of those times that I really long for C++ scope-based destructors, to automatically release locks when you exit a scope.

```
> > +     ssize_t max_bytes = sizeof(static_buffer) - 1;
> > +     if (!cft->write && !cft->trigger) {
> > +             if (!nbytes)
> > +                     return -EINVAL;
> > +             if (nbytes >= max_bytes)
> > +                     return -E2BIG;
> > +             if (nbytes >= sizeof(static_buffer)) {
```

>
> afaict this can't happen - we would have already returned -E2BIG?

You're right, with this patch it can't. The code that makes it necessary to allocate a dynamic buffer gets added in the write_string patch, so I'll move it there. (These two were originally one patch and I guess I did a poor job of splitting them).

>
>
> > +   /* +1 for nul-terminator */
> > +   buffer = kmalloc(nbytes + 1, GFP_KERNEL);
> > +   if (buffer == NULL)
> > +   return -ENOMEM;
> > +   }
> > +   if (copy_from_user(buffer, userbuf, nbytes)) {
> > +   retval = -EFAULT;
> > +   goto out_free;
> > +   }
> > +   buffer[nbytes] = 0;   /* nul-terminate */
> > +   strstrip(buffer);   /* strip -just- trailing whitespace */
> >   }
> > -   return -EINVAL;
> > -}
>
> I'm trying to work out what protects static_buffer?
>
> Why does it need to be static anyway? 64 bytes on-stack is OK.
>

As Matt observed, this is just a poorly-named variable. How about "small_buffer"?

Paul