Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Sat, 06 Oct 2007 19:59:04 GMT
View Forum Message <> Reply to Message

David wrote:
> It would probably be better to just save references to the tasks.
>
> struct cgroup_iter it;
> struct task_struct *p, **tasks;
> int i = 0;
>
> cgroup_iter_start(cs->css.cgroup, &it);
> while ((p = cgroup_iter_next(cs->css.cgroup, &it))) {
>   get_task_struct(p);
>   tasks[i++] = p;
> }
> cgroup_iter_end(cs->css.cgroup, &it);

Hmmm ... guess I'd have to loop over the cgroup twice, once to count them (the 'count' field is not claimed to be accurate) and then again, after I've kmalloc'd the tasks[] array, filling in the tasks[] array.

On a big cgroup on a big system, this could easily be thousands of iteration loops.

And I've have to drop the css_set_lock spinlock between the two loops, since I can't hold a spinlock while calling kmalloc.

So then I'd have to be prepared for the possibility that the second loop found more cgroups on the list than what I counted in the first loop.

This is doable ... indeed I've done such before, in the code that is now known as kernel/cgroup.c:cgroup_tasks_open(). Look for how pidarray[] is setup.

And note that that code doesn't deal with the case that more cgroups showed up after they were counted. When supporting the reading of the 'tasks' file by user code, this is ok - it's inherently racey anyway - so not worth trying too hard just to close the window part way.

If I need to close the window all the way, completely solving the race condition, then I have the code in kernel/cpuset.c:update_nodemask(), which builds an mmarray[] using two loops and some retries if newly forked tasks are showing up too rapidly at the same time. The first of the two loops is hidden in the cgroup_task_count() call.

That's a bunch of code, mate. If some other solution was adequate

(no worse than the current situation, which forces user space to
rewrite every pid in the tasks file back to itself if they want
a 'cpus' change to actually be applied) but took much less code,
then I'd have to give it serious consideration, as I did before.

I don't mind a bunch of code, but kernel text has to earn its keep.
I'm not yet convinced that the above page or two of somewhat fussy code
(see again the code in kernel/cpuset.c:update_nodemask() ...) has
sufficient real user value per byte of kernel text space to justify its
existence.

... by the way ... tell me again why css_set_lock is a spinlock?

> I didn't think it was such a good idea to hold a spinlock while
> iterating over a major list, doing lord knows what (the loops
> over cgroup_iter_next() do user provided code, as in this case.)
> Shouldn't that be a mutex?

Or, if there is a good reason that must remain a spinlock, then the
smallest amount of new code, and the easiest code to write, would
perhaps be adding another cgroup callback, called only by cgroup attach
() requests back to the same group. Then code that wants to do
something odd, such as cpusets, for what seems like a no-op, can do so.

--

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