Subject: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Wed, 03 Oct 2007 08:42:41 GMT
View Forum Message <> Reply to Message

From: Paul Jackson <pj@sgi.com>

The code in kernel/cgroup.c attach_task() which skips the
attachment of a task to the group it is already in has to be
removed. Cpusets depends on reattaching a task to its current
cpuset, in order to trigger updating the cpus_allowed mask in the
task struct.

The dependency of cpusets on this is a hack, granted, but an
important one. It lets us avoid checking for a changed cpuset
'cpus' setting in critical scheduler code paths.

Signed-off-by: Paul Jackson <pj@sgi.com>
Cc: Paul Menage <menage@google.com>

Andrew - this patch applies directly following the patch:
    task-containersv11-add-tasks-file-interface.patch

 kernel/cgroup.c | 3 ---
 1 file changed, 3 deletions(-)

--- 2.6.23-rc8-mm1.orig/kernel/cgroup.c 2007-10-02 20:24:11.078925442 -0700
+++ 2.6.23-rc8-mm1/kernel/cgroup.c 2007-10-02 20:25:41.352279374 -0700
@@ -739,10 +739,7 @@ static int attach_task(struct cgroup *co

  get_first_subsys(cont, NULL, &subsys_id);

- /* Nothing to do if the task is already in that cgroup */
  oldcont = task_cgroup(tsk, subsys_id);
- if (cont == oldcont)
- return 0;

  for_each_subsys(root, ss) {
   if (ss->can_attach) {

--
                I won't rest till it's the best ...
                Programmer, Linux Scalability
                Paul Jackson <pj@sgi.com> 1.650.933.1373

_____
Containers mailing list
Containers@lists.linux-foundation.org

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Wed, 03 Oct 2007 15:51:41 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Jackson <pj@sgi.com> wrote:
> From: Paul Jackson <pj@sgi.com>
>
> The code in kernel/cgroup.c attach_task() which skips the
> attachment of a task to the group it is already in has to be
> removed. Cpusets depends on reattaching a task to its current
> cpuset, in order to trigger updating the cpus_allowed mask in the
> task struct.

Can you explain a bit more about why this is needed? (i.e. specific scenarios where cpusets will break without this change).

What triggers the reattach in normal use? Something from userspace?

Paul

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Wed, 03 Oct 2007 17:58:17 GMT
View Forum Message <> Reply to Message

Paul M wrote:
> > The code in kernel/cgroup.c attach_task() which skips the
> > attachment of a task to the group it is already in has to be
> > removed. Cpusets depends on reattaching a task to its current
> > cpuset, in order to trigger updating the cpus_allowed mask in the
> > task struct.
>
> Can you explain a bit more about why this is needed? (i.e. specific
> scenarios where cpusets will break without this change).
>
> What triggers the reattach in normal use? Something from userspace?

Yes, something in user space has to do it. It's part of the
kernel-user cpuset API. If you change a cpuset's 'cpus', then
you have to rewrite each pid in its 'tasks' file back to that

'tasks' file in order to get that 'cpus' change to be applied
to the task struct cpus_allowed of each task, and thereby visible
to the scheduler.

--

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

_____

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Wed, 03 Oct 2007 18:10:58 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Jackson <pj@sgi.com> wrote:
>
> Yes, something in user space has to do it. It's part of the
> kernel-user cpuset API. If you change a cpuset's 'cpus', then
> you have to rewrite each pid in its 'tasks' file back to that
> 'tasks' file in order to get that 'cpus' change to be applied
> to the task struct cpus_allowed of each task, and thereby visible
> to the scheduler.

What's the rationale for this?

Given that later in cpusets.txt you say:

>If hotplug functionality is used
>to remove all the CPUs that are currently assigned to a cpuset,
>then the kernel will automatically update the cpus_allowed of all
>tasks attached to CPUs in that cpuset to allow all CPUs

why can't the same thing be done when changing the 'cpus' file manually.

What's wrong with, in update_cpumask(), doing a loop across all
members of the cgroup and updating their cpus_allowed fields?

The existing cpusets API is broken, since a new child can always be
forked between reading the tasks file and doing the writes.

Paul

_____

Containers mailing list
Containers@lists.linux-foundation.org

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Wed, 03 Oct 2007 18:25:21 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Menage <menage@google.com> wrote:
>
> What's wrong with, in update_cpumask(), doing a loop across all
> members of the cgroup and updating their cpus_allowed fields?

i.e. something like:

```
struct cgroup_iter it;
struct task_struct *p;
while ((p = cgroup_iter_next(cs->css.cgroup, &it))) {
 set_cpus_allowed(p, cs->cpus_allowed);
}
cgroup_iter_end(cs->css.cgroup, &it);
```

Paul

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

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Wed, 03 Oct 2007 20:16:32 GMT
View Forum Message <> Reply to Message

Paul M wrote:
> Given that later in cpusets.txt you say:
>
> >If hotplug functionality is used
> >to remove all the CPUs that are currently assigned to a cpuset,
> >then the kernel will automatically update the cpus_allowed of all
> >tasks attached to CPUs in that cpuset to allow all CPUs
>
> why can't the same thing be done when changing the 'cpus' file manually.
>
> What's wrong with, in update_cpumask(), doing a loop across all
> members of the cgroup and updating their cpus_allowed fields?
>
> The existing cpusets API is broken, since a new child can always be
> forked between reading the tasks file and doing the writes.

Hmmm ... interesting. Yes - that race, between fork and rewriting
the tasks file is there, by design, and has been there for years.

So far as I know, it is a benign race. In practice, no one that I know
has tripped over it. This doesn't imply that we should not look for
opportunities to fix it, but it does mean it is not an urgent fix, in
my experience.

But there may be more issues here than that:
 1) The above cpusets.txt text might be both incorrect (at first
    glance now, I suspect that it is the cpuset cpus_allowed that
    is updated on hotplug events, not the tasks cpus_allowed) and
    perhaps stale as well (given some recent cpuset hotplug patches
    being worked by Cliff Wickman.)
 2) It is not immediately clear to me how the hotplug can work at
    present - if it updates just the cpusets cpus_allowed but doesn't
    get to the tasks cpus_allowed.
 3) So far as I can recall, the only kernel code path to update a
    tasks cpus_allowed goes via the "set_cpus_allowed()" routine,
    which the kernel/cpuset.c code only calls from one place, that
    being the code that handles writing a pid to a cpuset tasks file.
    This provides more evidence that (2) above is a problem -- no
    hotplug code paths in the cpuset code seem to invoke this key
    "set_cpus_allowed()" routine.

Vedy vedy interesting. Like lifting up a rock and seeing a bunch
of weird creepy crawlies under it.

I'm wondering how to proceed from here, in two regards:

 1) The cc list on this thread is long and not well focused for the
    detailed discussion that it may take to unravel this.
 2) I am not optimistic that we can unravel this in time for the
    rapidly approaching start of 2.6.24.

At present, the only one of the bugs or potential bugs noted here
that are actually biting me is the bug this patch addressed; the
current (yes, knowingly broken) cpuset code requires the attach
code to be run even when reattaching to the same cgroup/cpuset to
which the task is already attached.

If the 2.6.24 kernel releases with this bug, such that reattaching
tasks to their current cpuset becomes a no-op, that will hurt.

Does my patch (the one opening this thread) cause any problems
that you know of, Paul M?

So far as I can tell, this patch just removes a minor optimization, and other than its allowing a hack in the cpuset code to persist (a side affect of what logically should be a no-op), this patch should be unnoticeable to user land.

If that's so, can we allow this patch to proceed, likely for the 2.6.24 kernel series, while we work on killing off all the creepy crawlies that your good questions have exposed?

And I'm tempted to continue this discussion on a new thread, as it seems that it could be an involved discussion not well focused on the interests of many on the current cc list of this message. There may be more hotplug interest, and less container interest, than currently seen on this threads cc list.

Any further suggestions, or embarrassing (;) questions?

Thanks!

--

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

_____

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Wed, 03 Oct 2007 20:31:17 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Jackson <pj@sgi.com> wrote:
>
> So far as I can tell, this patch just removes a minor optimization,

It's not minor for any subsystem that has a non-trivial attach() operation.

>
> Any further suggestions, or embarrassing (;) questions?
>

What was wrong with my suggestion from a couple of emails back? Adding the following in cpuset_attach():

struct cgroup_iter it;
struct task_struct *p;

```
while ((p = cgroup_iter_next(cs->css.cgroup, &it))) {
      set_cpus_allowed(p, cs->cpus_allowed);
}
cgroup_iter_end(cs->css.cgroup, &it);
```

Paul

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Wed, 03 Oct 2007 20:52:32 GMT
View Forum Message <> Reply to Message

> What was wrong with my suggestion from a couple of emails back? Adding
> the following in cpuset_attach():
>
> struct cgroup_iter it;
> struct task_struct *p;
> while ((p = cgroup_iter_next(cs->css.cgroup, &it))) {
>   set_cpus_allowed(p, cs->cpus_allowed);
> }
> cgroup_iter_end(cs->css.cgroup, &it);

Hmmm ... that just might work.

And this brings to light the reason (justification, excuse, whatever
you call it) that I probably didn't do this earlier.

In the dark ages before cgroups (aka containers) we did not have an
efficient way to walk the tasks in a cpuset. One had to walk the entire
task list, comparing task struct cpuset pointers. On big honking NUMA
iron, one should avoid task list walks as much as one can get away
with, even if it meant sneaking in a little bit racey API. Since some
updates of a cpusets 'cpus' mask don't need it (you happen to know that
all tasks in that cpuset are pause'd anyway) I might have made the
tradeoff to make this task list walk an explicitly invoked user action,
to be done only when needed.

But now (correct me if I'm wrong here) cgroups has a per-cgroup task
list, and the above loop has cost linear in the number of tasks
actually in the cgroup, plus (unfortunate but necessary and tolerable)
the cost of taking a global css_set_lock, right?

And I take it the above code snipped is missing the cgroup_iter_start,
correct?

I'll ask Andrew to kill this patch of mine, and I will test out your
suggestion this evening.

This still leaves the other creepy crawlies involving cpusets and hot
plug that I glimpsed slithering by in my last message. I guess I'll
start a separate discussion with Cliff Wickman and whomever else I think
might want to be involved on those issues.

Nice work - thanks.

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

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Wed, 03 Oct 2007 20:56:15 GMT
View Forum Message <> Reply to Message

Andrew - please kill this patch.

Looks like Paul Menage has a better solution
that I will be trying out.

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

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Wed, 03 Oct 2007 20:58:26 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Jackson <pj@sgi.com> wrote:
>
> But now (correct me if I'm wrong here) cgroups has a per-cgroup task

> list, and the above loop has cost linear in the number of tasks
> actually in the cgroup, plus (unfortunate but necessary and tolerable)
> the cost of taking a global css_set_lock, right?

Yes.

>
> And I take it the above code snipped is missing the cgroup_iter_start,
> correct?

Oops, yes.

Paul

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Sat, 06 Oct 2007 08:24:37 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> What was wrong with my suggestion from a couple of emails back? Adding
> the following in cpuset_attach():
>
> struct cgroup_iter it;
> struct task_struct *p;
> cgroup_iter_start(cs->css.cgroup, &it);
> while ((p = cgroup_iter_next(cs->css.cgroup, &it)))
> set_cpus_allowed(p, cs->cpus_allowed);
> cgroup_iter_end(cs->css.cgroup, &it);

This isn't working for me.

The key kernel routine for updating a tasks cpus_allowed
cannot be called while holding a spinlock.

But the above loop holds a spinlock, css_set_lock, between
the cgroup_iter_start and the cgroup_iter_end.

I end up generating complaints of:
 BUG: scheduling while atomic
when I invoke the set_cpus_allowed() above.

Should css_set_lock be a mutex? Locking changes like that
can be risky.

Or perhaps there should be another callback, called only by
attach() requests back to the same group. Likely cpusets would
be the only subsystem interested in plugging that callback.

That, or my original patch, which calls the attach routine
even if re-attaching to the current cgroup ...

--

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

_____

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by David Rientjes on Sat, 06 Oct 2007 17:54:35 GMT
View Forum Message <> Reply to Message

On Sat, 6 Oct 2007, Paul Jackson wrote:

> This isn't working for me.
>
> The key kernel routine for updating a tasks cpus_allowed
> cannot be called while holding a spinlock.
>
> But the above loop holds a spinlock, css_set_lock, between
> the cgroup_iter_start and the cgroup_iter_end.
>
> I end up generating complaints of:
> BUG: scheduling while atomic
> when I invoke the set_cpus_allowed() above.
>
> Should css_set_lock be a mutex? Locking changes like that
> can be risky.
>

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);

while (--i >= 0) {
 set_cpus_allowed(tasks[i], cs->cpus_allowed);
 put_task_struct(tasks[i]);
}
```

The getting and putting of the tasks will prevent them from exiting or being deallocated prematurely. But this is also a critical section that will need to be protected by some mutex so it doesn't race with other set_cpus_allowed().

David

_____

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

---

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

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Sat, 06 Oct 2007 20:53:53 GMT
View Forum Message <> Reply to Message

On 10/6/07, Paul Jackson <pj@sgi.com> wrote:
>
> This isn't working for me.
>
> The key kernel routine for updating a tasks cpus_allowed
> cannot be called while holding a spinlock.
>
> But the above loop holds a spinlock, css_set_lock, between
> the cgroup_iter_start and the cgroup_iter_end.
>
> I end up generating complaints of:
> BUG: scheduling while atomic
> when I invoke the set_cpus_allowed() above.
>
> Should css_set_lock be a mutex? Locking changes like that
> can be risky.

css_set_lock is an rwlock currently; I'd rather not turn it into an rw mutex since there are places that it gets taken where we can't afford to sleep.

>
> Or perhaps there should be another callback, called only by
> attach() requests back to the same group. Likely cpusets would
> be the only subsystem interested in plugging that callback.
>
> That, or my original patch, which calls the attach routine
> even if re-attaching to the current cgroup ...

I'd prefer David's solution of grabbing references to tasks during the iteration and then doing set_cpus_allowed outside the tasklist_lock.

Paul

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Sat, 06 Oct 2007 21:09:52 GMT
View Forum Message <> Reply to Message

On 10/6/07, Paul Jackson <pj@sgi.com> wrote:
> 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.

But if userspace has to do it, the effect will be far more expensive.

>
> 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.

In general, the loop inside cgroup_task_count() will only have a
single iteration. (It's iterating across the shared css_set objects,
not across member tasks.

What's wrong with:

```
  allocate a page of task_struct pointers
again:
  need_repeat = false;
  cgroup_iter_start();
```

```
 while (cgroup_iter_next()) {
   if (p->cpus_allowed != new_cpumask) {
     store p;
     if (page is full) {
       need_repeat = true;
       break;
     }
   }
 }
 for each saved task p {
   set_cpus_allowed(p, new_cpumask);
   release p;
 }
 if (need_repeat)
   goto again;
```

Advantages:

- no vmalloc needed
- just one iteration in the case where a cgroup has fewer than 512 members
- additional iterations only need to deal with tasks that don't have
the right cpu mask
- automatically handles fork races

Another option would be to have a cpuset fork callback that forces
p->cpus_allowed to its cpuset's cpus_allowed if another thread is in
the middle of update_cpumask(). Then you don't need to worry about
fork races at all, since all new threads will get the right cpumask.

> 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'd much rather not perpetuate that broken API requirement. The fact
that cpusets wants this odd behaviour is based on a nasty hack.

Paul

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets

Posted by Paul Menage on Sat, 06 Oct 2007 21:11:43 GMT
View Forum Message <> Reply to Message

On 10/6/07, David Rientjes <rientjes@google.com> wrote:
> The getting and putting of the tasks will prevent them from exiting or
> being deallocated prematurely. But this is also a critical section that
> will need to be protected by some mutex so it doesn't race with other
> set_cpus_allowed().

Is that necessary? If some other process calls set_cpus_allowed()
concurrently with a cpuset cpus update, it's not clear that there's
any defined serialization semantics that have to be achieved, as long
as the end result is that the task's cpus_allowed are within the
cpuset's cpus_allowed.

Paul

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

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Sat, 06 Oct 2007 21:41:57 GMT
View Forum Message <> Reply to Message

Paul M wrote:
>
> What's wrong with:
>
> allocate a page of task_struct pointers
> again:
> need_repeat = false;
> cgroup_iter_start();
> while (cgroup_iter_next()) {
> if (p->cpus_allowed != new_cpumask) {
> store p;
> if (page is full) {
> need_repeat = true;
> break;
> }
> }
> }
> for each saved task p {
> set_cpus_allowed(p, new_cpumask);
> release p;
> }
> if (need_repeat)

> goto again;

That might work ... nice idea there, comparing the two masks, so one
only needs to store the ones not yet fixed. Uunfortunately, I need to
put this aside until I return in four days, from a short trip.

Given that cgroups is targeted for 2.6.24, and that cpusets is broken
without this, I'll have to do something soon. But this, or some such,
should work, soon enough.

Thanks.

> I'd much rather not perpetuate that broken API requirement. The fact
> that cpusets wants this odd behaviour is based on a nasty hack.

Well ... yeah ... it's a bit of an ugly child. But I'm its daddy.
The kid looks fine in my eyes. <grin>

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

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by David Rientjes on Sun, 07 Oct 2007 06:13:10 GMT
View Forum Message <> Reply to Message

On Sat, 6 Oct 2007, Paul Jackson wrote:

> > 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.

>

Use the struct task_struct **tasks above. Any call here should only be
about 300 bytes into the stack so, with 8K stacks on x86_64, you should be
able to store pointers to a little under 1,000 tasks before corrupting it.

Or, if you think there's going to be more than 1,000 tasks in this cgroup,
just do an arbitrary number of them and then reuse your stack before
calling set_cpus_allowed() and put_task_struct() on all tasks you've
collected so far; then finally call cgroup_iter_end() when you're done
with all of them. Use the task_struct usage counter as your locking so
that tasks don't prematurely exit or deallocate before you migrate them.

Keep in mind that simply iterating over the tasks and using
set_cpus_allowed() isn't good enough not only because you're holding a
spinlock but also because the task can exit during the call, unless you're
using some sort of cgroups locking I'm not aware of that prevents that.
My solution prevents that with {get,put}_task_struct().

The current update_cpu_domains(), or -mm's equivalent of it with this
patchset, cannot call lock_cpu_hotplug() with callback_mutex held, so
it'll need to be unlocked after all the tasks have been put. If you're
preempted here before update_cpu_domains(), attach_task() will take care
of the cpus_allowed reassignment.

  David

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by David Rientjes on Sun, 07 Oct 2007 06:15:14 GMT
View Forum Message <> Reply to Message

On Sat, 6 Oct 2007, Paul Menage wrote:

> > The getting and putting of the tasks will prevent them from exiting or
> > being deallocated prematurely. But this is also a critical section that
> > will need to be protected by some mutex so it doesn't race with other
> > set_cpus_allowed().
>
> Is that necessary? If some other process calls set_cpus_allowed()
> concurrently with a cpuset cpus update, it's not clear that there's
> any defined serialization semantics that have to be achieved, as long
> as the end result is that the task's cpus_allowed are within the
> cpuset's cpus_allowed.

>

It can race with sched_setaffinity(). It has to give up tasklist_lock as
well to call set_cpus_allowed() and can race

```
cpus_allowed = cpuset_cpus_allowed(p);
cpus_and(new_mask, new_mask, cpus_allowed);
retval = set_cpus_allowed(p, new_mask);
```

and allow a task to have a cpu outside of the cpuset's new cpus_allowed if
you've taken it away between cpuset_cpus_allowed() and set_cpus_allowed().

David

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Menage on Wed, 10 Oct 2007 20:46:36 GMT
View Forum Message <> Reply to Message

On 10/6/07, David Rientjes <rientjes@google.com> wrote:
>
> It can race with sched_setaffinity(). It has to give up tasklist_lock as
> well to call set_cpus_allowed() and can race
>
> cpus_allowed = cpuset_cpus_allowed(p);
> cpus_and(new_mask, new_mask, cpus_allowed);
> retval = set_cpus_allowed(p, new_mask);
>
> and allow a task to have a cpu outside of the cpuset's new cpus_allowed if
> you've taken it away between cpuset_cpus_allowed() and set_cpus_allowed().

cpuset_cpus_allowed() takes callback_mutex, which is held by
update_cpumask() when it updates cs->cpus_allowed. So if we continue
to hold callback_mutex across the task update loop this wouldn't be a
race. Having said that, holding callback mutex for that long might not
be a good idea.

A cleaner solution might be to drop callback_mutex after updating
cs->cpus_allowed in update_cpumask() and then make sched_setaffinity()
do a post-check:

```
cpus_allowed = cpuset_cpus_allowed(p);
again:
cpus_and(new_mask, new_mask, cpus_allowed);
```

```
 retval = set_cpus_allowed(p, new_mask);
 if (!retval) {
  /* Check for races with cpuset updates */
  cpus_allowed = cpuset_cpus_allowed(p);
  if (!cpus_subset(new_mask, cpus_allowed)) {
   /*
    * We raced with a change to cpuset update,
    * and our cpumask is now outside the
    * permitted cpumask for the cpuset.  Since a
    * change to the cpuset's cpus resets the
    * cpumask for each task, do the same thing
    * here.
    */
   new_mask = cpus_allowed;
   goto again;
  }
 }
```

Paul

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by David Rientjes on Wed, 10 Oct 2007 20:59:44 GMT
View Forum Message <> Reply to Message

On Wed, 10 Oct 2007, Paul Menage wrote:

> On 10/6/07, David Rientjes <rientjes@google.com> wrote:
> >
> > It can race with sched_setaffinity(). It has to give up tasklist_lock as
> > well to call set_cpus_allowed() and can race
> >
> >   cpus_allowed = cpuset_cpus_allowed(p);
> >   cpus_and(new_mask, new_mask, cpus_allowed);
> >   retval = set_cpus_allowed(p, new_mask);
> >
> > and allow a task to have a cpu outside of the cpuset's new cpus_allowed if
> > you've taken it away between cpuset_cpus_allowed() and set_cpus_allowed().
>
> cpuset_cpus_allowed() takes callback_mutex, which is held by
> update_cpumask() when it updates cs->cpus_allowed. So if we continue
> to hold callback_mutex across the task update loop this wouldn't be a
> race. Having said that, holding callback mutex for that long might not
> be a good idea.

>

Moving the actual use of set_cpus_allowed() from the cgroup code to sched.c would probably be the best in terms of a clean interface. Then you could protect the whole thing by sched_hotcpu_mutex, which is expressly designed for migrations.

Something like this:

```
 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);

 while (--i >= 0) {
  sched_migrate_task(tasks[i], cs->cpus_allowed);
  put_task_struct(tasks[i]);
 }
```

kernel/sched.c:

```
 void sched_migrate_task(struct task_struct *task,
    cpumask_t cpus_allowed)
 {
  mutex_lock(&sched_hotcpu_mutex);
  set_cpus_allowed(task, cpus_allowed);
  mutex_unlock(&sched_hotcpu_mutex);
 }
```

It'd be faster to just pass **tasks to a sched.c function with the number of tasks to migrate to reduce the contention on sched_hotcpu_mutex, but then the put_task_struct() is left dangling over in cgroup code. sched_hotcpu_mutex should be rarely contended, anyway.

David

_____

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

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Thu, 11 Oct 2007 22:03:15 GMT
View Forum Message <> Reply to Message

Several days ago, Paul M replied to Paul J:
> > 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.
>
> But if userspace has to do it, the effect will be far more expensive.

Ah - but I'm not trying to optimize this particular operation (changing
a cpusets 'cpus'). It's not at all performance critical.

I'm trying to minimize the amount of special purpose code in the
kernel.

The maintenance costs of a line of kernel code are quite a bit higher
than for a line of user code. I work hard to have most of my lines of
kernel code be on well traveled code paths, of general usefulness, even
if this means that some infrequent operations require yet more user
source code lines and user CPU cycles, in order to be refactored as the
combination of multiple system call primitives.

... all within reasonable limits, of course.

Corner case, special situation, non-trivial chunks of kernel code are
very expensive. They don't get very good testing coverage in the
real world, and end up harboring latent bugs for months or years,
by which time it can be expensive to deal with them.

Be that as it may, I've just started digesting the actual code
suggestions posted by yourself and David (thanks!) this last week.
I just couldn't resist a bit of philosophizing ... sorry.

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

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

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Thu, 11 Oct 2007 23:15:59 GMT
View Forum Message <> Reply to Message

David wrote:
> you could protect the whole thing by sched_hotcpu_mutex, which is
> expressly designed for migrations.
>
> Something like this:
>
> 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);
>
> while (--i >= 0) {
>   sched_migrate_task(tasks[i], cs->cpus_allowed);
>   put_task_struct(tasks[i]);
> }
>
> kernel/sched.c:
>
> void sched_migrate_task(struct task_struct *task,
>   cpumask_t cpus_allowed)
> {
>   mutex_lock(&sched_hotcpu_mutex);
>   set_cpus_allowed(task, cpus_allowed);
>   mutex_unlock(&sched_hotcpu_mutex);
> }

Hmmm ... I hadn't noticed that sched_hotcpu_mutex before.

I wonder what it is guarding? As best as I can guess, it seems, at
least in part, to be keeping the following two items consistent:
 1) cpu_online_map
 2) the per-task cpus_allowed masks

That is, it seems to ensure that a task is allowed to run on some
online CPU.

If that's approximately true, then shouldn't I take sched_hotcpu_mutex
around the entire chunk of code that handles updating a cpusets 'cpus',

from the time it verifies that the requested CPUs are online, until the time that every affected task has its cpus_allowed updated?

Furthermore, I should probably guard changes to and verifications against the top_cpuset's cpus_allowed with this mutex as well, as it is supposed to be a copy of cpu_online_map.

And since all descendent cpusets have to have 'cpus' masks that are subsets of their parents, this means guarding other chunks of cpuset code that depend on the consistency of various per-cpuset cpus_allowed masks and cpu_online_map.

My current intuition is that this expanded use of sched_hotcpu_mutex in the cpuset code involving various cpus_allowed masks would be a good thing.

In sum, perhaps sched_hotcpu_mutex is guarding the dispersed kernel state that depends on what CPUs are online. This includes the per-task and per-cpuset cpus_allowed masks, all of which are supposed to be some non-empty subset of the online CPUs.

Taking and dropping the sched_hotcpu_mutex for each task, just around the call to set_cpus_allowed(), as you suggested above, doesn't seem to accomplish much that I can see, and certainly doesn't seem to guard the consistency of cpu_online_map with the tasks cpus_allowed masks.

... lurkers beware ... good chance I haven't a friggin clue ;).
In other words: Thirty minutes ago I couldn't even spell sched_hotcpu_mutex, and now I'm pontificating on it ;);).

--

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

_____

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by ebiederm on Thu, 11 Oct 2007 23:20:43 GMT
View Forum Message <> Reply to Message

Stupid question.

Would it help at all if we structured this as:
- Take the control group off of the cpus and runqueues.

- Modify the tasks in the control group.
- Place the control group back on the runqueues.

Essentially this is what ptrace does (except for one task at a time).

Since we know that the tasks are not running, and that we have
exclusive access to the tasks in the control group we can take action
as if we were the actual tasks themselves. Which should simplify
locking.

So I think with just a touch of care we can significantly simplify the
locking by being much more coarse grained, at the expense of this kind
of control group operation being more expensive. Plus we would have
an operation that was reusable.

I think the stop everything. modify everything, start everything
could also be used instead of an array of task structures.

Eric

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by Paul Jackson on Fri, 12 Oct 2007 01:23:59 GMT
View Forum Message <> Reply to Message

> Since we know that the tasks are not running, and that we have
> exclusive access to the tasks in the control group we can take action
> as if we were the actual tasks themselves. Which should simplify
> locking.

The Big Kernel Lock (BKL), born again, as a Medium Sized Cgroup Lock ?

This only simplifies things if it enables us to remove finer grain
locking, but some finer grain locking is essential for performance on
higher processor count systems (which if Intel and AMD have their way,
will be just about any system bigger than a cell phone.)

There is no escaping actually having to think about these things, and
clearly understand and document what locks what. Locks don't just
guard code sections, and they don't just guard particular data items.

Rather, in my view, they ensure certain invariants on your data.
Non-atomic code sections that depend on, or modify, data subject to
such invariants must be done while holding the appropriate lock.

One is guaranteed not to see the invariant violated while holding
the lock, nor to expose others to temporary violations of the invariant
done while locked.

In this case, we have multiple copies of cpumasks in task structs and
cpusets that must honor the invariant that they are equal or subsets
of, cpu_online_map. Changes to the set of online CPUs must hold some
lock, apparently sched_hotcpu_mutex, until all those cpumasks are
adjusted to once again honor, this invariant.

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

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

---

Subject: Re: [PATCH] task containersv11 add tasks file interface fix for cpusets
Posted by David Rientjes on Fri, 12 Oct 2007 15:13:35 GMT
View Forum Message <> Reply to Message

On Thu, 11 Oct 2007, Paul Jackson wrote:

> Hmmm ... I hadn't noticed that sched_hotcpu_mutex before.
>
> I wonder what it is guarding? As best as I can guess, it seems, at
> least in part, to be keeping the following two items consistent:
> 1) cpu_online_map

Yes, it protects against cpu hot-plug or hot-unplug; cpu_online_map is
guaranteed to be unchanged while the mutex is being held.

> 2) the per-task cpus_allowed masks
>

It doesn't need to protect the per-task cpus_allowed per se, that's
already protected. If a task's cpu affinity changes during a call to
set_cpus_allowed(), the migration thread will notice the change when it
tries to deactive the task and activate it on the destination cpu. It
then becomes a no-op.

That's a consequence of the fact that we can't migrate current and need a
kthread, particularly the source cpu's runqueue migration thread, to do it
when it's scheduled. A migration request such as that includes a
completion variable so that the set_cpus_allowed() waits until it has

either been migrated or changed cpu affinity again.

> That is, it seems to ensure that a task is allowed to run on some
> online CPU.
>

Right, the destination cpu will not be hot-unplugged out from underneath
the task during migration.

> If that's approximately true, then shouldn't I take sched_hotcpu_mutex
> around the entire chunk of code that handles updating a cpusets 'cpus',
> from the time it verifies that the requested CPUs are online, until the
> time that every affected task has its cpus_allowed updated?
>

Not necessarily, you can iterate through a list of tasks and change their
cpu affinity (represented by task->cpus_allowed) by migrating them away
while task->cpuset->cpus_allowed remains unchanged. The hotcpu notifier
cpuset_handle_cpuhp() will update that when necessary for cpu hot-plug or
hot-unplug events.

So it's entirely possible that a cpu will be downed during your iteration
of tasks, but that's fine. Just as long as it isn't downed during the
migration. The cpuset's cpus_allowed will be updated by the hotcpu
notifier and sched_hotcpu_mutex will protect from unplugged cpus around
the set_cpus_allowed() call, which checks for intersection between your
new cpumask and cpu_online_map.

> Furthermore, I should probably guard changes to and verifications
> against the top_cpuset's cpus_allowed with this mutex as well, as it is
> supposed to be a copy of cpu_online_map.
>

The hotcpu notifier protects you there as well.
common_cpu_mem_hotplug_unplug() explicitly sets them.

> And since all descendent cpusets have to have 'cpus' masks that are
> subsets of their parents, this means guarding other chunks of cpuset
> code that depend on the consistency of various per-cpuset cpus_allowed
> masks and cpu_online_map.
>

Same as above, except now you're using
guarantee_online_cpus_mems_in_subtree().

> My current intuition is that this expanded use of sched_hotcpu_mutex in
> the cpuset code involving various cpus_allowed masks would be a good
> thing.

>
> In sum, perhaps sched_hotcpu_mutex is guarding the dispersed kernel
> state that depends on what CPUs are online. This includes the per-task
> and per-cpuset cpus_allowed masks, all of which are supposed to be some
> non-empty subset of the online CPUs.
>

It guards cpu_online_map from being changed while it's held.

> Taking and dropping the sched_hotcpu_mutex for each task, just around
> the call to set_cpus_allowed(), as you suggested above, doesn't seem to
> accomplish much that I can see, and certainly doesn't seem to guard the
> consistency of cpu_online_map with the tasks cpus_allowed masks.
>

It's needed to serialize with other migrations such as sched_setaffinity()
and you can use it since all migrations will inherently need this type of
protection. It makes the new cpumask consistent with cpu_online_map only
so far as that it's a subset; otherwise, set_cpus_allowed() will fail.
The particular destination cpu is chosen as any online cpu, which we know
won't be downed because we're holding sched_hotcpu_mutex.

  David