

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Matt Helsley](#) on Sat, 12 Jul 2008 00:03:54 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Wed, 2008-07-09 at 23:46 -0700, Paul Menage wrote:

> This is an initial design for a transactional task attachment  
> framework for cgroups. There are probably some potential simplifications  
> that I've missed, particularly in the area of locking. Comments  
> appreciated.

>  
> The Problem  
> =====

>  
> Currently cgroups task movements are done in three phases

>  
> 1) all subsystems in the destination cgroup get the chance to veto the  
> movement (via their can\_attach()) callback  
> 2) task->cgroups is updated (while holding task->alloc\_lock)  
> 3) all subsystems get an attach() callback to let them perform any  
> housekeeping updates

>  
> The problems with this include:

>  
> - There's no way to ensure that the result of can\_attach() remains  
> valid up until the attach() callback, unless any invalidating  
> operations call cgroup\_lock() to synchronize with the change. This is  
> fine for something like cpusets, where invalidating operations are  
> rare slow events like the user removing all cpus from a cpuset, or cpu  
> hotplug triggering removal of a cpuset's last cpu. It's not so good  
> for the virtual address space controller where the can\_attach() check  
> might be that the res\_counter has enough space, and an invalidating  
> operation might be another task in the cgroup allocating another page  
> of virtual address space.

>  
> - It doesn't handle the case of the proposed "cgroup.procs" file which  
> can move multiple threads into a cgroup in one operation; the  
> can\_attach() and attach() calls should be able to atomically allow all  
> or none of the threads to move.

>  
> - it can create races around the time of the movement regarding to  
> which cgroup a resource charge/uncharge should be assigned (e.g.  
> between steps 2 and 3 new resource usage will be charged to the  
> destination cgroup, but step 3 might involve migrating a charge equal  
> to the task's resource usage from the old cgroup to the new, resulting  
> in over/under-charges.

I agree with Balbir: this is a good description of the problems.

>  
> Conceptual solution  
> =====  
>  
> In ideal terms, a solution for this problem would meet the following  
> requirements:  
>  
> - support movement of an arbitrary set of threads between an arbitrary  
> set of cgroups  
>  
> - allow arbitrarily complex locking from the subsystems involved so  
> that they can synchronize against concurrent changes, etc  
> - allow rollback at any point in the process  
>  
> But in practice that would probably be way more complex than we'd want  
> in the kernel. We don't want to encourage excessively-complex locking  
> from subsystems, and we don't need to support arbitrary task  
> movements.  
>  
>  
> (Hopefully!) Practical solution  
> =====  
>  
> So here's a more practical solution, which hopefully catches the  
> important parts of the requirements without being quite so complex.  
>  
> The restrictions are:  
>  
> - only supporting movement to one destination cgroup (in the same  
> hierarchy, of course); if an entire process is being moved, then  
> potentially its threads could be coming from different source cgroups  
> - a subsystem may optionally fail such an attach if it can't handle  
> the synchronization this would entail.  
>  
> - supporting moving either one thread, one entire thread group or (for  
> the future) "all threads". This supports the existing "tasks" file,  
> the proposed "procs" file and also allows scope for things like adding  
> a subsystem to an existing hierarchy.  
>  
> - locking/checking performed in two phases - one to support sleeping  
> locks, and one to support spinlocks. This is to support both  
> subsystems that use mutexes to protect their data, and subsystems that  
> use spinlocks  
>  
> - no locks allowed to be shared between multiple subsystems during the  
> transaction, with the single exception of the mmap\_sem of the  
> thread/process being moved. This is because multiple subsystems use  
> the mmap\_sem for synchronization, and are quite likely to be mounted

> in the same hierarchy. The alternative would be to introduce a  
> down\_read\_unfair() operation that would skip ahead of waiting writers,  
> to safely allow a single thread to recursively lock mm->mmap\_sem.

>  
> First we define the state for the transaction:

>  
> struct cgroup\_attach\_state {

nit: How about naming it cgroup\_attach\_request or  
cgroup\_attach\_request\_state? I suggest this because it's not really  
"state" that's kept beyond the prepare-then-(commit|abort) sequence.

> // The thread or process being moved, NULL if moving (potentially) all threads

> struct task\_struct \*task;

> enum {

> CGROUP\_ATTACH\_THREAD,

> CGROUP\_ATTACH\_PROCESS,

> CGROUP\_ATTACH\_ALL

> } mode;

>

>

> // The destination cgroup

> struct cgroup \*dest;

>

> // The source cgroup for "task" (child threads \*may\* have different  
> groups; subsystem must handle this if it needs to)

> struct cgroup \*src;

>

> // Private state for the attach operation per-subsys. Subsystems are  
> completely responsible for managing this

> void \*subsys\_state[CGROUP\_SUBSYS\_STATE];

>

> // "Recursive lock count" for task->mm->mmap\_sem (needed if we don't  
> introduce down\_read\_unfair())

> int mmap\_sem\_lock\_count;

> };

>

> New cgroup subsystem callbacks (all optional):

>

> -----

>

> int prepare\_attach\_sleep(struct cgroup\_attach\_state \*state);

>

> Called during the first preparation phase for each subsystem. The  
> subsystem may perform any sleeping behaviour, including waiting for  
> mutexes and doing sleeping memory allocations, but may not disable  
> interrupts or take any spinlocks. Return a -ve error on failure or 0  
> on success. If it returns failure, then no further callbacks will be

- > made for this attach; if it returns success then exactly one of
- > `abort_attach_sleep()` or `commit_attach()` is guaranteed to be called in
- > the future
- >
- > No two subsystems may take the same lock as part of their
- > `prepare_attach_sleep()` callback. A special case is made for `mmap_sem`:
- > if this callback needs to `down_read(&state->task->mmap_sem)` it should
- > only do so if `state->mmap_sem_lock_count++ == 0`. (A helper function
- > will be provided for this). The callback should not
- > `write_lock(&state->task->mmap_sem)`.

What about the `task->alloc_lock`? Might that need to be taken by multiple subsystems? See my next comment.

- > Called with `group_mutex` (which prevents any other task movement
- > between cgroups) held plus any mutexes/semaphores taken by earlier
- > subsystems's callbacks.
- >
- > -----
- >
- > `int prepare_attach_nosleep(struct cgroup_attach_state *state);`
- >
- > Called during the second preparation phase (assuming no subsystem
- > failed in the first phase). The subsystem may not sleep in any way,
- > but may disable interrupts or take spinlocks. Return a -ve error on
- > failure or 0 on success. If it returns failure, then
- > `abort_attach_sleep()` will be called; if it returns success then either
- > `abort_attach_nosleep()` followed by `abort_attach_sleep()` will be
- > called, or `commit_attach()` will be called
- >
- > Called with `cgroup_mutex` and `alloc_lock` for task held (plus any
- > mutexes/semaphores taken by subsystems in the `prepare_attach_nosleep()`
- > phase, and any spinlocks taken by earlier subsystems in this phase .
- > If `state->mode == CGROUP_ATTACH_PROCESS` then `alloc_lock` for all
- > threads in task's `thread_group` are held. (Is this a really bad idea?
- > Maybe we should call this without any `task->alloc_lock` held?)

With `task->alloc_lock` held would avoid the case where multiple subsystems need it (assuming the case exists of course).

- > ----
- >
- > `void abort_attach_sleep(struct cgroup_attach_state *state);`
- >
- > Called following a successful return from `prepare_attach_sleep()`.
- > Indicates that the attach operation was aborted and the subsystem
- > should unwind any state changes made and locks taken by
- > `prepare_attach_sleep()`.

```

>
> Called with same locks as prepare_attach_sleep()
>
> ----
>
> void abort_attach_nosleep(struct cgroup_attach_state *state);
>
> Called following a successful return from prepare_attach_nosleep().
> Indicates that the attach operation was aborted and the subsystem
> should unwind any state changes made and locks taken by
> prepare_attach_nosleep().
>
> Called with the same locks as prepare_attach_nosleep();
>
> ----
>
> void commit_attach(struct cgroup_attach_state *state);
>
> Called following a successful return from prepare_attach_sleep() for a
> subsystem that has no prepare_attach_nosleep(), or following a
> successful return from prepare_attach_nosleep(). Indicates that the
> attach operation is going ahead, and
> any partially-committed state should be finalized, and any taken locks
> should be released. No further callbacks will be made for this attach.
>
> This is called immediately after updating task->cgroups (and threads
> if necessary) to point to the new cgroup set.
>
> Called with the same locks held as prepare_attach_nosleep()

```

Rather than describing what might be called later for each API entry separately it might be simpler to prefix the whole API/protocol description with something like:

=====

A successful return from prepare\_X will cause abort\_X to be called if any of the preparatory calls fail. (where X is either sleep or nosleep)

A successful return from prepare\_X will cause commit to be called if all of the preparatory calls succeed. (where X is either sleep or nosleep)

Otherwise no calls to abort\_X or commit will be made. (where X is either sleep or nosleep)

=====

I think that's correct based on your descriptions. Of course changing this only makes sense if this proposal will go into Documentation/ in some form..

>

> Examples

> =====

>

> Here are a few examples of how you might use this. They're not  
 > intended to be syntactically correct or compilable - they're just an  
 > idea of what the routines might look like.

>

>

> 1) cpusets

>

> cpusets (currently) uses cgroup\_mutex for most of its changes that can  
 > invalidate a task attach. thus it can assume that any checks performed  
 > by prepare\_attach\_\*() will remain valid without needing any additional  
 > locking. The existing callback\_mutex used to synchronize cpuset  
 > changes can't be taken in commit\_attach() since spinlocks are held at  
 > that point. However, I think that all the current uses of  
 > callback\_mutex could actually be replaced with an rwlock, which would  
 > be permitted to be taken during commit\_attach(). The cpuset subsystem  
 > wouldn't need to maintain any special state for the transaction. So:

>

> - prepare\_attach\_nosleep(): same as existing cpuset\_can\_attach()

>

> - commit\_attach(): update tasks' allowed cpus; schedule memory  
 > migration in a workqueue if necessary (since we can't take locks at  
 > this point.

>

>

> 2) memrlimit

>

> memrlimit needs to be able to ensure that:

>

> - changes to an mm's virtual address space size can't occur  
 > concurrently with the mm's owner moving between cgroups (including via  
 > a change of mm ownership).

>

> - moving the mm's owner doesn't over-commit the destination cgroup

>

> - once the destination cgroup has been checked, additional charges  
 > can't be made that result in the original move becoming invalid

>

> Currently all normal charges and uncharges are done under the  
 > protection of down\_write(&mm->mmap\_sem); uncharging following a change  
 > that was charged but failed for other reasons isn't done under  
 > mmap\_sem, but isn't a critical path so could probably be changed to do  
 > so (it wouldn't have to be all one big critical section).  
 > Additionally, mm->owner changes are also done under  
 > down\_write(&mmap\_sem). Thus holding down\_read(&mmap\_sem) across the

```

> transaction is sufficient. So (roughly):
>
> prepare_attach_sleep() {
> // prevent mm->owner and mm->total_vm changes
> down_read(&mm->mmap_sem);
> // Nothing to do if we're not moving the owner
> if (mm->owner != state->task) return 0;
> if ((ret = res_counter_charge(state->dest, mm->total_vm)) {
> // If we failed to allocate in the destination, clean up
> up_read(&mm->mmap_sem);
> }
> return ret;
> }
>
> commit_attach() {
> if (mm->owner == state->task) {
> // Release the charge from the source
> res_counter_uncharge(state->src, mm->total_vm);
> }
> // Clean up locks
> up_read(&mm->mmap_sem);
> }
>
> abort_attach_sleep() {
> if (mm->owner == state->task) {
> // Remove the temporary charge from the destination
> res_counter_uncharge(state->dest_cgroup, mm->total_vm);
> }
> // Clean up locks
> up_read(&mm->mmap_sem);
> }
>
> As mentioned above, to handle the case where multiple subsystems need
> to down_read(&mm->mmap_sem), these down/up operations may actually end
> up being done via helper functions to avoid recursive locks.
>
>
> 3) memory
>
> Currently the memory cgroup only uses the mm->owner's cgroup at charge
> time, and keeps a reference to the cgroup on the page. However,
> patches have been proposed that would move all non-shared (page count
> == 1) pages to the destination cgroup when the mm->owner moves to a
> new cgroup. Since it's not possible to prevent page count changes
> without locking all mms on the system, even this transaction approach
> can't really give guarantees. However, something like the following
> would probably be suitable. It's very similar to the memrlimit
> approach, except for the fact that we have to handle the fact that the

```

```

> number of pages we finally move might not be exactly the same as the
> number of pages we thought we'd be moving.
>
> prepare_attach_sleep() {
>   down_read(&mm->mmap_sem);
>   if (mm->owner != state->task) return 0;
>   count = count_unshared_pages(mm);
>   // save the count charged to the new cgroup
>   state->subsys[memcggroup_subsys_id] = (void *)count;
>   if ((ret = res_counter_charge(state->dest, count)) {
>     up_read(&mm->mmap_sem);
>   }
>   return ret;
> }
>
> commit_attach() {
>   if (mm->owner == state->task) {
>     final_count = move_unshared_pages(mm, state->dest);
>     res_counter_uncharge(state->src, final_count);
>     count = state->subsys[memcggroup_subsys_id];
>     res_counter_force_charge(state->dest, final_count - count);
>   }
>   up_read(&mm->mmap_sem);
> }
>
> abort_attach_sleep() {
>   if (mm->owner == state->task) {
>     count = state->subsys[memcggroup_subsys_id];
>     res_counter_uncharge(state->dest, count);
>   }
>   up_read(&mm->mmap_sem);
> }
>
> 4) numtasks:
>
> Numtasks is different from the two memory-related controllers in that
> it may need to move charges from multiple source cgroups (for
> different threads); the memory cgroups only have to deal with the mm
> of a thread-group leader, and all threads in an attach operation are
> from the same thread_group. So numtasks has to be able to handle
> uncharging multiple source cgroups in the commit_attach() operation.
> In order to do this, it requires additional state:
>
> struct numtasks_attach_state {
>   int count;
>   struct cgroup *cg;
>   struct numtasks_attach_state *next;
> }

```

```

>
> It will build a list of numtasks_attach_state objects, one for each
> distinct source cgroup; in the general case either there will only be
> a single thread moving or else all the threads in the thread group
> will belong to the same cgroup, in which case this list will only be a
> single element; the list is very unlikely to get to more than a small
> number of elements.
>
> The prepare_attach_sleep() function can rely on the fact that although
> tasks can fork/exit concurrently with the attach, since cgroup_mutex
> is held, no tasks can change cgroups, and therefore a complete list of
> source cgroups can be constructed.
>
> prepare_attach_sleep() {
>     for each thread being moved:
>         if the list doesn't yet have an entry for thread->cgroup:
>             allocate new entry with cg = thread->cgroup, count = 0;
>             add new entry to list
>         store list in state->subsys[numtasks_subsys_id];
>     return 0;
> }
>
> Then prepare_attach_nosleep() can move counts under protection of
> tasklist_lock, to prevent any forks/exits
>
> prepare_attach_nosleep() {
>     read_lock(&tasklist_lock);
>     for each thread being moved {
>         find entry for thread->cgroup in list
>         entry->count++;
>         total_count++;
>     }
>     if ((ret = res_counter_charge(state->dest, total_count) != 0) {
>         read_unlock(&tasklist_lock);
>     }
>     return ret;
> }
>
> commit_attach() {
>     for each entry in list {
>         res_counter_uncharge(entry->cg, entry->count);
>     }
>     read_unlock(&tasklist_lock);
>     free list;
> }
> abort_attach_nosleep() {
>     // just needs to clear up prepare_attach_nosleep()
>     res_counter_uncharge(state->dest, total_count);

```

```
> read_unlock(&tasklist_lock);
> }
>
> abort_attach_sleep() {
> // just needs to clean up the list allocated in prepare_attach_sleep()
> free list;
> }
>
>
> So, thoughts? Is this just way to complex? Have I missed something
> that means this approach can't work?
```

This proposal for attaching tasks works for the proposed freezer subsystem too.

Allowing prepare\_X to hold locks when it has exited seems ripe for introducing two separate subsystems that inadvertently take locks out of order. I guess lockdep will warn us about this assuming lockdep and all the cgroup subsystems have been configured and tested in the same hierarchy.

Cheers,  
-Matt Helsley

---

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

---