

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.

## 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 {

    // 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).`

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?)

----

`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()`

## 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?

Paul

---

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

---



---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Jackson](#) on Thu, 10 Jul 2008 11:23:06 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Paul M wrote:  
> So, thoughts?

The cpuset portions seem reasonable to me, at first glance.  
But cpusets are easy, as you note.

--

I won't rest till it's the best ...  
Programmer, Linux Scalability

---

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>

---

---

Subject: Re: [RFC] Transactional CGroup task attachment

Posted by [KAMEZAWA Hiroyuki](#) on Fri, 11 Jul 2008 00:17:45 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

Thank you for your effort.

On Wed, 9 Jul 2008 23:46:33 -0700

"Paul Menage" <menage@google.com> wrote:

> 3) memory

>

> Curently 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);
> }
>

```

At first look, maybe works well. we need some special codes (to move resource) but that's all.

My small concern is a state change between `prepare_attach_sleep()` -> `commit_attach()`. Hmm...but as you say, we cannot do `down_write(mmap_sem)`. Maybe inserting some check codes to `mem_cgrouop_charge()` to stop charge while move is the last thing we can do.

Anyway, if unwinding is supported officially, I think we can find a way to go.

Thanks,  
-Kame

```

> 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?  
>  
> Paul  
>

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Li Zefan](#) on Fri, 11 Jul 2008 02:14:33 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

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.  
>

Though not so elegant, it sounds workable in practise.

Support for CGROUP\_ATTACH\_PROCESS adds complexity and some subsystems may have to aware of this and write code to support it, but fortunately it seems only numtasks needs to deal with this. Or we can implement this new task attachment framework for CGROUP\_ATTACH\_THREAD mode only, to see how good it works, and later we add support for CGROUP\_ATTACH\_PROCESS.

And I'll do testing if this gets implemented. :)

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Menage](#) on Fri, 11 Jul 2008 03:31:26 GMT

---

On Thu, Jul 10, 2008 at 7:14 PM, Li Zefan <lizf@cn.fujitsu.com> wrote:

>  
> Support for CGROUP\_ATTACH\_PROCESS adds complexity and some subsystems  
> may have to aware of this and write code to support it, but fortunately  
> it seems only numtasks needs to deal with this. Or we can implement  
> this new task attachment framework for CGROUP\_ATTACH\_THREAD mode only,  
> to see how good it works, and later we add support for  
> CGROUP\_ATTACH\_PROCESS.

Yes, that's the right way to go - but getting the  
CGROUP\_ATTACH\_PROCESS mode working is a prerequisite for adding the  
"procs" file.

Paul

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [serue](#) on Fri, 11 Jul 2008 14:27:39 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

Quoting Paul Menage (menage@google.com):

> 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.

>

>

> 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 charges, 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 {
- >
- > // 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).
>
> 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?)
>

```

```

> ----
>
> 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()
>
>
> 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[memcgsubsys_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?

It seems well thought out on first reading.

It does feel like it may be too much designed for one particular user  
(i.e. is there a reason not to expect a future cgroup to need a check  
under a spinlock before a check under a mutex - say an i\_sem - in the  
can\_attach sequence?), but it solves the current real problems first,  
what else can you do? :)

-serge

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Menage](#) on Fri, 11 Jul 2008 15:23:22 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, Jul 11, 2008 at 7:27 AM, Serge E. Hallyn <serue@us.ibm.com> wrote:

>  
> It does feel like it may be too much designed for one particular user  
> (i.e. is there a reason not to expect a future cgroup to need a check  
> under a spinlock before a check under a mutex - say an i\_sem - in the  
> can\_attach sequence?),

It would be fine as long as the code didn't want to \*keep\* holding the  
spinlock after the first check, while taking the mutex - and since  
that style of code is invalid under the existing locking rules, I  
don't see that as a problem. There's nothing to stop a  
prepare\_attach\_sleep() method from taking a spinlock as long as it  
releases it before it returns.

Paul

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [serue](#) on Fri, 11 Jul 2008 15:34:21 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Quoting Paul Menage (menage@google.com):

> On Fri, Jul 11, 2008 at 7:27 AM, Serge E. Hallyn <serue@us.ibm.com> wrote:

> >

> > It does feel like it may be too much designed for one particular user  
> > (i.e. is there a reason not to expect a future cgroup to need a check  
> > under a spinlock before a check under a mutex - say an i\_sem - in the  
> > can\_attach sequence?),

>

> It would be fine as long as the code didn't want to \*keep\* holding the  
> spinlock after the first check, while taking the mutex - and since  
> that style of code is invalid under the existing locking rules, I  
> don't see that as a problem. There's nothing to stop a  
> prepare\_attach\_sleep() method from taking a spinlock as long as it  
> releases it before it returns.

>

> Paul

Good point. For some stupid reason i was thinking don't take a spinlock at all.

Have you started an implementation?

thanks,  
-serge

---

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>

---

Subject: Re: [RFC] Transactional CGroup task attachment

Posted by [Paul Menage](#) on Fri, 11 Jul 2008 15:36:32 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, Jul 11, 2008 at 8:34 AM, Serge E. Hallyn <serue@us.ibm.com> wrote:

>

> Have you started an implementation?

>

No, not yet. I wanted to get general agreement that the API made sense first.

Paul

---

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>



Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Balbir Singh](#) on Fri, 11 Jul 2008 16:12:39 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

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.
- >

Precisely!

- > - 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.
- >

>

> 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.  
>

This would ideally be recursive mutexes, Linus does not like recursive mutexes.  
Adding an unfair variant would mean that we need to support a more generic  
locking class.

> First we define the state for the transaction:

```
>
> struct cgroup_attach_state {
>
> // 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);
>
```

Is \_sleep really required to be specified? The function name sounds as if the  
callback processor will sleep.

> 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).
- >
- > Called with group\_mutex (which prevents any other task movement
- > between cgroups) held plus any mutexes/semaphores taken by earlier
- > subsystems's callbacks.
- >

This sounds almost like the BKL for cgroups :) I see where you are going with this, but I am afraid the implementation and rules sound complex. It will be hard to verify that two subsystems are not going to take the same lock. I would rather prefer to do the following

1. Prepare\_attach() the subsystem does it's or fails
2. If someone failed, send out failed notifications to successful callbacks
3. On receiving a failed notification (due to a different cgroup failure), clients undo their operation (done in prepare\_attach())
4. If all was successful, move the task and call attached() after the task is attached.

[snip]

- >
- > So, thoughts? Is this just way to complex? Have I missed something
- > that means this approach can't work?
- >

I read through the rest of it. The sleep/nosleep might make sense (to help the task acquire the type of lock it wants to acquire), but isn't sleep a generic case for nosleep as well?

Can we manage with steps I've listed above?

--

Warm Regards,  
 Balbir Singh  
 Linux Technology Center

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Menage](#) on Fri, 11 Jul 2008 20:41:34 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, Jul 11, 2008 at 9:12 AM, Balbir Singh <balbir@linux.vnet.ibm.com> wrote:

>  
> This would ideally be recursive mutexes, Linus does not like recursive mutexes.

But we already have them in function, if not in name, with things like  
cpu\_hotplug.lock, which is an open-coded reader-writer mutex with  
recursion.

>>  
>> int prepare\_attach\_sleep(struct cgroup\_attach\_state \*state);  
>>  
>  
> Is \_sleep really required to be specified? The function name sounds as if the  
> callback processor will sleep.

I wasn't quite sure about the name of that method either. We could  
call it prepare\_attach\_maysleep(); just prepare\_attach() seems a  
little under-descriptive.

>> Called with group\_mutex (which prevents any other task movement  
>> between cgroups) held plus any mutexes/semaphores taken by earlier  
>> subsystems's callbacks.  
>>  
>  
> This sounds almost like the BKL for cgroups :)

Yes, cgroup\_mutex is indeed currently the BKL for cgroups. I'm working  
separately to reduce that with things like the per-subsystem  
hierarchy\_mutex, which will also protect against task movements.

>  
> 1. Prepare\_attach() the subsystem does it's or fails

Did you miss a word here?

> 2. If someone failed, send out failed notifications to successfull callbacks  
> 3. On receiving a failed notification (due to a different cgroup failure),

> clients undo their operation (done in prepare\_attach())  
> 4. If all was successful, move the task and call attached() after the task is  
> attached.

That's pretty much what we have - except that I've got two prepare\_attach() methods to handle the case that some subsystems might need mutexes and others might need spinlocks in order to handle synchronization between the move and their resource charge/uncharge mechanisms - because any mutexes (and memory allocations) need to be handled before spinlocks.

>  
> I read through the rest of it. The sleep/nosleep might make sense (to help the  
> task acquire the type of lock it wants to acquire), but isn't sleep a generic  
> case for nosleep as well?

Can you clarify what you mean by that?

Paul

---

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

---

---

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 charges, 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
>
> Currenly 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

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Menage](#) on Sat, 12 Jul 2008 00:18:18 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, Jul 11, 2008 at 5:03 PM, Matt Helsley <matthl@us.ibm.com> wrote:

```
>> 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.
```

State doesn't have to be long-lived to be state. But I'm not too worried about the exact name for it, if people have other preferences.

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

My thought was that cgroups would take that anyway prior to calling `prepare_attach_nosleep()`, since it's a requirement for changing `task->cgroups` anyway.

```
>  
> 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'll play with working that into the description.

```
> 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..
```



Yes, we'd definitely need to document this in some detail.

>  
>     Allowing prepare\_X to hold locks when it has exited seems ripe for  
> introducing two separate subsystems that inadvertently take locks out of  
> order.

Yes, but I'm not sure that there's much that we can do about that. If we want to guarantee to be able to rollback one subsystem when a later subsystem fails then we have to let the earlier subsystems continue to hold locks. Or is this too ambitious a goal to support?

Paul

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Matt Helsley](#) on Sat, 12 Jul 2008 00:48:14 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, 2008-07-11 at 17:18 -0700, Paul Menage wrote:  
> On Fri, Jul 11, 2008 at 5:03 PM, Matt Helsley <matthltc@us.ibm.com> wrote:  
> >> 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.  
>  
> State doesn't have to be long-lived to be state. But I'm not too  
> worried about the exact name for it, if people have other preferences.  
>  
> >  
> > What about the task->alloc\_lock? Might that need to be taken by multiple  
> > subsystems? See my next comment.  
>  
> My thought was that cgroups would take that anyway prior to calling  
> prepare\_attach\_nosleep(), since it's a requirement for changing  
> task->cgroups anyway.

Yeah, that makes sense.

> >  
> > 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'll play with working that into the description.  
>  
> > 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..  
>  
> Yes, we'd definitely need to document this in some detail.  
>  
> >  
> >     Allowing prepare\_X to hold locks when it has exited seems ripe for  
> > introducing two separate subsystems that inadvertently take locks out of  
> > order.  
>  
> Yes, but I'm not sure that there's much that we can do about that. If  
> we want to guarantee to be able to rollback one subsystem when a later  
> subsystem fails then we have to let the earlier subsystems continue to  
> hold locks. Or is this too ambitious a goal to support?

I can't see a better way to support that goal and it doesn't seem  
overly ambitious to me. Just needs a somewhat specific test  
configuration for new subsystem patches to detect the deadlock issue.

Cheers,  
-Matt Helsley

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Menage](#) on Sat, 12 Jul 2008 01:49:17 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, Jul 11, 2008 at 5:48 PM, Matt Helsley <[matthlrc@us.ibm.com](mailto:matthlrc@us.ibm.com)> wrote:  
>

> I can't see a better way to support that goal and it doesn't seem  
> overly ambitious to me. Just needs a somewhat specific test  
> configuration for new subsystem patches to detect the deadlock issue.

Right - but all non-trivial development should be done with lockdep enabled anyway.

Paul

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Matt Helsley](#) on Sat, 12 Jul 2008 02:03:36 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, 2008-07-11 at 18:49 -0700, Paul Menage wrote:  
> On Fri, Jul 11, 2008 at 5:48 PM, Matt Helsley <matthl@us.ibm.com> wrote:  
> >  
> > I can't see a better way to support that goal and it doesn't seem  
> > overly ambitious to me. Just needs a somewhat specific test  
> > configuration for new subsystem patches to detect the deadlock issue.  
>  
> Right - but all non-trivial development should be done with lockdep  
> enabled anyway.  
>  
> Paul

True, though I don't think it's as simple as just enabling lockdep. My understanding is you won't be able to determine if locks could ever be taken out of order unless all of the cgroup systems are enabled and they are all in the same cgroup hierarchy.

Cheers,  
-Matt Helsley

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Menage](#) on Sat, 12 Jul 2008 02:09:42 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, Jul 11, 2008 at 7:03 PM, Matt Helsley <matthltc@us.ibm.com> wrote:

>  
> True, though I don't think it's as simple as just enabling lockdep. My  
> understanding is you won't be able to determine if locks could ever be  
> taken out of order unless all of the cgroup systems are enabled and they  
> are all in the same cgroup hierarchy.

I wonder how hard it would be to extend lockdep to give a "signature"  
of locking operations from point A to point B? Then you could just  
compare a new subsystem with all existing subsystems without actually  
running with it.

Paul

---

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>

---

---

Subject: Re: [RFC] Transactional CGroup task attachment

Posted by [Daisuke Nishimura](#) on Mon, 14 Jul 2008 06:28:22 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, 11 Jul 2008 09:20:58 +0900, KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com> wrote:

> Thank you for your effort.

>

> On Wed, 9 Jul 2008 23:46:33 -0700

> "Paul Menage" <menage@google.com> wrote:

> > 3) memory

> >

> > Curently 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);
> > }
> >
>
> At first look, maybe works well. we need some special codes (to move resource)
> but that's all.
>
> My small concern is a state change between prepare_attach_sleep() ->
> commit_attach(). Hmm...but as you say, we cannot do down_write(mmap_sem).
> Maybe inserting some check codes to mem_cgroup_charge() to stop charge while
> move is the last thing we can do.
>
> I have two comments.

```

- I think page reclaiming code decreases the memory charge without holding mmap\_sem(e.g. try\_to\_unmap(), \_\_remove\_mapping()). Shouldn't we handle these cases?
- When swap controller is merged, I should implement prepare\_attach\_nosleep() which holds swap\_lock.

```

> Anyway, if unwinding is supported officially, I think we can find a way
> to go.
>
> I think so too.

```

Thanks,  
Daisuke Nishimura.

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [KAMEZAWA Hiroyuki](#) on Mon, 14 Jul 2008 07:50:51 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Mon, 14 Jul 2008 15:28:22 +0900  
Daisuke Nishimura <nishimura@mxp.nes.nec.co.jp> wrote:

> On Fri, 11 Jul 2008 09:20:58 +0900, KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>  
wrote:

> > Thank you for your effort.

> >

> > On Wed, 9 Jul 2008 23:46:33 -0700

> > "Paul Menage" <menage@google.com> wrote:

> > > 3) memory

> > >

> > > Curently 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);
>>> }
>>>
>>
>> At frist look, maybe works well. we need some special codes (to move resource)
>> but that's all.
>>
>> My small concern is a state change between prepare_attach_sleep() ->
>> commit_attach(). Hmm...but as you say, we cannot do down_write(mmap_sem).
>> Maybe inserting some check codes to mem_cgrouop_charge() to stop charge while
>> move is the last thing we can do.
>>
>> I have two comments.
>
> - I think page reclaiming code decreases the memory charge
>   without holding mmap_sem(e.g. try_to_unmap(), __remove_mapping()).
>   Shouldn't we handle these cases?

```

I think decreasing is not problem, here.

I don't like handle mmap->sem by some unclear way. I'd like to add some flag to mm\_struct or page\_struct to stop(skip/avoid) charge/uncharge while task move.

```

> - When swap controller is merged, I should implement
>   prepare_attach_nosleep() which holds swap_lock.
>
just making add_to_swap() fail during move is not enough ?

```

Thanks,  
-Kame

---

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

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Peter Zijlstra](#) on Mon, 14 Jul 2008 10:50:01 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, 2008-07-11 at 17:03 -0700, Matt Helsley wrote:  
> On Wed, 2008-07-09 at 23:46 -0700, Paul Menage wrote:

> > 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.

Other alternatives: cgroup\_attach\_context, cgroup\_attach\_txn

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Peter Zijlstra](#) on Mon, 14 Jul 2008 10:56:49 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, 2008-07-11 at 19:09 -0700, Paul Menage wrote:  
> On Fri, Jul 11, 2008 at 7:03 PM, Matt Helsley <matthl@us.ibm.com> wrote:  
> >  
> > True, though I don't think it's as simple as just enabling lockdep. My  
> > understanding is you won't be able to determine if locks could ever be  
> > taken out of order unless all of the cgroup systems are enabled and they  
> > are all in the same cgroup hierarchy.  
>  
> I wonder how hard it would be to extend lockdep to give a "signature"  
> of locking operations from point A to point B? Then you could just  
> compare a new subsystem with all existing subsystems without actually  
> running with it.

something like: <http://lkml.org/lkml/2008/6/22/308> ?

---

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

---



Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Daisuke Nishimura](#) on Mon, 14 Jul 2008 12:36:09 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Mon, 14 Jul 2008 16:54:44 +0900, KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com> wrote:

> On Mon, 14 Jul 2008 15:28:22 +0900

> Daisuke Nishimura <nishimura@mxp.nes.nec.co.jp> wrote:

>

> > On Fri, 11 Jul 2008 09:20:58 +0900, KAMEZAWA Hiroyuki

<kamezawa.hiroyu@jp.fujitsu.com> wrote:

> > > Thank you for your effort.

> > >

> > > On Wed, 9 Jul 2008 23:46:33 -0700

> > > "Paul Menage" <menage@google.com> wrote:

> > > > 3) memory

> > > >

> > > > Curently 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[memcgroup\_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[memcgroup\_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);
>>>> }
>>>>
>>>>
>>> At first look, maybe works well. we need some special codes (to move resource)
>>> but that's all.
>>>
>>> My small concern is a state change between prepare_attach_sleep() ->
>>> commit_attach(). Hmm...but as you say, we cannot do down_write(mmap_sem).
>>> Maybe inserting some check codes to mem_cgroup_charge() to stop charge while
>>> move is the last thing we can do.
>>>
>> I have two comments.
>>
>> - I think page reclaiming code decreases the memory charge
>> without holding mmap_sem(e.g. try_to_unmap(), __remove_mapping()).
>> Shouldn't we handle these cases?
>
> I think decreasing is not problem, here.
> I don't like handle mmap->sem by some unclear way. I'd like to add some flag to
> mm_struct or page_struct to stop(skip/avoid) charge/uncharge while task move.
>
It would be a good idea.

```

```

>> - When swap controller is merged, I should implement
>> prepare_attach_nosleep() which holds swap_lock.
>>
> just making add_to_swap() fail during move is not enough ?
>
This can only avoid increasing, I think.

```

I thought it would be better to avoid decreasing too,  
just because some special handling on uncharged usage  
would be needed in rollback or commit.

Anyway, I think it depends on how to implement move and rollback,  
and I will consider more.  
Thank you for your suggestion.

Thanks,

Daisuke Nishimura.

---

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

---

---

Subject: Re: [RFC] Transactional CGroup task attachment  
Posted by [Paul Menage](#) on Mon, 14 Jul 2008 19:16:08 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Sun, Jul 13, 2008 at 11:28 PM, Daisuke Nishimura  
<nishimura@mxp.nes.nec.co.jp> wrote:

>  
> - I think page reclaiming code decreases the memory charge  
> without holding mmap\_sem(e.g. try\_to\_unmap(), \_\_remove\_mapping()).  
> Shouldn't we handle these cases?

The prepare\_attach\_nosleep() call could take the res\_counter's  
spinlock, which would lock out all other charges and uncharges until  
the transaction was completed; that might be enough for what you want.  
We'd need to export lock/unlock functions from res\_counter.

Paul

---

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

---