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 simplications
> 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 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.
>

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 specfied? 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 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.

[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
IBM, ISTL

_____

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