
Subject: Re: [PATCH] Move cgroups destroy() callbacks to cgroup_diput()
Posted by [Paul Menage](#) on Wed, 24 Oct 2007 02:20:19 GMT
[View Forum Message](#) <> [Reply to Message](#)

OK, if no-one has any complaints about this I'll send it on to akpm.

Paul

On 10/23/07, Paul Menage <menage@google.com> wrote:

```
> Move the calls to the cgroup subsystem destroy() methods from
> cgroup_rmdir() to cgroup_diput(). This allows control file reads and
> writes to access their subsystem state without having to be concerned
> with locking against cgroup destruction - the control file dentry will
> keep the cgroup and its subsystem state objects alive until the file
> is closed.
>
> The documentation is updated to reflect the changed semantics of
> destroy(); additionally the locking comments for destroy() and some
> other methods were clarified and decrustified.
>
> Signed-off-by: Paul Menage <menage@google.com>
>
> ---
> Documentation/cgroups.txt | 22 ++++++-----
> kernel/cgroup.c          | 36 ++++++-----
> 2 files changed, 35 insertions(+), 23 deletions(-)
>
> Index: container-2.6.23-mm1/kernel/cgroup.c
> =====
> --- container-2.6.23-mm1.orig/kernel/cgroup.c
> +++ container-2.6.23-mm1/kernel/cgroup.c
> @@ -592,6 +592,7 @@ static void cgroup_diput(struct dentry *
>     /* is dentry a directory ? if so, kfree() associated cgroup */
>     if (S_ISDIR(inode->i_mode)) {
>         struct cgroup *cgrp = dentry->d_fsdata;
> +         struct cgroup_subsys *ss;
>         BUG_ON(!(cgroup_is_removed(cgrp)));
>         /* It's possible for external users to be holding css
>          * reference counts on a cgroup; css_put() needs to
> @@ -600,6 +601,23 @@ static void cgroup_diput(struct dentry *
>          * queue the cgroup to be handled by the release
>          * agent */
>         synchronize_rcu();
> +
> +         mutex_lock(&cgroup_mutex);
> +         /*
> +          * Release the subsystem state objects.
> +          */
```

```

> +     for_each_subsys(cgrp->root, ss) {
> +         if (cgrp->subsys[ss->subsys_id])
> +             ss->destroy(ss, cgrp);
> +     }
> +
> +     cgrp->root->number_of_cgroups--;
> +     mutex_unlock(&cgroup_mutex);
> +
> +     /* Drop the active superblock reference that we took when we
> +      * created the cgroup */
> +     deactivate_super(cgrp->root->sb);
> +
>     kfree(cgrp);
> }
> iput(inode);
> @@ -1333,6 +1351,10 @@ static ssize_t cgroup_common_file_write(
>
>     mutex_lock(&cgroup_mutex);
>
> +     /*
> +      * This was already checked for in cgroup_file_write(), but
> +      * check again now we're holding cgroup_mutex.
> +      */
>     if (cgroup_is_removed(cgrp)) {
>         retval = -ENODEV;
>         goto out2;
> @@ -1388,7 +1410,7 @@ static ssize_t cgroup_file_write(struct
>     struct cftype *cft = __d_cft(file->f_dentry);
>     struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
>
> -     if (!cft)
> +     if (!cft || cgroup_is_removed(cgrp))
>         return -ENODEV;
>     if (cft->write)
>         return cft->write(cgrp, cft, file, buf, nbytes, ppos);
> @@ -1458,7 +1480,7 @@ static ssize_t cgroup_file_read(struct f
>     struct cftype *cft = __d_cft(file->f_dentry);
>     struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
>
> -     if (!cft)
> +     if (!cft || cgroup_is_removed(cgrp))
>         return -ENODEV;
>
>     if (cft->read)
> @@ -2139,7 +2161,6 @@ static int cgroup_rmdir(struct inode *un
>     struct cgroup *cgrp = dentry->d_fsdata;
>     struct dentry *d;
>     struct cgroup *parent;

```

```

> - struct cgroup_subsys *ss;
> struct super_block *sb;
> struct cgroupfs_root *root;
>
> @@ -2164,11 +2185,6 @@ static int cgroup_rmdir(struct inode *un
>         return -EBUSY;
>     }
>
> - for_each_subsys(root, ss) {
> -     if (cgrp->subsys[ss->subsys_id])
> -         ss->destroy(ss, cgrp);
> - }
> -
> spin_lock(&release_list_lock);
> set_bit(CGRP_REMOVED, &cgrp->flags);
> if (!list_empty(&cgrp->release_list))
> @@ -2183,15 +2199,11 @@ static int cgroup_rmdir(struct inode *un
>
> cgroup_d_remove_dir(d);
> dput(d);
> - root->number_of_cgroups--;
>
> set_bit(CGRP_RELEASABLE, &parent->flags);
> check_for_release(parent);
>
> mutex_unlock(&cgroup_mutex);
> - /* Drop the active superblock reference that we took when we
> -  * created the cgroup */
> - deactivate_super(sb);
> return 0;
> }
>
> Index: container-2.6.23-mm1/Documentation/cgroups.txt
> =====
> --- container-2.6.23-mm1.orig/Documentation/cgroups.txt
> +++ container-2.6.23-mm1/Documentation/cgroups.txt
> @@ -456,7 +456,7 @@ methods are create/destroy. Any others t
> be successful no-ops.
>
> struct cgroup_subsys_state *create(struct cgroup *cont)
> -LL=cgroup_mutex
> +(cgroup_mutex held by caller)
>
> Called to create a subsystem state object for a cgroup. The
> subsystem should allocate its subsystem state object for the passed
> @@ -471,14 +471,19 @@ it's the root of the hierarchy) and may
> initialization code.
>

```

```

> void destroy(struct cgroup *cont)
> -LL=cgroup_mutex
> +(cgroup_mutex held by caller)
>
> -The cgroup system is about to destroy the passed cgroup; the
> -subsystem should do any necessary cleanup
> +The cgroup system is about to destroy the passed cgroup; the subsystem
> +should do any necessary cleanup and free its subsystem state
> +object. By the time this method is called, the cgroup has already been
> +unlinked from the file system and from the child list of its parent;
> +cgroup->parent is still valid. (Note - can also be called for a
> +newly-created cgroup if an error occurs after this subsystem's
> +create() method has been called for the new cgroup).
>
> int can_attach(struct cgroup_subsys *ss, struct cgroup *cont,
>               struct task_struct *task)
> -LL=cgroup_mutex
> +(cgroup_mutex held by caller)
>
> Called prior to moving a task into a cgroup; if the subsystem
> returns an error, this will abort the attach operation. If a NULL
> @@ -489,25 +494,20 @@ remain valid while the caller holds cgro
>
> void attach(struct cgroup_subsys *ss, struct cgroup *cont,
>            struct cgroup *old_cont, struct task_struct *task)
> -LL=cgroup_mutex
> -
>
> Called after the task has been attached to the cgroup, to allow any
> post-attachment activity that requires memory allocations or blocking.
>
> void fork(struct cgroup_subsys *ss, struct task_struct *task)
> -LL=callback_mutex, maybe read_lock(tasklist_lock)
>
> Called when a task is forked into a cgroup. Also called during
> registration for all existing tasks.
>
> void exit(struct cgroup_subsys *ss, struct task_struct *task)
> -LL=callback_mutex
>
> Called during task exit
>
> int populate(struct cgroup_subsys *ss, struct cgroup *cont)
> -LL=none
>
> Called after creation of a cgroup to allow a subsystem to populate
> the cgroup directory with file entries. The subsystem should make
> @@ -524,7 +524,7 @@ example in cpusets, no task may attach b

```

> up.
>
> void bind(struct cgroup_subsys *ss, struct cgroup *root)
> -LL=callback_mutex
> +(cgroup_mutex held by caller)
>
> Called when a cgroup subsystem is rebound to a different hierarchy
> and root cgroup. Currently this will only involve movement between
> -
> To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at <http://vger.kernel.org/majordomo-info.html>
> Please read the FAQ at <http://www.tux.org/lkml/>
>

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