
Subject: [PATCH v8 3/3] cgroups: make procs file writable
Posted by [Ben Blum](#) on Tue, 08 Feb 2011 01:39:50 GMT
[View Forum Message](#) <> [Reply to Message](#)

Makes procs file writable to move all threads by tgid at once

From: Ben Blum <bblum@andrew.cmu.edu>

This patch adds functionality that enables users to move all threads in a threadgroup at once to a cgroup by writing the tgid to the 'cgroup.procs' file. This current implementation makes use of a per-threadgroup rwsem that's taken for reading in the fork() path to prevent newly forking threads within the threadgroup from "escaping" while the move is in progress.

Signed-off-by: Ben Blum <bblum@andrew.cmu.edu>

Documentation/cgroups/cgroups.txt | 9 +
kernel/cgroup.c | 437 ++++++-----
2 files changed, 397 insertions(+), 49 deletions(-)

diff --git a/Documentation/cgroups/cgroups.txt b/Documentation/cgroups/cgroups.txt
index d3c9a24..92d93d6 100644

--- a/Documentation/cgroups/cgroups.txt

+++ b/Documentation/cgroups/cgroups.txt

@@ -236,7 +236,8 @@ containing the following files describing that cgroup:

- cgroup.procs: list of tgid's in the cgroup. This list is not guaranteed to be sorted or free of duplicate tgid's, and userspace should sort/uniquify the list if this property is required.
- This is a read-only file, for now.
- + Writing a thread group id into this file moves all threads in that
- + group into this cgroup.
- notify_on_release flag: run the release agent on exit?
- release_agent: the path to use for release notifications (this file exists in the top cgroup only)

@@ -426,6 +427,12 @@ You can attach the current shell task by echoing 0:

echo 0 > tasks

+You can use the cgroup.procs file instead of the tasks file to move all
+threads in a threadgroup at once. Echoing the pid of any task in a
+threadgroup to cgroup.procs causes all tasks in that threadgroup to be
+be attached to the cgroup. Writing 0 to cgroup.procs moves all tasks
+in the writing task's threadgroup.

+

2.3 Mounting hierarchies by name

diff --git a/kernel/cgroup.c b/kernel/cgroup.c

index 616f27a..58b364a 100644

--- a/kernel/cgroup.c

+++ b/kernel/cgroup.c

```
@@ -1726,6 +1726,76 @@ int cgroup_path(const struct cgroup *cgrp, char *buf, int buflen)
}
EXPORT_SYMBOL_GPL(cgroup_path);
```

```
+/*
```

```
+ * cgroup_task_migrate - move a task from one cgroup to another.
```

```
+ *
```

```
+ * 'guarantee' is set if the caller promises that a new css_set for the task
```

```
+ * will already exist. If not set, this function might sleep, and can fail with
```

```
+ * -ENOMEM. Otherwise, it can only fail with -ESRCH.
```

```
+ */
```

```
+static int cgroup_task_migrate(struct cgroup *cgrp, struct cgroup *oldcgrp,
+                               struct task_struct *tsk, bool guarantee)
```

```
+{
```

```
+ struct css_set *oldcg;
```

```
+ struct css_set *newcg;
```

```
+
```

```
+ /*
```

```
+ * get old css_set. we need to take task_lock and refcount it, because
```

```
+ * an exiting task can change its css_set to init_css_set and drop its
```

```
+ * old one without taking cgroup_mutex.
```

```
+ */
```

```
+ task_lock(tsk);
```

```
+ oldcg = tsk->cgroups;
```

```
+ get_css_set(oldcg);
```

```
+ task_unlock(tsk);
```

```
+
```

```
+ /* locate or allocate a new css_set for this task. */
```

```
+ if (guarantee) {
```

```
+ /* we know the css_set we want already exists. */
```

```
+ struct cgroup_subsys_state *template[CGROUP_SUBSYS_COUNT];
```

```
+ read_lock(&css_set_lock);
```

```
+ newcg = find_existing_css_set(oldcg, cgrp, template);
```

```
+ BUG_ON(!newcg);
```

```
+ get_css_set(newcg);
```

```
+ read_unlock(&css_set_lock);
```

```
+ } else {
```

```
+ might_sleep();
```

```
+ /* find_css_set will give us newcg already referenced. */
```

```
+ newcg = find_css_set(oldcg, cgrp);
```

```
+ if (!newcg) {
```

```
+ put_css_set(oldcg);
```

```
+ return -ENOMEM;
```

```
+ }
```

```
+ }
```

```

+ put_css_set(oldcg);
+
+ /* if PF_EXITING is set, the tsk->cgroups pointer is no longer safe. */
+ task_lock(tsk);
+ if (tsk->flags & PF_EXITING) {
+ task_unlock(tsk);
+ put_css_set(newcg);
+ return -ESRCH;
+ }
+ rcu_assign_pointer(tsk->cgroups, newcg);
+ task_unlock(tsk);
+
+ /* Update the css_set linked lists if we're using them */
+ write_lock(&css_set_lock);
+ if (!list_empty(&tsk->cg_list))
+ list_move(&tsk->cg_list, &newcg->tasks);
+ write_unlock(&css_set_lock);
+
+ /*
+ * We just gained a reference on oldcg by taking it from the task. As
+ * trading it for newcg is protected by cgroup_mutex, we're safe to drop
+ * it here; it will be freed under RCU.
+ */
+ put_css_set(oldcg);
+
+ set_bit(CGRP_RELEASABLE, &oldcgrp->flags);
+ return 0;
+}
+
+/**
+ * cgroup_attach_task - attach task 'tsk' to cgroup 'cgrp'
+ * @cgrp: the cgroup the task is attaching to
+ @@ -1736,11 +1806,9 @@ EXPORT_SYMBOL_GPL(cgroup_path);
+ */
+int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
+{
+ int retval = 0;
+ int retval;
+ struct cgroup_subsys *ss, *failed_ss = NULL;
+ struct cgroup *oldcgrp;
+ struct css_set *cg;
+ struct css_set *newcg;
+ struct cgroupfs_root *root = cgrp->root;
+
+ /* Nothing to do if the task is already in that cgroup */
+ @@ -1771,38 +1839,9 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
+ }
+ }

```

```

- task_lock(tsk);
- cg = tsk->cgroups;
- get_css_set(cg);
- task_unlock(tsk);
- /*
-  * Locate or allocate a new css_set for this task,
-  * based on its final set of cgroups
-  */
- newcg = find_css_set(cg, cgrp);
- put_css_set(cg);
- if (!newcg) {
-     retval = -ENOMEM;
-     goto out;
- }
-
- task_lock(tsk);
- if (tsk->flags & PF_EXITING) {
-     task_unlock(tsk);
-     put_css_set(newcg);
-     retval = -ESRCH;
+     retval = cgroup_task_migrate(cgrp, oldcgrp, tsk, false);
+     if (retval)
+         goto out;
- }
- rcu_assign_pointer(tsk->cgroups, newcg);
- task_unlock(tsk);
-
- /* Update the css_set linked lists if we're using them */
- write_lock(&css_set_lock);
- if (!list_empty(&tsk->cg_list)) {
-     list_del(&tsk->cg_list);
-     list_add(&tsk->cg_list, &newcg->tasks);
- }
- write_unlock(&css_set_lock);

for_each_subsys(root, ss) {
    if (ss->pre_attach)
@@ -1812,9 +1851,8 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
    if (ss->attach)
        ss->attach(ss, cgrp, oldcgrp, tsk);
}
- set_bit(CGRP_RELEASABLE, &oldcgrp->flags);
+
+ synchronize_rcu();
- put_css_set(cg);

/*

```

```

    * wake up rmdir() waiter. the rmdir should fail since the cgroup
@@ -1864,49 +1902,352 @@ int cgroup_attach_task_all(struct task_struct *from, struct
task_struct *tsk)
EXPORT_SYMBOL_GPL(cgroup_attach_task_all);

```

```

/*
- * Attach task with pid 'pid' to cgroup 'cgrp'. Call with cgroup_mutex
- * held. May take task_lock of task
+ * cgroup_attach_proc works in two stages, the first of which prefetches all
+ * new css_sets needed (to make sure we have enough memory before committing
+ * to the move) and stores them in a list of entries of the following type.
+ * TODO: possible optimization: use css_set->rcu_head for chaining instead
+ */
+struct cg_list_entry {
+ struct css_set *cg;
+ struct list_head links;
+};
+
+static bool css_set_check_fetched(struct cgroup *cgrp,
+ struct task_struct *tsk, struct css_set *cg,
+ struct list_head *newcg_list)
+{
+ struct css_set *newcg;
+ struct cg_list_entry *cg_entry;
+ struct cgroup_subsys_state *template[CGROUP_SUBSYS_COUNT];
+
+ read_lock(&css_set_lock);
+ newcg = find_existing_css_set(cg, cgrp, template);
+ if (newcg)
+ get_css_set(newcg);
+ read_unlock(&css_set_lock);
+
+ /* doesn't exist at all? */
+ if (!newcg)
+ return false;
+ /* see if it's already in the list */
+ list_for_each_entry(cg_entry, newcg_list, links) {
+ if (cg_entry->cg == newcg) {
+ put_css_set(newcg);
+ return true;
+ }
+ }
+
+ /* not found */
+ put_css_set(newcg);
+ return false;
+}
+

```

```

+/*
+ * Find the new css_set and store it in the list in preparation for moving the
+ * given task to the given cgroup. Returns 0 or -ENOMEM.
+ */
+static int css_set_prefetch(struct cgroup *cgrp, struct css_set *cg,
+    struct list_head *newcg_list)
+{
+    struct css_set *newcg;
+    struct cg_list_entry *cg_entry;
+
+    /* ensure a new css_set will exist for this thread */
+    newcg = find_css_set(cg, cgrp);
+    if (!newcg)
+        return -ENOMEM;
+    /* add it to the list */
+    cg_entry = kmalloc(sizeof(struct cg_list_entry), GFP_KERNEL);
+    if (!cg_entry) {
+        put_css_set(newcg);
+        return -ENOMEM;
+    }
+    cg_entry->cg = newcg;
+    list_add(&cg_entry->links, newcg_list);
+    return 0;
+}
+
+/**
+ * cgroup_attach_proc - attach all threads in a threadgroup to a cgroup
+ * @cgrp: the cgroup to attach to
+ * @leader: the threadgroup leader task_struct of the group to be attached
+ *
+ * Call holding cgroup_mutex and the threadgroup_fork_lock of the leader. Will
+ * take task_lock of each thread in leader's threadgroup individually in turn.
+ */
+int cgroup_attach_proc(struct cgroup *cgrp, struct task_struct *leader)
+{
+    int retval, i, group_size;
+    struct cgroup_subsys *ss, *failed_ss = NULL;
+    /* guaranteed to be initialized later, but the compiler needs this */
+    struct cgroup *oldcgrp = NULL;
+    struct css_set *oldcg;
+    struct cgroupfs_root *root = cgrp->root;
+    /* threadgroup list cursor and array */
+    struct task_struct *tsk;
+    struct task_struct **group;
+
+    /*
+     * we need to make sure we have css_sets for all the tasks we're
+     * going to move -before- we actually start moving them, so that in
+     * case we get an ENOMEM we can bail out before making any changes.

```

```

+ */
+ struct list_head newcg_list;
+ struct cg_list_entry *cg_entry, *temp_nobe;
+
+ /*
+  * step 0: in order to do expensive, possibly blocking operations for
+  * every thread, we cannot iterate the thread group list, since it needs
+  * rcu or tasklist locked. instead, build an array of all threads in the
+  * group - threadgroup_fork_lock prevents new threads from appearing,
+  * and if threads exit, this will just be an over-estimate.
+  */
+ group_size = get_nr_threads(leader);
+ group = kmalloc(group_size * sizeof(*group), GFP_KERNEL);
+ if (!group)
+ return -ENOMEM;
+
+ /* prevent changes to the threadgroup list while we take a snapshot. */
+ rcu_read_lock();
+ if (!thread_group_leader(leader)) {
+ /*
+  * a race with de_thread from another thread's exec() may strip
+  * us of our leadership, making while_each_thread unsafe to use
+  * on this task. if this happens, there is no choice but to
+  * throw this task away and try again (from cgroup_procs_write);
+  * this is "double-double-toil-and-trouble-check locking".
+  */
+ rcu_read_unlock();
+ retval = -EAGAIN;
+ goto out_free_group_list;
+ }
+ /* take a reference on each task in the group to go in the array. */
+ tsk = leader;
+ i = 0;
+ do {
+ /* as per above, nr_threads may decrease, but not increase. */
+ BUG_ON(i >= group_size);
+ get_task_struct(tsk);
+ group[i] = tsk;
+ i++;
+ } while_each_thread(leader, tsk);
+ /* remember the number of threads in the array for later. */
+ BUG_ON(i == 0);
+ group_size = i;
+ rcu_read_unlock();
+
+ /*
+  * step 1: check that we can legitimately attach to the cgroup.
+  */

```

```

+ for_each_subsys(root, ss) {
+   if (ss->can_attach) {
+     retval = ss->can_attach(ss, cgrp, leader);
+     if (retval) {
+       failed_ss = ss;
+       goto out_cancel_attach;
+     }
+   }
+   /* a callback to be run on every thread in the threadgroup. */
+   if (ss->can_attach_task) {
+     /* run on each task in the threadgroup. */
+     for (i = 0; i < group_size; i++) {
+       retval = ss->can_attach_task(cgrp, group[i]);
+       if (retval) {
+         failed_ss = ss;
+         goto out_cancel_attach;
+       }
+     }
+   }
+ }
+
+ /*
+  * step 2: make sure css_sets exist for all threads to be migrated.
+  * we use find_css_set, which allocates a new one if necessary.
+  */
+ INIT_LIST_HEAD(&newcg_list);
+ for (i = 0; i < group_size; i++) {
+   tsk = group[i];
+   /* nothing to do if this task is already in the cgroup */
+   oldcgrp = task_cgroup_from_root(tsk, root);
+   if (cgrp == oldcgrp)
+     continue;
+   /* get old css_set pointer */
+   task_lock(tsk);
+   if (tsk->flags & PF_EXITING) {
+     /* ignore this task if it's going away */
+     task_unlock(tsk);
+     continue;
+   }
+   oldcg = tsk->cgroups;
+   get_css_set(oldcg);
+   task_unlock(tsk);
+   /* see if the new one for us is already in the list? */
+   if (css_set_check_fetched(cgrp, tsk, oldcg, &newcg_list)) {
+     /* was already there, nothing to do. */
+     put_css_set(oldcg);
+   } else {
+     /* we don't already have it. get new one. */

```



```

+ retval = css_set_prefetch(cgrp, oldcgrp, &newcgrp_list);
+ put_css_set(oldcgrp);
+ if (retval)
+   goto out_list_tearardown;
+ }
+ }
+
+ /*
+  * step 3: now that we're guaranteed success wrt the css_sets, proceed
+  * to move all tasks to the new cgroup, calling ss->attach_task for each
+  * one along the way. there are no failure cases after here, so this is
+  * the commit point.
+  */
+ for_each_subsys(root, ss) {
+   if (ss->pre_attach)
+     ss->pre_attach(cgrp);
+ }
+ for (i = 0; i < group_size; i++) {
+   tsk = group[i];
+   /* leave current thread as it is if it's already there */
+   oldcgrp = task_cgroup_from_root(tsk, root);
+   if (cgrp == oldcgrp)
+     continue;
+   /* attach each task to each subsystem */
+   for_each_subsys(root, ss) {
+     if (ss->attach_task)
+       ss->attach_task(cgrp, tsk);
+   }
+   /* if the thread is PF_EXITING, it can just get skipped. */
+   retval = cgroup_task_migrate(cgrp, oldcgrp, tsk, true);
+   BUG_ON(retval != 0 && retval != -ESRCH);
+ }
+ /* nothing is sensitive to fork() after this point. */
+
+ /*
+  * step 4: do expensive, non-thread-specific subsystem callbacks.
+  * TODO: if ever a subsystem needs to know the oldcgrp for each task
+  * being moved, this call will need to be reworked to communicate that.
+  */
+ for_each_subsys(root, ss) {
+   if (ss->attach)
+     ss->attach(ss, cgrp, oldcgrp, leader);
+ }
+
+ /*
+  * step 5: success! and cleanup
+  */
+ synchronize_rcu();

```

```

+ cgroup_wakeup_rmdir_waiter(cgrp);
+ retval = 0;
+out_list_tearardown:
+ /* clean up the list of prefetched css_sets. */
+ list_for_each_entry_safe(cg_entry, temp_nobe, &newcg_list, links) {
+ list_del(&cg_entry->links);
+ put_css_set(cg_entry->cg);
+ kfree(cg_entry);
+ }
+out_cancel_attach:
+ /* same deal as in cgroup_attach_task */
+ if (retval) {
+ for_each_subsys(root, ss) {
+ if (ss == failed_ss)
+ break;
+ if (ss->cancel_attach)
+ ss->cancel_attach(ss, cgrp, leader);
+ }
+ }
+ /* clean up the array of referenced threads in the group. */
+ for (i = 0; i < group_size; i++)
+ put_task_struct(group[i]);
+out_free_group_list:
+ kfree(group);
+ return retval;
+}
+
+/*
+ * Find the task_struct of the task to attach by vpid and pass it along to the
+ * function to attach either it or all tasks in its threadgroup. Will take
+ * cgroup_mutex; may take task_lock of task.
+ */
+static int attach_task_by_pid(struct cgroup *cgrp, u64 pid)
+static int attach_task_by_pid(struct cgroup *cgrp, u64 pid, bool threadgroup)
+{
+ struct task_struct *tsk;
+ const struct cred *cred = current_cred(), *tcred;
+ int ret;

+ if (!cgroup_lock_live_group(cgrp))
+ return -ENODEV;
+
+ if (pid) {
+ rcu_read_lock();
+ tsk = find_task_by_vpid(pid);
+ if (!tsk || tsk->flags & PF_EXITING) {
+ if (!tsk) {
+ rcu_read_unlock();

```

```

+ cgroup_unlock();
+ return -ESRCH;
+ }
+ if (threadgroup) {
+ /*
+  * it is safe to find group_leader because tsk was found
+  * in the tid map, meaning it can't have been unhashed
+  * by someone in de_thread changing the leadership.
+  */
+ tsk = tsk->group_leader;
+ BUG_ON(!thread_group_leader(tsk));
+ } else if (tsk->flags & PF_EXITING) {
+ /* optimization for the single-task-only case */
+ rcu_read_unlock();
+ cgroup_unlock();
+ return -ESRCH;
+ }

+ /*
+  * even if we're attaching all tasks in the thread group, we
+  * only need to check permissions on one of them.
+  */
+ tcred = __task_cred(tsk);
+ if (cred->euid &&
+     cred->euid != tcred->uid &&
+     cred->euid != tcred->suid) {
+ rcu_read_unlock();
+ cgroup_unlock();
+ return -EACCES;
+ }
+ get_task_struct(tsk);
+ rcu_read_unlock();
+ } else {
- tsk = current;
+ if (threadgroup)
+ tsk = current->group_leader;
+ else
+ tsk = current;
+ get_task_struct(tsk);
+ }

- ret = cgroup_attach_task(cgrp, tsk);
+ if (threadgroup) {
+ threadgroup_fork_write_lock(tsk);
+ ret = cgroup_attach_proc(cgrp, tsk);
+ threadgroup_fork_write_unlock(tsk);
+ } else {
+ ret = cgroup_attach_task(cgrp, tsk);

```

```

+ }
  put_task_struct(tsk);
+ cgroup_unlock();
  return ret;
}

static int cgroup_tasks_write(struct cgroup *cgrp, struct cftype *cft, u64 pid)
{
+ return attach_task_by_pid(cgrp, pid, false);
+}
+
+static int cgroup_procs_write(struct cgroup *cgrp, struct cftype *cft, u64 tgid)
+{
  int ret;
- if (!cgroup_lock_live_group(cgrp))
- return -ENODEV;
- ret = attach_task_by_pid(cgrp, pid);
- cgroup_unlock();
+ do {
+ /*
+  * attach_proc fails with -EAGAIN if threadgroup leadership
+  * changes in the middle of the operation, in which case we need
+  * to find the task_struct for the new leader and start over.
+  */
+ ret = attach_task_by_pid(cgrp, tgid, true);
+ } while (ret == -EAGAIN);
  return ret;
}

```

```

@@ -3260,9 +3601,9 @@ static struct cftype files[] = {
{
  .name = CGROUP_FILE_GENERIC_PREFIX "procs",
  .open = cgroup_procs_open,
- /* .write_u64 = cgroup_procs_write, TODO */
+ .write_u64 = cgroup_procs_write,
  .release = cgroup_pidlist_release,
- .mode = S_IRUGO,
+ .mode = S_IRUGO | S_IWUSR,
},
{
  .name = "notify_on_release",

```

Containers mailing list

Containers@lists.linux-foundation.org

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