Subject: [PATCH v8 0/3] cgroups: implement moving a threadgroup's threads atomically with cgroup.procs
Posted by Ben Blum on Tue, 08 Feb 2011 01:35:42 GMT
View Forum Message <> Reply to Message

On Sun, Dec 26, 2010 at 07:09:19AM -0500, Ben Blum wrote:
> On Fri, Dec 24, 2010 at 03:22:26AM -0500, Ben Blum wrote:
> > On Wed, Aug 11, 2010 at 01:46:04AM -0400, Ben Blum wrote:
> > > On Fri, Jul 30, 2010 at 07:56:49PM -0400, Ben Blum wrote:
> > > > This patch series is a revision of http://lkml.org/lkml/2010/6/25/11 .
> > > >
> > > > This patch series implements a write function for the 'cgroup.procs'
> > > > per-cgroup file, which enables atomic movement of multithreaded
> > > > applications between cgroups. Writing the thread-ID of any thread in a
> > > > threadgroup to a cgroup's procs file causes all threads in the group to
> > > > be moved to that cgroup safely with respect to threads forking/exiting.
> > > > (Possible usage scenario: If running a multithreaded build system that
> > > > sucks up system resources, this lets you restrict it all at once into a
> > > > new cgroup to keep it under control.)
> > > >
> > > > Example: Suppose pid 31337 clones new threads 31338 and 31339.
> > > >
> > > > # cat /dev/cgroup/tasks
> > > > ...
> > > > 31337
> > > > 31338
> > > > 31339
> > > > # mkdir /dev/cgroup/foo
> > > > # echo 31337 > /dev/cgroup/foo/cgroup.procs
> > > > # cat /dev/cgroup/foo/tasks
> > > > 31337
> > > > 31338
> > > > 31339
> > > >
> > > > A new lock, called threadgroup_fork_lock and living in signal_struct, is
> > > > introduced to ensure atomicity when moving threads between cgroups. It's
> > > > taken for writing during the operation, and taking for reading in fork()
> > > > around the calls to cgroup_fork() and cgroup_post_fork().
> >
> > Well this time everything here is actually safe and correct, as far as
> > my best efforts and keen eyes can tell. I dropped the per_thread call
> > from the last series in favour of revising the subsystem callback
> > interface. It now looks like this:
> >
> > ss->can_attach()
> > - Thread-independent, possibly expensive/sleeping.
> >
> > ss->can_attach_task()

> - Called per-thread, run with rcu_read so must not sleep.
>
> ss->pre_attach()
> - Thread independent, must be atomic, happens before attach_task.
>
> ss->attach_task()
> - Called per-thread, run with tasklist_lock so must not sleep.
>
> ss->attach()
> - Thread independent, possibly expensive/sleeping, called last.

Okay, so.

I've revamped the cgroup_attach_proc implementation a bunch and this version should be a lot easier on the eyes (and brains). Issues that are addressed:

1) cgroup_attach_proc now iterates over leader->thread_group once, at the very beginning, and puts each task_struct that we want to move into an array, using get_task_struct to make sure they stick around.
 - threadgroup_fork_lock ensures no threads not in the array can appear, and allows us to use signal->nr_threads to determine the size of the array when kmallocing it.
 - This simplifies the rest of the function a bunch, since now we never need to do rcu_read_lock after building the array. All the subsystem callbacks are the same as described just above, but the "can't sleep" restriction is gone, so it's nice and clean.
 - Checking for a race with de_thread (the manoeuvre I refer to as "double-double-toil-and-trouble-check locking") now needs to be done only once, at the beginning (before building the array).

2) The nodemask allocation problem in cpuset is fixed the same way as before - the masks are shared between the three attach callbacks, so are made as static global variables.

3) The introduction of threadgroup_fork_lock in sched.h (specifically, in signal_struct) requires rwsem.h; the new include appears in the first patch. (An alternate plan would be to make it a struct pointer with an incomplete forward declaration and kmalloc/kfree it during housekeeping, but adding an include seems better than that particular complication.) In light of this, the definitions for threadgroup_fork_{read,write}_{un,}lock are also in sched.h.

-- Ben

 Documentation/cgroups/cgroups.txt | 39 ++-
 block/blk-cgroup.c | 18 -

```
 include/linux/cgroup.h          |   10
 include/linux/init_task.h       |    9
 include/linux/sched.h           |   37 +++
 kernel/cgroup.c                 |  454 ++++++++++++++++++++++++++++++++++-----
 kernel/cgroup_freezer.c         |   26 --
 kernel/cpuset.c                 |  105 +++++-----
 kernel/fork.c                   |   10
 kernel/ns_cgroup.c              |   23 -
 kernel/sched.c                  |   38 ---
 mm/memcontrol.c                 |   18 -
 security/device_cgroup.c        |    3
 13 files changed, 575 insertions(+), 215 deletions(-)
```

_____

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

---

Subject: [PATCH v8 1/3] cgroups: read-write lock CLONE_THREAD forking per threadgroup

Posted by Ben Blum on Tue, 08 Feb 2011 01:37:41 GMT
View Forum Message <> Reply to Message

Adds functionality to read/write lock CLONE_THREAD fork()ing per-threadgroup

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

This patch adds an rwsem that lives in a threadgroup's signal_struct that's
taken for reading in the fork path, under CONFIG_CGROUPS. If another part of
the kernel later wants to use such a locking mechanism, the CONFIG_CGROUPS
ifdefs should be changed to a higher-up flag that CGROUPS and the other system
would both depend on.

This is a pre-patch for cgroup-procs-write.patch.

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

```
---
 include/linux/init_task.h |    9 +++++++++
 include/linux/sched.h     |   37 +++++++++++++++++++++++++++++++++++++
 kernel/fork.c             |   10 ++++++++++
 3 files changed, 56 insertions(+), 0 deletions(-)

diff --git a/include/linux/init_task.h b/include/linux/init_task.h
index 6b281fa..b560381 100644
--- a/include/linux/init_task.h
+++ b/include/linux/init_task.h
@@ -15,6 +15,14 @@
 extern struct files_struct init_files;
```

```
 extern struct fs_struct init_fs;

+#ifdef CONFIG_CGROUPS
+#define INIT_THREADGROUP_FORK_LOCK(sig)     \
+ .threadgroup_fork_lock =     \
+  __RWSEM_INITIALIZER(sig.threadgroup_fork_lock),
+#else
+#define INIT_THREADGROUP_FORK_LOCK(sig)
+#endif
+
 #define INIT_SIGNALS(sig) {      \
  .nr_threads = 1,      \
  .wait_chldexit = __WAIT_QUEUE_HEAD_INITIALIZER(sig.wait_chldexit),\
@@ -31,6 +39,7 @@ extern struct fs_struct init_fs;
  },       \
  .cred_guard_mutex =      \
   __MUTEX_INITIALIZER(sig.cred_guard_mutex),  \
+ INIT_THREADGROUP_FORK_LOCK(sig)     \
 }

 extern struct nsproxy init_nsproxy;
diff --git a/include/linux/sched.h b/include/linux/sched.h
index 8580dc6..2fdbeb1 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -509,6 +509,8 @@ struct thread_group_cputimer {
 spinlock_t lock;
 };

+#include <linux/rwsem.h>
+
 /*
  * NOTE! "signal_struct" does not have it's own
  * locking, because a shared signal_struct always
@@ -623,6 +625,16 @@ struct signal_struct {
 unsigned audit_tty;
 struct tty_audit_buf *tty_audit_buf;
 #endif
+#ifdef CONFIG_CGROUPS
+ /*
+  * The threadgroup_fork_lock prevents threads from forking with
+  * CLONE_THREAD while held for writing. Use this for fork-sensitive
+  * threadgroup-wide operations. It's taken for reading in fork.c in
+  * copy_process().
+  * Currently only needed write-side by cgroups.
+  */
+ struct rw_semaphore threadgroup_fork_lock;
+#endif
```

```
  int oom_adj;  /* OOM kill score adjustment (bit shift) */
  int oom_score_adj; /* OOM kill score adjustment */
@@ -2270,6 +2282,31 @@ static inline void unlock_task_sighand(struct task_struct *tsk,
  spin_unlock_irqrestore(&tsk->sighand->siglock, *flags);
 }

+/* See the declaration of threadgroup_fork_lock in signal_struct. */
+#ifdef CONFIG_CGROUPS
+static inline void threadgroup_fork_read_lock(struct task_struct *tsk)
+{
+ down_read(&tsk->signal->threadgroup_fork_lock);
+}
+static inline void threadgroup_fork_read_unlock(struct task_struct *tsk)
+{
+ up_read(&tsk->signal->threadgroup_fork_lock);
+}
+static inline void threadgroup_fork_write_lock(struct task_struct *tsk)
+{
+ down_write(&tsk->signal->threadgroup_fork_lock);
+}
+static inline void threadgroup_fork_write_unlock(struct task_struct *tsk)
+{
+ up_write(&tsk->signal->threadgroup_fork_lock);
+}
+#else
+static inline void threadgroup_fork_read_lock(struct task_struct *tsk) {}
+static inline void threadgroup_fork_read_unlock(struct task_struct *tsk) {}
+static inline void threadgroup_fork_write_lock(struct task_struct *tsk) {}
+static inline void threadgroup_fork_write_unlock(struct task_struct *tsk) {}
+#endif
+
 #ifndef __HAVE_THREAD_FUNCTIONS

 #define task_thread_info(task) ((struct thread_info *)(task)->stack)
diff --git a/kernel/fork.c b/kernel/fork.c
index 0979527..aefe61f 100644
--- a/kernel/fork.c
+++ b/kernel/fork.c
@@ -905,6 +905,10 @@ static int copy_signal(unsigned long clone_flags, struct task_struct *tsk)

  tty_audit_fork(sig);

+#ifdef CONFIG_CGROUPS
+ init_rwsem(&sig->threadgroup_fork_lock);
+#endif
+
  sig->oom_adj = current->signal->oom_adj;
```

```
  sig->oom_score_adj = current->signal->oom_score_adj;
  sig->oom_score_adj_min = current->signal->oom_score_adj_min;
@@ -1087,6 +1091,8 @@ static struct task_struct *copy_process(unsigned long clone_flags,
  monotonic_to_bootbased(&p->real_start_time);
  p->io_context = NULL;
  p->audit_context = NULL;
+ if (clone_flags & CLONE_THREAD)
+  threadgroup_fork_read_lock(current);
  cgroup_fork(p);
 #ifdef CONFIG_NUMA
  p->mempolicy = mpol_dup(p->mempolicy);
@@ -1294,6 +1300,8 @@ static struct task_struct *copy_process(unsigned long clone_flags,
  write_unlock_irq(&tasklist_lock);
  proc_fork_connector(p);
  cgroup_post_fork(p);
+ if (clone_flags & CLONE_THREAD)
+  threadgroup_fork_read_unlock(current);
  perf_event_fork(p);
  return p;

@@ -1332,6 +1340,8 @@ bad_fork_cleanup_policy:
  mpol_put(p->mempolicy);
 bad_fork_cleanup_cgroup:
 #endif
+ if (clone_flags & CLONE_THREAD)
+  threadgroup_fork_read_unlock(current);
  cgroup_exit(p, cgroup_callbacks_done);
  delayacct_tsk_free(p);
  module_put(task_thread_info(p)->exec_domain->module);
```

_____

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

---

Subject: [PATCH v8 2/3] cgroups: add per-thread subsystem callbacks
Posted by Ben Blum on Tue, 08 Feb 2011 01:39:15 GMT
View Forum Message <> Reply to Message

Add cgroup subsystem callbacks for per-thread attachment

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

This patch adds can_attach_task, pre_attach, and attach_task as new callbacks for cgroups's subsystem interface. Unlike can_attach and attach, these are for per-thread operations, to be called potentially many times when attaching an entire threadgroup.

Also, the old "bool threadgroup" interface is removed, as replaced by this.
All subsystems are modified for the new interface - of note is cpuset, which
requires from/to nodemasks for attach to be globally scoped (though per-cpuset
would work too) to persist from its pre_attach to attach_task and attach.

This is a pre-patch for cgroup-procs-writable.patch.

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

 Documentation/cgroups/cgroups.txt | 30 ++++++++---
 block/blk-cgroup.c | 18 ++----
 include/linux/cgroup.h | 10 ++--
 kernel/cgroup.c | 17 +++++-
 kernel/cgroup_freezer.c | 26 ++++-----
 kernel/cpuset.c | 105 +++++++++++++++++++++----------------
 kernel/ns_cgroup.c | 23 +++----
 kernel/sched.c | 38 +++----------
 mm/memcontrol.c | 18 ++----
 security/device_cgroup.c | 3 -
 10 files changed, 122 insertions(+), 166 deletions(-)

diff --git a/Documentation/cgroups/cgroups.txt b/Documentation/cgroups/cgroups.txt
index 190018b..d3c9a24 100644
--- a/Documentation/cgroups/cgroups.txt
+++ b/Documentation/cgroups/cgroups.txt
@@ -563,7 +563,7 @@ rmdir() will fail with it. From this behavior, pre_destroy() can be
 called multiple times against a cgroup.

 int can_attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
-	 struct task_struct *task, bool threadgroup)
+	 struct task_struct *task)
 (cgroup_mutex held by caller)

 Called prior to moving a task into a cgroup; if the subsystem
@@ -572,9 +572,14 @@ task is passed, then a successful result indicates that *any*
 unspecified task can be moved into the cgroup. Note that this isn't
 called on a fork. If this method returns 0 (success) then this should
 remain valid while the caller holds cgroup_mutex and it is ensured that either
-attach() or cancel_attach() will be called in future. If threadgroup is
-true, then a successful result indicates that all threads in the given
-thread's threadgroup can be moved together.
+attach() or cancel_attach() will be called in future.
+
+int can_attach_task(struct cgroup *cgrp, struct task_struct *tsk);
+(cgroup_mutex held by caller)
+
+As can_attach, but for operations that must be run once per task to be
+attached (possibly many when using cgroup_attach_proc). Called after

```
+can_attach.

 void cancel_attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
         struct task_struct *task, bool threadgroup)
@@ -586,15 +591,24 @@ function, so that the subsystem can implement a rollback. If not, not
necessary.
 This will be called only about subsystems whose can_attach() operation have
 succeeded.

+void pre_attach(struct cgroup *cgrp);
+(cgroup_mutex held by caller)
+
+For any non-per-thread attachment work that needs to happen before
+attach_task. Needed by cpuset.
+
 void attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
-     struct cgroup *old_cgrp, struct task_struct *task,
-     bool threadgroup)
+     struct cgroup *old_cgrp, struct task_struct *task)
 (cgroup_mutex held by caller)

 Called after the task has been attached to the cgroup, to allow any
 post-attachment activity that requires memory allocations or blocking.
-If threadgroup is true, the subsystem should take care of all threads
-in the specified thread's threadgroup. Currently does not support any
+
+void attach_task(struct cgroup *cgrp, struct task_struct *tsk);
+(cgroup_mutex held by caller)
+
+As attach, but for operations that must be run once per task to be attached,
+like can_attach_task. Called before attach. Currently does not support any
 subsystem that might need the old_cgrp for every thread in the group.

 void fork(struct cgroup_subsy *ss, struct task_struct *task)
diff --git a/block/blk-cgroup.c b/block/blk-cgroup.c
index b1febd0..45b3809 100644
--- a/block/blk-cgroup.c
+++ b/block/blk-cgroup.c
@@ -30,10 +30,8 @@ EXPORT_SYMBOL_GPL(blkio_root_cgroup);

 static struct cgroup_subsys_state *blkiocg_create(struct cgroup_subsys *,
         struct cgroup *);
-static int blkiocg_can_attach(struct cgroup_subsys *, struct cgroup *,
-         struct task_struct *, bool);
-static void blkiocg_attach(struct cgroup_subsys *, struct cgroup *,
-      struct cgroup *, struct task_struct *, bool);
+static int blkiocg_can_attach_task(struct cgroup *, struct task_struct *);
+static void blkiocg_attach_task(struct cgroup *, struct task_struct *);
```

```
 static void blkiocg_destroy(struct cgroup_subsys *, struct cgroup *);
 static int blkiocg_populate(struct cgroup_subsys *, struct cgroup *);

@@ -46,8 +44,8 @@ static int blkiocg_populate(struct cgroup_subsys *, struct cgroup *);
 struct cgroup_subsys blkio_subsys = {
  .name = "blkio",
  .create = blkiocg_create,
- .can_attach = blkiocg_can_attach,
- .attach = blkiocg_attach,
+ .can_attach_task = blkiocg_can_attach_task,
+ .attach_task = blkiocg_attach_task,
  .destroy = blkiocg_destroy,
  .populate = blkiocg_populate,
 #ifdef CONFIG_BLK_CGROUP
@@ -1475,9 +1473,7 @@ done:
  * of the main cic data structures.  For now we allow a task to change
  * its cgroup only if it's the only owner of its ioc.
  */
-static int blkiocg_can_attach(struct cgroup_subsys *subsys,
-    struct cgroup *cgroup, struct task_struct *tsk,
-    bool threadgroup)
+static int blkiocg_can_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
 {
  struct io_context *ioc;
  int ret = 0;
@@ -1492,9 +1488,7 @@ static int blkiocg_can_attach(struct cgroup_subsys *subsys,
  return ret;
 }

-static void blkiocg_attach(struct cgroup_subsys *subsys, struct cgroup *cgroup,
-    struct cgroup *prev, struct task_struct *tsk,
-    bool threadgroup)
+static void blkiocg_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
 {
  struct io_context *ioc;

diff --git a/include/linux/cgroup.h b/include/linux/cgroup.h
index ce104e3..35b69b4 100644
--- a/include/linux/cgroup.h
+++ b/include/linux/cgroup.h
@@ -467,12 +467,14 @@ struct cgroup_subsys {
  int (*pre_destroy)(struct cgroup_subsys *ss, struct cgroup *cgrp);
  void (*destroy)(struct cgroup_subsys *ss, struct cgroup *cgrp);
  int (*can_attach)(struct cgroup_subsys *ss, struct cgroup *cgrp,
-     struct task_struct *tsk, bool threadgroup);
+     struct task_struct *tsk);
+ int (*can_attach_task)(struct cgroup *cgrp, struct task_struct *tsk);
  void (*cancel_attach)(struct cgroup_subsys *ss, struct cgroup *cgrp,
```

```
-     struct task_struct *tsk, bool threadgroup);
+          struct task_struct *tsk);
+ void (*pre_attach)(struct cgroup *cgrp);
+ void (*attach_task)(struct cgroup *cgrp, struct task_struct *tsk);
  void (*attach)(struct cgroup_subsys *ss, struct cgroup *cgrp,
-    struct cgroup *old_cgrp, struct task_struct *tsk,
-    bool threadgroup);
+          struct cgroup *old_cgrp, struct task_struct *tsk);
  void (*fork)(struct cgroup_subsys *ss, struct task_struct *task);
  void (*exit)(struct cgroup_subsys *ss, struct task_struct *task);
  int (*populate)(struct cgroup_subsys *ss,
diff --git a/kernel/cgroup.c b/kernel/cgroup.c
index 66a416b..616f27a 100644
--- a/kernel/cgroup.c
+++ b/kernel/cgroup.c
@@ -1750,7 +1750,7 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)

  for_each_subsys(root, ss) {
   if (ss->can_attach) {
-    retval = ss->can_attach(ss, cgrp, tsk, false);
+    retval = ss->can_attach(ss, cgrp, tsk);
     if (retval) {
      /*
       * Remember on which subsystem the can_attach()
@@ -1762,6 +1762,13 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
      goto out;
     }
    }
+   if (ss->can_attach_task) {
+    retval = ss->can_attach_task(cgrp, tsk);
+    if (retval) {
+     failed_ss = ss;
+     goto out;
+    }
+   }
  }

  task_lock(tsk);
@@ -1798,8 +1805,12 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
  write_unlock(&css_set_lock);

  for_each_subsys(root, ss) {
+   if (ss->pre_attach)
+    ss->pre_attach(cgrp);
+   if (ss->attach_task)
+    ss->attach_task(cgrp, tsk);
    if (ss->attach)
-    ss->attach(ss, cgrp, oldcgrp, tsk, false);
```

```
+   ss->attach(ss, cgrp, oldcgrp, tsk);
  }
  set_bit(CGRP_RELEASABLE, &oldcgrp->flags);
  synchronize_rcu();
@@ -1822,7 +1833,7 @@ out:
       */
      break;
    if (ss->cancel_attach)
-    ss->cancel_attach(ss, cgrp, tsk, false);
+    ss->cancel_attach(ss, cgrp, tsk);
   }
  }
  return retval;
diff --git a/kernel/cgroup_freezer.c b/kernel/cgroup_freezer.c
index e7bebb7..e691818 100644
--- a/kernel/cgroup_freezer.c
+++ b/kernel/cgroup_freezer.c
@@ -160,7 +160,7 @@ static void freezer_destroy(struct cgroup_subsys *ss,
  */
 static int freezer_can_attach(struct cgroup_subsys *ss,
          struct cgroup *new_cgroup,
-         struct task_struct *task, bool threadgroup)
+         struct task_struct *task)
 {
  struct freezer *freezer;

@@ -172,26 +172,17 @@ static int freezer_can_attach(struct cgroup_subsys *ss,
  if (freezer->state != CGROUP_THAWED)
   return -EBUSY;

+ return 0;
+}
+
+static int freezer_can_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
+{
  rcu_read_lock();
- if (__cgroup_freezing_or_frozen(task)) {
+ if (__cgroup_freezing_or_frozen(tsk)) {
   rcu_read_unlock();
   return -EBUSY;
  }
  rcu_read_unlock();
-
- if (threadgroup) {
-  struct task_struct *c;
-
-  rcu_read_lock();
-  list_for_each_entry_rcu(c, &task->thread_group, thread_group) {
```

```
-   if (__cgroup_freezing_or_frozen(c)) {
-    rcu_read_unlock();
-    return -EBUSY;
-   }
-  }
-  rcu_read_unlock();
- }
-
  return 0;
 }

@@ -390,6 +381,9 @@ struct cgroup_subsys freezer_subsys = {
  .populate = freezer_populate,
  .subsys_id = freezer_subsys_id,
  .can_attach = freezer_can_attach,
+ .can_attach_task = freezer_can_attach_task,
+ .pre_attach = NULL,
+ .attach_task = NULL,
  .attach  = NULL,
  .fork  = freezer_fork,
  .exit  = NULL,
diff --git a/kernel/cpuset.c b/kernel/cpuset.c
index 4349935..5f71ca2 100644
--- a/kernel/cpuset.c
+++ b/kernel/cpuset.c
@@ -1372,14 +1372,10 @@ static int fmeter_getrate(struct fmeter *fmp)
  return val;
 }

-/* Protected by cgroup_lock */
-static cpumask_var_t cpus_attach;
-
 /* Called by cgroups to determine if a cpuset is usable; cgroup_mutex held */
 static int cpuset_can_attach(struct cgroup_subsys *ss, struct cgroup *cont,
-        struct task_struct *tsk, bool threadgroup)
+        struct task_struct *tsk)
 {
- int ret;
  struct cpuset *cs = cgroup_cs(cont);

  if (cpumask_empty(cs->cpus_allowed) || nodes_empty(cs->mems_allowed))
@@ -1396,29 +1392,42 @@ static int cpuset_can_attach(struct cgroup_subsys *ss, struct
cgroup *cont,
  if (tsk->flags & PF_THREAD_BOUND)
   return -EINVAL;

- ret = security_task_setscheduler(tsk);
- if (ret)
```

```
-  return ret;
- if (threadgroup) {
-  struct task_struct *c;
-
-  rcu_read_lock();
-  list_for_each_entry_rcu(c, &tsk->thread_group, thread_group) {
-   ret = security_task_setscheduler(c);
-   if (ret) {
-    rcu_read_unlock();
-    return ret;
-   }
-  }
-  rcu_read_unlock();
- }
  return 0;
 }

-static void cpuset_attach_task(struct task_struct *tsk, nodemask_t *to,
-          struct cpuset *cs)
+static int cpuset_can_attach_task(struct cgroup *cgrp, struct task_struct *task)
+{
+ return security_task_setscheduler(task);
+}
+
+/*
+ * Protected by cgroup_lock. The nodemasks must be stored globally because
+ * dynamically allocating them is not allowed in pre_attach, and they must
+ * persist among pre_attach, attach_task, and attach.
+ */
+static cpumask_var_t cpus_attach;
+static nodemask_t cpuset_attach_nodemask_from;
+static nodemask_t cpuset_attach_nodemask_to;
+
+/* Set-up work for before attaching each task. */
+static void cpuset_pre_attach(struct cgroup *cont)
+{
+ struct cpuset *cs = cgroup_cs(cont);
+
+ if (cs == &top_cpuset)
+  cpumask_copy(cpus_attach, cpu_possible_mask);
+ else
+  guarantee_online_cpus(cs, cpus_attach);
+
+ guarantee_online_mems(cs, &cpuset_attach_nodemask_to);
+}
+
+/* Per-thread attachment work. */
+static void cpuset_attach_task(struct cgroup *cont, struct task_struct *tsk)
```

```
 {
  int err;
+ struct cpuset *cs = cgroup_cs(cont);
+
  /*
   * can_attach beforehand should guarantee that this doesn't fail.
   * TODO: have a better way to handle failure here
@@ -1426,56 +1435,31 @@ static void cpuset_attach_task(struct task_struct *tsk, nodemask_t
*to,
  err = set_cpus_allowed_ptr(tsk, cpus_attach);
  WARN_ON_ONCE(err);

- cpuset_change_task_nodemask(tsk, to);
+ cpuset_change_task_nodemask(tsk, &cpuset_attach_nodemask_to);
  cpuset_update_task_spread_flag(cs, tsk);
-
 }

 static void cpuset_attach(struct cgroup_subsys *ss, struct cgroup *cont,
-      struct cgroup *oldcont, struct task_struct *tsk,
-      bool threadgroup)
+      struct cgroup *oldcont, struct task_struct *tsk)
 {
  struct mm_struct *mm;
  struct cpuset *cs = cgroup_cs(cont);
  struct cpuset *oldcs = cgroup_cs(oldcont);
- NODEMASK_ALLOC(nodemask_t, from, GFP_KERNEL);
- NODEMASK_ALLOC(nodemask_t, to, GFP_KERNEL);
-
- if (from == NULL || to == NULL)
-  goto alloc_fail;

- if (cs == &top_cpuset) {
-  cpumask_copy(cpus_attach, cpu_possible_mask);
- } else {
-  guarantee_online_cpus(cs, cpus_attach);
- }
- guarantee_online_mems(cs, to);
-
- /* do per-task migration stuff possibly for each in the threadgroup */
- cpuset_attach_task(tsk, to, cs);
- if (threadgroup) {
-  struct task_struct *c;
-  rcu_read_lock();
-  list_for_each_entry_rcu(c, &tsk->thread_group, thread_group) {
-   cpuset_attach_task(c, to, cs);
-  }
-  rcu_read_unlock();
```

```
- }
-
- /* change mm; only needs to be done once even if threadgroup */
- *from = oldcs->mems_allowed;
- *to = cs->mems_allowed;
+ /*
+  * Change mm, possibly for multiple threads in a threadgroup. This is
+  * expensive and may sleep.
+  */
+ cpuset_attach_nodemask_from = oldcs->mems_allowed;
+ cpuset_attach_nodemask_to = cs->mems_allowed;
  mm = get_task_mm(tsk);
  if (mm) {
-  mpol_rebind_mm(mm, to);
+  mpol_rebind_mm(mm, &cpuset_attach_nodemask_to);
   if (is_memory_migrate(cs))
-   cpuset_migrate_mm(mm, from, to);
+   cpuset_migrate_mm(mm, &cpuset_attach_nodemask_from,
+       &cpuset_attach_nodemask_to);
   mmput(mm);
  }
-
-alloc_fail:
- NODEMASK_FREE(from);
- NODEMASK_FREE(to);
 }

 /* The various types of files and directories in a cpuset file system */
@@ -1928,6 +1912,9 @@ struct cgroup_subsys cpuset_subsys = {
  .create = cpuset_create,
  .destroy = cpuset_destroy,
  .can_attach = cpuset_can_attach,
+ .can_attach_task = cpuset_can_attach_task,
+ .pre_attach = cpuset_pre_attach,
+ .attach_task = cpuset_attach_task,
  .attach = cpuset_attach,
  .populate = cpuset_populate,
  .post_clone = cpuset_post_clone,
diff --git a/kernel/ns_cgroup.c b/kernel/ns_cgroup.c
index 2c98ad9..1fc2b1b 100644
--- a/kernel/ns_cgroup.c
+++ b/kernel/ns_cgroup.c
@@ -43,7 +43,7 @@ int ns_cgroup_clone(struct task_struct *task, struct pid *pid)
  *       ancestor cgroup thereof)
  */
 static int ns_can_attach(struct cgroup_subsys *ss, struct cgroup *new_cgroup,
-    struct task_struct *task, bool threadgroup)
+    struct task_struct *task)
```

```
 {
  if (current != task) {
   if (!capable(CAP_SYS_ADMIN))
@@ -53,21 +53,13 @@ static int ns_can_attach(struct cgroup_subsys *ss, struct cgroup
*new_cgroup,
    return -EPERM;
  }

- if (!cgroup_is_descendant(new_cgroup, task))
-  return -EPERM;
-
- if (threadgroup) {
-  struct task_struct *c;
-  rcu_read_lock();
-  list_for_each_entry_rcu(c, &task->thread_group, thread_group) {
-   if (!cgroup_is_descendant(new_cgroup, c)) {
-    rcu_read_unlock();
-    return -EPERM;
-   }
-  }
-  rcu_read_unlock();
- }
+ return 0;
+}

+static int ns_can_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
+{
+ if (!cgroup_is_descendant(cgrp, tsk))
+  return -EPERM;
  return 0;
 }

@@ -112,6 +104,7 @@ static void ns_destroy(struct cgroup_subsys *ss,
 struct cgroup_subsys ns_subsys = {
  .name = "ns",
  .can_attach = ns_can_attach,
+ .can_attach_task = ns_can_attach_task,
  .create = ns_create,
  .destroy  = ns_destroy,
  .subsys_id = ns_subsys_id,
diff --git a/kernel/sched.c b/kernel/sched.c
index 218ef20..d619f1d 100644
--- a/kernel/sched.c
+++ b/kernel/sched.c
@@ -8655,42 +8655,10 @@ cpu_cgroup_can_attach_task(struct cgroup *cgrp, struct task_struct
*tsk)
  return 0;
 }
```

```
-static int
-cpu_cgroup_can_attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
-        struct task_struct *tsk, bool threadgroup)
-{
- int retval = cpu_cgroup_can_attach_task(cgrp, tsk);
- if (retval)
-  return retval;
- if (threadgroup) {
-  struct task_struct *c;
-  rcu_read_lock();
-  list_for_each_entry_rcu(c, &tsk->thread_group, thread_group) {
-   retval = cpu_cgroup_can_attach_task(cgrp, c);
-   if (retval) {
-    rcu_read_unlock();
-    return retval;
-   }
-  }
-  rcu_read_unlock();
- }
- return 0;
-}
-
 static void
-cpu_cgroup_attach(struct cgroup_subsys *ss, struct cgroup *cgrp,
-    struct cgroup *old_cont, struct task_struct *tsk,
-    bool threadgroup)
+cpu_cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
 {
  sched_move_task(tsk);
- if (threadgroup) {
-  struct task_struct *c;
-  rcu_read_lock();
-  list_for_each_entry_rcu(c, &tsk->thread_group, thread_group) {
-   sched_move_task(c);
-  }
-  rcu_read_unlock();
- }
 }

 #ifdef CONFIG_FAIR_GROUP_SCHED
@@ -8763,8 +8731,8 @@ struct cgroup_subsys cpu_cgroup_subsys = {
  .name  = "cpu",
  .create  = cpu_cgroup_create,
  .destroy = cpu_cgroup_destroy,
- .can_attach = cpu_cgroup_can_attach,
- .attach  = cpu_cgroup_attach,
+ .can_attach_task = cpu_cgroup_can_attach_task,
```

```
+ .attach_task = cpu_cgroup_attach_task,
  .populate = cpu_cgroup_populate,
  .subsys_id = cpu_cgroup_subsys_id,
  .early_init = 1,
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 729beb7..995f0b9 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -4720,8 +4720,7 @@ static void mem_cgroup_clear_mc(void)

 static int mem_cgroup_can_attach(struct cgroup_subsys *ss,
      struct cgroup *cgroup,
-     struct task_struct *p,
-     bool threadgroup)
+     struct task_struct *p)
 {
  int ret = 0;
  struct mem_cgroup *mem = mem_cgroup_from_cont(cgroup);
@@ -4775,8 +4774,7 @@ static int mem_cgroup_can_attach(struct cgroup_subsys *ss,

 static void mem_cgroup_cancel_attach(struct cgroup_subsys *ss,
      struct cgroup *cgroup,
-     struct task_struct *p,
-     bool threadgroup)
+     struct task_struct *p)
 {
  mem_cgroup_clear_mc();
 }
@@ -4880,8 +4878,7 @@ static void mem_cgroup_move_charge(struct mm_struct *mm)
 static void mem_cgroup_move_task(struct cgroup_subsys *ss,
      struct cgroup *cont,
      struct cgroup *old_cont,
-     struct task_struct *p,
-     bool threadgroup)
+     struct task_struct *p)
 {
  if (!mc.mm)
   /* no need to move charge */
@@ -4893,22 +4890,19 @@ static void mem_cgroup_move_task(struct cgroup_subsys *ss,
 #else /* !CONFIG_MMU */
 static int mem_cgroup_can_attach(struct cgroup_subsys *ss,
      struct cgroup *cgroup,
-     struct task_struct *p,
-     bool threadgroup)
+     struct task_struct *p)
 {
  return 0;
 }
```

```
 static void mem_cgroup_cancel_attach(struct cgroup_subsys *ss,
     struct cgroup *cgroup,
-    struct task_struct *p,
-    bool threadgroup)
+    struct task_struct *p)
 {
 }
 static void mem_cgroup_move_task(struct cgroup_subsys *ss,
     struct cgroup *cont,
     struct cgroup *old_cont,
-    struct task_struct *p,
-    bool threadgroup)
+    struct task_struct *p)
 {
 }
 #endif
diff --git a/security/device_cgroup.c b/security/device_cgroup.c
index 8d9c48f..cd1f779 100644
--- a/security/device_cgroup.c
+++ b/security/device_cgroup.c
@@ -62,8 +62,7 @@ static inline struct dev_cgroup *task_devcgroup(struct task_struct *task)
 struct cgroup_subsys devices_subsys;

 static int devcgroup_can_attach(struct cgroup_subsys *ss,
-  struct cgroup *new_cgroup, struct task_struct *task,
-  bool threadgroup)
+  struct cgroup *new_cgroup, struct task_struct *task)
 {
   if (current != task && !capable(CAP_SYS_ADMIN))
     return -EPERM;
```

_____

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

---

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 tgids in the cgroup. This list is not
    guaranteed to be sorted or free of duplicate tgids, 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, oldcg, &newcg_list);
+   put_css_set(oldcg);
+   if (retval)
+    goto out_list_teardown;
+  }
+ }
+
+ /*
+  * 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_teardown:
+ /* 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/containe rs

---

Subject: Re: [PATCH v8 0/3] cgroups: implement moving a threadgroup's threads atomically with cgroup.procs

Posted by akpm on Wed, 09 Feb 2011 23:10:46 GMT

View Forum Message <> Reply to Message

On Mon, 7 Feb 2011 20:35:42 -0500
Ben Blum <bblum@andrew.cmu.edu> wrote:

> On Sun, Dec 26, 2010 at 07:09:19AM -0500, Ben Blum wrote:
> > On Fri, Dec 24, 2010 at 03:22:26AM -0500, Ben Blum wrote:
> > > On Wed, Aug 11, 2010 at 01:46:04AM -0400, Ben Blum wrote:
> > > > On Fri, Jul 30, 2010 at 07:56:49PM -0400, Ben Blum wrote:
> > > > > This patch series is a revision of http://lkml.org/lkml/2010/6/25/11 .

> > > > >
> > > > > This patch series implements a write function for the 'cgroup.procs'
> > > > > per-cgroup file, which enables atomic movement of multithreaded
> > > > > applications between cgroups. Writing the thread-ID of any thread in a
> > > > > threadgroup to a cgroup's procs file causes all threads in the group to
> > > > > be moved to that cgroup safely with respect to threads forking/exiting.
> > > > > (Possible usage scenario: If running a multithreaded build system that
> > > > > sucks up system resources, this lets you restrict it all at once into a
> > > > > new cgroup to keep it under control.)
> > > > >
> > > > > Example: Suppose pid 31337 clones new threads 31338 and 31339.
> > > > >
> > > > > # cat /dev/cgroup/tasks
> > > > > ...
> > > > > 31337
> > > > > 31338
> > > > > 31339
> > > > > # mkdir /dev/cgroup/foo
> > > > > # echo 31337 > /dev/cgroup/foo/cgroup.procs
> > > > > # cat /dev/cgroup/foo/tasks
> > > > > 31337
> > > > > 31338
> > > > > 31339
> > > > >
> > > > > A new lock, called threadgroup_fork_lock and living in signal_struct, is
> > > > > introduced to ensure atomicity when moving threads between cgroups. It's
> > > > > taken for writing during the operation, and taking for reading in fork()
> > > > > around the calls to cgroup_fork() and cgroup_post_fork().

The above six month old text is the best (and almost the only)
explanation of the rationale for the entire patch series. Is
it still correct and complete?

Assuming "yes", then... how do we determine whether the feature is
sufficiently useful to justify merging and maintaining it? Will people
actually use it?

Was there some particular operational situation which led you to think
that the kernel should have this capability? If so, please help us out here
and lavishly describe it.

_____

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

Subject: Re: [PATCH v8 0/3] cgroups: implement moving a threadgroup's threads atomically with cgroup.procs
Posted by KAMEZAWA Hiroyuki on Thu, 10 Feb 2011 01:02:10 GMT
View Forum Message <> Reply to Message

On Wed, 9 Feb 2011 15:10:46 -0800
Andrew Morton <akpm@linux-foundation.org> wrote:

> On Mon, 7 Feb 2011 20:35:42 -0500
> Ben Blum <bblum@andrew.cmu.edu> wrote:
>
> > On Sun, Dec 26, 2010 at 07:09:19AM -0500, Ben Blum wrote:
> > > On Fri, Dec 24, 2010 at 03:22:26AM -0500, Ben Blum wrote:
> > > > On Wed, Aug 11, 2010 at 01:46:04AM -0400, Ben Blum wrote:
> > > > > On Fri, Jul 30, 2010 at 07:56:49PM -0400, Ben Blum wrote:
> > > > > > This patch series is a revision of http://lkml.org/lkml/2010/6/25/11 .
> > > > > >
> > > > > > This patch series implements a write function for the 'cgroup.procs'
> > > > > > per-cgroup file, which enables atomic movement of multithreaded
> > > > > > applications between cgroups. Writing the thread-ID of any thread in a
> > > > > > threadgroup to a cgroup's procs file causes all threads in the group to
> > > > > > be moved to that cgroup safely with respect to threads forking/exiting.
> > > > > > (Possible usage scenario: If running a multithreaded build system that
> > > > > > sucks up system resources, this lets you restrict it all at once into a
> > > > > > new cgroup to keep it under control.)
> > > > > >
> > > > > > Example: Suppose pid 31337 clones new threads 31338 and 31339.
> > > > > >
> > > > > > # cat /dev/cgroup/tasks
> > > > > > ...
> > > > > > 31337
> > > > > > 31338
> > > > > > 31339
> > > > > > # mkdir /dev/cgroup/foo
> > > > > > # echo 31337 > /dev/cgroup/foo/cgroup.procs
> > > > > > # cat /dev/cgroup/foo/tasks
> > > > > > 31337
> > > > > > 31338
> > > > > > 31339
> > > > > >
> > > > > > A new lock, called threadgroup_fork_lock and living in signal_struct, is
> > > > > > introduced to ensure atomicity when moving threads between cgroups. It's
> > > > > > taken for writing during the operation, and taking for reading in fork()
> > > > > > around the calls to cgroup_fork() and cgroup_post_fork().
>
> The above six month old text is the best (and almost the only)
> explanation of the rationale for the entire patch series. Is
> it still correct and complete?
>

>
> Assuming "yes", then... how do we determine whether the feature is
> sufficiently useful to justify merging and maintaining it? Will people
> actually use it?
>
> Was there some particular operational situation which led you to think
> that the kernel should have this capability? If so, please help us out here
> and lavishly describe it.
>

In these months, I saw following questions as
==
Q. I think I put qemu to xxxx cgroup but it never works!
A. You need to put all threads in qemu to cgroup.
==

'tasks' file is not useful interface for users, I think.
(Even if users tend to use put-task-before-exec scheme.)

IMHO, from user's side of view, 'tasks' file is a mystery.

TID(thread-ID) is one of secrets in Linux + pthread library. For example,
on RHEL6, to use gettid(), users has to use syscall() directly. And end-user
may not know about thread-ID which is hidden under pthreads.

IIRC, there are no interface other than /proc/<pid>/tasks which shows all
thread IDs of a process. But it's not atomic.

So, I think it's ok to have 'procs' interface for cgroup if
overhead/impact of patch is not heavy.

Thanks,
-Kame

Containers mailing list
Containers@lists.linux-foundation.org

Subject: Re: [PATCH v8 0/3] cgroups: implement moving a threadgroup's threads atomically with cgroup.procs
Posted by Ben Blum on Thu, 10 Feb 2011 01:36:06 GMT
View Forum Message <> Reply to Message

On Thu, Feb 10, 2011 at 10:02:10AM +0900, KAMEZAWA Hiroyuki wrote:
> On Wed, 9 Feb 2011 15:10:46 -0800
> Andrew Morton <akpm@linux-foundation.org> wrote:
>
> > On Mon, 7 Feb 2011 20:35:42 -0500
> > Ben Blum <bblum@andrew.cmu.edu> wrote:
> >
> > > On Sun, Dec 26, 2010 at 07:09:19AM -0500, Ben Blum wrote:
> > > > On Fri, Dec 24, 2010 at 03:22:26AM -0500, Ben Blum wrote:
> > > > > On Wed, Aug 11, 2010 at 01:46:04AM -0400, Ben Blum wrote:
> > > > > > On Fri, Jul 30, 2010 at 07:56:49PM -0400, Ben Blum wrote:
> > > > > > > This patch series is a revision of http://lkml.org/lkml/2010/6/25/11 .
> > > > > > >
> > > > > > > This patch series implements a write function for the 'cgroup.procs'
> > > > > > > per-cgroup file, which enables atomic movement of multithreaded
> > > > > > > applications between cgroups. Writing the thread-ID of any thread in a
> > > > > > > threadgroup to a cgroup's procs file causes all threads in the group to
> > > > > > > be moved to that cgroup safely with respect to threads forking/exiting.
> > > > > > > (Possible usage scenario: If running a multithreaded build system that
> > > > > > > sucks up system resources, this lets you restrict it all at once into a
> > > > > > > new cgroup to keep it under control.)
> > > > > > >
> > > > > > > Example: Suppose pid 31337 clones new threads 31338 and 31339.
> > > > > > >
> > > > > > > # cat /dev/cgroup/tasks
> > > > > > > ...
> > > > > > > 31337
> > > > > > > 31338
> > > > > > > 31339
> > > > > > > # mkdir /dev/cgroup/foo
> > > > > > > # echo 31337 > /dev/cgroup/foo/cgroup.procs
> > > > > > > # cat /dev/cgroup/foo/tasks
> > > > > > > 31337
> > > > > > > 31338
> > > > > > > 31339
> > > > > > >
> > > > > > > A new lock, called threadgroup_fork_lock and living in signal_struct, is
> > > > > > > introduced to ensure atomicity when moving threads between cgroups. It's
> > > > > > > taken for writing during the operation, and taking for reading in fork()
> > > > > > > around the calls to cgroup_fork() and cgroup_post_fork().

> >
> > The above six month old text is the best (and almost the only)
> > explanation of the rationale for the entire patch series. Is
> > it still correct and complete?

Yep, it's still fresh. (That's why I kept it around!)

> >
> >
> > Assuming "yes", then... how do we determine whether the feature is
> > sufficiently useful to justify merging and maintaining it? Will people
> > actually use it?
> >
> > Was there some particular operational situation which led you to think
> > that the kernel should have this capability? If so, please help us out here
> > and lavishly describe it.
> >
>
> In these months, I saw following questions as
> ==
> Q. I think I put qemu to xxxx cgroup but it never works!
> A. You need to put all threads in qemu to cgroup.
> ==
>
> 'tasks' file is not useful interface for users, I think.
> (Even if users tend to use put-task-before-exec scheme.)
>
>
> IMHO, from user's side of view, 'tasks' file is a mystery.
>
> TID(thread-ID) is one of secrets in Linux + pthread library. For example,
> on RHEL6, to use gettid(), users has to use syscall() directly. And end-user
> may not know about thread-ID which is hidden under pthreads.

I think glibc in general is to blame for the fact that you need to
syscall(__NR_gettid)? Regardless - yes, exposing an interface dealing
with task_structs can be less than perfect for a world that deals in
userland applications.

> IIRC, there are no interface other than /proc/<pid>/tasks which shows all
> thread IDs of a process. But it's not atomic.

I tend to use pgrep, which is a bit of a hassle.

Also, like in the six-month-old-text, many resource-sucking programs
nowadays (web browsers) are multithreaded.

> So, I think it's ok to have 'procs' interface for cgroup if

> overhead/impact of patch is not heavy.
>
> Thanks,
> -Kame

Thanks for the reasoning. ;)

-- Ben

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containe rs

---

Subject: Re: [PATCH v8 0/3] cgroups: implement moving a threadgroup's threads atomically with cgroup.procs
Posted by Paul Menage on Mon, 14 Feb 2011 06:12:19 GMT
View Forum Message <> Reply to Message

On Wed, Feb 9, 2011 at 5:02 PM, KAMEZAWA Hiroyuki
<kamezawa.hiroyu@jp.fujitsu.com> wrote:
>
> So, I think it's ok to have 'procs' interface for cgroup if
> overhead/impact of patch is not heavy.
>

Agreed - it's definitely an operation that comes up as either
confusing or annoying for users, depending on whether or not they
understand how threads and cgroups interact. (We've been getting
people wanting to do this internally at Google, and I'm guessing that
we're one of the bigger users of cgroups.)

In theory it's something that could be handled in userspace, in one of two ways:

- repeatedly scan the old cgroup's tasks file and sweep any threads
from the given process into the destination cgroup, until you complete
a clean sweep finding none. (Possibly even this is racy if a thread is
being slow to fork)

- use a process event notifier to catch thread fork events and keep
track of any newly created threads that appear after your first sweep
of threads, and be prepared to handle them for some reasonable length
of time (tens of milliseconds?) after the last thread has been
apparently moved.

(The alternative approach, of course, is to give up and never try to
move a process into a cgroup except right when you're in the middle of
forking it, before the exec(), when you know that it has only a single

thread and you're in control of it.)

These are both painful procedures, compared to the very simple
approach of letting the kernel move the entire process atomically.

It's true that it's a pretty heavyweight operation, but that weight is
only paid when you actually use it on a very large process (and which
would be even more expensive to do in userspace). For the rest of the
kernel, it's just an extra read lock in the fork path on a semaphore
in a structure that's pretty much guaranteed to be in cache.

Paul

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containe rs

---

Subject: Re: [PATCH v8 0/3] cgroups: implement moving a threadgroup's threads atomically with cgroup.procs
Posted by Paul Menage on Mon, 14 Feb 2011 06:12:51 GMT
View Forum Message <> Reply to Message

On Wed, Feb 9, 2011 at 3:10 PM, Andrew Morton <akpm@linux-foundation.org> wrote:
> On Mon, 7 Feb 2011 20:35:42 -0500
> Ben Blum <bblum@andrew.cmu.edu> wrote:
>
>> On Sun, Dec 26, 2010 at 07:09:19AM -0500, Ben Blum wrote:
>> > On Fri, Dec 24, 2010 at 03:22:26AM -0500, Ben Blum wrote:
>> > > On Wed, Aug 11, 2010 at 01:46:04AM -0400, Ben Blum wrote:
>> > > > On Fri, Jul 30, 2010 at 07:56:49PM -0400, Ben Blum wrote:
>> > > > > This patch series is a revision of http://lkml.org/lkml/2010/6/25/11 .
>> > > > >
>> > > > > This patch series implements a write function for the 'cgroup.procs'
>> > > > > per-cgroup file, which enables atomic movement of multithreaded
>> > > > > applications between cgroups. Writing the thread-ID of any thread in a
>> > > > > threadgroup to a cgroup's procs file causes all threads in the group to
>> > > > > be moved to that cgroup safely with respect to threads forking/exiting.
>> > > > > (Possible usage scenario: If running a multithreaded build system that
>> > > > > sucks up system resources, this lets you restrict it all at once into a
>> > > > > new cgroup to keep it under control.)
>> > > > >
>
> The above six month old text is the best (and almost the only)
> explanation of the rationale for the entire patch series. Is
> it still correct and complete?
>

It's still correct, but I'm sure we could come up with a more detailed
justification if necessary.

Paul

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containe rs

---

Subject: [PATCH v8 4/3] cgroups: use flex_array in attach_proc
Posted by Ben Blum on Wed, 16 Feb 2011 19:22:00 GMT
View Forum Message <> Reply to Message

Convert cgroup_attach_proc to use flex_array.

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

The cgroup_attach_proc implementation requires a pre-allocated array to store
task pointers to atomically move a thread-group, but asking for a monolithic
array with kmalloc() may be unreliable for very large groups. Using flex_array
provides the same functionality with less risk of failure.

This is a post-patch for cgroup-procs-write.patch.

Signed-off-by: Ben Blum <bblum@andrew.cmu.edu>
---
kernel/cgroup.c | 37 ++++++++++++++++++++++++++++++++---------
1 files changed, 28 insertions(+), 9 deletions(-)

diff --git a/kernel/cgroup.c b/kernel/cgroup.c
index 58b364a..feba784 100644
--- a/kernel/cgroup.c
+++ b/kernel/cgroup.c
@@ -57,6 +57,7 @@
#include <linux/vmalloc.h> /* TODO: replace with more sophisticated array */
#include <linux/eventfd.h>
#include <linux/poll.h>
+#include <linux/flex_array.h> /* used in cgroup_attach_proc */

#include <asm/atomic.h>

@@ -1985,7 +1986,7 @@ int cgroup_attach_proc(struct cgroup *cgrp, struct task_struct *leader)
struct cgroupfs_root *root = cgrp->root;
/* threadgroup list cursor and array */
struct task_struct *tsk;
- struct task_struct **group;
+ struct flex_array *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
@@ -2002,9 +2003,15 @@ int cgroup_attach_proc(struct cgroup *cgrp, struct task_struct
*leader)
   * 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);
+ /* flex_array supports very large thread-groups better than kmalloc. */
+ group = flex_array_alloc(sizeof(struct task_struct *), group_size,
+     GFP_KERNEL);
  if (!group)
   return -ENOMEM;
+ /* pre-allocate to guarantee space while iterating in rcu read-side. */
+ retval = flex_array_prealloc(group, 0, group_size - 1, GFP_KERNEL);
+ if (retval)
+  goto out_free_group_list;

  /* prevent changes to the threadgroup list while we take a snapshot. */
  rcu_read_lock();
@@ -2027,7 +2034,12 @@ int cgroup_attach_proc(struct cgroup *cgrp, struct task_struct
*leader)
   /* as per above, nr_threads may decrease, but not increase. */
   BUG_ON(i >= group_size);
   get_task_struct(tsk);
-  group[i] = tsk;
+  /*
+   * saying GFP_ATOMIC has no effect here because we did prealloc
+   * earlier, but it's good form to communicate our expectations.
+   */
+  retval = flex_array_put_ptr(group, i, tsk, GFP_ATOMIC);
+  BUG_ON(retval != 0);
   i++;
  } while_each_thread(leader, tsk);
  /* remember the number of threads in the array for later. */
@@ -2050,7 +2062,9 @@ int cgroup_attach_proc(struct cgroup *cgrp, struct task_struct *leader)
   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]);
+    tsk = flex_array_get_ptr(group, i);
+    BUG_ON(tsk == NULL);
+    retval = ss->can_attach_task(cgrp, tsk);
     if (retval) {
      failed_ss = ss;
      goto out_cancel_attach;
@@ -2065,7 +2079,8 @@ int cgroup_attach_proc(struct cgroup *cgrp, struct task_struct *leader)
```

```
   */
  INIT_LIST_HEAD(&newcg_list);
  for (i = 0; i < group_size; i++) {
-  tsk = group[i];
+  tsk = flex_array_get_ptr(group, i);
+  BUG_ON(tsk == NULL);
   /* nothing to do if this task is already in the cgroup */
   oldcgrp = task_cgroup_from_root(tsk, root);
   if (cgrp == oldcgrp)
@@ -2104,7 +2119,8 @@ int cgroup_attach_proc(struct cgroup *cgrp, struct task_struct *leader)
    ss->pre_attach(cgrp);
  }
  for (i = 0; i < group_size; i++) {
-  tsk = group[i];
+  tsk = flex_array_get_ptr(group, i);
+  BUG_ON(tsk == NULL);
   /* leave current thread as it is if it's already there */
   oldcgrp = task_cgroup_from_root(tsk, root);
   if (cgrp == oldcgrp)
@@ -2154,10 +2170,13 @@ out_cancel_attach:
   }
  }
  /* clean up the array of referenced threads in the group. */
- for (i = 0; i < group_size; i++)
-  put_task_struct(group[i]);
+ for (i = 0; i < group_size; i++) {
+  tsk = flex_array_get_ptr(group, i);
+  BUG_ON(tsk == NULL);
+  put_task_struct(tsk);
+ }
 out_free_group_list:
- kfree(group);
+ flex_array_free(group);
  return retval;
 }
```

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