Subject: [PATCH 03/29] task containersv11 add tasks file interface
Posted by Paul Menage on Tue, 11 Sep 2007 19:52:42 GMT
View Forum Message <> Reply to Message

From: Paul Menage <menage@google.com>

Add the per-directory "tasks" file for cgroupfs mounts; this allows the
user to determine which tasks are members of a cgroup by reading a
cgroup's "tasks", and to move a task into a cgroup by writing its pid to
its "tasks".

Signed-off-by: Paul Menage <menage@google.com>
Cc: Serge E. Hallyn <serue@us.ibm.com>
Cc: "Eric W. Biederman" <ebiederm@xmission.com>
Cc: Dave Hansen <haveblue@us.ibm.com>
Cc: Balbir Singh <balbir@in.ibm.com>
Cc: Paul Jackson <pj@sgi.com>
Cc: Kirill Korotaev <dev@openvz.org>
Cc: Herbert Poetzl <herbert@13thfloor.at>
Cc: Srivatsa Vaddagiri <vatsa@in.ibm.com>
Cc: Cedric Le Goater <clg@fr.ibm.com>
Signed-off-by: Andrew Morton <akpm@linux-foundation.org>
---

 include/linux/cgroup.h | 10 +
 kernel/cgroup.c | 359 ++-
 2 files changed, 368 insertions(+), 1 deletion(-)

diff -puN include/linux/cgroup.h~task-cgroupsv11-add-tasks-file-interface include/linux/cgroup.h
--- a/include/linux/cgroup.h~task-cgroupsv11-add-tasks-file-interface
+++ a/include/linux/cgroup.h
@@ -144,6 +144,16 @@ int cgroup_is_removed(const struct co

 int cgroup_path(const struct cgroup *cont, char *buf, int buflen);

+int __cgroup_task_count(const struct cgroup *cont);
+static inline int cgroup_task_count(const struct cgroup *cont)
+{
+ int task_count;
+ rcu_read_lock();
+ task_count = __cgroup_task_count(cont);
+ rcu_read_unlock();
+ return task_count;
+}
+
 /* Return true if the cgroup is a descendant of the current cgroup */
 int cgroup_is_descendant(const struct cgroup *cont);

```
diff -puN kernel/cgroup.c~task-cgroupsv11-add-tasks-file-interface kernel/cgroup.c
--- a/kernel/cgroup.c~task-cgroupsv11-add-tasks-file-interface
+++ a/kernel/cgroup.c
@@ -40,7 +40,7 @@
 #include <linux/magic.h>
 #include <linux/spinlock.h>
 #include <linux/string.h>
-
+#include <linux/sort.h>
 #include <asm/atomic.h>

 /* Generate an array of cgroup subsystem pointers */
@@ -713,6 +713,127 @@ int cgroup_path(const struct containe
  return 0;
 }

+/*
+ * Return the first subsystem attached to a cgroup's hierarchy, and
+ * its subsystem id.
+ */
+
+static void get_first_subsys(const struct cgroup *cont,
+   struct cgroup_subsys_state **css, int *subsys_id)
+{
+ const struct cgroupfs_root *root = cont->root;
+ const struct cgroup_subsys *test_ss;
+ BUG_ON(list_empty(&root->subsys_list));
+ test_ss = list_entry(root->subsys_list.next,
+        struct cgroup_subsys, sibling);
+ if (css) {
+  *css = cont->subsys[test_ss->subsys_id];
+  BUG_ON(!*css);
+ }
+ if (subsys_id)
+  *subsys_id = test_ss->subsys_id;
+}
+
+/*
+ * Attach task 'tsk' to cgroup 'cont'
+ *
+ * Call holding cgroup_mutex.  May take task_lock of
+ * the task 'pid' during call.
+ */
+static int attach_task(struct cgroup *cont, struct task_struct *tsk)
+{
+ int retval = 0;
+ struct cgroup_subsys *ss;
+ struct cgroup *oldcont;
```

```
+ struct css_set *cg = &tsk->cgroups;
+ struct cgroupfs_root *root = cont->root;
+ int i;
+ int subsys_id;
+
+ get_first_subsys(cont, NULL, &subsys_id);
+
+ /* Nothing to do if the task is already in that cgroup */
+ oldcont = task_cgroup(tsk, subsys_id);
+ if (cont == oldcont)
+  return 0;
+
+ for_each_subsys(root, ss) {
+  if (ss->can_attach) {
+   retval = ss->can_attach(ss, cont, tsk);
+   if (retval) {
+    return retval;
+   }
+  }
+ }
+
+ task_lock(tsk);
+ if (tsk->flags & PF_EXITING) {
+  task_unlock(tsk);
+  return -ESRCH;
+ }
+ /* Update the css_set pointers for the subsystems in this
+  * hierarchy */
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+  if (root->subsys_bits & (1ull << i)) {
+   /* Subsystem is in this hierarchy. So we want
+    * the subsystem state from the new
+    * cgroup. Transfer the refcount from the
+    * old to the new */
+   atomic_inc(&cont->count);
+   atomic_dec(&cg->subsys[i]->cgroup->count);
+   rcu_assign_pointer(cg->subsys[i], cont->subsys[i]);
+  }
+ }
+ task_unlock(tsk);
+
+ for_each_subsys(root, ss) {
+  if (ss->attach) {
+   ss->attach(ss, cont, oldcont, tsk);
+  }
+ }
+
+ synchronize_rcu();
```

```
+ return 0;
+}
+
+/*
+ * Attach task with pid 'pid' to cgroup 'cont'. Call with
+ * cgroup_mutex, may take task_lock of task
+ */
+static int attach_task_by_pid(struct cgroup *cont, char *pidbuf)
+{
+ pid_t pid;
+ struct task_struct *tsk;
+ int ret;
+
+ if (sscanf(pidbuf, "%d", &pid) != 1)
+  return -EIO;
+
+ if (pid) {
+  rcu_read_lock();
+  tsk = find_task_by_pid(pid);
+  if (!tsk || tsk->flags & PF_EXITING) {
+   rcu_read_unlock();
+   return -ESRCH;
+  }
+  get_task_struct(tsk);
+  rcu_read_unlock();
+
+  if ((current->euid) && (current->euid != tsk->uid)
+      && (current->euid != tsk->suid)) {
+   put_task_struct(tsk);
+   return -EACCES;
+  }
+ } else {
+  tsk = current;
+  get_task_struct(tsk);
+ }
+
+ ret = attach_task(cont, tsk);
+ put_task_struct(tsk);
+ return ret;
+}
+
 /* The various types of files and directories in a cgroup file system */

 enum cgroup_filetype {
@@ -721,6 +842,55 @@ enum cgroup_filetype {
  FILE_TASKLIST,
 };
```

```
+static ssize_t cgroup_common_file_write(struct cgroup *cont,
+        struct cftype *cft,
+        struct file *file,
+        const char __user *userbuf,
+        size_t nbytes, loff_t *unused_ppos)
+{
+ enum cgroup_filetype type = cft->private;
+ char *buffer;
+ int retval = 0;
+
+ if (nbytes >= PATH_MAX)
+  return -E2BIG;
+
+ /* +1 for nul-terminator */
+ buffer = kmalloc(nbytes + 1, GFP_KERNEL);
+ if (buffer == NULL)
+  return -ENOMEM;
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+  retval = -EFAULT;
+  goto out1;
+ }
+ buffer[nbytes] = 0; /* nul-terminate */
+
+ mutex_lock(&cgroup_mutex);
+
+ if (cgroup_is_removed(cont)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch (type) {
+ case FILE_TASKLIST:
+  retval = attach_task_by_pid(cont, buffer);
+  break;
+ default:
+  retval = -EINVAL;
+  goto out2;
+ }
+
+ if (retval == 0)
+  retval = nbytes;
+out2:
+ mutex_unlock(&cgroup_mutex);
+out1:
+ kfree(buffer);
+ return retval;
+}
```

```
+
 static ssize_t cgroup_file_write(struct file *file, const char __user *buf,
       size_t nbytes, loff_t *ppos)
 {
@@ -924,6 +1094,189 @@ int cgroup_add_files(struct cgroup
  return 0;
 }

+/* Count the number of tasks in a cgroup. Could be made more
+ * time-efficient but less space-efficient with more linked lists
+ * running through each cgroup and the css_set structures that
+ * referenced it. Must be called with tasklist_lock held for read or
+ * write or in an rcu critical section.
+ */
+int __cgroup_task_count(const struct cgroup *cont)
+{
+ int count = 0;
+ struct task_struct *g, *p;
+ struct cgroup_subsys_state *css;
+ int subsys_id;
+
+ get_first_subsys(cont, &css, &subsys_id);
+ do_each_thread(g, p) {
+  if (task_subsys_state(p, subsys_id) == css)
+   count ++;
+ } while_each_thread(g, p);
+ return count;
+}
+
+/*
+ * Stuff for reading the 'tasks' file.
+ *
+ * Reading this file can return large amounts of data if a cgroup has
+ * *lots* of attached tasks. So it may need several calls to read(),
+ * but we cannot guarantee that the information we produce is correct
+ * unless we produce it entirely atomically.
+ *
+ * Upon tasks file open(), a struct ctr_struct is allocated, that
+ * will have a pointer to an array (also allocated here).  The struct
+ * ctr_struct * is stored in file->private_data.  Its resources will
+ * be freed by release() when the file is closed.  The array is used
+ * to sprintf the PIDs and then used by read().
+ */
+struct ctr_struct {
+ char *buf;
+ int bufsz;
+};
+
```

```
+/*
+ * Load into 'pidarray' up to 'npids' of the tasks using cgroup
+ * 'cont'.  Return actual number of pids loaded.  No need to
+ * task_lock(p) when reading out p->cgroup, since we're in an RCU
+ * read section, so the css_set can't go away, and is
+ * immutable after creation.
+ */
+static int pid_array_load(pid_t *pidarray, int npids, struct cgroup *cont)
+{
+ int n = 0;
+ struct task_struct *g, *p;
+ struct cgroup_subsys_state *css;
+ int subsys_id;
+
+ get_first_subsys(cont, &css, &subsys_id);
+ rcu_read_lock();
+ do_each_thread(g, p) {
+  if (task_subsys_state(p, subsys_id) == css) {
+   pidarray[n++] = pid_nr(task_pid(p));
+   if (unlikely(n == npids))
+    goto array_full;
+  }
+ } while_each_thread(g, p);
+
+array_full:
+ rcu_read_unlock();
+ return n;
+}
+
+static int cmppid(const void *a, const void *b)
+{
+ return *(pid_t *)a - *(pid_t *)b;
+}
+
+/*
+ * Convert array 'a' of 'npids' pid_t's to a string of newline separated
+ * decimal pids in 'buf'.  Don't write more than 'sz' chars, but return
+ * count 'cnt' of how many chars would be written if buf were large enough.
+ */
+static int pid_array_to_buf(char *buf, int sz, pid_t *a, int npids)
+{
+ int cnt = 0;
+ int i;
+
+ for (i = 0; i < npids; i++)
+  cnt += snprintf(buf + cnt, max(sz - cnt, 0), "%d\n", a[i]);
+ return cnt;
+}
```

```
+
+/*
+ * Handle an open on 'tasks' file.  Prepare a buffer listing the
+ * process id's of tasks currently attached to the cgroup being opened.
+ *
+ * Does not require any specific cgroup mutexes, and does not take any.
+ */
+static int cgroup_tasks_open(struct inode *unused, struct file *file)
+{
+ struct cgroup *cont = __d_cont(file->f_dentry->d_parent);
+ struct ctr_struct *ctr;
+ pid_t *pidarray;
+ int npids;
+ char c;
+
+ if (!(file->f_mode & FMODE_READ))
+  return 0;
+
+ ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
+ if (!ctr)
+  goto err0;
+
+ /*
+  * If cgroup gets more users after we read count, we won't have
+  * enough space - tough.  This race is indistinguishable to the
+  * caller from the case that the additional cgroup users didn't
+  * show up until sometime later on.
+  */
+ npids = cgroup_task_count(cont);
+ if (npids) {
+  pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
+  if (!pidarray)
+   goto err1;
+
+  npids = pid_array_load(pidarray, npids, cont);
+  sort(pidarray, npids, sizeof(pid_t), cmppid, NULL);
+
+  /* Call pid_array_to_buf() twice, first just to get bufsz */
+  ctr->bufsz = pid_array_to_buf(&c, sizeof(c), pidarray, npids) + 1;
+  ctr->buf = kmalloc(ctr->bufsz, GFP_KERNEL);
+  if (!ctr->buf)
+   goto err2;
+  ctr->bufsz = pid_array_to_buf(ctr->buf, ctr->bufsz, pidarray, npids);
+
+  kfree(pidarray);
+ } else {
+  ctr->buf = 0;
+  ctr->bufsz = 0;
```

```
+ }
+ file->private_data = ctr;
+ return 0;
+
+err2:
+ kfree(pidarray);
+err1:
+ kfree(ctr);
+err0:
+ return -ENOMEM;
+}
+
+static ssize_t cgroup_tasks_read(struct cgroup *cont,
+        struct cftype *cft,
+        struct file *file, char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct ctr_struct *ctr = file->private_data;
+
+ return simple_read_from_buffer(buf, nbytes, ppos, ctr->buf, ctr->bufsz);
+}
+
+static int cgroup_tasks_release(struct inode *unused_inode,
+     struct file *file)
+{
+ struct ctr_struct *ctr;
+
+ if (file->f_mode & FMODE_READ) {
+  ctr = file->private_data;
+  kfree(ctr->buf);
+  kfree(ctr);
+ }
+ return 0;
+}
+
+/*
+ * for the common functions, 'private' gives the type of file
+ */
+static struct cftype cft_tasks = {
+ .name = "tasks",
+ .open = cgroup_tasks_open,
+ .read = cgroup_tasks_read,
+ .write = cgroup_common_file_write,
+ .release = cgroup_tasks_release,
+ .private = FILE_TASKLIST,
+};
+
 static int cgroup_populate_dir(struct cgroup *cont)
```

```
 {
  int err;
@@ -932,6 +1285,10 @@ static int cgroup_populate_dir(struct
  /* First clear out any existing files */
  cgroup_clear_directory(cont->dentry);

+ err = cgroup_add_file(cont, NULL, &cft_tasks);
+ if (err < 0)
+  return err;
+
  for_each_subsys(cont->root, ss) {
   if (ss->populate && (err = ss->populate(ss, cont)) < 0)
    return err;
_

--
```