
Subject: [PATCH 03/10] Containers(V10): Add tasks file interface

Posted by [Paul Menage](#) on Tue, 29 May 2007 13:01:07 GMT

[View Forum Message](#) <> [Reply to Message](#)

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

Signed-off-by: Paul Menage <menage@google.com>

include/linux/container.h | 10 +
kernel/container.c | 335 ++++++
2 files changed, 345 insertions(+)

Index: container-2.6.22-rc2-mm1/include/linux/container.h

=====

--- container-2.6.22-rc2-mm1.orig/include/linux/container.h

+++ container-2.6.22-rc2-mm1/include/linux/container.h

@@ -128,6 +128,16 @@ int container_is_removed(const struct co

int container_path(const struct container *cont, char *buf, int buflen);

+int __container_task_count(const struct container *cont);

+static inline int container_task_count(const struct container *cont)

+{

+ int task_count;

+ rcu_read_lock();

+ task_count = __container_task_count(cont);

+ rcu_read_unlock();

+ return task_count;

+}

+

/* Return true if the container is a descendant of the current container */

int container_is_descendant(const struct container *cont);

Index: container-2.6.22-rc2-mm1/kernel/container.c

=====

--- container-2.6.22-rc2-mm1.orig/kernel/container.c

+++ container-2.6.22-rc2-mm1/kernel/container.c

@@ -679,6 +679,109 @@ static inline void get_first_subsys(cons

*subsys_id = test_ss->subsys_id;

}

+/

+ * Attach task 'tsk' to container 'cont'

+ *

```

+ * Call holding container_mutex. May take task_lock of
+ * the task 'pid' during call.
+ */
+
+static int attach_task(struct container *cont, struct task_struct *tsk)
+{
+ int retval = 0;
+ struct container_subsys *ss;
+ struct container *oldcont;
+ struct css_group *cg = &tsk->containers;
+ struct containerfs_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 container */
+ oldcont = task_container(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_group pointers for the subsystems in this
+ * hierarchy */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ if (root->subsys_bits & (1ull << i)) {
+ /* Subsystem is in this hierarchy. So we want
+ * the subsystem state from the new
+ * container. Transfer the refcount from the
+ * old to the new */
+ atomic_inc(&cont->count);
+ atomic_dec(&cg->subsys[i]->container->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 container 'cont'. Call with
+ * container_mutex, may take task_lock of task
+ *
+ */
+
+static int attach_task_by_pid(struct container *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 container file system */

typedef enum {
@@ -687,6 +790,54 @@ typedef enum {
    FILE_TASKLIST,
} container_filetype_t;

+static ssize_t container_common_file_write(struct container *cont,
+    struct cftype *cft,
+    struct file *file,
+    const char __user *userbuf,
+    size_t nbytes, loff_t *unused_ppos)
+{
+ container_filetype_t type = cft->private;
+ char *buffer;
+ int retval = 0;
+
+
+ if (nbytes >= PATH_MAX)
+ return -E2BIG;
+
+
+ /* +1 for nul-terminator */
+ if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
+ return -ENOMEM;
+
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+ retval = -EFAULT;
+ goto out1;
+ }
+
+ buffer[nbytes] = 0; /* nul-terminate */
+
+
+ mutex_lock(&container_mutex);
+
+
+ if (container_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(&container_mutex);
+out1:
+ kfree(buffer);
+ return retval;
+}
+
static ssize_t container_file_write(struct file *file, const char __user *buf,
    size_t nbytes, loff_t *ppos)
{
@@ -875,6 +1026,187 @@ int container_add_files(struct container
    return 0;
}

+/* Count the number of tasks in a container. Could be made more
+ * time-efficient but less space-efficient with more linked lists
+ * running through each container and the css_group structures that
+ * referenced it. Must be called with tasklist_lock held for read or
+ * write or in an rcu critical section. */
+
+int __container_task_count(const struct container *cont)
+{
+   int count = 0;
+   struct task_struct *g, *p;
+   struct container_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 container 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().
+ */
+
+/* containers_tasks_read array */
+
+struct ctr_struct {
+ char *buf;
+ int bufsz;
+};
+
+/*
+ * Load into 'pidarray' up to 'npids' of the tasks using container
+ * 'cont'. Return actual number of pids loaded. No need to
+ * task_lock(p) when reading out p->container, since we're in an RCU
+ * read section, so the css_group can't go away, and is
+ * immutable after creation.
+ */
+static int pid_array_load(pid_t *pidarray, int npids, struct container *cont)
+{
+ int n = 0;
+ struct task_struct *g, *p;
+ struct container_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 += sprintf(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 container being opened.
+ *
+ * Does not require any specific container mutexes, and does not take any.
+ */
+static int container_tasks_open(struct inode *unused, struct file *file)
+{
+ struct container *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 container 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 container users didn't
+ * show up until sometime later on.
+ */
+ npids = container_task_count(cont);
+ 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), cmp_pid, 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);
+ file->private_data = ctr;
+ return 0;
+
+err2:
+ kfree(pidarray);
+err1:
+ kfree(ctr);
+err0:
+ return -ENOMEM;
+}
+
+static ssize_t container_tasks_read(struct container *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 container_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 = container_tasks_open,
+ .read = container_tasks_read,
+ .write = container_common_file_write,
+ .release = container_tasks_release,
+ .private = FILE_TASKLIST,
+};
+
static int container_populate_dir(struct container *cont)
{
    int err;
@@ -883,6 +1215,9 @@ static int container_populate_dir(struct
/* First clear out any existing files */
container_clear_directory(cont->dentry);

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

```

Subject: Re: [PATCH 03/10] Containers(V10): Add tasks file interface
 Posted by [Cedric Le Goater](#) on Thu, 07 Jun 2007 14:00:22 GMT
[View Forum Message](#) <> [Reply to Message](#)

Hello Paul !

menage@google.com wrote:

> This patch adds the per-directory "tasks" file for containerfs mounts;
 > this allows the user to determine which tasks are members of a
 > container by reading a container's "tasks", and to move a task into a
 > container by writing its pid to its "tasks".

here's a small fix for 2.6.22-rc4-mm2.

C.

when there's no tasks in a container, opening

<containerfs>/<container>/tasks

spits the following warning because we are trying to
 kmalloc(0).

WARNING: at mm/slab.c:777 __find_general_cachep()

```

[<c0102b01>] show_trace_log_lvl+0x1a/0x2f
[<c0103627>] show_trace+0x12/0x14
[<c010363e>] dump_stack+0x15/0x17
[<c0148040>] __kmalloc+0x56/0xf3
[<c012d030>] container_tasks_open+0x56/0x131
[<c012cdf8>] container_file_open+0x32/0x3a
[<c014964d>] __dentry_open+0x99/0x13c
[<c0149771>] nameidata_to_filp+0x27/0x37
[<c01497b4>] do_filp_open+0x33/0x3b
[<c0149801>] do_sys_open+0x45/0xc9
[<c01498bd>] sys_open+0x1c/0x1e
[<c0102540>] syscall_call+0x7/0xb

```

Signed-off-by: Cedric Le Goater <clg@fr.ibm.com>

Cc: Paul Menage <menage@google.com>

```

kernel/container.c | 33 ++++++-----
1 file changed, 19 insertions(+), 14 deletions(-)

```

Index: 2.6.22-rc4-mm2/kernel/container.c

=====

--- 2.6.22-rc4-mm2.orig/kernel/container.c

+++ 2.6.22-rc4-mm2/kernel/container.c

@@ -1651,21 +1651,26 @@ static int container_tasks_open(struct i

* show up until sometime later on.

*/

npids = container_task_count(cont);

- 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), cmp_pid, 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);

+ 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), cmp_pid, 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);
+ kfree(pidarray);
+ } else {
+   ctr->buf = 0;
+   ctr->bufsz = 0;
+ }
  file->private_data = ctr;
  return 0;
```

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

Subject: Re: [PATCH 03/10] Containers(V10): Add tasks file interface
Posted by [Paul Menage](#) on Thu, 07 Jun 2007 17:12:19 GMT
[View Forum Message](#) <> [Reply to Message](#)

On 6/7/07, Cedric Le Goater <clg@fr.ibm.com> wrote:

```
>
> when there's no tasks in a container, opening
>
> <containerfs>/<container>/tasks
>
> spits the following warning because we are trying to
> kmalloc(0).
```

I guess I'm not opposed to this change - but isn't there still discussion going on about whether kmalloc(0) should actually produce a warning or not?

Thanks,

Paul

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