Subject: [PATCH 03/33] task containersv11 add tasks file interface
Posted by Paul Menage on Mon, 17 Sep 2007 21:03:10 GMT
View Forum Message <> Reply to Message

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>
---

```
 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;

–

--

Subject: Re: [PATCH 03/33] task containersv11 add tasks file interface
Posted by Paul Jackson on Wed, 03 Oct 2007 08:09:23 GMT
View Forum Message <> Reply to Message

Cgroup (aka container) code review:

Except for the very last item below, my other comments are minor.
And the last item is pretty easy too - just more important.

Overall - nice stuff.  I like this generalization of the cpuset
hierarchy.  Thanks.

===========

Review comments on Documentation/cgroups.txt

 - You can drop the cpuset mini-history at the top of cgroups.txt:
 Modified by Paul Jackson <pj@sgi.com>
 Modified by Christoph Lameter <clameter@sgi.com>

 - The "Definitions" section is well done, thanks.

 - Could you change the present tense in:
 There are multiple efforts to provide ...  These all require ...
   to past tense:
 There have been multiple efforts to provide ...  These all required ...

   The present tense suggests to me that there are, as of the time
   I am reading this (perhaps years in the future) implementations of
   cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server
   namespaces all somewhere in the main line kernel.  I don't think so.

 - I guess that the items "CPU", "Memory", Disk", and "Network" in
   the example are separate hierarchies - but that's not clear.
   Perhaps "CPU" and "Memory" are in the same hierarchy - cpusets ??
   There is a mix of terminology in this example that is confusing.
   The word "class" is out of context.  I guess that "the WWW Network
   class" refers to the "WWW browsing" thingie -- is that a cgroup --

  but not to the "Network:" thingie -- is that a cgroup hierarchy ?
  I got lost in this "example of a scenario ...".  It reads like
  a cut+paste from some other effort, incompletely reworded.

- What are these apparent 'exec notifications' that are provided to
  user space that the following mentions - I cannot find any other
  mention of them:

With the ability to classify tasks differently for different
resources (by putting those resource subsystems in different
hierarchies) then the admin can easily set up a script which
receives exec notifications and depending on who is launching
the browser he can

- Try replacing the following with a single period character - it's
  intruding on some description of a network "resource class" (whatever
  that is), and ends in a trailing comma:

  OR give one of the students simulation
apps enhanced CPU power,

- Could you add a two or three line example 'cgroup' following:

Each task under /proc has an added file named 'cgroup' displaying,
for each active hierarchy, the subsystem names and the cgroup name
as the path relative to the root of the cgroup file system.

- I don't see any documentation of the "/proc/cgroups" file.  And for
  that matter, the format of this cgroups file is not "self-documenting"
  and neither just one value nor even an array of <key, value> pairs.

  Here's a sample /proc/cgroups file from my test system:

```
# cat /proc/cgroups
Hierarchies:
e000003015400000: bits=1 cgroups=2 (cpuset) s_active=2
a00000010118f520: bits=0 cgroups=1 ()
Subsystems:
0: name=cpuset hierarchy=e000003015400000
1: name=cpuacct hierarchy=a00000010118f520
2: name=debug hierarchy=a00000010118f520
3: name=ns hierarchy=a00000010118f520
4: name=memory hierarchy=a00000010118f520
Control Group groups: 2
```

  Can this file be both simplified and documented?

- It states in cgroups.txt:

  *** notify_on_release is disabled in the current patch set. It will be
  *** reactivated in a future patch in a less-intrusive manner

  This doesn't seem to be true, and had better not be true.
  From what I can tell, notify_on_release still works for cpusets,
  and it is important that it continue to work when cgroups are
  folded into the main line kernel.

- Typo in:

 To mount a cgroup hierarchy will all available subsystems,

 Should be:

 To mount a cgroup hierarchy with all available subsystems,

- In the following:

Each cgroup object created by the system has an array of pointers,
indexed by subsystem id; this pointer is entirely managed by the
subsystem; the generic cgroup code will never touch this pointer.

 Is plural "pointers", or singular "pointer", the correct wording?

- The following seems confused - I'd guess that all mentions of
 'callback_mutex' in cgroup code and comments should be removed.

Subsystems can take/release the cgroup_mutex via the functions
cgroup_lock()/cgroup_unlock(), and can
take/release the callback_mutex via the functions
cgroup_lock()/cgroup_unlock().

- Several lines near the end of cgroups.txt start with "LL".
 I guess they list what locks are held while taking the call,
 but the notation seems cryptic and unfamiliar to me, and its
 meaning here undocumented.

- Could you indent something (either the function declarations
 or the commentary) under "3.3 Subsystem API", to make it more
 readable?

- There are many instances of the local variable 'cont', referring
 to a struct cgroup pointer. I presume the spelling 'cont' is a
 holdover from the time when we called these containers. Perhaps
 some other spelling, such as 'cgp', would be less obscure now.

===========

Review comments on include/linux/cgroup.h:

 - I don't see a Google copyright - should I?

 - One mention of 'callback_mutex' probably should be 'cgroup_mutex'.

 - In the following:

```
 /*
  * Call css_get() to hold a reference on the cgroup;
  *
  */
```
   Remove extra comment line, replace ';' with '.'.

 - In the following:

```
	struct cgroup *parent;	/* my parent */
```

   one more tab is needed before the comment.

===========

Review comments on kernel/cgroup.c:

 - Could you state at the top that 'cgroup' stands for "control group" ?

 - The Google copyright is year 2006 - should that be 2006-2007 now?

 - You can drop the cpuset mini-history at the top of cgroup.c:
```
  *  2003-10-10 Written by Simon Derr.
  *  2003-10-22 Updates by Stephen Hemminger.
  *  2004 May-July Rework by Paul Jackson.
```
   It's still in kernel/cpuset.c, which is the more useful spot.

 - Would it be a bit clearer that the define SUBSYS in kernel/cgroup.c
   is there for the use of the following subsys[] definition if you
   replaced:

```
 /* Generate an array of cgroup subsystem pointers */
 #define SUBSYS(_x) &_x ## _subsys,

 static struct cgroup_subsys *subsys[] = {
 #include <linux/cgroup_subsys.h>
 };
```

   with:

```
/* Generate subsys[] array of cgroup subsystem pointers */
#define SUBSYS(_x) &_x ## _subsys,
static struct cgroup_subsys *subsys[] = {
#include <linux/cgroup_subsys.h>
};
#undef SUBSYS
```

I removed a blank line, undef'd SUBSYS when done with it, and reworded the comment, so it's clear that this is a single unit of code. On first reading it seemed that (1) the comment was claiming that the define SUBSYS itself was supposed to generate some array, which made no sense, and (2) there was no apparent use of that define SUBSYS in all of kernel/cgroup.c.

- See Documentation/CodingStyle "Section 8: Commenting" for the preferred comment style. For example, replace:

```
/* css_set_lock protects the list of css_set objects, and the
 * chain of tasks off each css_set.  Nests outside task->alloc_lock
 * due to cgroup_iter_start() */
```

with:

```
/*
 * css_set_lock protects the list of css_set objects, and the
 * chain of tasks off each css_set.  Nests outside task->alloc_lock
 * due to cgroup_iter_start()
 */
```

- The code in attach_task which skips the attachment of a task to the group it is already in has to be removed. Cpusets depends on reattaching a task to its current cpuset, in order to trigger updating the cpus_allowed mask in the task struct. This is a hack, granted, but an important one. It avoids checking for a changed cpuset 'cpus' setting in critical scheduler code paths.

I have a patch that I will send for this change shortly, as it is more critical than the other issues I note in this review. Main line cpusets would be broken without this patch.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

Subject: Re: [PATCH 03/33] task containersv11 add tasks file interface
Posted by Paul Menage on Wed, 03 Oct 2007 15:16:53 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Jackson <pj@sgi.com> wrote:
>
> - What are these apparent 'exec notifications' that are provided to
> user space that the following mentions - I cannot find any other
> mention of them:
>
> With the ability to classify tasks differently for different
> resources (by putting those resource subsystems in different
> hierarchies) then the admin can easily set up a script which
> receives exec notifications and depending on who is launching
> the browser he can

It's the process connector netlink notifier. It can report
fork/exit/exec/setuid events to userspace. See
drivers/connector/cn_proc.c

>
>
> - It states in cgroups.txt:
>
> *** notify_on_release is disabled in the current patch set. It will be
> *** reactivated in a future patch in a less-intrusive manner
>
> This doesn't seem to be true, and had better not be true.
> From what I can tell, notify_on_release still works for cpusets,
> and it is important that it continue to work when cgroups are
> folded into the main line kernel.

Correct, it's reactivated in a later patch in the series, but this
intermediate comment snuck through.

>
> Each cgroup object created by the system has an array of pointers,
> indexed by subsystem id; this pointer is entirely managed by the
> subsystem; the generic cgroup code will never touch this pointer.
>
> Is plural "pointers", or singular "pointer", the correct wording?

Probably plural.

>

>  - Several lines near the end of cgroups.txt start with "LL".
>    I guess they list what locks are held while taking the call,
>    but the notation seems cryptic and unfamiliar to me, and its
>    meaning here undocumented.

"Locking Level", describing which locks *are* held, and which are
*not* held during a call. I thought it was a more generally
widely-used commenting convention, but I don't see any other uses of
it in the kernel. I can replace them with "holds cgroup_mutex" or
"doesn't hold cgroup_mutex" for clarity.

>
>  - There are many instances of the local variable 'cont', referring
>    to a struct cgroup pointer.  I presume the spelling 'cont' is a
>    holdover from the time when we called these containers.

Yes, and since cgroup is short for "control group", "cont" still
seemed like a reasonable abbreviation. (And made the automatic
renaming much simpler).
>
>  - The code in attach_task which skips the attachment of a task to
>    the group it is already in has to be removed.  Cpusets depends
>    on reattaching a task to its current cpuset, in order to trigger
>    updating the cpus_allowed mask in the task struct.  This is a
>    hack, granted, but an important one.  It avoids checking for a
>    changed cpuset 'cpus' setting in critical scheduler code paths.

I don't quite understand how this is meant to work - under what
circumstances would it occur? Are there cases when userspace is
required to try to reattach a task to its current cpuset in order to
get a cpu mask change to stick?

Other comments noted, thanks.

Paul

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

---

## Subject: Re: [PATCH 03/33] task containersv11 add tasks file interface
## Posted by Paul Jackson on Wed, 03 Oct 2007 17:51:57 GMT

View Forum Message <> Reply to Message

Paul M wrote:
> Are there cases when userspace is
> required to try to reattach a task to its current cpuset in order to

> get a cpu mask change to stick?

Yes, there are such cases.

If tasks are running in a cpuset, and someone changes the
'cpus' of that cpuset, the tasks already in that cpuset don't
move.  They stay with their task structs cpus_allowed pointing
to the old value.  The only code path that user space can trigger
that leads to changing an existing tasks cpus_allowed mask is to
write that tasks pid to a cpuset 'tasks' file.

So after changing the 'cpus' of a cpuset, you (something in
user space) then has to rewrite every pid in that cpusets
tasks file back to that same tasks file, in order to get the
change to be applied to each of those tasks, and get them to
start running on their new CPUs.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

## Subject: Re: [PATCH 03/33] task containersv11 add tasks file interface
Posted by Paul Menage on Wed, 03 Oct 2007 18:15:57 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Jackson <pj@sgi.com> wrote:
>
> So after changing the 'cpus' of a cpuset, you (something in
> user space) then has to rewrite every pid in that cpusets
> tasks file back to that same tasks file, in order to get the
> change to be applied to each of those tasks, and get them to
> start running on their new CPUs.

What happens to the new child that got forked right between userspace
reading the 'tasks' file and writing back the pids?

Paul

Subject: Re: [PATCH 03/33] task containersv11 add tasks file interface
Posted by Paul Jackson on Thu, 04 Oct 2007 02:46:52 GMT
View Forum Message <> Reply to Message

> >  - There are many instances of the local variable 'cont', referring
> >    to a struct cgroup pointer.  I presume the spelling 'cont' is a
> >    holdover from the time when we called these containers.
>
> Yes, and since cgroup is short for "control group", "cont" still
> seemed like a reasonable abbreviation. (And made the automatic
> renaming much simpler).

The following will change all 'cont' words to your choice (I doubt
you want to use 'XXXX' as I did here) in cgroup.c:

```
    sed -i -r 's/(\W|^)cont(\W|$)/\1XXXX\2/g' kernel/cgroup.c
```

I can't say for sure, but I suspect that if cgroups had always
been cgroups (short for control groups), then these local 'cont'
variables would have a different name.  One can often, as in this
case, find some justification for most any name.  The question is
which name is most quickly and easily understood.

... yes ... I'm a stickler for names ... sorry.

--
                  I won't rest till it's the best ...
                  Programmer, Linux Scalability
                  Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [PATCH 03/33] task containersv11 add tasks file interface
Posted by Paul Menage on Thu, 04 Oct 2007 02:53:03 GMT
View Forum Message <> Reply to Message

On 10/3/07, Paul Jackson <pj@sgi.com> wrote:
>
> I can't say for sure, but I suspect that if cgroups had always
> been cgroups (short for control groups), then these local 'cont'
> variables would have a different name.

Oh, absolutely. I just refrained from changing them in the rename
since the name was sort of still relevant and it made the change
simpler.

Paul

Subject: Re: [PATCH 03/33] task containersv11 add tasks file interface
Posted by Paul Jackson on Thu, 04 Oct 2007 02:55:03 GMT
View Forum Message <> Reply to Message

One more cgroup code review detail ...

The following is evidence of some more stale comments in
kernel/cpuset.c.  Some routines which used to be in that file, but
which are now reimplemented in cgroups, are still named in cpuset.c
comments:

```
$ grep -E 'cpuset_rmdir|cpuset_exit|cpuset_fork' kernel/cpuset.c
 * knows that the cpuset won't be removed, as cpuset_rmdir() needs
 * The fork and exit callbacks cpuset_fork() and cpuset_exit(), don't
 * critical pieces of code here.  The exception occurs on cpuset_exit(),
 *    the_top_cpuset_hack in cpuset_exit(), which sets an exiting tasks
```

The downside of my writing too many comments ... its more of a
maintenance burden on those changing the code ;).

--
                  I won't rest till it's the best ...
                  Programmer, Linux Scalability
                  Paul Jackson <pj@sgi.com> 1.925.600.0401