Subject: [PATCH 00/10] Task Containers(V11): Introduction
Posted by Paul Menage on Fri, 20 Jul 2007 18:31:52 GMT
View Forum Message <> Reply to Message

This is an update to the task containers patchset.

Changes since V10 (May 30th) include:

- Based on 2.6.22-rc6-mm1 (minus existing container patches, see below)

- Rolled in various fix/tidy patches contributed by akpm and others

- Reorganisation of the mount/unmount code to use sget(); the new
approach is modelled on the NFS superblock code. This fixes some
potential lock inversions pointed out by lockdep.

- Fix various lockdep warnings

- Changed the create() subsystem callback to return a pointer to the
new state object rather than updating the subsystem pointer in the
container directly.

- Changed container_add_file() to automatically prefix the subsystem
name (and a period) on to all container files unless the filesystem
is mounted with the "noprefix" option (intended for use by the
legacy cpuset filesystem emulation).

- Added a release_agent= mount option to allow the release agent path
to be specified at mount time.

- css_put() is now completely non-blocking

- css_get()/css_put() avoid taking/dropping reference counts on the
root state since this can't be freed anyway; this saves some atomic ops

API changes (for subsystem writers):
1) return your new css object from create() callback
2) remove the subsystem name prefix from your cftype structures
3) pass your subsystem pointer as an additional new parameter to
container_add_file() and container_add_files()

Still TODO:

- finalize the naming

- add a hash-table based lookup for css_group objects.

- use seq_file properly in container tasks files to avoid having to

allocate a big array for all the container's task pointers.

- add virtualization support to allow delegation to virtual servers

- fix a lockdep false-positive - container_mutex nests inside
  inode->i_mutex, but there's a point in the mount code where we need to
  lock a newly-created (and hence guaranteed unlocked) directory from
  within container_mutex.

- more subsystems

Generic Process Containers

There have recently been various proposals floating around for
resource management/accounting and other task grouping subsystems in
the kernel, including ResGroups, User BeanCounters, NSProxy
containers, and others. These all need the basic abstraction of being
able to group together multiple processes in an aggregate, in order to
track/limit the resources permitted to those processes, or control
other behaviour of the processes, and all implement this grouping in
different ways.

This patchset provides a framework for tracking and grouping processes
into arbitrary "containers" and assigning arbitrary state to those
groupings, in order to control the behaviour of the container as an
aggregate.

The intention is that the various resource management and
virtualization/container efforts can also become task container
clients, with the result that:

- the userspace APIs are (somewhat) normalised

- it's easier to test e.g. the ResGroups CPU controller in
 conjunction with the BeanCounters memory controller, or use either of
them as the resource-control portion of a virtual server system.

- the additional kernel footprint of any of the competing resource
 management systems is substantially reduced, since it doesn't need
 to provide process grouping/containment, hence improving their
 chances of getting into the kernel

The patch set is structured as follows:

1) Basic container framework - filesystem and tracking structures

2) Support for the "tasks" control file

3) Hooks for fork() and exit()

4) Support for the container_clone() operation

5) Add /proc reporting interface

6) Share container subsystem pointer arrays between tasks with the
same assignments

7) Support for a userspace "release agent", similar to the cpusets
release agent functionality

8) Make cpusets a container subsystem

9) Simple CPU Accounting example subsystem

10) Simple container debugging subsystem

It applies to 2.6.22-rc6-mm1, *minus* the following patches (available
from http://www.kernel.org/pub/linux/kernel/people/akpm/mm/broken
-out-2007-06-27-03-28.tar.gz)

containersv10-basic-container-framework.patch
containersv10-basic-container-framework-fix.patch
containersv10-basic-container-framework-fix-2.patch
containersv10-basic-container-framework-fix-3.patch
containersv10-example-cpu-accounting-subsystem.patch
containersv10-example-cpu-accounting-subsystem-fix.patch
containersv10-add-tasks-file-interface.patch
containersv10-add-tasks-file-interface-fix.patch
containersv10-add-tasks-file-interface-fix-2.patch
containersv10-add-fork-exit-hooks.patch
containersv10-add-fork-exit-hooks-fix.patch
containersv10-add-container_clone-interface.patch
containersv10-add-container_clone-interface-fix.patch
containersv10-add-procfs-interface.patch
containersv10-add-procfs-interface-fix.patch
containersv10-make-cpusets-a-client-of-containers.patch
containersv10-make-cpusets-a-client-of-containers-whitespace .patch
containersv10-share-css_group-arrays-between-tasks-with-same -container-memberships.patch
containersv10-share-css_group-arrays-between-tasks-with-same
-container-memberships-fix.patch
containersv10-share-css_group-arrays-between-tasks-with-same
-container-memberships-cpuset-zero-malloc-fix-for-new-contai ners.patch
containersv10-simple-debug-info-subsystem.patch
containersv10-simple-debug-info-subsystem-fix.patch

containersv10-simple-debug-info-subsystem-fix-2.patch
containersv10-support-for-automatic-userspace-release-agents .patch
containersv10-support-for-automatic-userspace-release-agents -whitespace.patch
add-containerstats-v3.patch
add-containerstats-v3-fix.patch
update-getdelays-to-become-containerstats-aware.patch
containers-implement-subsys-post_clone.patch
containers-implement-namespace-tracking-subsystem-v3.patch

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

--

Subject: [PATCH 01/10] Task Containers(V11): Basic task container framework
Posted by Paul Menage on Fri, 20 Jul 2007 18:31:53 GMT
View Forum Message <> Reply to Message

This patch adds the main task containers framework - the container
filesystem, and the basic structures for tracking membership and
associating subsystem state objects to tasks.

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

```
 Documentation/containers.txt     |  526 ++++++++++++++++++
 include/linux/container.h        |  214 ++++++
 include/linux/container_subsys.h |   10
 include/linux/magic.h            |    1
 include/linux/sched.h            |   34 +
 init/Kconfig                     |    8
 init/main.c                      |    3
 kernel/Makefile                  |    1
 kernel/container.c               | 1199 +++++++++++++++++++++++++++++++++++++++
 9 files changed, 1995 insertions(+), 1 deletion(-)

Index: container-2.6.22-rc6-mm1/Documentation/containers.txt
 ===================================================================== =======
--- /dev/null
+++ container-2.6.22-rc6-mm1/Documentation/containers.txt
@@ -0,0 +1,526 @@
+    CONTAINERS
+    -------
+
+Written by Paul Menage <menage@google.com> based on Documentation/cpusets.txt
+
+Original copyright statements from cpusets.txt:
```

+Portions Copyright (C) 2004 BULL SA.
+Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
+Modified by Paul Jackson <pj@sgi.com>
+Modified by Christoph Lameter <clameter@sgi.com>
+
+CONTENTS:
+=========
+
+1. Containers
+ 1.1 What are containers ?
+ 1.2 Why are containers needed ?
+ 1.3 How are containers implemented ?
+ 1.4 What does notify_on_release do ?
+ 1.5 How do I use containers ?
+2. Usage Examples and Syntax
+ 2.1 Basic Usage
+ 2.2 Attaching processes
+3. Kernel API
+ 3.1 Overview
+ 3.2 Synchronization
+ 3.3 Subsystem API
+4. Questions
+
+1. Containers
+==========
+
+1.1 What are containers ?
+----------------------
+
+Containers provide a mechanism for aggregating/partitioning sets of
+tasks, and all their future children, into hierarchical groups with
+specialized behaviour.
+
+Definitions:
+
+A *container* associates a set of tasks with a set of parameters for one
+or more subsystems.
+
+A *subsystem* is a module that makes use of the task grouping
+facilities provided by containers to treat groups of tasks in
+particular ways. A subsystem is typically a "resource controller" that
+schedules a resource or applies per-container limits, but it may be
+anything that wants to act on a group of processes, e.g. a
+virtualization subsystem.
+
+A *hierarchy* is a set of containers arranged in a tree, such that
+every task in the system is in exactly one of the containers in the
+hierarchy, and a set of subsystems; each subsystem has system-specific

+state attached to each container in the hierarchy. Each hierarchy has
+an instance of the container virtual filesystem associated with it.
+
+At any one time there may be multiple active hierachies of task
+containers. Each hierarchy is a partition of all tasks in the system.
+
+User level code may create and destroy containers by name in an
+instance of the container virtual file system, specify and query to
+which container a task is assigned, and list the task pids assigned to
+a container. Those creations and assignments only affect the hierarchy
+associated with that instance of the container file system.
+
+On their own, the only use for containers is for simple job
+tracking. The intention is that other subsystems hook into the generic
+container support to provide new attributes for containers, such as
+accounting/limiting the resources which processes in a container can
+access. For example, cpusets (see Documentation/cpusets.txt) allows
+you to associate a set of CPUs and a set of memory nodes with the
+tasks in each container.
+
+1.2 Why are containers needed ?
+---------------------------
+
+There are multiple efforts to provide process aggregations in the
+Linux kernel, mainly for resource tracking purposes. Such efforts
+include cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server
+namespaces. These all require the basic notion of a
+grouping/partitioning of processes, with newly forked processes ending
+in the same group (container) as their parent process.
+
+The kernel container patch provides the minimum essential kernel
+mechanisms required to efficiently implement such groups. It has
+minimal impact on the system fast paths, and provides hooks for
+specific subsystems such as cpusets to provide additional behaviour as
+desired.
+
+Multiple hierarchy support is provided to allow for situations where
+the division of tasks into containers is distinctly different for
+different subsystems - having parallel hierarchies allows each
+hierarchy to be a natural division of tasks, without having to handle
+complex combinations of tasks that would be present if several
+unrelated subsystems needed to be forced into the same tree of
+containers.
+
+At one extreme, each resource controller or subsystem could be in a
+separate hierarchy; at the other extreme, all subsystems
+would be attached to the same hierarchy.
+

+As an example of a scenario (originally proposed by vatsa@in.ibm.com)
+that can benefit from multiple hierarchies, consider a large
+university server with various users - students, professors, system
+tasks etc. The resource planning for this server could be along the
+following lines:
+
+	CPU :		Top cpuset
+			/	\
+		CPUSet1		CPUSet2
+		 |		 |
+		(Profs)		(Students)
+
+		In addition (system tasks) are attached to topcpuset (so
+		that they can run anywhere) with a limit of 20%
+
+	Memory : Professors (50%), students (30%), system (20%)
+
+	Disk : Prof (50%), students (30%), system (20%)
+
+	Network : WWW browsing (20%), Network File System (60%), others (20%)
+			/ \
+		Prof (15%) students (5%)
+
+Browsers like firefox/lynx go into the WWW network class, while (k)nfsd go
+into NFS network class.
+
+At the same time firefox/lynx will share an appropriate CPU/Memory class
+depending on who launched it (prof/student).
+
+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
+
+	# echo browser_pid > /mnt/<restype>/<userclass>/tasks
+
+With only a single hierarchy, he now would potentially have to create
+a separate container for every browser launched and associate it with
+approp network and other resource class. This may lead to
+proliferation of such containers.
+
+Also lets say that the administrator would like to give enhanced network
+access temporarily to a student's browser (since it is night and the user
+wants to do online gaming :) OR give one of the students simulation
+apps enhanced CPU power,
+
+With ability to write pids directly to resource classes, its just a
+matter of :

+
+	# echo pid > /mnt/network/<new_class>/tasks
+	(after some time)
+	# echo pid > /mnt/network/<orig_class>/tasks
+
+Without this ability, he would have to split the container into
+multiple separate ones and then associate the new containers with the
+new resource classes.
+
+
+
+1.3 How are containers implemented ?
+---------------------------------
+
+Containers extends the kernel as follows:
+
+ - Each task in the system has a reference-counted pointer to a
+   css_group.
+
+ - A css_group contains a set of reference-counted pointers to
+   container_subsys_state objects, one for each container subsystem
+   registered in the system. There is no direct link from a task to
+   the container of which it's a member in each hierarchy, but this
+   can be determined by following pointers through the
+   container_subsys_state objects. This is because accessing the
+   subsystem state is something that's expected to happen frequently
+   and in performance-critical code, whereas operations that require a
+   task's actual container assignments (in particular, moving between
+   containers) are less common.
+
+ - A container hierarchy filesystem can be mounted for browsing and
+   manipulation from user space.
+
+ - You can list all the tasks (by pid) attached to any container.
+
+The implementation of containers requires a few, simple hooks
+into the rest of the kernel, none in performance critical paths:
+
+ - in init/main.c, to initialize the root containers and initial
+   css_group at system boot.
+
+ - in fork and exit, to attach and detach a task from its css_group.
+
+In addition a new file system, of type "container" may be mounted, to
+enable browsing and modifying the containers presently known to the
+kernel. When mounting a container hierarchy, you may specify a
+comma-separated list of subsystems to mount as the filesystem mount
+options. By default, mounting the container filesystem attempts to

+mount a hierarchy containing all registered subsystems.
+
+If an active hierarchy with exactly the same set of subsystems already
+exists, it will be reused for the new mount. If no existing hierarchy
+matches, and any of the requested subsystems are in use in an existing
+hierarchy, the mount will fail with -EBUSY. Otherwise, a new hierarchy
+is activated, associated with the requested subsystems.
+
+It's not currently possible to bind a new subsystem to an active
+container hierarchy, or to unbind a subsystem from an active container
+hierarchy. This may be possible in future, but is fraught with nasty
+error-recovery issues.
+
+When a container filesystem is unmounted, if there are any
+subcontainers created below the top-level container, that hierarchy
+will remain active even though unmounted; if there are no
+subcontainers then the hierarchy will be deactivated.
+
+No new system calls are added for containers - all support for
+querying and modifying containers is via this container file system.
+
+Each task under /proc has an added file named 'container' displaying,
+for each active hierarchy, the subsystem names and the container name
+as the path relative to the root of the container file system.
+
+Each container is represented by a directory in the container file system
+containing the following files describing that container:
+
+ - tasks: list of tasks (by pid) attached to that container
+ - notify_on_release flag: run /sbin/container_release_agent on exit?
+
+Other subsystems such as cpusets may add additional files in each
+container dir
+
+New containers are created using the mkdir system call or shell
+command. The properties of a container, such as its flags, are
+modified by writing to the appropriate file in that containers
+directory, as listed above.
+
+The named hierarchical structure of nested containers allows partitioning
+a large system into nested, dynamically changeable, "soft-partitions".
+
+The attachment of each task, automatically inherited at fork by any
+children of that task, to a container allows organizing the work load
+on a system into related sets of tasks. A task may be re-attached to
+any other container, if allowed by the permissions on the necessary
+container file system directories.
+

+When a task is moved from one container to another, it gets a new
+css_group pointer - if there's an already existing css_group with the
+desired collection of containers then that group is reused, else a new
+css_group is allocated. Note that the current implementation uses a
+linear search to locate an appropriate existing css_group, so isn't
+very efficient. A future version will use a hash table for better
+performance.
+
+The use of a Linux virtual file system (vfs) to represent the
+container hierarchy provides for a familiar permission and name space
+for containers, with a minimum of additional kernel code.
+
+1.4 What does notify_on_release do ?
+------------------------------------
+
+*** notify_on_release is disabled in the current patch set. It will be
+*** reactivated in a future patch in a less-intrusive manner
+
+If the notify_on_release flag is enabled (1) in a container, then
+whenever the last task in the container leaves (exits or attaches to
+some other container) and the last child container of that container
+is removed, then the kernel runs the command specified by the contents
+of the "release_agent" file in that hierarchy's root directory,
+supplying the pathname (relative to the mount point of the container
+file system) of the abandoned container. This enables automatic
+removal of abandoned containers. The default value of
+notify_on_release in the root container at system boot is disabled
+(0). The default value of other containers at creation is the current
+value of their parents notify_on_release setting. The default value of
+a container hierarchy's release_agent path is empty.
+
+1.5 How do I use containers ?
+-----------------------------
+
+To start a new job that is to be contained within a container, using
+the "cpuset" container subsystem, the steps are something like:
+
+ 1) mkdir /dev/container
+ 2) mount -t container -ocpuset cpuset /dev/container
+ 3) Create the new container by doing mkdir's and write's (or echo's) in
+   the /dev/container virtual file system.
+ 4) Start a task that will be the "founding father" of the new job.
+ 5) Attach that task to the new container by writing its pid to the
+   /dev/container tasks file for that container.
+ 6) fork, exec or clone the job tasks from this founding father task.
+
+For example, the following sequence of commands will setup a container
+named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,

+and then start a subshell 'sh' in that container:
+
+ mount -t container cpuset -ocpuset /dev/container
+ cd /dev/container
+ mkdir Charlie
+ cd Charlie
+ /bin/echo 2-3 > cpus
+ /bin/echo 1 > mems
+ /bin/echo $$ > tasks
+ sh
+ # The subshell 'sh' is now running in container Charlie
+ # The next line should display '/Charlie'
+ cat /proc/self/container
+
+2. Usage Examples and Syntax
+============================
+
+2.1 Basic Usage
+---------------
+
+Creating, modifying, using the containers can be done through the container
+virtual filesystem.
+
+To mount a container hierarchy will all available subsystems, type:
+# mount -t container xxx /dev/container
+
+The "xxx" is not interpreted by the container code, but will appear in
+/proc/mounts so may be any useful identifying string that you like.
+
+To mount a container hierarchy with just the cpuset and numtasks
+subsystems, type:
+# mount -t container -o cpuset,numtasks hier1 /dev/container
+
+To change the set of subsystems bound to a mounted hierarchy, just
+remount with different options:
+
+# mount -o remount,cpuset,ns /dev/container
+
+Note that changing the set of subsystems is currently only supported
+when the hierarchy consists of a single (root) container. Supporting
+the ability to arbitrarily bind/unbind subsystems from an existing
+container hierarchy is intended to be implemented in the future.
+
+Then under /dev/container you can find a tree that corresponds to the
+tree of the containers in the system. For instance, /dev/container
+is the container that holds the whole system.
+
+If you want to create a new container under /dev/container:

+# cd /dev/container
+# mkdir my_container
+
+Now you want to do something with this container.
+# cd my_container
+
+In this directory you can find several files:
+# ls
+notify_on_release release_agent tasks
+(plus whatever files are added by the attached subsystems)
+
+Now attach your shell to this container:
+# /bin/echo $$ > tasks
+
+You can also create containers inside your container by using mkdir in this
+directory.
+# mkdir my_sub_cs
+
+To remove a container, just use rmdir:
+# rmdir my_sub_cs
+
+This will fail if the container is in use (has containers inside, or
+has processes attached, or is held alive by other subsystem-specific
+reference).
+
+2.2 Attaching processes
+-----------------------
+
+# /bin/echo PID > tasks
+
+Note that it is PID, not PIDs. You can only attach ONE task at a time.
+If you have several tasks to attach, you have to do it one after another:
+
+# /bin/echo PID1 > tasks
+# /bin/echo PID2 > tasks
+ ...
+# /bin/echo PIDn > tasks
+
+3. Kernel API
+=============
+
+3.1 Overview
+------------
+
+Each kernel subsystem that wants to hook into the generic container
+system needs to create a container_subsys object. This contains
+various methods, which are callbacks from the container system, along
+with a subsystem id which will be assigned by the container system.

+
+Other fields in the container_subsys object include:
+
+- subsys_id: a unique array index for the subsystem, indicating which
+ entry in container->subsys[] this subsystem should be
+ managing. Initialized by container_register_subsys(); prior to this
+ it should be initialized to -1
+
+- hierarchy: an index indicating which hierarchy, if any, this
+ subsystem is currently attached to. If this is -1, then the
+ subsystem is not attached to any hierarchy, and all tasks should be
+ considered to be members of the subsystem's top_container. It should
+ be initialized to -1.
+
+- name: should be initialized to a unique subsystem name prior to
+ calling container_register_subsystem. Should be no longer than
+ MAX_CONTAINER_TYPE_NAMELEN
+
+Each container object created by the system has an array of pointers,
+indexed by subsystem id; this pointer is entirely managed by the
+subsystem; the generic container code will never touch this pointer.
+
+3.2 Synchronization
+-------------------
+
+There is a global mutex, container_mutex, used by the container
+system. This should be taken by anything that wants to modify a
+container. It may also be taken to prevent containers from being
+modified, but more specific locks may be more appropriate in that
+situation.
+
+See kernel/container.c for more details.
+
+Subsystems can take/release the container_mutex via the functions
+container_lock()/container_unlock(), and can
+take/release the callback_mutex via the functions
+container_lock()/container_unlock().
+
+Accessing a task's container pointer may be done in the following ways:
+- while holding container_mutex
+- while holding the task's alloc_lock (via task_lock())
+- inside an rcu_read_lock() section via rcu_dereference()
+
+3.3 Subsystem API
+-------------------------
+
+Each subsystem should:
+

+- add an entry in linux/container_subsys.h
+- define a container_subsys object called <name>_subsys
+
+Each subsystem may export the following methods. The only mandatory
+methods are create/destroy. Any others that are null are presumed to
+be successful no-ops.
+
+struct container_subsys_state *create(struct container *cont)
+LL=container_mutex
+
+Called to create a subsystem state object for a container. The
+subsystem should allocate its subsystem state object for the passed
+container, returning a pointer to the new object on success or a
+negative error code. On success, the subsystem pointer should point to
+a structure of type container_subsys_state (typically embedded in a
+larger subsystem-specific object), which will be initialized by the
+container system. Note that this will be called at initialization to
+create the root subsystem state for this subsystem; this case can be
+identified by the passed container object having a NULL parent (since
+it's the root of the hierarchy) and may be an appropriate place for
+initialization code.
+
+void destroy(struct container *cont)
+LL=container_mutex
+
+The container system is about to destroy the passed container; the
+subsystem should do any necessary cleanup
+
+int can_attach(struct container_subsys *ss, struct container *cont,
+	 struct task_struct *task)
+LL=container_mutex
+
+Called prior to moving a task into a container; if the subsystem
+returns an error, this will abort the attach operation. If a NULL
+task is passed, then a successful result indicates that *any*
+unspecified task can be moved into the container. Note that this isn't
+called on a fork. If this method returns 0 (success) then this should
+remain valid while the caller holds container_mutex.
+
+void attach(struct container_subsys *ss, struct container *cont,
+	 struct container *old_cont, struct task_struct *task)
+LL=container_mutex
+
+
+Called after the task has been attached to the container, to allow any
+post-attachment activity that requires memory allocations or blocking.
+
+void fork(struct container_subsy *ss, struct task_struct *task)

+LL=callback_mutex, maybe read_lock(tasklist_lock)
+
+Called when a task is forked into a container. Also called during
+registration for all existing tasks.
+
+void exit(struct container_subsys *ss, struct task_struct *task)
+LL=callback_mutex
+
+Called during task exit
+
+int populate(struct container_subsys *ss, struct container *cont)
+LL=none
+
+Called after creation of a container to allow a subsystem to populate
+the container directory with file entries. The subsystem should make
+calls to container_add_file() with objects of type cftype (see
+include/linux/container.h for details). Note that although this
+method can return an error code, the error code is currently not
+always handled well.
+
+void bind(struct container_subsys *ss, struct container *root)
+LL=callback_mutex
+
+Called when a container subsystem is rebound to a different hierarchy
+and root container. Currently this will only involve movement between
+the default hierarchy (which never has sub-containers) and a hierarchy
+that is being created/destroyed (and hence has no sub-containers).
+
+4. Questions
+============
+
+Q: what's up with this '/bin/echo' ?
+A: bash's builtin 'echo' command does not check calls to write() against
+   errors. If you use it in the container file system, you won't be
+   able to tell whether a command succeeded or failed.
+
+Q: When I attach processes, only the first of the line gets really attached !
+A: We can only return one error code per call to write(). So you should also
+   put only ONE pid.
+
Index: container-2.6.22-rc6-mm1/include/linux/container.h
 == =======
--- /dev/null
+++ container-2.6.22-rc6-mm1/include/linux/container.h
@@ -0,0 +1,214 @@
+#ifndef _LINUX_CONTAINER_H
+#define _LINUX_CONTAINER_H
+/*

```
+ *  container interface
+ *
+ *  Copyright (C) 2003 BULL SA
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ */
+
+#include <linux/sched.h>
+#include <linux/kref.h>
+#include <linux/cpumask.h>
+#include <linux/nodemask.h>
+#include <linux/rcupdate.h>
+
+#ifdef CONFIG_CONTAINERS
+
+struct containerfs_root;
+struct container_subsys;
+struct inode;
+
+extern int container_init_early(void);
+extern int container_init(void);
+extern void container_init_smp(void);
+extern void container_lock(void);
+extern void container_unlock(void);
+
+/* Per-subsystem/per-container state maintained by the system. */
+struct container_subsys_state {
+ /* The container that this subsystem is attached to. Useful
+  * for subsystems that want to know about the container
+  * hierarchy structure */
+ struct container *container;
+
+ /* State maintained by the container system to allow
+  * subsystems to be "busy". Should be accessed via css_get()
+  * and css_put() */
+
+ atomic_t refcnt;
+
+ unsigned long flags;
+};
+
+/* bits in struct container_subsys_state flags field */
+enum {
+ CSS_ROOT, /* This CSS is the root of the subsystem */
+};
+
+/*
+ * Call css_get() to hold a reference on the container;
```

```
+ *
+ */
+
+static inline void css_get(struct container_subsys_state *css)
+{
+ /* We don't need to reference count the root state */
+ if (!test_bit(CSS_ROOT, &css->flags))
+  atomic_inc(&css->refcnt);
+}
+/*
+ * css_put() should be called to release a reference taken by
+ * css_get()
+ */
+
+static inline void css_put(struct container_subsys_state *css)
+{
+ if (!test_bit(CSS_ROOT, &css->flags))
+  atomic_dec(&css->refcnt);
+}
+
+struct container {
+ unsigned long flags;  /* "unsigned long" so bitops work */
+
+ /* count users of this container. >0 means busy, but doesn't
+  * necessarily indicate the number of tasks in the
+  * container */
+ atomic_t count;
+
+ /*
+  * We link our 'sibling' struct into our parent's 'children'.
+  * Our children link their 'sibling' into our 'children'.
+  */
+ struct list_head sibling; /* my parent's children */
+ struct list_head children; /* my children */
+
+ struct container *parent; /* my parent */
+ struct dentry *dentry;    /* container fs entry */
+
+ /* Private pointers for each registered subsystem */
+ struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
+
+ struct containerfs_root *root;
+ struct container *top_container;
+};
+
+/* struct cftype:
+ *
+ * The files in the container filesystem mostly have a very simple read/write
```

```
+ * handling, some common function will take care of it. Nevertheless some cases
+ * (read tasks) are special and therefore I define this structure for every
+ * kind of file.
+ *
+ *
+ * When reading/writing to a file:
+ * - the container to use in file->f_dentry->d_parent->d_fsdata
+ * - the 'cftype' of the file is file->f_dentry->d_fsdata
+ */
+
+#define MAX_CFTYPE_NAME 64
+struct cftype {
+ /* By convention, the name should begin with the name of the
+  * subsystem, followed by a period */
+ char name[MAX_CFTYPE_NAME];
+ int private;
+ int (*open) (struct inode *inode, struct file *file);
+ ssize_t (*read) (struct container *cont, struct cftype *cft,
+    struct file *file,
+    char __user *buf, size_t nbytes, loff_t *ppos);
+ /*
+  * read_uint() is a shortcut for the common case of returning a
+  * single integer. Use it in place of read()
+  */
+ u64 (*read_uint) (struct container *cont, struct cftype *cft);
+ ssize_t (*write) (struct container *cont, struct cftype *cft,
+     struct file *file,
+     const char __user *buf, size_t nbytes, loff_t *ppos);
+ int (*release) (struct inode *inode, struct file *file);
+};
+
+/* Add a new file to the given container directory. Should only be
+ * called by subsystems from within a populate() method */
+int container_add_file(struct container *cont, struct container_subsys *subsys,
+         const struct cftype *cft);
+
+/* Add a set of new files to the given container directory. Should
+ * only be called by subsystems from within a populate() method */
+int container_add_files(struct container *cont,
+   struct container_subsys *subsys,
+   const struct cftype cft[],
+   int count);
+
+int container_is_removed(const struct container *cont);
+
+int container_path(const struct container *cont, char *buf, int buflen);
+
+/* Return true if the container is a descendant of the current container */
```

```
+int container_is_descendant(const struct container *cont);
+
+/* Container subsystem type. See Documentation/containers.txt for details */
+
+struct container_subsys {
+ struct container_subsys_state *(*create)(struct container_subsys *ss,
+        struct container *cont);
+ void (*destroy)(struct container_subsys *ss, struct container *cont);
+ int (*can_attach)(struct container_subsys *ss,
+     struct container *cont, struct task_struct *tsk);
+ void (*attach)(struct container_subsys *ss, struct container *cont,
+   struct container *old_cont, struct task_struct *tsk);
+ void (*fork)(struct container_subsys *ss, struct task_struct *task);
+ void (*exit)(struct container_subsys *ss, struct task_struct *task);
+ int (*populate)(struct container_subsys *ss,
+   struct container *cont);
+ void (*bind)(struct container_subsys *ss, struct container *root);
+ int subsys_id;
+ int active;
+ int early_init;
+#define MAX_CONTAINER_TYPE_NAMELEN 32
+ const char *name;
+
+ /* Protected by RCU */
+ struct containerfs_root *root;
+
+ struct list_head sibling;
+
+ void *private;
+};
+
+#define SUBSYS(_x) extern struct container_subsys _x ## _subsys;
+#include <linux/container_subsys.h>
+#undef SUBSYS
+
+static inline struct container_subsys_state *container_subsys_state(
+ struct container *cont, int subsys_id)
+{
+ return cont->subsys[subsys_id];
+}
+
+static inline struct container_subsys_state *task_subsys_state(
+ struct task_struct *task, int subsys_id)
+{
+ return rcu_dereference(task->containers.subsys[subsys_id]);
+}
+
+static inline struct container* task_container(struct task_struct *task,
```

```
+           int subsys_id)
+{
+ return task_subsys_state(task, subsys_id)->container;
+}
+
+int container_path(const struct container *cont, char *buf, int buflen);
+
+#else /* !CONFIG_CONTAINERS */
+
+static inline int container_init_early(void) { return 0; }
+static inline int container_init(void) { return 0; }
+static inline void container_init_smp(void) {}
+
+static inline void container_lock(void) {}
+static inline void container_unlock(void) {}
+
+#endif /* !CONFIG_CONTAINERS */
+
+#endif /* _LINUX_CONTAINER_H */
Index: container-2.6.22-rc6-mm1/include/linux/container_subsys.h
 ===================================================================
--- /dev/null
+++ container-2.6.22-rc6-mm1/include/linux/container_subsys.h
@@ -0,0 +1,10 @@
+/* Add subsystem definitions of the form SUBSYS(<name>) in this
+ * file. Surround each one by a line of comment markers so that
+ * patches don't collide
+ */
+
+/* */
+
+/* */
+
+/* */
Index: container-2.6.22-rc6-mm1/include/linux/sched.h
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/include/linux/sched.h
+++ container-2.6.22-rc6-mm1/include/linux/sched.h
@@ -909,6 +909,34 @@ struct sched_entity {
  unsigned long wait_runtime_overruns, wait_runtime_underruns;
 };

+#ifdef CONFIG_CONTAINERS
+
+#define SUBSYS(_x) _x ## _subsys_id,
+enum container_subsys_id {
+#include <linux/container_subsys.h>
+ CONTAINER_SUBSYS_COUNT
```

```
+};
+#undef SUBSYS
+
+/* A css_group is a structure holding pointers to a set of
+ * container_subsys_state objects.
+ */
+
+struct css_group {
+
+ /* Set of subsystem states, one for each subsystem. NULL for
+  * subsystems that aren't part of this hierarchy. These
+  * pointers reduce the number of dereferences required to get
+  * from a task to its state for a given container, but result
+  * in increased space usage if tasks are in wildly different
+  * groupings across different hierarchies. This array is
+  * immutable after creation */
+ struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
+
+};
+
+#endif /* CONFIG_CONTAINERS */
+
 struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
@@ -1145,6 +1173,9 @@ struct task_struct {
  int cpuset_mems_generation;
  int cpuset_mem_spread_rotor;
 #endif
+#ifdef CONFIG_CONTAINERS
+ struct css_group containers;
+#endif
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
  struct compat_robust_list_head __user *compat_robust_list;
@@ -1593,7 +1624,8 @@ static inline int thread_group_empty(str
 /*
  * Protects ->fs, ->files, ->mm, ->group_info, ->comm, keyring
  * subscriptions and synchronises with wait4().  Also used in procfs.  Also
- * pins the final release of task.io_context.  Also protects ->cpuset.
+ * pins the final release of task.io_context.  Also protects ->cpuset and
+ * ->container.subsys[].
  *
  * Nests both inside and outside of read_lock(&tasklist_lock).
  * It must not be nested with write_lock_irq(&tasklist_lock),
Index: container-2.6.22-rc6-mm1/init/Kconfig
 ===================================================================== =======
--- container-2.6.22-rc6-mm1.orig/init/Kconfig
```

```
+++ container-2.6.22-rc6-mm1/init/Kconfig
@@ -295,6 +295,14 @@ config LOG_BUF_SHIFT
        13 =>  8 KB
        12 =>  4 KB

+config CONTAINERS
+ bool "Container support"
+ help
+   This option will let you use process container subsystems
+   such as Cpusets
+
+   Say N if unsure.
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
Index: container-2.6.22-rc6-mm1/init/main.c
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/init/main.c
+++ container-2.6.22-rc6-mm1/init/main.c
@@ -39,6 +39,7 @@
 #include <linux/writeback.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/efi.h>
 #include <linux/tick.h>
 #include <linux/interrupt.h>
@@ -519,6 +520,7 @@ asmlinkage void __init start_kernel(void
   */
  unwind_init();
  lockdep_init();
+ container_init_early();

  local_irq_disable();
  early_boot_irqs_off();
@@ -640,6 +642,7 @@ asmlinkage void __init start_kernel(void
 #ifdef CONFIG_PROC_FS
  proc_root_init();
 #endif
+ container_init();
  cpuset_init();
  taskstats_init_early();
  delayacct_init();
Index: container-2.6.22-rc6-mm1/kernel/Makefile
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/kernel/Makefile
+++ container-2.6.22-rc6-mm1/kernel/Makefile
```

```
@@ -37,6 +37,7 @@ obj-$(CONFIG_PM) += power/
 obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
+obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
Index: container-2.6.22-rc6-mm1/kernel/container.c
 ===================================================================== =======
--- /dev/null
+++ container-2.6.22-rc6-mm1/kernel/container.c
@@ -0,0 +1,1199 @@
+/*
+ *  kernel/container.c
+ *
+ *  Generic process-grouping system.
+ *
+ *  Based originally on the cpuset system, extracted by Paul Menage
+ *  Copyright (C) 2006 Google, Inc
+ *
+ *  Copyright notices from the original cpuset code:
+ *  --------------------------------------------------
+ *  Copyright (C) 2003 BULL SA.
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ *  Portions derived from Patrick Mochel's sysfs code.
+ *  sysfs is Copyright (c) 2001-3 Patrick Mochel
+ *
+ *  2003-10-10 Written by Simon Derr.
+ *  2003-10-22 Updates by Stephen Hemminger.
+ *  2004 May-July Rework by Paul Jackson.
+ *  ---------------------------------------------------
+ *
+ *  This file is subject to the terms and conditions of the GNU General Public
+ *  License.  See the file COPYING in the main directory of the Linux
+ *  distribution for more details.
+ */
+
+#include <linux/container.h>
+#include <linux/errno.h>
+#include <linux/fs.h>
+#include <linux/kernel.h>
+#include <linux/list.h>
+#include <linux/mm.h>
+#include <linux/mutex.h>
+#include <linux/mount.h>
+#include <linux/pagemap.h>
```

```
+#include <linux/rcupdate.h>
+#include <linux/sched.h>
+#include <linux/seq_file.h>
+#include <linux/slab.h>
+#include <linux/magic.h>
+#include <linux/spinlock.h>
+#include <linux/string.h>
+
+#include <asm/atomic.h>
+
+/* Generate an array of container subsystem pointers */
+#define SUBSYS(_x) &_x ## _subsys,
+
+static struct container_subsys *subsys[] = {
+#include <linux/container_subsys.h>
+};
+
+/*
+ * A containerfs_root represents the root of a container hierarchy,
+ * and may be associated with a superblock to form an active
+ * hierarchy
+ */
+struct containerfs_root {
+ struct super_block *sb;
+
+ /*
+  * The bitmask of subsystems intended to be attached to this
+  * hierarchy
+  */
+ unsigned long subsys_bits;
+
+ /* The bitmask of subsystems currently attached to this hierarchy */
+ unsigned long actual_subsys_bits;
+
+ /* A list running through the attached subsystems */
+ struct list_head subsys_list;
+
+ /* The root container for this hierarchy */
+ struct container top_container;
+
+ /* Tracks how many containers are currently defined in hierarchy.*/
+ int number_of_containers;
+
+ /* A list running through the mounted hierarchies */
+ struct list_head root_list;
+
+ /* Hierarchy-specific flags */
+ unsigned long flags;
```

```
+};
+
+
+/*
+ * The "rootnode" hierarchy is the "dummy hierarchy", reserved for the
+ * subsystems that are otherwise unattached - it never has more than a
+ * single container, and all tasks are part of that container.
+ */
+static struct containerfs_root rootnode;
+
+/* The list of hierarchy roots */
+
+static LIST_HEAD(roots);
+
+/* dummytop is a shorthand for the dummy hierarchy's top container */
+#define dummytop (&rootnode.top_container)
+
+/* This flag indicates whether tasks in the fork and exit paths should
+ * take callback_mutex and check for fork/exit handlers to call. This
+ * avoids us having to do extra work in the fork/exit path if none of the
+ * subsystems need to be called.
+ */
+static int need_forkexit_callback;
+
+/* bits in struct container flags field */
+enum {
+ CONT_REMOVED,
+};
+
+/* convenient tests for these bits */
+inline int container_is_removed(const struct container *cont)
+{
+ return test_bit(CONT_REMOVED, &cont->flags);
+}
+
+/* bits in struct containerfs_root flags field */
+enum {
+ ROOT_NOPREFIX, /* mounted subsystems have no named prefix */
+};
+
+/*
+ * for_each_subsys() allows you to iterate on each subsystem attached to
+ * an active hierarchy
+ */
+#define for_each_subsys(_root, _ss) \
+list_for_each_entry(_ss, &_root->subsys_list, sibling)
+
+/* for_each_root() allows you to iterate across the active hierarchies */
```

```
+#define for_each_root(_root) \
+list_for_each_entry(_root, &roots, root_list)
+
+/*
+ * There is one global container mutex. We also require taking
+ * task_lock() when dereferencing a task's container subsys pointers.
+ * See "The task_lock() exception", at the end of this comment.
+ *
+ * A task must hold container_mutex to modify containers.
+ *
+ * Any task can increment and decrement the count field without lock.
+ * So in general, code holding container_mutex can't rely on the count
+ * field not changing.  However, if the count goes to zero, then only
+ * attach_task() can increment it again.  Because a count of zero
+ * means that no tasks are currently attached, therefore there is no
+ * way a task attached to that container can fork (the other way to
+ * increment the count).  So code holding container_mutex can safely
+ * assume that if the count is zero, it will stay zero. Similarly, if
+ * a task holds container_mutex on a container with zero count, it
+ * knows that the container won't be removed, as container_rmdir()
+ * needs that mutex.
+ *
+ * The container_common_file_write handler for operations that modify
+ * the container hierarchy holds container_mutex across the entire operation,
+ * single threading all such container modifications across the system.
+ *
+ * The fork and exit callbacks container_fork() and container_exit(), don't
+ * (usually) take container_mutex.  These are the two most performance
+ * critical pieces of code here.  The exception occurs on container_exit(),
+ * when a task in a notify_on_release container exits.  Then container_mutex
+ * is taken, and if the container count is zero, a usermode call made
+ * to /sbin/container_release_agent with the name of the container (path
+ * relative to the root of container file system) as the argument.
+ *
+ * A container can only be deleted if both its 'count' of using tasks
+ * is zero, and its list of 'children' containers is empty.  Since all
+ * tasks in the system use _some_ container, and since there is always at
+ * least one task in the system (init, pid == 1), therefore, top_container
+ * always has either children containers and/or using tasks.  So we don't
+ * need a special hack to ensure that top_container cannot be deleted.
+ *
+ * The task_lock() exception
+ *
+ * The need for this exception arises from the action of
+ * attach_task(), which overwrites one tasks container pointer with
+ * another.  It does so using container_mutexe, however there are
+ * several performance critical places that need to reference
+ * task->container without the expense of grabbing a system global
```

```
+ * mutex.  Therefore except as noted below, when dereferencing or, as
+ * in attach_task(), modifying a task'ss container pointer we use
+ * task_lock(), which acts on a spinlock (task->alloc_lock) already in
+ * the task_struct routinely used for such matters.
+ *
+ * P.S.  One more locking exception.  RCU is used to guard the
+ * update of a tasks container pointer by attach_task()
+ */
+
+static DEFINE_MUTEX(container_mutex);
+
+/**
+ * container_lock - lock out any changes to container structures
+ *
+ */
+
+void container_lock(void)
+{
+ mutex_lock(&container_mutex);
+}
+
+/**
+ * container_unlock - release lock on container changes
+ *
+ * Undo the lock taken in a previous container_lock() call.
+ */
+
+void container_unlock(void)
+{
+ mutex_unlock(&container_mutex);
+}
+
+/*
+ * A couple of forward declarations required, due to cyclic reference loop:
+ * container_mkdir -> container_create -> container_populate_dir ->
+ * container_add_file -> container_create_file -> container_dir_inode_operations
+ * -> container_mkdir.
+ */
+
+static int container_mkdir(struct inode *dir, struct dentry *dentry, int mode);
+static int container_rmdir(struct inode *unused_dir, struct dentry *dentry);
+static int container_populate_dir(struct container *cont);
+static struct inode_operations container_dir_inode_operations;
+
+static struct inode *container_new_inode(mode_t mode, struct super_block *sb)
+{
+ struct inode *inode = new_inode(sb);
+ static struct backing_dev_info container_backing_dev_info = {
```

```
+  .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
+ };
+
+ if (inode) {
+  inode->i_mode = mode;
+  inode->i_uid = current->fsuid;
+  inode->i_gid = current->fsgid;
+  inode->i_blocks = 0;
+  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
+  inode->i_mapping->backing_dev_info = &container_backing_dev_info;
+ }
+ return inode;
+}
+
+static void container_diput(struct dentry *dentry, struct inode *inode)
+{
+ /* is dentry a directory ? if so, kfree() associated container */
+ if (S_ISDIR(inode->i_mode)) {
+  struct container *cont = dentry->d_fsdata;
+  BUG_ON(!(container_is_removed(cont)));
+  kfree(cont);
+ }
+ iput(inode);
+}
+
+static struct dentry *container_get_dentry(struct dentry *parent,
+        const char *name)
+{
+ struct dentry *d = lookup_one_len(name, parent, strlen(name));
+ static struct dentry_operations container_dops = {
+  .d_iput = container_diput,
+ };
+
+ if (!IS_ERR(d))
+  d->d_op = &container_dops;
+ return d;
+}
+
+static void remove_dir(struct dentry *d)
+{
+ struct dentry *parent = dget(d->d_parent);
+
+ d_delete(d);
+ simple_rmdir(parent->d_inode, d);
+ dput(parent);
+}
+
+static void container_clear_directory(struct dentry *dentry)
```

```
+{
+ struct list_head *node;
+
+ BUG_ON(!mutex_is_locked(&dentry->d_inode->i_mutex));
+ spin_lock(&dcache_lock);
+ node = dentry->d_subdirs.next;
+ while (node != &dentry->d_subdirs) {
+  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
+  list_del_init(node);
+  if (d->d_inode) {
+   /* This should never be called on a container
+    * directory with child containers */
+   BUG_ON(d->d_inode->i_mode & S_IFDIR);
+   d = dget_locked(d);
+   spin_unlock(&dcache_lock);
+   d_delete(d);
+   simple_unlink(dentry->d_inode, d);
+   dput(d);
+   spin_lock(&dcache_lock);
+  }
+  node = dentry->d_subdirs.next;
+ }
+ spin_unlock(&dcache_lock);
+}
+
+/*
+ * NOTE : the dentry must have been dget()'ed
+ */
+static void container_d_remove_dir(struct dentry *dentry)
+{
+ container_clear_directory(dentry);
+
+ spin_lock(&dcache_lock);
+ list_del_init(&dentry->d_u.d_child);
+ spin_unlock(&dcache_lock);
+ remove_dir(dentry);
+}
+
+static int rebind_subsystems(struct containerfs_root *root,
+         unsigned long final_bits)
+{
+ unsigned long added_bits, removed_bits;
+ struct container *cont = &root->top_container;
+ int i;
+
+ removed_bits = root->actual_subsys_bits & ~final_bits;
+ added_bits = final_bits & ~root->actual_subsys_bits;
+ /* Check that any added subsystems are currently free */
```

```
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  unsigned long long bit = 1ull << i;
+  struct container_subsys *ss = subsys[i];
+  if (!(bit & added_bits))
+   continue;
+  if (ss->root != &rootnode) {
+   /* Subsystem isn't free */
+   return -EBUSY;
+  }
+ }
+
+ /* Currently we don't handle adding/removing subsystems when
+  * any subcontainers exist. This is theoretically supportable
+  * but involves complex error handling, so it's being left until
+  * later */
+ if (!list_empty(&cont->children))
+  return -EBUSY;
+
+ /* Process each subsystem */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  struct container_subsys *ss = subsys[i];
+  unsigned long bit = 1UL << i;
+  if (bit & added_bits) {
+   /* We're binding this subsystem to this hierarchy */
+   BUG_ON(cont->subsys[i]);
+   BUG_ON(!dummytop->subsys[i]);
+   BUG_ON(dummytop->subsys[i]->container != dummytop);
+   cont->subsys[i] = dummytop->subsys[i];
+   cont->subsys[i]->container = cont;
+   list_add(&ss->sibling, &root->subsys_list);
+   rcu_assign_pointer(ss->root, root);
+   if (ss->bind)
+    ss->bind(ss, cont);
+
+  } else if (bit & removed_bits) {
+   /* We're removing this subsystem */
+   BUG_ON(cont->subsys[i] != dummytop->subsys[i]);
+   BUG_ON(cont->subsys[i]->container != cont);
+   if (ss->bind)
+    ss->bind(ss, dummytop);
+   dummytop->subsys[i]->container = dummytop;
+   cont->subsys[i] = NULL;
+   rcu_assign_pointer(subsys[i]->root, &rootnode);
+   list_del(&ss->sibling);
+  } else if (bit & final_bits) {
+   /* Subsystem state should already exist */
+   BUG_ON(!cont->subsys[i]);
+  } else {
```

```
+   /* Subsystem state shouldn't exist */
+   BUG_ON(cont->subsys[i]);
+  }
+ }
+ root->subsys_bits = root->actual_subsys_bits = final_bits;
+ synchronize_rcu();
+
+ return 0;
+}
+
+static int container_show_options(struct seq_file *seq, struct vfsmount *vfs)
+{
+ struct containerfs_root *root = vfs->mnt_sb->s_fs_info;
+ struct container_subsys *ss;
+
+ mutex_lock(&container_mutex);
+ for_each_subsys(root, ss)
+  seq_printf(seq, ",%s", ss->name);
+ if (test_bit(ROOT_NOPREFIX, &root->flags))
+  seq_puts(seq, ",noprefix");
+ mutex_unlock(&container_mutex);
+ return 0;
+}
+
+struct container_sb_opts {
+ unsigned long subsys_bits;
+ unsigned long flags;
+};
+
+/* Convert a hierarchy specifier into a bitmask of subsystems and
+ * flags. */
+static int parse_containerfs_options(char *data,
+         struct container_sb_opts *opts)
+{
+ char *token, *o = data ?: "all";
+
+ opts->subsys_bits = 0;
+ opts->flags = 0;
+
+ while ((token = strsep(&o, ",")) != NULL) {
+  if (!*token)
+   return -EINVAL;
+  if (!strcmp(token, "all")) {
+   opts->subsys_bits = (1 << CONTAINER_SUBSYS_COUNT) - 1;
+  } else if (!strcmp(token, "noprefix")) {
+   set_bit(ROOT_NOPREFIX, &opts->flags);
+  } else {
+   struct container_subsys *ss;
```

```
+   int i;
+   for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+    ss = subsys[i];
+    if (!strcmp(token, ss->name)) {
+     set_bit(i, &opts->subsys_bits);
+     break;
+    }
+   }
+   if (i == CONTAINER_SUBSYS_COUNT)
+    return -ENOENT;
+  }
+ }
+
+ /* We can't have an empty hierarchy */
+ if (!opts->subsys_bits)
+  return -EINVAL;
+
+ return 0;
+}
+
+static int container_remount(struct super_block *sb, int *flags, char *data)
+{
+ int ret = 0;
+ struct containerfs_root *root = sb->s_fs_info;
+ struct container *cont = &root->top_container;
+ struct container_sb_opts opts;
+
+ mutex_lock(&cont->dentry->d_inode->i_mutex);
+ mutex_lock(&container_mutex);
+
+ /* See what subsystems are wanted */
+ ret = parse_containerfs_options(data, &opts);
+ if (ret)
+  goto out_unlock;
+
+ /* Don't allow flags to change at remount */
+ if (opts.flags != root->flags) {
+  ret = -EINVAL;
+  goto out_unlock;
+ }
+
+ ret = rebind_subsystems(root, opts.subsys_bits);
+
+ /* (re)populate subsystem files */
+ if (!ret)
+  container_populate_dir(cont);
+
+ out_unlock:
```

```
+ mutex_unlock(&container_mutex);
+ mutex_unlock(&cont->dentry->d_inode->i_mutex);
+ return ret;
+}
+
+static struct super_operations container_ops = {
+ .statfs = simple_statfs,
+ .drop_inode = generic_delete_inode,
+ .show_options = container_show_options,
+ .remount_fs = container_remount,
+};
+
+static void init_container_root(struct containerfs_root *root)
+{
+ struct container *cont = &root->top_container;
+ INIT_LIST_HEAD(&root->subsys_list);
+ INIT_LIST_HEAD(&root->root_list);
+ root->number_of_containers = 1;
+ cont->root = root;
+ cont->top_container = cont;
+ INIT_LIST_HEAD(&cont->sibling);
+ INIT_LIST_HEAD(&cont->children);
+}
+
+static int container_test_super(struct super_block *sb, void *data)
+{
+ struct containerfs_root *new = data;
+ struct containerfs_root *root = sb->s_fs_info;
+
+ /* First check subsystems */
+ if (new->subsys_bits != root->subsys_bits)
+     return 0;
+
+ /* Next check flags */
+ if (new->flags != root->flags)
+  return 0;
+
+ return 1;
+}
+
+static int container_set_super(struct super_block *sb, void *data)
+{
+ int ret;
+ struct containerfs_root *root = data;
+
+ ret = set_anon_super(sb, NULL);
+ if (ret)
+  return ret;
```

```
+
+ sb->s_fs_info = root;
+ root->sb = sb;
+
+ sb->s_blocksize = PAGE_CACHE_SIZE;
+ sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
+ sb->s_magic = CONTAINER_SUPER_MAGIC;
+ sb->s_op = &container_ops;
+
+ return 0;
+}
+
+static int container_get_rootdir(struct super_block *sb)
+{
+ struct inode *inode =
+  container_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR, sb);
+ struct dentry *dentry;
+
+ if (!inode)
+  return -ENOMEM;
+
+ inode->i_op = &simple_dir_inode_operations;
+ inode->i_fop = &simple_dir_operations;
+ inode->i_op = &container_dir_inode_operations;
+ /* directories start off with i_nlink == 2 (for "." entry) */
+ inc_nlink(inode);
+ dentry = d_alloc_root(inode);
+ if (!dentry) {
+  iput(inode);
+  return -ENOMEM;
+ }
+ sb->s_root = dentry;
+ return 0;
+}
+
+static int container_get_sb(struct file_system_type *fs_type,
+    int flags, const char *unused_dev_name,
+    void *data, struct vfsmount *mnt)
+{
+ struct container_sb_opts opts;
+ int ret = 0;
+ struct super_block *sb;
+ struct containerfs_root *root;
+
+ /* First find the desired set of subsystems */
+ ret = parse_containerfs_options(data, &opts);
+ if (ret)
+  return ret;
```

```
+
+ root = kzalloc(sizeof(*root), GFP_KERNEL);
+ if (!root)
+  return -ENOMEM;
+
+ init_container_root(root);
+ root->subsys_bits = opts.subsys_bits;
+ root->flags = opts.flags;
+
+ sb = sget(fs_type, container_test_super, container_set_super, root);
+
+ if (IS_ERR(sb)) {
+  kfree(root);
+  return PTR_ERR(sb);
+ }
+
+ if (sb->s_fs_info != root) {
+  /* Reusing an existing superblock */
+  BUG_ON(sb->s_root == NULL);
+  kfree(root);
+  root = NULL;
+ } else {
+  /* New superblock */
+  struct container *cont = &root->top_container;
+
+  BUG_ON(sb->s_root != NULL);
+
+  ret = container_get_rootdir(sb);
+  if (ret)
+   goto drop_new_super;
+
+  mutex_lock(&container_mutex);
+
+  ret = rebind_subsystems(root, root->subsys_bits);
+  if (ret == -EBUSY) {
+   mutex_unlock(&container_mutex);
+   goto drop_new_super;
+  }
+
+  /* EBUSY should be the only error here */
+  BUG_ON(ret);
+
+  list_add(&root->root_list, &roots);
+
+  sb->s_root->d_fsdata = &root->top_container;
+  root->top_container.dentry = sb->s_root;
+
+  BUG_ON(!list_empty(&cont->sibling));
```

```
+  BUG_ON(!list_empty(&cont->children));
+  BUG_ON(root->number_of_containers != 1);
+
+  /*
+   * I believe that it's safe to nest i_mutex inside
+   * container_mutex in this case, since no-one else can
+   * be accessing this directory yet. But we still need
+   * to teach lockdep that this is the case - currently
+   * a containerfs remount triggers a lockdep warning
+   */
+  mutex_lock(&cont->dentry->d_inode->i_mutex);
+  container_populate_dir(cont);
+  mutex_unlock(&cont->dentry->d_inode->i_mutex);
+  mutex_unlock(&container_mutex);
+ }
+
+ return simple_set_mnt(mnt, sb);
+
+ drop_new_super:
+ up_write(&sb->s_umount);
+ deactivate_super(sb);
+ return ret;
+}
+
+static void container_kill_sb(struct super_block *sb) {
+ struct containerfs_root *root = sb->s_fs_info;
+ struct container *cont = &root->top_container;
+ int ret;
+
+ BUG_ON(!root);
+
+ BUG_ON(root->number_of_containers != 1);
+ BUG_ON(!list_empty(&cont->children));
+ BUG_ON(!list_empty(&cont->sibling));
+
+ mutex_lock(&container_mutex);
+
+ /* Rebind all subsystems back to the default hierarchy */
+ ret = rebind_subsystems(root, 0);
+ /* Shouldn't be able to fail ... */
+ BUG_ON(ret);
+
+ if (!list_empty(&root->root_list))
+  list_del(&root->root_list);
+ mutex_unlock(&container_mutex);
+
+ kfree(root);
+ kill_litter_super(sb);
```

```
+}
+
+static struct file_system_type container_fs_type = {
+ .name = "container",
+ .get_sb = container_get_sb,
+ .kill_sb = container_kill_sb,
+};
+
+static inline struct container *__d_cont(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+static inline struct cftype *__d_cft(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+/*
+ * Called with container_mutex held.  Writes path of container into buf.
+ * Returns 0 on success, -errno on error.
+ */
+int container_path(const struct container *cont, char *buf, int buflen)
+{
+ char *start;
+
+ start = buf + buflen;
+
+ *--start = '\0';
+ for (;;) {
+  int len = cont->dentry->d_name.len;
+  if ((start -= len) < buf)
+   return -ENAMETOOLONG;
+  memcpy(start, cont->dentry->d_name.name, len);
+  cont = cont->parent;
+  if (!cont)
+   break;
+  if (!cont->parent)
+   continue;
+  if (--start < buf)
+   return -ENAMETOOLONG;
+  *start = '/';
+ }
+ memmove(buf, start, buf + buflen - start);
+ return 0;
+}
+
+/* The various types of files and directories in a container file system */
```

```
+
+enum container_filetype {
+ FILE_ROOT,
+ FILE_DIR,
+ FILE_TASKLIST,
+};
+
+static ssize_t container_file_write(struct file *file, const char __user *buf,
+      size_t nbytes, loff_t *ppos)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct container *cont = __d_cont(file->f_dentry->d_parent);
+
+ if (!cft)
+  return -ENODEV;
+ if (!cft->write)
+  return -EINVAL;
+
+ return cft->write(cont, cft, file, buf, nbytes, ppos);
+}
+
+static ssize_t container_read_uint(struct container *cont, struct cftype *cft,
+       struct file *file,
+       char __user *buf, size_t nbytes,
+       loff_t *ppos)
+{
+ char tmp[64];
+ u64 val = cft->read_uint(cont, cft);
+ int len = sprintf(tmp, "%llu\n", (unsigned long long) val);
+
+ return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
+}
+
+static ssize_t container_file_read(struct file *file, char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct container *cont = __d_cont(file->f_dentry->d_parent);
+
+ if (!cft)
+  return -ENODEV;
+
+ if (cft->read)
+  return cft->read(cont, cft, file, buf, nbytes, ppos);
+ if (cft->read_uint)
+  return container_read_uint(cont, cft, file, buf, nbytes, ppos);
+ return -EINVAL;
+}
```

```
+
+static int container_file_open(struct inode *inode, struct file *file)
+{
+ int err;
+ struct cftype *cft;
+
+ err = generic_file_open(inode, file);
+ if (err)
+  return err;
+
+ cft = __d_cft(file->f_dentry);
+ if (!cft)
+  return -ENODEV;
+ if (cft->open)
+  err = cft->open(inode, file);
+ else
+  err = 0;
+
+ return err;
+}
+
+static int container_file_release(struct inode *inode, struct file *file)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ if (cft->release)
+  return cft->release(inode, file);
+ return 0;
+}
+
+/*
+ * container_rename - Only allow simple rename of directories in place.
+ */
+static int container_rename(struct inode *old_dir, struct dentry *old_dentry,
+       struct inode *new_dir, struct dentry *new_dentry)
+{
+ if (!S_ISDIR(old_dentry->d_inode->i_mode))
+  return -ENOTDIR;
+ if (new_dentry->d_inode)
+  return -EEXIST;
+ if (old_dir != new_dir)
+  return -EIO;
+ return simple_rename(old_dir, old_dentry, new_dir, new_dentry);
+}
+
+static struct file_operations container_file_operations = {
+ .read = container_file_read,
+ .write = container_file_write,
+ .llseek = generic_file_llseek,
```

```
+ .open = container_file_open,
+ .release = container_file_release,
+};
+
+static struct inode_operations container_dir_inode_operations = {
+ .lookup = simple_lookup,
+ .mkdir = container_mkdir,
+ .rmdir = container_rmdir,
+ .rename = container_rename,
+};
+
+static int container_create_file(struct dentry *dentry, int mode,
+    struct super_block *sb)
+{
+ struct inode *inode;
+
+ if (!dentry)
+  return -ENOENT;
+ if (dentry->d_inode)
+  return -EEXIST;
+
+ inode = container_new_inode(mode, sb);
+ if (!inode)
+  return -ENOMEM;
+
+ if (S_ISDIR(mode)) {
+  inode->i_op = &container_dir_inode_operations;
+  inode->i_fop = &simple_dir_operations;
+
+  /* start off with i_nlink == 2 (for "." entry) */
+  inc_nlink(inode);
+
+  /* start with the directory inode held, so that we can
+   * populate it without racing with another mkdir */
+  mutex_lock(&inode->i_mutex);
+ } else if (S_ISREG(mode)) {
+  inode->i_size = 0;
+  inode->i_fop = &container_file_operations;
+ }
+
+ d_instantiate(dentry, inode);
+ dget(dentry); /* Extra count - pin the dentry in core */
+ return 0;
+}
+
+/*
+ * container_create_dir - create a directory for an object.
+ * cont: the container we create the directory for.
```

```
+ *  It must have a valid ->parent field
+ *  And we are going to fill its ->dentry field.
+ * name: The name to give to the container directory. Will be copied.
+ * mode: mode to set on new directory.
+ */
+static int container_create_dir(struct container *cont, struct dentry *dentry,
+    int mode)
+{
+ struct dentry *parent;
+ int error = 0;
+
+ parent = cont->parent->dentry;
+ if (IS_ERR(dentry))
+  return PTR_ERR(dentry);
+ error = container_create_file(dentry, S_IFDIR | mode, cont->root->sb);
+ if (!error) {
+  dentry->d_fsdata = cont;
+  inc_nlink(parent->d_inode);
+  cont->dentry = dentry;
+ }
+ dput(dentry);
+
+ return error;
+}
+
+int container_add_file(struct container *cont,
+        struct container_subsys *subsys,
+        const struct cftype *cft)
+{
+ struct dentry *dir = cont->dentry;
+ struct dentry *dentry;
+ int error;
+
+ char name[MAX_CONTAINER_TYPE_NAMELEN + MAX_CFTYPE_NAME + 2] = { 0 };
+ if (subsys && !test_bit(ROOT_NOPREFIX, &cont->root->flags)) {
+  strcpy(name, subsys->name);
+  strcat(name, ".");
+ }
+ strcat(name, cft->name);
+ BUG_ON(!mutex_is_locked(&dir->d_inode->i_mutex));
+ dentry = container_get_dentry(dir, name);
+ if (!IS_ERR(dentry)) {
+  error = container_create_file(dentry, 0644 | S_IFREG,
+      cont->root->sb);
+  if (!error)
+   dentry->d_fsdata = (void *)cft;
+  dput(dentry);
+ } else
```

```
+  error = PTR_ERR(dentry);
+ return error;
+}
+
+int container_add_files(struct container *cont,
+   struct container_subsys *subsys,
+   const struct cftype cft[],
+   int count)
+{
+ int i, err;
+ for (i = 0; i < count; i++) {
+  err = container_add_file(cont, subsys, &cft[i]);
+  if (err)
+   return err;
+ }
+ return 0;
+}
+
+static int container_populate_dir(struct container *cont)
+{
+ int err;
+ struct container_subsys *ss;
+
+ /* First clear out any existing files */
+ container_clear_directory(cont->dentry);
+
+ for_each_subsys(cont->root, ss) {
+  if (ss->populate && (err = ss->populate(ss, cont)) < 0)
+   return err;
+ }
+
+ return 0;
+}
+
+static void init_container_css(struct container_subsys_state *css,
+         struct container_subsys *ss,
+         struct container *cont)
+{
+ css->container = cont;
+ atomic_set(&css->refcnt, 0);
+ css->flags = 0;
+ if (cont == dummytop)
+  set_bit(CSS_ROOT, &css->flags);
+ BUG_ON(cont->subsys[ss->subsys_id]);
+ cont->subsys[ss->subsys_id] = css;
+}
+
+/*
```

```
+ * container_create - create a container
+ * parent: container that will be parent of the new container.
+ * name:  name of the new container. Will be strcpy'ed.
+ * mode:  mode to set on new inode
+ *
+ * Must be called with the mutex on the parent inode held
+ */
+
+static long container_create(struct container *parent, struct dentry *dentry,
+       int mode)
+{
+ struct container *cont;
+ struct containerfs_root *root = parent->root;
+ int err = 0;
+ struct container_subsys *ss;
+ struct super_block *sb = root->sb;
+
+ cont = kzalloc(sizeof(*cont), GFP_KERNEL);
+ if (!cont)
+  return -ENOMEM;
+
+ /* Grab a reference on the superblock so the hierarchy doesn't
+  * get deleted on unmount if there are child containers.  This
+  * can be done outside container_mutex, since the sb can't
+  * disappear while someone has an open control file on the
+  * fs */
+ atomic_inc(&sb->s_active);
+
+ mutex_lock(&container_mutex);
+
+ cont->flags = 0;
+ INIT_LIST_HEAD(&cont->sibling);
+ INIT_LIST_HEAD(&cont->children);
+
+ cont->parent = parent;
+ cont->root = parent->root;
+ cont->top_container = parent->top_container;
+
+ for_each_subsys(root, ss) {
+  struct container_subsys_state *css = ss->create(ss, cont);
+  if (IS_ERR(css)) {
+   err = PTR_ERR(css);
+   goto err_destroy;
+  }
+  init_container_css(css, ss, cont);
+ }
+
+ list_add(&cont->sibling, &cont->parent->children);
```

```
+ root->number_of_containers++;
+
+ err = container_create_dir(cont, dentry, mode);
+ if (err < 0)
+  goto err_remove;
+
+ /* The container directory was pre-locked for us */
+ BUG_ON(!mutex_is_locked(&cont->dentry->d_inode->i_mutex));
+
+ err = container_populate_dir(cont);
+ /* If err < 0, we have a half-filled directory - oh well ;) */
+
+ mutex_unlock(&container_mutex);
+ mutex_unlock(&cont->dentry->d_inode->i_mutex);
+
+ return 0;
+
+ err_remove:
+
+ list_del(&cont->sibling);
+ root->number_of_containers--;
+
+ err_destroy:
+
+ for_each_subsys(root, ss) {
+  if (cont->subsys[ss->subsys_id])
+   ss->destroy(ss, cont);
+ }
+
+ mutex_unlock(&container_mutex);
+
+ /* Release the reference count that we took on the superblock */
+ deactivate_super(sb);
+
+ kfree(cont);
+ return err;
+}
+
+static int container_mkdir(struct inode *dir, struct dentry *dentry, int mode)
+{
+ struct container *c_parent = dentry->d_parent->d_fsdata;
+
+ /* the vfs holds inode->i_mutex already */
+ return container_create(c_parent, dentry, mode | S_IFDIR);
+}
+
+static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)
+{
```

```
+ struct container *cont = dentry->d_fsdata;
+ struct dentry *d;
+ struct container *parent;
+ struct container_subsys *ss;
+ struct super_block *sb;
+ struct containerfs_root *root;
+ int css_busy = 0;
+
+ /* the vfs holds both inode->i_mutex already */
+
+ mutex_lock(&container_mutex);
+ if (atomic_read(&cont->count) != 0) {
+  mutex_unlock(&container_mutex);
+  return -EBUSY;
+ }
+ if (!list_empty(&cont->children)) {
+  mutex_unlock(&container_mutex);
+  return -EBUSY;
+ }
+
+ parent = cont->parent;
+ root = cont->root;
+ sb = root->sb;
+
+ /* Check the reference count on each subsystem. Since we
+  * already established that there are no tasks in the
+  * container, if the css refcount is also 0, then there should
+  * be no outstanding references, so the subsystem is safe to
+  * destroy */
+ for_each_subsys(root, ss) {
+  struct container_subsys_state *css;
+  css = cont->subsys[ss->subsys_id];
+  if (atomic_read(&css->refcnt)) {
+   css_busy = 1;
+   break;
+  }
+ }
+ if (css_busy) {
+  mutex_unlock(&container_mutex);
+  return -EBUSY;
+ }
+
+ for_each_subsys(root, ss) {
+  if (cont->subsys[ss->subsys_id])
+   ss->destroy(ss, cont);
+ }
+
+ set_bit(CONT_REMOVED, &cont->flags);
```

```
+ /* delete my sibling from parent->children */
+ list_del(&cont->sibling);
+ spin_lock(&cont->dentry->d_lock);
+ d = dget(cont->dentry);
+ cont->dentry = NULL;
+ spin_unlock(&d->d_lock);
+
+ container_d_remove_dir(d);
+ dput(d);
+ root->number_of_containers--;
+
+ mutex_unlock(&container_mutex);
+ /* Drop the active superblock reference that we took when we
+  * created the container */
+ deactivate_super(sb);
+ return 0;
+}
+
+static void container_init_subsys(struct container_subsys *ss)
+{
+ struct task_struct *g, *p;
+ struct container_subsys_state *css;
+ printk(KERN_ERR "Initializing container subsys %s\n", ss->name);
+
+ /* Create the top container state for this subsystem */
+ ss->root = &rootnode;
+ css = ss->create(ss, dummytop);
+ /* We don't handle early failures gracefully */
+ BUG_ON(IS_ERR(css));
+ init_container_css(css, ss, dummytop);
+
+ /* Update all tasks to contain a subsys pointer to this state
+  * - since the subsystem is newly registered, all tasks are in
+  * the subsystem's top container. */
+
+  /* If this subsystem requested that it be notified with fork
+   * events, we should send it one now for every process in the
+   * system */
+
+ read_lock(&tasklist_lock);
+ init_task.containers.subsys[ss->subsys_id] = css;
+ if (ss->fork)
+  ss->fork(ss, &init_task);
+
+ do_each_thread(g, p) {
+  printk(KERN_INFO "Setting task %p css to %p (%d)\n", css, p, p->pid);
+  p->containers.subsys[ss->subsys_id] = css;
+  if (ss->fork)
```

```
+   ss->fork(ss, p);
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+
+ need_forkexit_callback |= ss->fork || ss->exit;
+
+ ss->active = 1;
+}
+
+/**
+ * container_init_early - initialize containers at system boot, and
+ * initialize any subsystems that request early init.
+ */
+int __init container_init_early(void)
+{
+ int i;
+ init_container_root(&rootnode);
+ list_add(&rootnode.root_list, &roots);
+
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+   struct container_subsys *ss = subsys[i];
+
+   BUG_ON(!ss->name);
+   BUG_ON(strlen(ss->name) > MAX_CONTAINER_TYPE_NAMELEN);
+   BUG_ON(!ss->create);
+   BUG_ON(!ss->destroy);
+   if (ss->subsys_id != i) {
+     printk(KERN_ERR "Subsys %s id == %d\n",
+            ss->name, ss->subsys_id);
+     BUG();
+   }
+
+   if (ss->early_init)
+     container_init_subsys(ss);
+ }
+ return 0;
+}
+
+/**
+ * container_init - register container filesystem and /proc file, and
+ * initialize any subsystems that didn't request early init.
+ */
+int __init container_init(void)
+{
+ int err;
+ int i;
+
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
```

```
+  struct container_subsys *ss = subsys[i];
+  if (!ss->early_init)
+   container_init_subsys(ss);
+ }
+
+ err = register_filesystem(&container_fs_type);
+ if (err < 0)
+  goto out;
+
+out:
+ return err;
+}
Index: container-2.6.22-rc6-mm1/include/linux/magic.h
 ===================================================== =======
--- container-2.6.22-rc6-mm1.orig/include/linux/magic.h
+++ container-2.6.22-rc6-mm1/include/linux/magic.h
@@ -41,5 +41,6 @@

 #define SMB_SUPER_MAGIC  0x517B
 #define USBDEVICE_SUPER_MAGIC 0x9fa2
+#define CONTAINER_SUPER_MAGIC 0x27e0eb

 #endif /* __LINUX_MAGIC_H__ */

--
```

Subject: [PATCH 02/10] Task Containers(V11): Add tasks file interface
Posted by Paul Menage on Fri, 20 Jul 2007 18:31:54 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        |  359 ++++++++++++++++++++++++++++++++++++++++++++-
 2 files changed, 368 insertions(+), 1 deletion(-)

Index: container-2.6.22-rc6-mm1/include/linux/container.h
 ===================================================== =======
--- container-2.6.22-rc6-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc6-mm1/include/linux/container.h
@@ -144,6 +144,16 @@ 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-rc6-mm1/kernel/container.c
 ================================================================== =======
--- container-2.6.22-rc6-mm1.orig/kernel/container.c
+++ container-2.6.22-rc6-mm1/kernel/container.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 container subsystem pointers */
@@ -704,6 +704,127 @@ int container_path(const struct containe
  return 0;
 }

+/*
+ * Return the first subsystem attached to a container's hierarchy, and
+ * its subsystem id.
+ */
+
+static void get_first_subsys(const struct container *cont,
+   struct container_subsys_state **css, int *subsys_id)
+{
+ const struct containerfs_root *root = cont->root;
+ const struct container_subsys *test_ss;
+ BUG_ON(list_empty(&root->subsys_list));
+ test_ss = list_entry(root->subsys_list.next,
+        struct container_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 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 */

 enum container_filetype {
@@ -712,6 +833,55 @@ enum container_filetype {
  FILE_TASKLIST,
 };

+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)
+{
+ enum container_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(&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)
 {
@@ -915,6 +1085,189 @@ 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().
+ */
+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 += 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 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);
+ 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 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;
@@ -923,6 +1276,10 @@ static int container_populate_dir(struct
  /* First clear out any existing files */
  container_clear_directory(cont->dentry);

+ err = container_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: [PATCH 04/10] Task Containers(V11): Add container_clone() interface.
Posted by Paul Menage on Fri, 20 Jul 2007 18:31:56 GMT
View Forum Message <> Reply to Message

This patch adds support for container_clone(), a way to create new
containers intended to be used for systems such as namespace
unsharing. A new subsystem callback, post_clone(), is added to allow
subsystems to automatically configure cloned containers.

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

 Documentation/containers.txt | 7 ++
 include/linux/container.h | 3

```
 kernel/container.c          |  135 ++++++++++++++++++++++++++++++++++++++++++++++
 3 files changed, 145 insertions(+)

Index: container-2.6.22-rc6-mm1/include/linux/container.h
 ===================================================================== =======
--- container-2.6.22-rc6-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc6-mm1/include/linux/container.h
@@ -174,6 +174,7 @@ struct container_subsys {
  void (*exit)(struct container_subsys *ss, struct task_struct *task);
  int (*populate)(struct container_subsys *ss,
    struct container *cont);
+ void (*post_clone)(struct container_subsys *ss, struct container *cont);
  void (*bind)(struct container_subsys *ss, struct container *root);
  int subsys_id;
  int active;
@@ -213,6 +214,8 @@ static inline struct container* task_con

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

+int container_clone(struct task_struct *tsk, struct container_subsys *ss);
+
 #else /* !CONFIG_CONTAINERS */

 static inline int container_init_early(void) { return 0; }
Index: container-2.6.22-rc6-mm1/kernel/container.c
 ===================================================================== =======
--- container-2.6.22-rc6-mm1.orig/kernel/container.c
+++ container-2.6.22-rc6-mm1/kernel/container.c
@@ -1675,3 +1675,138 @@ void container_exit(struct task_struct *
  tsk->containers = init_task.containers;
  task_unlock(tsk);
 }
+
+/**
+ * container_clone - duplicate the current container in the hierarchy
+ * that the given subsystem is attached to, and move this task into
+ * the new child
+ */
+int container_clone(struct task_struct *tsk, struct container_subsys *subsys)
+{
+ struct dentry *dentry;
+ int ret = 0;
+ char nodename[MAX_CONTAINER_TYPE_NAMELEN];
+ struct container *parent, *child;
+ struct inode *inode;
+ struct css_group *cg;
+ struct containerfs_root *root;
+ struct container_subsys *ss;
```

```
+
+ /* We shouldn't be called by an unregistered subsystem */
+ BUG_ON(!subsys->active);
+
+ /* First figure out what hierarchy and container we're dealing
+  * with, and pin them so we can drop container_mutex */
+ mutex_lock(&container_mutex);
+ again:
+ root = subsys->root;
+ if (root == &rootnode) {
+  printk(KERN_INFO
+         "Not cloning container for unused subsystem %s\n",
+         subsys->name);
+  mutex_unlock(&container_mutex);
+  return 0;
+ }
+ cg = &tsk->containers;
+ parent = task_container(tsk, subsys->subsys_id);
+
+ snprintf(nodename, MAX_CONTAINER_TYPE_NAMELEN, "node_%d", tsk->pid);
+
+ /* Pin the hierarchy */
+ atomic_inc(&parent->root->sb->s_active);
+
+ mutex_unlock(&container_mutex);
+
+ /* Now do the VFS work to create a container */
+ inode = parent->dentry->d_inode;
+
+ /* Hold the parent directory mutex across this operation to
+  * stop anyone else deleting the new container */
+ mutex_lock(&inode->i_mutex);
+ dentry = container_get_dentry(parent->dentry, nodename);
+ if (IS_ERR(dentry)) {
+  printk(KERN_INFO
+         "Couldn't allocate dentry for %s: %ld\n", nodename,
+         PTR_ERR(dentry));
+  ret = PTR_ERR(dentry);
+  goto out_release;
+ }
+
+ /* Create the container directory, which also creates the container */
+ ret = vfs_mkdir(inode, dentry, S_IFDIR | 0755);
+ child = __d_cont(dentry);
+ dput(dentry);
+ if (ret) {
+  printk(KERN_INFO
+         "Failed to create container %s: %d\n", nodename,
```

```
+         ret);
+  goto out_release;
+ }
+
+ if (!child) {
+  printk(KERN_INFO
+         "Couldn't find new container %s\n", nodename);
+  ret = -ENOMEM;
+  goto out_release;
+ }
+
+ /* The container now exists. Retake container_mutex and check
+  * that we're still in the same state that we thought we
+  * were. */
+ mutex_lock(&container_mutex);
+ if ((root != subsys->root) ||
+     (parent != task_container(tsk, subsys->subsys_id))) {
+  /* Aargh, we raced ... */
+  mutex_unlock(&inode->i_mutex);
+
+  deactivate_super(parent->root->sb);
+  /* The container is still accessible in the VFS, but
+   * we're not going to try to rmdir() it at this
+   * point. */
+  printk(KERN_INFO
+         "Race in container_clone() - leaking container %s\n",
+         nodename);
+  goto again;
+ }
+
+ /* do any required auto-setup */
+ for_each_subsys(root, ss) {
+  if (ss->post_clone)
+   ss->post_clone(ss, child);
+ }
+
+ /* All seems fine. Finish by moving the task into the new container */
+ ret = attach_task(child, tsk);
+ mutex_unlock(&container_mutex);
+
+ out_release:
+ mutex_unlock(&inode->i_mutex);
+ deactivate_super(parent->root->sb);
+ return ret;
+}
+
+/*
+ * See if "cont" is a descendant of the current task's container in
```

```
+ * the appropriate hierarchy
+ *
+ * If we are sending in dummytop, then presumably we are creating
+ * the top container in the subsystem.
+ *
+ * Called only by the ns (nsproxy) container.
+ */
+int container_is_descendant(const struct container *cont)
+{
+ int ret;
+ struct container *target;
+ int subsys_id;
+
+ if (cont == dummytop)
+  return 1;
+
+ get_first_subsys(cont, NULL, &subsys_id);
+ target = task_container(current, subsys_id);
+ while (cont != target && cont!= cont->top_container)
+  cont = cont->parent;
+ ret = (cont == target);
+ return ret;
+}
Index: container-2.6.22-rc6-mm1/Documentation/containers.txt
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/Documentation/containers.txt
+++ container-2.6.22-rc6-mm1/Documentation/containers.txt
@@ -504,6 +504,13 @@ include/linux/container.h for details).
 method can return an error code, the error code is currently not
 always handled well.

+void post_clone(struct container_subsys *ss, struct container *cont)
+
+Called at the end of container_clone() to do any paramater
+initialization which might be required before a task could attach.  For
+example in cpusets, no task may attach before 'cpus' and 'mems' are set
+up.
+
 void bind(struct container_subsys *ss, struct container *root)
 LL=callback_mutex


--
```

Subject: [PATCH 05/10] Task Containers(V11): Add procfs interface
Posted by Paul Menage on Fri, 20 Jul 2007 18:31:57 GMT

This patch adds:

/proc/containers - general system info

/proc/*/container - per-task container membership info

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

```
 fs/proc/base.c           |    7 ++
 include/linux/container.h |    2
 kernel/container.c       |  132 ++++++++++++++++++++++++++++++++++++++++++++++
 3 files changed, 141 insertions(+)

Index: container-2.6.22-rc6-mm1/fs/proc/base.c
 ===================================================================== =======
--- container-2.6.22-rc6-mm1.orig/fs/proc/base.c
+++ container-2.6.22-rc6-mm1/fs/proc/base.c
@@ -67,6 +67,7 @@
 #include <linux/mount.h>
 #include <linux/security.h>
 #include <linux/ptrace.h>
+#include <linux/container.h>
 #include <linux/cpuset.h>
 #include <linux/audit.h>
 #include <linux/poll.h>
@@ -2050,6 +2051,9 @@ static const struct pid_entry tgid_base_
 #ifdef CONFIG_CPUSETS
   REG("cpuset",     S_IRUGO, cpuset),
 #endif
+#ifdef CONFIG_CONTAINERS
+ REG("container",  S_IRUGO, container),
+#endif
   INF("oom_score",  S_IRUGO, oom_score),
   REG("oom_adj",    S_IRUGO|S_IWUSR, oom_adjust),
 #ifdef CONFIG_AUDITSYSCALL
@@ -2341,6 +2345,9 @@ static const struct pid_entry tid_base_s
 #ifdef CONFIG_CPUSETS
   REG("cpuset",    S_IRUGO, cpuset),
 #endif
+#ifdef CONFIG_CONTAINERS
+ REG("container",  S_IRUGO, container),
+#endif
   INF("oom_score", S_IRUGO, oom_score),
   REG("oom_adj",   S_IRUGO|S_IWUSR, oom_adjust),
 #ifdef CONFIG_AUDITSYSCALL
Index: container-2.6.22-rc6-mm1/kernel/container.c
```

```
 ========================================================== =======
--- container-2.6.22-rc6-mm1.orig/kernel/container.c
+++ container-2.6.22-rc6-mm1/kernel/container.c
@@ -33,6 +33,7 @@
 #include <linux/mutex.h>
 #include <linux/mount.h>
 #include <linux/pagemap.h>
+#include <linux/proc_fs.h>
 #include <linux/rcupdate.h>
 #include <linux/sched.h>
 #include <linux/seq_file.h>
@@ -247,6 +248,7 @@ static int container_mkdir(struct inode
 static int container_rmdir(struct inode *unused_dir, struct dentry *dentry);
 static int container_populate_dir(struct container *cont);
 static struct inode_operations container_dir_inode_operations;
+static struct file_operations proc_containerstats_operations;

 static struct inode *container_new_inode(mode_t mode, struct super_block *sb)
 {
@@ -1567,6 +1569,7 @@ int __init container_init(void)
 {
  int err;
  int i;
+ struct proc_dir_entry *entry;

  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
   struct container_subsys *ss = subsys[i];
@@ -1578,10 +1581,139 @@ int __init container_init(void)
  if (err < 0)
   goto out;

+ entry = create_proc_entry("containers", 0, NULL);
+ if (entry)
+  entry->proc_fops = &proc_containerstats_operations;
+
 out:
  return err;
 }

+/*
+ * proc_container_show()
+ *  - Print task's container paths into seq_file, one line for each hierarchy
+ *  - Used for /proc/<pid>/container.
+ *  - No need to task_lock(tsk) on this tsk->container reference, as it
+ *    doesn't really matter if tsk->container changes after we read it,
+ *    and we take container_mutex, keeping attach_task() from changing it
+ *    anyway.  No need to check that tsk->container != NULL, thanks to
+ *    the_top_container_hack in container_exit(), which sets an exiting tasks
```

```
+ *    container to top_container.
+ */
+
+/* TODO: Use a proper seq_file iterator */
+static int proc_container_show(struct seq_file *m, void *v)
+{
+ struct pid *pid;
+ struct task_struct *tsk;
+ char *buf;
+ int retval;
+ struct containerfs_root *root;
+
+ retval = -ENOMEM;
+ buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
+ if (!buf)
+  goto out;
+
+ retval = -ESRCH;
+ pid = m->private;
+ tsk = get_pid_task(pid, PIDTYPE_PID);
+ if (!tsk)
+  goto out_free;
+
+ retval = 0;
+
+ mutex_lock(&container_mutex);
+
+ for_each_root(root) {
+  struct container_subsys *ss;
+  struct container *cont;
+  int subsys_id;
+  int count = 0;
+
+  /* Skip this hierarchy if it has no active subsystems */
+  if (!root->actual_subsys_bits)
+   continue;
+  for_each_subsys(root, ss)
+   seq_printf(m, "%s%s", count++ ? "," : "", ss->name);
+  seq_putc(m, ':');
+  get_first_subsys(&root->top_container, NULL, &subsys_id);
+  cont = task_container(tsk, subsys_id);
+  retval = container_path(cont, buf, PAGE_SIZE);
+  if (retval < 0)
+   goto out_unlock;
+  seq_puts(m, buf);
+  seq_putc(m, '\n');
+ }
+
```

```
+out_unlock:
+ mutex_unlock(&container_mutex);
+ put_task_struct(tsk);
+out_free:
+ kfree(buf);
+out:
+ return retval;
+}
+
+static int container_open(struct inode *inode, struct file *file)
+{
+ struct pid *pid = PROC_I(inode)->pid;
+ return single_open(file, proc_container_show, pid);
+}
+
+struct file_operations proc_container_operations = {
+ .open  = container_open,
+ .read  = seq_read,
+ .llseek  = seq_lseek,
+ .release = single_release,
+};
+
+/* Display information about each subsystem and each hierarchy */
+static int proc_containerstats_show(struct seq_file *m, void *v)
+{
+ int i;
+ struct containerfs_root *root;
+
+ mutex_lock(&container_mutex);
+ seq_puts(m, "Hierarchies:\n");
+ for_each_root(root) {
+  struct container_subsys *ss;
+  int first = 1;
+  seq_printf(m, "%p: bits=%lx containers=%d (", root,
+      root->subsys_bits, root->number_of_containers);
+  for_each_subsys(root, ss) {
+   seq_printf(m, "%s%s", first ? "" : ", ", ss->name);
+   first = false;
+  }
+  seq_putc(m, ')');
+  if (root->sb) {
+   seq_printf(m, " s_active=%d",
+       atomic_read(&root->sb->s_active));
+  }
+  seq_putc(m, '\n');
+ }
+ seq_puts(m, "Subsystems:\n");
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
```

```
+  struct container_subsys *ss = subsys[i];
+  seq_printf(m, "%d: name=%s hierarchy=%p\n",
+       i, ss->name, ss->root);
+ }
+ mutex_unlock(&container_mutex);
+ return 0;
+}
+
+static int containerstats_open(struct inode *inode, struct file *file)
+{
+ return single_open(file, proc_containerstats_show, 0);
+}
+
+static struct file_operations proc_containerstats_operations = {
+ .open = containerstats_open,
+ .read = seq_read,
+ .llseek = seq_lseek,
+ .release = single_release,
+};
+
 /**
  * container_fork - attach newly forked task to its parents container.
  * @tsk: pointer to task_struct of forking parent process.
Index: container-2.6.22-rc6-mm1/include/linux/container.h
 ========================================================== =======
--- container-2.6.22-rc6-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc6-mm1/include/linux/container.h
@@ -29,6 +29,8 @@ extern void container_fork(struct task_s
 extern void container_fork_callbacks(struct task_struct *p);
 extern void container_exit(struct task_struct *p, int run_callbacks);

+extern struct file_operations proc_container_operations;
+
 /* Per-subsystem/per-container state maintained by the system. */
 struct container_subsys_state {
  /* The container that this subsystem is attached to. Useful

--
```

Subject: [PATCH 06/10] Task Containers(V11): Shared container subsystem group arrays

Posted by Paul Menage on Fri, 20 Jul 2007 18:31:58 GMT

View Forum Message <> Reply to Message

This patch replaces the struct css_group embedded in task_struct with a pointer; all tasks that have the same set of memberships across all hierarchies will share a css_group object, and will be linked via their

css_groups field to the "tasks" list_head in the css_group.

Assuming that many tasks share the same container assignments, this reduces overall space usage and keeps the size of the task_struct down (three pointers added to task_struct compared to a non-containers kernel, no matter how many subsystems are registered).

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

```
 Documentation/containers.txt |   14
 include/linux/container.h    |   89 +++++-
 include/linux/sched.h        |   33 --
 kernel/container.c           |  606 +++++++++++++++++++++++++++++++++++++++------
 kernel/fork.c                |    1
 5 files changed, 620 insertions(+), 123 deletions(-)

Index: container-2.6.22-rc6-mm1/Documentation/containers.txt
=================================================================== =======
--- container-2.6.22-rc6-mm1.orig/Documentation/containers.txt
+++ container-2.6.22-rc6-mm1/Documentation/containers.txt
@@ -176,7 +176,9 @@ Containers extends the kernel as follows
    subsystem state is something that's expected to happen frequently
    and in performance-critical code, whereas operations that require a
    task's actual container assignments (in particular, moving between
-   containers) are less common.
+   containers) are less common. A linked list runs through the cg_list
+   field of each task_struct using the css_group, anchored at
+   css_group->tasks.

  - A container hierarchy filesystem can be mounted  for browsing and
    manipulation from user space.
@@ -252,6 +254,16 @@ linear search to locate an appropriate e
 very efficient. A future version will use a hash table for better
 performance.

+To allow access from a container to the css_groups (and hence tasks)
+that comprise it, a set of cg_container_link objects form a lattice;
+each cg_container_link is linked into a list of cg_container_links for
+a single container on its cont_link_list field, and a list of
+cg_container_links for a single css_group on its cg_link_list.
+
+Thus the set of tasks in a container can be listed by iterating over
+each css_group that references the container, and sub-iterating over
+each css_group's task set.
+
 The use of a Linux virtual file system (vfs) to represent the
 container hierarchy provides for a familiar permission and name space
```

```
 for containers, with a minimum of additional kernel code.
Index: container-2.6.22-rc6-mm1/include/linux/container.h
 ===================================================== =======
--- container-2.6.22-rc6-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc6-mm1/include/linux/container.h
@@ -27,10 +27,19 @@ extern void container_lock(void);
 extern void container_unlock(void);
 extern void container_fork(struct task_struct *p);
 extern void container_fork_callbacks(struct task_struct *p);
+extern void container_post_fork(struct task_struct *p);
 extern void container_exit(struct task_struct *p, int run_callbacks);

 extern struct file_operations proc_container_operations;

+/* Define the enumeration of all container subsystems */
+#define SUBSYS(_x) _x ## _subsys_id,
+enum container_subsys_id {
+#include <linux/container_subsys.h>
+ CONTAINER_SUBSYS_COUNT
+};
+#undef SUBSYS
+
 /* Per-subsystem/per-container state maintained by the system. */
 struct container_subsys_state {
  /* The container that this subsystem is attached to. Useful
@@ -97,6 +106,52 @@ struct container {

  struct containerfs_root *root;
  struct container *top_container;
+
+ /*
+  * List of cg_container_links pointing at css_groups with
+  * tasks in this container. Protected by css_group_lock
+  */
+ struct list_head css_groups;
+};
+
+/* A css_group is a structure holding pointers to a set of
+ * container_subsys_state objects. This saves space in the task struct
+ * object and speeds up fork()/exit(), since a single inc/dec and a
+ * list_add()/del() can bump the reference count on the entire
+ * container set for a task.
+ */
+
+struct css_group {
+
+ /* Reference count */
+ struct kref ref;
```

```
+
+ /*
+  * List running through all container groups. Protected by
+  * css_group_lock
+  */
+ struct list_head list;
+
+ /*
+  * List running through all tasks using this container
+  * group. Protected by css_group_lock
+  */
+ struct list_head tasks;
+
+ /*
+  * List of cg_container_link objects on link chains from
+  * containers referenced from this css_group. Protected by
+  * css_group_lock
+  */
+ struct list_head cg_links;
+
+ /*
+  * Set of subsystem states, one for each subsystem. This array
+  * is immutable after creation apart from the init_css_group
+  * during subsystem registration (at boot time).
+  */
+ struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
+
 };

 /* struct cftype:
@@ -149,15 +204,7 @@ 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;
-}
+int container_task_count(const struct container *cont);

 /* Return true if the container is a descendant of the current container */
 int container_is_descendant(const struct container *cont);
@@ -205,7 +252,7 @@ static inline struct container_subsys_st
```

```
 static inline struct container_subsys_state *task_subsys_state(
  struct task_struct *task, int subsys_id)
 {
- return rcu_dereference(task->containers.subsys[subsys_id]);
+ return rcu_dereference(task->containers->subsys[subsys_id]);
 }

 static inline struct container* task_container(struct task_struct *task,
@@ -218,6 +265,27 @@ int container_path(const struct containe

 int container_clone(struct task_struct *tsk, struct container_subsys *ss);

+/* A container_iter should be treated as an opaque object */
+struct container_iter {
+ struct list_head *cg_link;
+ struct list_head *task;
+};
+
+/* To iterate across the tasks in a container:
+ *
+ * 1) call container_iter_start to intialize an iterator
+ *
+ * 2) call container_iter_next() to retrieve member tasks until it
+ *    returns NULL or until you want to end the iteration
+ *
+ * 3) call container_iter_end() to destroy the iterator.
+ */
+void container_iter_start(struct container *cont, struct container_iter *it);
+struct task_struct *container_iter_next(struct container *cont,
+     struct container_iter *it);
+void container_iter_end(struct container *cont, struct container_iter *it);
+
+
 #else /* !CONFIG_CONTAINERS */

 static inline int container_init_early(void) { return 0; }
@@ -225,6 +293,7 @@ static inline int container_init(void) {
 static inline void container_init_smp(void) {}
 static inline void container_fork(struct task_struct *p) {}
 static inline void container_fork_callbacks(struct task_struct *p) {}
+static inline void container_post_fork(struct task_struct *p) {}
 static inline void container_exit(struct task_struct *p, int callbacks) {}

 static inline void container_lock(void) {}
Index: container-2.6.22-rc6-mm1/include/linux/sched.h
 ===================================================== =======
--- container-2.6.22-rc6-mm1.orig/include/linux/sched.h
+++ container-2.6.22-rc6-mm1/include/linux/sched.h
```

```
@@ -909,34 +909,6 @@ struct sched_entity {
  unsigned long wait_runtime_overruns, wait_runtime_underruns;
 };

-#ifdef CONFIG_CONTAINERS
-
-#define SUBSYS(_x) _x ## _subsys_id,
-enum container_subsys_id {
-#include <linux/container_subsys.h>
- CONTAINER_SUBSYS_COUNT
-};
-#undef SUBSYS
-
-/* A css_group is a structure holding pointers to a set of
- * container_subsys_state objects.
- */
-
-struct css_group {
-
- /* Set of subsystem states, one for each subsystem. NULL for
-  * subsystems that aren't part of this hierarchy. These
-  * pointers reduce the number of dereferences required to get
-  * from a task to its state for a given container, but result
-  * in increased space usage if tasks are in wildly different
-  * groupings across different hierarchies. This array is
-  * immutable after creation */
- struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
-
-};
-
-#endif /* CONFIG_CONTAINERS */
-
 struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
@@ -1174,7 +1146,10 @@ struct task_struct {
  int cpuset_mem_spread_rotor;
 #endif
 #ifdef CONFIG_CONTAINERS
- struct css_group containers;
+ /* Container info protected by css_group_lock */
+ struct css_group *containers;
+ /* cg_list protected by css_group_lock and tsk->alloc_lock */
+ struct list_head cg_list;
 #endif
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
Index: container-2.6.22-rc6-mm1/kernel/container.c
```

```
 ========================================================== =======
--- container-2.6.22-rc6-mm1.orig/kernel/container.c
+++ container-2.6.22-rc6-mm1/kernel/container.c
@@ -95,6 +95,7 @@ static struct containerfs_root rootnode;
 /* The list of hierarchy roots */

 static LIST_HEAD(roots);
+static int root_count;

 /* dummytop is a shorthand for the dummy hierarchy's top container */
 #define dummytop (&rootnode.top_container)
@@ -133,12 +134,49 @@ list_for_each_entry(_ss, &_root->subsys_
 #define for_each_root(_root) \
 list_for_each_entry(_root, &roots, root_list)

-/* Each task_struct has an embedded css_group, so the get/put
- * operation simply takes a reference count on all the containers
- * referenced by subsystems in this css_group. This can end up
- * multiple-counting some containers, but that's OK - the ref-count is
- * just a busy/not-busy indicator; ensuring that we only count each
- * container once would require taking a global lock to ensure that no
+/* Link structure for associating css_group objects with containers */
+struct cg_container_link {
+ /*
+  * List running through cg_container_links associated with a
+  * container, anchored on container->css_groups
+  */
+ struct list_head cont_link_list;
+ /*
+  * List running through cg_container_links pointing at a
+  * single css_group object, anchored on css_group->cg_links
+  */
+ struct list_head cg_link_list;
+ struct css_group *cg;
+};
+
+/* The default css_group - used by init and its children prior to any
+ * hierarchies being mounted. It contains a pointer to the root state
+ * for each subsystem. Also used to anchor the list of css_groups. Not
+ * reference-counted, to improve performance when child containers
+ * haven't been created.
+ */
+
+static struct css_group init_css_group;
+static struct cg_container_link init_css_group_link;
+
+/* css_group_lock protects the list of css_group objects, and the
+ * chain of tasks off each css_group.  Nests outside task->alloc_lock
```

```
+ * due to container_iter_start() */
+static DEFINE_RWLOCK(css_group_lock);
+static int css_group_count;
+
+/* We don't maintain the lists running through each css_group to its
+ * task until after the first call to container_iter_start(). This
+ * reduces the fork()/exit() overhead for people who have containers
+ * compiled into their kernel but not actually in use */
+static int use_task_css_group_links;
+
+/* When we create or destroy a css_group, the operation simply
+ * takes/releases a reference count on all the containers referenced
+ * by subsystems in this css_group. This can end up multiple-counting
+ * some containers, but that's OK - the ref-count is just a
+ * busy/not-busy indicator; ensuring that we only count each container
+ * once would require taking a global lock to ensure that no
  * subsystems moved between hierarchies while we were doing so.
  *
  * Possible TODO: decide at boot time based on the number of
@@ -146,18 +184,230 @@ list_for_each_entry(_root, &roots, root_
  * it's better for performance to ref-count every subsystem, or to
  * take a global lock and only add one ref count to each hierarchy.
  */
-static void get_css_group(struct css_group *cg)
+
+/*
+ * unlink a css_group from the list and free it
+ */
+static void release_css_group(struct kref *k)
 {
+ struct css_group *cg = container_of(k, struct css_group, ref);
  int i;
+
+ write_lock(&css_group_lock);
+ list_del(&cg->list);
+ css_group_count--;
+ while (!list_empty(&cg->cg_links)) {
+  struct cg_container_link *link;
+  link = list_entry(cg->cg_links.next,
+       struct cg_container_link, cg_link_list);
+  list_del(&link->cg_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+ write_unlock(&css_group_lock);
  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++)
-  atomic_inc(&cg->subsys[i]->container->count);
+  atomic_dec(&cg->subsys[i]->container->count);
```

```
+ kfree(cg);
+}
+
+/*
+ * refcounted get/put for css_group objects
+ */
+static inline void get_css_group(struct css_group *cg)
+{
+ kref_get(&cg->ref);
 }

-static void put_css_group(struct css_group *cg)
+static inline void put_css_group(struct css_group *cg)
+{
+ kref_put(&cg->ref, release_css_group);
+}
+
+/*
+ * find_existing_css_group() is a helper for
+ * find_css_group(), and checks to see whether an existing
+ * css_group is suitable. This currently walks a linked-list for
+ * simplicity; a later patch will use a hash table for better
+ * performance
+ *
+ * oldcg: the container group that we're using before the container
+ * transition
+ *
+ * cont: the container that we're moving into
+ *
+ * template: location in which to build the desired set of subsystem
+ * state objects for the new container group
+ */
+
+static struct css_group *find_existing_css_group(
+ struct css_group *oldcg,
+ struct container *cont,
+ struct container_subsys_state *template[])
 {
  int i;
- for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++)
-  atomic_dec(&cg->subsys[i]->container->count);
+ struct containerfs_root *root = cont->root;
+ struct list_head *l = &init_css_group.list;
+
+ /* Built the set of subsystem state objects that we want to
+  * see in the new css_group */
+ 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 */
+   template[i] = cont->subsys[i];
+  } else {
+   /* Subsystem is not in this hierarchy, so we
+    * don't want to change the subsystem state */
+   template[i] = oldcg->subsys[i];
+  }
+ }
+
+ /* Look through existing container groups to find one to reuse */
+ do {
+  struct css_group *cg =
+   list_entry(l, struct css_group, list);
+
+  if (!memcmp(template, cg->subsys, sizeof(cg->subsys))) {
+   /* All subsystems matched */
+   return cg;
+  }
+  /* Try the next container group */
+  l = l->next;
+ } while (l != &init_css_group.list);
+
+ /* No existing container group matched */
+ return NULL;
+}
+
+/*
+ * allocate_cg_links() allocates "count" cg_container_link structures
+ * and chains them on tmp through their cont_link_list fields. Returns 0 on
+ * success or a negative error
+ */
+
+static int allocate_cg_links(int count, struct list_head *tmp)
+{
+ struct cg_container_link *link;
+ int i;
+ INIT_LIST_HEAD(tmp);
+ for (i = 0; i < count; i++) {
+  link = kmalloc(sizeof(*link), GFP_KERNEL);
+  if (!link) {
+   while (!list_empty(tmp)) {
+    link = list_entry(tmp->next,
+        struct cg_container_link,
+        cont_link_list);
+    list_del(&link->cont_link_list);
+    kfree(link);
```

```
+   }
+   return -ENOMEM;
+  }
+  list_add(&link->cont_link_list, tmp);
+ }
+ return 0;
+}
+
+static void free_cg_links(struct list_head *tmp)
+{
+ while (!list_empty(tmp)) {
+  struct cg_container_link *link;
+  link = list_entry(tmp->next,
+       struct cg_container_link,
+       cont_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+}
+
+/*
+ * find_css_group() takes an existing container group and a
+ * container object, and returns a css_group object that's
+ * equivalent to the old group, but with the given container
+ * substituted into the appropriate hierarchy. Must be called with
+ * container_mutex held
+ */
+
+static struct css_group *find_css_group(
+ struct css_group *oldcg, struct container *cont)
+{
+ struct css_group *res;
+ struct container_subsys_state *template[CONTAINER_SUBSYS_COUNT];
+ int i;
+
+ struct list_head tmp_cg_links;
+ struct cg_container_link *link;
+
+ /* First see if we already have a container group that matches
+  * the desired set */
+ write_lock(&css_group_lock);
+ res = find_existing_css_group(oldcg, cont, template);
+ if (res)
+  get_css_group(res);
+ write_unlock(&css_group_lock);
+
+ if (res)
+  return res;
```

```
+
+ res = kmalloc(sizeof(*res), GFP_KERNEL);
+ if (!res)
+  return NULL;
+
+ /* Allocate all the cg_container_link objects that we'll need */
+ if (allocate_cg_links(root_count, &tmp_cg_links) < 0) {
+  kfree(res);
+  return NULL;
+ }
+
+ kref_init(&res->ref);
+ INIT_LIST_HEAD(&res->cg_links);
+ INIT_LIST_HEAD(&res->tasks);
+
+ /* Copy the set of subsystem state objects generated in
+  * find_existing_css_group() */
+ memcpy(res->subsys, template, sizeof(res->subsys));
+
+ write_lock(&css_group_lock);
+ /* Add reference counts and links from the new css_group. */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  struct container *cont = res->subsys[i]->container;
+  struct container_subsys *ss = subsys[i];
+  atomic_inc(&cont->count);
+  /*
+   * We want to add a link once per container, so we
+   * only do it for the first subsystem in each
+   * hierarchy
+   */
+  if (ss->root->subsys_list.next == &ss->sibling) {
+   BUG_ON(list_empty(&tmp_cg_links));
+   link = list_entry(tmp_cg_links.next,
+       struct cg_container_link,
+       cont_link_list);
+   list_del(&link->cont_link_list);
+   list_add(&link->cont_link_list, &cont->css_groups);
+   link->cg = res;
+   list_add(&link->cg_link_list, &res->cg_links);
+  }
+ }
+ if (list_empty(&rootnode.subsys_list)) {
+  link = list_entry(tmp_cg_links.next,
+      struct cg_container_link,
+      cont_link_list);
+  list_del(&link->cont_link_list);
+  list_add(&link->cont_link_list, &dummytop->css_groups);
+  link->cg = res;
```

```
+  list_add(&link->cg_link_list, &res->cg_links);
+ }
+
+ BUG_ON(!list_empty(&tmp_cg_links));
+
+ /* Link this container group into the list */
+ list_add(&res->list, &init_css_group.list);
+ css_group_count++;
+ INIT_LIST_HEAD(&res->tasks);
+ write_unlock(&css_group_lock);
+
+ return res;
 }

 /*
@@ -516,6 +766,7 @@ static void init_container_root(struct c
  cont->top_container = cont;
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
+ INIT_LIST_HEAD(&cont->css_groups);
 }

 static int container_test_super(struct super_block *sb, void *data)
@@ -585,6 +836,8 @@ static int container_get_sb(struct file_
  int ret = 0;
  struct super_block *sb;
  struct containerfs_root *root;
+ struct list_head tmp_cg_links, *l;
+ INIT_LIST_HEAD(&tmp_cg_links);

  /* First find the desired set of subsystems */
  ret = parse_containerfs_options(data, &opts);
@@ -623,6 +876,19 @@ static int container_get_sb(struct file_

   mutex_lock(&container_mutex);

+  /*
+   * We're accessing css_group_count without locking
+   * css_group_lock here, but that's OK - it can only be
+   * increased by someone holding container_lock, and
+   * that's us. The worst that can happen is that we
+   * have some link structures left over
+   */
+  ret = allocate_cg_links(css_group_count, &tmp_cg_links);
+  if (ret) {
+   mutex_unlock(&container_mutex);
+   goto drop_new_super;
+  }
```

```
+
   ret = rebind_subsystems(root, root->subsys_bits);
   if (ret == -EBUSY) {
    mutex_unlock(&container_mutex);
@@ -633,10 +899,34 @@ static int container_get_sb(struct file_
   BUG_ON(ret);

   list_add(&root->root_list, &roots);
+  root_count++;

   sb->s_root->d_fsdata = &root->top_container;
   root->top_container.dentry = sb->s_root;

+  /* Link the top container in this hierarchy into all
+   * the css_group objects */
+  write_lock(&css_group_lock);
+  l = &init_css_group.list;
+  do {
+   struct css_group *cg;
+   struct cg_container_link *link;
+   cg = list_entry(l, struct css_group, list);
+   BUG_ON(list_empty(&tmp_cg_links));
+   link = list_entry(tmp_cg_links.next,
+       struct cg_container_link,
+       cont_link_list);
+   list_del(&link->cont_link_list);
+   link->cg = cg;
+   list_add(&link->cont_link_list,
+     &root->top_container.css_groups);
+   list_add(&link->cg_link_list, &cg->cg_links);
+   l = l->next;
+  } while (l != &init_css_group.list);
+  write_unlock(&css_group_lock);
+
+  free_cg_links(&tmp_cg_links);
+
   BUG_ON(!list_empty(&cont->sibling));
   BUG_ON(!list_empty(&cont->children));
   BUG_ON(root->number_of_containers != 1);
@@ -659,6 +949,7 @@ static int container_get_sb(struct file_
  drop_new_super:
  up_write(&sb->s_umount);
  deactivate_super(sb);
+ free_cg_links(&tmp_cg_links);
  return ret;
 }

@@ -680,8 +971,25 @@ static void container_kill_sb(struct sup
```

```
  /* Shouldn't be able to fail ... */
  BUG_ON(ret);

- if (!list_empty(&root->root_list))
+ /*
+  * Release all the links from css_groups to this hierarchy's
+  * root container
+  */
+ write_lock(&css_group_lock);
+ while (!list_empty(&cont->css_groups)) {
+  struct cg_container_link *link;
+  link = list_entry(cont->css_groups.next,
+      struct cg_container_link, cont_link_list);
+  list_del(&link->cg_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+ write_unlock(&css_group_lock);
+
+ if (!list_empty(&root->root_list)) {
   list_del(&root->root_list);
+  root_count--;
+ }
  mutex_unlock(&container_mutex);

  kfree(root);
@@ -765,9 +1073,9 @@ static int attach_task(struct container
  int retval = 0;
  struct container_subsys *ss;
  struct container *oldcont;
- struct css_group *cg = &tsk->containers;
+ struct css_group *cg = tsk->containers;
+ struct css_group *newcg;
  struct containerfs_root *root = cont->root;
- int i;
  int subsys_id;

  get_first_subsys(cont, NULL, &subsys_id);
@@ -786,26 +1094,32 @@ static int attach_task(struct container
   }
  }

+ /*
+  * Locate or allocate a new css_group for this task,
+  * based on its final set of containers
+  */
+ newcg = find_css_group(cg, cont);
+ if (!newcg) {
```

```
+  return -ENOMEM;
+ }
+
  task_lock(tsk);
  if (tsk->flags & PF_EXITING) {
   task_unlock(tsk);
+  put_css_group(newcg);
   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]);
-  }
- }
+ rcu_assign_pointer(tsk->containers, newcg);
  task_unlock(tsk);

+ /* Update the css_group linked lists if we're using them */
+ write_lock(&css_group_lock);
+ if (!list_empty(&tsk->cg_list)) {
+  list_del(&tsk->cg_list);
+  list_add(&tsk->cg_list, &newcg->tasks);
+ }
+ write_unlock(&css_group_lock);
+
  for_each_subsys(root, ss) {
   if (ss->attach) {
    ss->attach(ss, cont, oldcont, tsk);
@@ -813,6 +1127,7 @@ static int attach_task(struct container
  }

  synchronize_rcu();
+ put_css_group(cg);
  return 0;
 }

@@ -1114,28 +1429,102 @@ 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)
+/* Count the number of tasks in a container. */
+
+int container_task_count(const struct container *cont)
 {
  int count = 0;
- struct task_struct *g, *p;
- struct container_subsys_state *css;
- int subsys_id;
+ struct list_head *l;

- 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);
+ read_lock(&css_group_lock);
+ l = cont->css_groups.next;
+ while (l != &cont->css_groups) {
+  struct cg_container_link *link =
+   list_entry(l, struct cg_container_link, cont_link_list);
+  count += atomic_read(&link->cg->ref.refcount);
+  l = l->next;
+ }
+ read_unlock(&css_group_lock);
  return count;
 }

 /*
+ * Advance a list_head iterator.  The iterator should be positioned at
+ * the start of a css_group
+ */
+static void container_advance_iter(struct container *cont,
+       struct container_iter *it)
+{
+ struct list_head *l = it->cg_link;
+ struct cg_container_link *link;
+ struct css_group *cg;
+
+ /* Advance to the next non-empty css_group */
+ do {
+  l = l->next;
```

```
+  if (l == &cont->css_groups) {
+   it->cg_link = NULL;
+   return;
+  }
+  link = list_entry(l, struct cg_container_link, cont_link_list);
+  cg = link->cg;
+ } while (list_empty(&cg->tasks));
+ it->cg_link = l;
+ it->task = cg->tasks.next;
+}
+
+void container_iter_start(struct container *cont, struct container_iter *it)
+{
+ /*
+  * The first time anyone tries to iterate across a container,
+  * we need to enable the list linking each css_group to its
+  * tasks, and fix up all existing tasks.
+  */
+ if (!use_task_css_group_links) {
+  struct task_struct *p, *g;
+  write_lock(&css_group_lock);
+  use_task_css_group_links = 1;
+   do_each_thread(g, p) {
+   task_lock(p);
+   if (list_empty(&p->cg_list))
+    list_add(&p->cg_list, &p->containers->tasks);
+   task_unlock(p);
+   } while_each_thread(g, p);
+  write_unlock(&css_group_lock);
+ }
+ read_lock(&css_group_lock);
+ it->cg_link = &cont->css_groups;
+ container_advance_iter(cont, it);
+}
+
+struct task_struct *container_iter_next(struct container *cont,
+     struct container_iter *it)
+{
+ struct task_struct *res;
+ struct list_head *l = it->task;
+
+ /* If the iterator cg is NULL, we have no tasks */
+ if (!it->cg_link)
+  return NULL;
+ res = list_entry(l, struct task_struct, cg_list);
+ /* Advance iterator to find next entry */
+ l = l->next;
+ if (l == &res->containers->tasks) {
```

```
+  /* We reached the end of this task list - move on to
+   * the next cg_container_link */
+  container_advance_iter(cont, it);
+ } else {
+  it->task = l;
+ }
+ return res;
+}
+
+void container_iter_end(struct container *cont, struct container_iter *it)
+{
+ read_unlock(&css_group_lock);
+}
+
+/*
  * Stuff for reading the 'tasks' file.
  *
  * Reading this file can return large amounts of data if a container has
@@ -1164,22 +1553,15 @@ struct ctr_struct {
 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();
+ struct container_iter it;
+ struct task_struct *tsk;
+ container_iter_start(cont, &it);
+ while ((tsk = container_iter_next(cont, &it))) {
+  if (unlikely(n == npids))
+   break;
+  pidarray[n++] = pid_nr(task_pid(tsk));
+ }
+ container_iter_end(cont, &it);
  return n;
```

```
 }

@@ -1364,6 +1746,7 @@ static long container_create(struct cont
  cont->flags = 0;
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
+ INIT_LIST_HEAD(&cont->css_groups);

  cont->parent = parent;
  cont->root = parent->root;
@@ -1495,8 +1878,8 @@ static int container_rmdir(struct inode

 static void container_init_subsys(struct container_subsys *ss)
 {
- struct task_struct *g, *p;
  struct container_subsys_state *css;
+ struct list_head *l;
  printk(KERN_ERR "Initializing container subsys %s\n", ss->name);

  /* Create the top container state for this subsystem */
@@ -1506,26 +1889,32 @@ static void container_init_subsys(struct
  BUG_ON(IS_ERR(css));
  init_container_css(css, ss, dummytop);

- /* Update all tasks to contain a subsys pointer to this state
-  * - since the subsystem is newly registered, all tasks are in
-  * the subsystem's top container. */
+ /* Update all container groups to contain a subsys
+  * pointer to this state - since the subsystem is
+  * newly registered, all tasks and hence all container
+  * groups are in the subsystem's top container. */
+ write_lock(&css_group_lock);
+ l = &init_css_group.list;
+ do {
+  struct css_group *cg =
+   list_entry(l, struct css_group, list);
+  cg->subsys[ss->subsys_id] = dummytop->subsys[ss->subsys_id];
+  l = l->next;
+ } while (l != &init_css_group.list);
+ write_unlock(&css_group_lock);

   /* If this subsystem requested that it be notified with fork
    * events, we should send it one now for every process in the
    * system */
+  if (ss->fork) {
+   struct task_struct *g, *p;

- read_lock(&tasklist_lock);
```

```
- init_task.containers.subsys[ss->subsys_id] = css;
- if (ss->fork)
-  ss->fork(ss, &init_task);
-
- do_each_thread(g, p) {
-  printk(KERN_INFO "Setting task %p css to %p (%d)\n", css, p, p->pid);
-  p->containers.subsys[ss->subsys_id] = css;
-  if (ss->fork)
-   ss->fork(ss, p);
- } while_each_thread(g, p);
- read_unlock(&tasklist_lock);
+   read_lock(&tasklist_lock);
+   do_each_thread(g, p) {
+    ss->fork(ss, p);
+   } while_each_thread(g, p);
+   read_unlock(&tasklist_lock);
+  }

  need_forkexit_callback |= ss->fork || ss->exit;

@@ -1539,8 +1928,22 @@ static void container_init_subsys(struct
 int __init container_init_early(void)
 {
  int i;
+ kref_init(&init_css_group.ref);
+ kref_get(&init_css_group.ref);
+ INIT_LIST_HEAD(&init_css_group.list);
+ INIT_LIST_HEAD(&init_css_group.cg_links);
+ INIT_LIST_HEAD(&init_css_group.tasks);
+ css_group_count = 1;
  init_container_root(&rootnode);
  list_add(&rootnode.root_list, &roots);
+ root_count = 1;
+ init_task.containers = &init_css_group;
+
+ init_css_group_link.cg = &init_css_group;
+ list_add(&init_css_group_link.cont_link_list,
+   &rootnode.top_container.css_groups);
+ list_add(&init_css_group_link.cg_link_list,
+   &init_css_group.cg_links);

  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
   struct container_subsys *ss = subsys[i];
@@ -1698,6 +2101,7 @@ static int proc_containerstats_show(stru
   seq_printf(m, "%d: name=%s hierarchy=%p\n",
       i, ss->name, ss->root);
  }
+ seq_printf(m, "Container groups: %d\n", css_group_count);
```

```
  mutex_unlock(&container_mutex);
  return 0;
 }
@@ -1724,18 +2128,19 @@ static struct file_operations proc_conta
  * fork.c by dup_task_struct().  However, we ignore that copy, since
  * it was not made under the protection of RCU or container_mutex, so
  * might no longer be a valid container pointer.  attach_task() might
- * have already changed current->container, allowing the previously
- * referenced container to be removed and freed.
+ * have already changed current->containers, allowing the previously
+ * referenced container group to be removed and freed.
  *
  * At the point that container_fork() is called, 'current' is the parent
  * task, and the passed argument 'child' points to the child task.
  */
 void container_fork(struct task_struct *child)
 {
- rcu_read_lock();
- child->containers = rcu_dereference(current->containers);
- get_css_group(&child->containers);
- rcu_read_unlock();
+ task_lock(current);
+ child->containers = current->containers;
+ get_css_group(child->containers);
+ task_unlock(current);
+ INIT_LIST_HEAD(&child->cg_list);
 }

 /**
@@ -1756,6 +2161,21 @@ void container_fork_callbacks(struct tas
 }

 /**
+ * container_post_fork - called on a new task after adding it to the
+ * task list. Adds the task to the list running through its css_group
+ * if necessary. Has to be after the task is visible on the task list
+ * in case we race with the first call to container_iter_start() - to
+ * guarantee that the new task ends up on its list. */
+void container_post_fork(struct task_struct *child)
+{
+ if (use_task_css_group_links) {
+  write_lock(&css_group_lock);
+  if (list_empty(&child->cg_list))
+   list_add(&child->cg_list, &child->containers->tasks);
+  write_unlock(&css_group_lock);
+ }
+}
+/**
```

```
  * container_exit - detach container from exiting task
  * @tsk: pointer to task_struct of exiting process
  *
@@ -1793,6 +2213,7 @@ void container_fork_callbacks(struct tas
 void container_exit(struct task_struct *tsk, int run_callbacks)
 {
  int i;
+ struct css_group *cg;

  if (run_callbacks && need_forkexit_callback) {
   for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
@@ -1801,11 +2222,26 @@ void container_exit(struct task_struct *
     ss->exit(ss, tsk);
   }
  }
+
+ /*
+  * Unlink from the css_group task list if necessary.
+  * Optimistically check cg_list before taking
+  * css_group_lock
+  */
+ if (!list_empty(&tsk->cg_list)) {
+  write_lock(&css_group_lock);
+  if (!list_empty(&tsk->cg_list))
+   list_del(&tsk->cg_list);
+  write_unlock(&css_group_lock);
+ }
+
  /* Reassign the task to the init_css_group. */
  task_lock(tsk);
- put_css_group(&tsk->containers);
- tsk->containers = init_task.containers;
+ cg = tsk->containers;
+ tsk->containers = &init_css_group;
  task_unlock(tsk);
+ if (cg)
+  put_css_group(cg);
 }

 /**
@@ -1839,7 +2275,7 @@ int container_clone(struct task_struct *
   mutex_unlock(&container_mutex);
   return 0;
  }
- cg = &tsk->containers;
+ cg = tsk->containers;
  parent = task_container(tsk, subsys->subsys_id);
```

```
  snprintf(nodename, MAX_CONTAINER_TYPE_NAMELEN, "node_%d", tsk->pid);
@@ -1847,6 +2283,8 @@ int container_clone(struct task_struct *
  /* Pin the hierarchy */
  atomic_inc(&parent->root->sb->s_active);

+ /* Keep the container alive */
+ get_css_group(cg);
  mutex_unlock(&container_mutex);

  /* Now do the VFS work to create a container */
@@ -1890,6 +2328,7 @@ int container_clone(struct task_struct *
      (parent != task_container(tsk, subsys->subsys_id))) {
   /* Aargh, we raced ... */
   mutex_unlock(&inode->i_mutex);
+  put_css_group(cg);

   deactivate_super(parent->root->sb);
   /* The container is still accessible in the VFS, but
@@ -1913,6 +2352,7 @@ int container_clone(struct task_struct *

  out_release:
  mutex_unlock(&inode->i_mutex);
+ put_css_group(cg);
  deactivate_super(parent->root->sb);
  return ret;
 }
Index: container-2.6.22-rc6-mm1/kernel/fork.c
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/kernel/fork.c
+++ container-2.6.22-rc6-mm1/kernel/fork.c
@@ -1288,6 +1288,7 @@ static struct task_struct *copy_process(
   put_user(p->pid, parent_tidptr);

  proc_fork_connector(p);
+ container_post_fork(p);
  return p;

 bad_fork_cleanup_namespaces:

--
```

Subject: [PATCH 07/10] Task Containers(V11): Automatic userspace notification of idle containers

Posted by Paul Menage on Fri, 20 Jul 2007 18:31:59 GMT

View Forum Message <> Reply to Message

This patch adds the following files to the container filesystem:

notify_on_release - configures/reports whether the container subsystem should
attempt to run a release script when this container becomes unused

release_agent - configures/reports the release agent to be used for
this hierarchy (top level in each hierarchy only)

releasable - reports whether this container would have been auto-released if
notify_on_release was true and a release agent was configured (mainly useful
for debugging)

To avoid locking issues, invoking the userspace release agent is done via a
workqueue task; containers that need to have their release agents invoked by
the workqueue task are linked on to a list.

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

 include/linux/container.h | 11 -
 kernel/container.c | 425 ++++++++++++++++++++++++++++++++++++++-----
 2 files changed, 393 insertions(+), 43 deletions(-)

Index: container-2.6.22-rc6-mm1/include/linux/container.h
 ========== =======
--- container-2.6.22-rc6-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc6-mm1/include/linux/container.h
@@ -77,10 +77,11 @@ static inline void css_get(struct contai
  * css_get()
  */

+extern void __css_put(struct container_subsys_state *css);
 static inline void css_put(struct container_subsys_state *css)
 {
  if (!test_bit(CSS_ROOT, &css->flags))
- atomic_dec(&css->refcnt);
+ __css_put(css);
 }

 struct container {
@@ -112,6 +113,13 @@ struct container {
   * tasks in this container. Protected by css_group_lock
   */
  struct list_head css_groups;
+
+ /*
+ * Linked list running through all containers that can
+ * potentially be reaped by the release agent. Protected by
+ * release_list_lock

```
+  */
+ struct list_head release_list;
 };

 /* A css_group is a structure holding pointers to a set of
@@ -285,7 +293,6 @@ struct task_struct *container_iter_next(
      struct container_iter *it);
 void container_iter_end(struct container *cont, struct container_iter *it);

-
 #else /* !CONFIG_CONTAINERS */

 static inline int container_init_early(void) { return 0; }
Index: container-2.6.22-rc6-mm1/kernel/container.c
 ===================================================================== =======
--- container-2.6.22-rc6-mm1.orig/kernel/container.c
+++ container-2.6.22-rc6-mm1/kernel/container.c
@@ -44,6 +44,8 @@
 #include <linux/sort.h>
 #include <asm/atomic.h>

+static DEFINE_MUTEX(container_mutex);
+
 /* Generate an array of container subsystem pointers */
 #define SUBSYS(_x) &_x ## _subsys,

@@ -82,6 +84,13 @@ struct containerfs_root {

   /* Hierarchy-specific flags */
   unsigned long flags;
+
+ /* The path to use for release notifications. No locking
+  * between setting and use - so if userspace updates this
+  * while subcontainers exist, you could miss a
+  * notification. We ensure that it's always a valid
+  * NUL-terminated string */
+ char release_agent_path[PATH_MAX];
 };


@@ -109,7 +118,13 @@ static int need_forkexit_callback;

 /* bits in struct container flags field */
 enum {
+ /* Container is dead */
   CONT_REMOVED,
+ /* Container has previously had a child container or a task,
+  * but no longer (only if CONT_NOTIFY_ON_RELEASE is set) */
```

```
+ CONT_RELEASABLE,
+ /* Container requires release notifications to userspace */
+ CONT_NOTIFY_ON_RELEASE,
 };

 /* convenient tests for these bits */
@@ -123,6 +138,19 @@ enum {
  ROOT_NOPREFIX, /* mounted subsystems have no named prefix */
 };

+inline int container_is_releasable(const struct container *cont)
+{
+ const int bits =
+  (1 << CONT_RELEASABLE) |
+  (1 << CONT_NOTIFY_ON_RELEASE);
+ return (cont->flags & bits) == bits;
+}
+
+inline int notify_on_release(const struct container *cont)
+{
+ return test_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+}
+
 /*
  * for_each_subsys() allows you to iterate on each subsystem attached to
  * an active hierarchy
@@ -134,6 +162,14 @@ list_for_each_entry(_ss, &_root->subsys_
 #define for_each_root(_root) \
 list_for_each_entry(_root, &roots, root_list)

+/* the list of containers eligible for automatic release. Protected by
+ * release_list_lock */
+static LIST_HEAD(release_list);
+static DEFINE_SPINLOCK(release_list_lock);
+static void container_release_agent(struct work_struct *work);
+static DECLARE_WORK(release_agent_work, container_release_agent);
+static void check_for_release(struct container *cont);
+
 /* Link structure for associating css_group objects with containers */
 struct cg_container_link {
  /*
@@ -188,11 +224,8 @@ static int use_task_css_group_links;
 /*
  * unlink a css_group from the list and free it
  */
-static void release_css_group(struct kref *k)
+static void unlink_css_group(struct css_group *cg)
 {
```

```
- struct css_group *cg = container_of(k, struct css_group, ref);
- int i;
-
  write_lock(&css_group_lock);
  list_del(&cg->list);
  css_group_count--;
@@ -205,11 +238,39 @@ static void release_css_group(struct kre
   kfree(link);
  }
  write_unlock(&css_group_lock);
- for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++)
-  atomic_dec(&cg->subsys[i]->container->count);
+}
+
+static void __release_css_group(struct kref *k, int taskexit)
+{
+ int i;
+ struct css_group *cg = container_of(k, struct css_group, ref);
+
+ unlink_css_group(cg);
+
+ rcu_read_lock();
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  struct container *cont = cg->subsys[i]->container;
+  if (atomic_dec_and_test(&cont->count) &&
+      notify_on_release(cont)) {
+   if (taskexit)
+    set_bit(CONT_RELEASABLE, &cont->flags);
+   check_for_release(cont);
+  }
+ }
+ rcu_read_unlock();
  kfree(cg);
 }

+static void release_css_group(struct kref *k)
+{
+ __release_css_group(k, 0);
+}
+
+static void release_css_group_taskexit(struct kref *k)
+{
+ __release_css_group(k, 1);
+}
+
 /*
  * refcounted get/put for css_group objects
  */
```

```
@@ -223,6 +284,11 @@ static inline void put_css_group(struct
  kref_put(&cg->ref, release_css_group);
 }

+static inline void put_css_group_taskexit(struct css_group *cg)
+{
+ kref_put(&cg->ref, release_css_group_taskexit);
+}
+
 /*
  * find_existing_css_group() is a helper for
  * find_css_group(), and checks to see whether an existing
@@ -464,8 +530,6 @@ static struct css_group *find_css_group(
  * update of a tasks container pointer by attach_task()
  */

-static DEFINE_MUTEX(container_mutex);
-
 /**
  * container_lock - lock out any changes to container structures
  *
@@ -524,6 +588,13 @@ static void container_diput(struct dentr
  if (S_ISDIR(inode->i_mode)) {
   struct container *cont = dentry->d_fsdata;
   BUG_ON(!(container_is_removed(cont)));
+  /* It's possible for external users to be holding css
+   * reference counts on a container; css_put() needs to
+   * be able to access the container after decrementing
+   * the reference count in order to know if it needs to
+   * queue the container to be handled by the release
+   * agent */
+  synchronize_rcu();
   kfree(cont);
  }
  iput(inode);
@@ -668,6 +739,8 @@ static int container_show_options(struct
   seq_printf(seq, ",%s", ss->name);
  if (test_bit(ROOT_NOPREFIX, &root->flags))
   seq_puts(seq, ",noprefix");
+ if (strlen(root->release_agent_path))
+  seq_printf(seq, ",release_agent=%s", root->release_agent_path);
  mutex_unlock(&container_mutex);
  return 0;
 }
@@ -675,6 +748,7 @@ static int container_show_options(struct
 struct container_sb_opts {
  unsigned long subsys_bits;
  unsigned long flags;
```

```
+ char *release_agent;
 };

 /* Convert a hierarchy specifier into a bitmask of subsystems and
@@ -686,6 +760,7 @@ static int parse_containerfs_options(cha

  opts->subsys_bits = 0;
  opts->flags = 0;
+ opts->release_agent = NULL;

  while ((token = strsep(&o, ",")) != NULL) {
   if (!*token)
@@ -694,6 +769,15 @@ static int parse_containerfs_options(cha
    opts->subsys_bits = (1 << CONTAINER_SUBSYS_COUNT) - 1;
   } else if (!strcmp(token, "noprefix")) {
    set_bit(ROOT_NOPREFIX, &opts->flags);
+  } else if (!strncmp(token, "release_agent=", 14)) {
+   /* Specifying two release agents is forbidden */
+   if (opts->release_agent)
+    return -EINVAL;
+   opts->release_agent = kzalloc(PATH_MAX, GFP_KERNEL);
+   if (!opts->release_agent)
+    return -ENOMEM;
+   strncpy(opts->release_agent, token + 14, PATH_MAX - 1);
+   opts->release_agent[PATH_MAX - 1] = 0;
   } else {
    struct container_subsys *ss;
    int i;
@@ -743,7 +827,11 @@ static int container_remount(struct supe
  if (!ret)
   container_populate_dir(cont);

+ if (opts.release_agent)
+  strcpy(root->release_agent_path, opts.release_agent);
  out_unlock:
+ if (opts.release_agent)
+  kfree(opts.release_agent);
  mutex_unlock(&container_mutex);
  mutex_unlock(&cont->dentry->d_inode->i_mutex);
  return ret;
@@ -767,6 +855,7 @@ static void init_container_root(struct c
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
  INIT_LIST_HEAD(&cont->css_groups);
+ INIT_LIST_HEAD(&cont->release_list);
 }

 static int container_test_super(struct super_block *sb, void *data)
```

```
@@ -841,8 +930,11 @@ static int container_get_sb(struct file_

  /* First find the desired set of subsystems */
  ret = parse_containerfs_options(data, &opts);
- if (ret)
+ if (ret) {
+  if (opts.release_agent)
+   kfree(opts.release_agent);
   return ret;
+ }

  root = kzalloc(sizeof(*root), GFP_KERNEL);
  if (!root)
@@ -851,6 +943,10 @@ static int container_get_sb(struct file_
  init_container_root(root);
  root->subsys_bits = opts.subsys_bits;
  root->flags = opts.flags;
+ if (opts.release_agent) {
+  strcpy(root->release_agent_path, opts.release_agent);
+  kfree(opts.release_agent);
+ }

  sb = sget(fs_type, container_test_super, container_set_super, root);

@@ -1125,7 +1221,7 @@ static int attach_task(struct container
    ss->attach(ss, cont, oldcont, tsk);
   }
  }
-
+ set_bit(CONT_RELEASABLE, &oldcont->flags);
  synchronize_rcu();
  put_css_group(cg);
  return 0;
@@ -1175,6 +1271,9 @@ enum container_filetype {
  FILE_ROOT,
  FILE_DIR,
  FILE_TASKLIST,
+ FILE_NOTIFY_ON_RELEASE,
+ FILE_RELEASABLE,
+ FILE_RELEASE_AGENT,
 };

 static ssize_t container_common_file_write(struct container *cont,
@@ -1212,6 +1311,32 @@ static ssize_t container_common_file_wri
  case FILE_TASKLIST:
   retval = attach_task_by_pid(cont, buffer);
   break;
+ case FILE_NOTIFY_ON_RELEASE:
```

```
+  clear_bit(CONT_RELEASABLE, &cont->flags);
+  if (simple_strtoul(buffer, NULL, 10) != 0)
+   set_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+  else
+   clear_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+  break;
+ case FILE_RELEASE_AGENT:
+ {
+  struct containerfs_root *root = cont->root;
+  /* Strip trailing newline */
+  if (nbytes && (buffer[nbytes-1] == '\n')) {
+   buffer[nbytes-1] = 0;
+  }
+  if (nbytes < sizeof(root->release_agent_path)) {
+   /* We never write anything other than '\0'
+    * into the last char of release_agent_path,
+    * so it always remains a NUL-terminated
+    * string */
+   strncpy(root->release_agent_path, buffer, nbytes);
+   root->release_agent_path[nbytes] = 0;
+  } else {
+   retval = -ENOSPC;
+  }
+  break;
+ }
  default:
   retval = -EINVAL;
   goto out2;
@@ -1252,6 +1377,49 @@ static ssize_t container_read_uint(struc
  return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
 }

+static ssize_t container_common_file_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ enum container_filetype type = cft->private;
+ char *page;
+ ssize_t retval = 0;
+ char *s;
+
+ if (!(page = (char *)__get_free_page(GFP_KERNEL)))
+  return -ENOMEM;
+
+ s = page;
+
```

```
+ switch (type) {
+ case FILE_RELEASE_AGENT:
+ {
+  struct containerfs_root *root;
+  size_t n;
+  mutex_lock(&container_mutex);
+  root = cont->root;
+  n = strnlen(root->release_agent_path,
+       sizeof(root->release_agent_path));
+  n = min(n, (size_t) PAGE_SIZE);
+  strncpy(s, root->release_agent_path, n);
+  mutex_unlock(&container_mutex);
+  s += n;
+  break;
+ }
+ default:
+  retval = -EINVAL;
+  goto out;
+ }
+ *s++ = '\n';
+
+ retval = simple_read_from_buffer(buf, nbytes, ppos, page, s - page);
+out:
+ free_page((unsigned long)page);
+ return retval;
+}
+
 static ssize_t container_file_read(struct file *file, char __user *buf,
        size_t nbytes, loff_t *ppos)
 {
@@ -1667,16 +1835,49 @@ static int container_tasks_release(struc
  return 0;
 }

+static u64 container_read_notify_on_release(struct container *cont,
+         struct cftype *cft)
+{
+ return notify_on_release(cont);
+}
+
+static u64 container_read_releasable(struct container *cont, struct cftype *cft)
+{
+ return test_bit(CONT_RELEASABLE, &cont->flags);
+}
+
 /*
  * 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,
+static struct cftype files[] = {
+ {
+  .name = "tasks",
+  .open = container_tasks_open,
+  .read = container_tasks_read,
+  .write = container_common_file_write,
+  .release = container_tasks_release,
+  .private = FILE_TASKLIST,
+ },
+
+ {
+  .name = "notify_on_release",
+  .read_uint = container_read_notify_on_release,
+  .write = container_common_file_write,
+  .private = FILE_NOTIFY_ON_RELEASE,
+ },
+
+ {
+  .name = "releasable",
+  .read_uint = container_read_releasable,
+  .private = FILE_RELEASABLE,
+ }
+};
+
+static struct cftype cft_release_agent = {
+ .name = "release_agent",
+ .read = container_common_file_read,
  .write = container_common_file_write,
- .release = container_tasks_release,
- .private = FILE_TASKLIST,
+ .private = FILE_RELEASE_AGENT,
 };

 static int container_populate_dir(struct container *cont)
@@ -1687,10 +1888,15 @@ static int container_populate_dir(struct
  /* First clear out any existing files */
  container_clear_directory(cont->dentry);

- err = container_add_file(cont, NULL, &cft_tasks);
+ err = container_add_files(cont, NULL, files, ARRAY_SIZE(files));
  if (err < 0)
   return err;

+ if (cont == cont->top_container) {
```

```
+  if ((err = container_add_file(cont, NULL, &cft_release_agent)) < 0)
+   return err;
+ }
+
  for_each_subsys(cont->root, ss) {
   if (ss->populate && (err = ss->populate(ss, cont)) < 0)
    return err;
@@ -1747,6 +1953,7 @@ static long container_create(struct cont
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
  INIT_LIST_HEAD(&cont->css_groups);
+ INIT_LIST_HEAD(&cont->release_list);

  cont->parent = parent;
  cont->root = parent->root;
@@ -1808,6 +2015,38 @@ static int container_mkdir(struct inode
  return container_create(c_parent, dentry, mode | S_IFDIR);
 }

+static inline int container_has_css_refs(struct container *cont)
+{
+ /* Check the reference count on each subsystem. Since we
+  * already established that there are no tasks in the
+  * container, if the css refcount is also 0, then there should
+  * be no outstanding references, so the subsystem is safe to
+  * destroy. We scan across all subsystems rather than using
+  * the per-hierarchy linked list of mounted subsystems since
+  * we can be called via check_for_release() with no
+  * synchronization other than RCU, and the subsystem linked
+  * list isn't RCU-safe */
+ int i;
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  struct container_subsys *ss = subsys[i];
+  struct container_subsys_state *css;
+  /* Skip subsystems not in this hierarchy */
+  if (ss->root != cont->root)
+   continue;
+  css = cont->subsys[ss->subsys_id];
+  /* When called from check_for_release() it's possible
+   * that by this point the container has been removed
+   * and the css deleted. But a false-positive doesn't
+   * matter, since it can only happen if the container
+   * has been deleted and hence no longer needs the
+   * release agent to be called anyway. */
+  if (css && atomic_read(&css->refcnt)) {
+   return 1;
+  }
+ }
```

```
+ return 0;
+}
+
 static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)
 {
  struct container *cont = dentry->d_fsdata;
@@ -1816,7 +2055,6 @@ static int container_rmdir(struct inode
  struct container_subsys *ss;
  struct super_block *sb;
  struct containerfs_root *root;
- int css_busy = 0;

  /* the vfs holds both inode->i_mutex already */

@@ -1834,20 +2072,7 @@ static int container_rmdir(struct inode
  root = cont->root;
  sb = root->sb;

- /* Check the reference count on each subsystem. Since we
-  * already established that there are no tasks in the
-  * container, if the css refcount is also 0, then there should
-  * be no outstanding references, so the subsystem is safe to
-  * destroy */
- for_each_subsys(root, ss) {
-  struct container_subsys_state *css;
-  css = cont->subsys[ss->subsys_id];
-  if (atomic_read(&css->refcnt)) {
-   css_busy = 1;
-   break;
-  }
- }
- if (css_busy) {
+ if (container_has_css_refs(cont)) {
   mutex_unlock(&container_mutex);
   return -EBUSY;
  }
@@ -1857,7 +2082,11 @@ static int container_rmdir(struct inode
    ss->destroy(ss, cont);
  }

+ spin_lock(&release_list_lock);
  set_bit(CONT_REMOVED, &cont->flags);
+ if (!list_empty(&cont->release_list))
+  list_del(&cont->release_list);
+ spin_unlock(&release_list_lock);
  /* delete my sibling from parent->children */
  list_del(&cont->sibling);
  spin_lock(&cont->dentry->d_lock);
```

```
@@ -1869,6 +2098,9 @@ static int container_rmdir(struct inode
  dput(d);
  root->number_of_containers--;

+ set_bit(CONT_RELEASABLE, &parent->flags);
+ check_for_release(parent);
+
  mutex_unlock(&container_mutex);
  /* Drop the active superblock reference that we took when we
   * created the container */
@@ -1906,15 +2138,15 @@ static void container_init_subsys(struct
   /* If this subsystem requested that it be notified with fork
    * events, we should send it one now for every process in the
    * system */
-  if (ss->fork) {
-   struct task_struct *g, *p;
+ if (ss->fork) {
+  struct task_struct *g, *p;

-   read_lock(&tasklist_lock);
-   do_each_thread(g, p) {
-    ss->fork(ss, p);
-   } while_each_thread(g, p);
-   read_unlock(&tasklist_lock);
-  }
+  read_lock(&tasklist_lock);
+  do_each_thread(g, p) {
+   ss->fork(ss, p);
+  } while_each_thread(g, p);
+  read_unlock(&tasklist_lock);
+ }

  need_forkexit_callback |= ss->fork || ss->exit;

@@ -2241,7 +2473,7 @@ void container_exit(struct task_struct *
  tsk->containers = &init_css_group;
  task_unlock(tsk);
  if (cg)
-  put_css_group(cg);
+  put_css_group_taskexit(cg);
 }

 /**
@@ -2352,7 +2584,10 @@ int container_clone(struct task_struct *

  out_release:
  mutex_unlock(&inode->i_mutex);
+
```

```
+ mutex_lock(&container_mutex);
  put_css_group(cg);
+ mutex_unlock(&container_mutex);
  deactivate_super(parent->root->sb);
  return ret;
 }
@@ -2382,3 +2617,111 @@ int container_is_descendant(const struct
  ret = (cont == target);
  return ret;
 }
+
+static void check_for_release(struct container *cont)
+{
+ /* All of these checks rely on RCU to keep the container
+  * structure alive */
+ if (container_is_releasable(cont) && !atomic_read(&cont->count)
+     && list_empty(&cont->children) && !container_has_css_refs(cont)) {
+  /* Container is currently removeable. If it's not
+   * already queued for a userspace notification, queue
+   * it now */
+  int need_schedule_work = 0;
+  spin_lock(&release_list_lock);
+  if (!container_is_removed(cont) &&
+      list_empty(&cont->release_list)) {
+   list_add(&cont->release_list, &release_list);
+   need_schedule_work = 1;
+  }
+  spin_unlock(&release_list_lock);
+  if (need_schedule_work)
+   schedule_work(&release_agent_work);
+ }
+}
+
+void __css_put(struct container_subsys_state *css)
+{
+ struct container *cont = css->container;
+ rcu_read_lock();
+ if (atomic_dec_and_test(&css->refcnt) && notify_on_release(cont)) {
+  set_bit(CONT_RELEASABLE, &cont->flags);
+  check_for_release(cont);
+ }
+ rcu_read_unlock();
+}
+
+/*
+ * Notify userspace when a container is released, by running the
+ * configured release agent with the name of the container (path
+ * relative to the root of container file system) as the argument.
```

```
+ *
+ * Most likely, this user command will try to rmdir this container.
+ *
+ * This races with the possibility that some other task will be
+ * attached to this container before it is removed, or that some other
+ * user task will 'mkdir' a child container of this container.  That's ok.
+ * The presumed 'rmdir' will fail quietly if this container is no longer
+ * unused, and this container will be reprieved from its death sentence,
+ * to continue to serve a useful existence.  Next time it's released,
+ * we will get notified again, if it still has 'notify_on_release' set.
+ *
+ * The final arg to call_usermodehelper() is UMH_WAIT_EXEC, which
+ * means only wait until the task is successfully execve()'d.  The
+ * separate release agent task is forked by call_usermodehelper(),
+ * then control in this thread returns here, without waiting for the
+ * release agent task.  We don't bother to wait because the caller of
+ * this routine has no use for the exit status of the release agent
+ * task, so no sense holding our caller up for that.
+ *
+ */
+
+static void container_release_agent(struct work_struct *work)
+{
+ BUG_ON(work != &release_agent_work);
+ mutex_lock(&container_mutex);
+ spin_lock(&release_list_lock);
+ while (!list_empty(&release_list)) {
+  char *argv[3], *envp[3];
+  int i;
+  char *pathbuf;
+  struct container *cont = list_entry(release_list.next,
+         struct container,
+         release_list);
+  list_del_init(&cont->release_list);
+  spin_unlock(&release_list_lock);
+  pathbuf = kmalloc(PAGE_SIZE, GFP_KERNEL);
+  if (!pathbuf) {
+   spin_lock(&release_list_lock);
+   continue;
+  }
+
+  if (container_path(cont, pathbuf, PAGE_SIZE) < 0) {
+   kfree(pathbuf);
+   spin_lock(&release_list_lock);
+   continue;
+  }
+
+  i = 0;
```

```
+  argv[i++] = cont->root->release_agent_path;
+  argv[i++] = (char *)pathbuf;
+  argv[i] = NULL;
+
+  i = 0;
+  /* minimal command environment */
+  envp[i++] = "HOME=/";
+  envp[i++] = "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
+  envp[i] = NULL;
+
+  /* Drop the lock while we invoke the usermode helper,
+   * since the exec could involve hitting disk and hence
+   * be a slow process */
+  mutex_unlock(&container_mutex);
+  call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
+  kfree(pathbuf);
+  mutex_lock(&container_mutex);
+  spin_lock(&release_list_lock);
+ }
+ spin_unlock(&release_list_lock);
+ mutex_unlock(&container_mutex);
+}

--
```

Subject: [PATCH 10/10] Task Containers(V11): Simple task container debug info subsystem
Posted by Paul Menage on Fri, 20 Jul 2007 18:32:02 GMT
View Forum Message <> Reply to Message

This example subsystem exports debugging information as an aid to diagnosing refcount leaks, etc, in the container framework.

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

```
 include/linux/container_subsys.h |    4 +
 init/Kconfig                     |   10 ++++
 kernel/Makefile                  |    1
 kernel/container_debug.c         |   97 +++++++++++++++++++++++++++++++++++++++++
 4 files changed, 112 insertions(+)

Index: container-2.6.22-rc6-mm1/include/linux/container_subsys.h
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/include/linux/container_subsys .h
+++ container-2.6.22-rc6-mm1/include/linux/container_subsys.h
@@ -19,4 +19,8 @@ SUBSYS(cpuacct)
```

```
 /* */

+#ifdef CONFIG_CONTAINER_DEBUG
+SUBSYS(debug)
+#endif
+
 /* */
Index: container-2.6.22-rc6-mm1/init/Kconfig
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/init/Kconfig
+++ container-2.6.22-rc6-mm1/init/Kconfig
@@ -303,6 +303,16 @@ config CONTAINERS

    Say N if unsure.

+config CONTAINER_DEBUG
+ bool "Example debug container subsystem"
+ depends on CONTAINERS
+ help
+   This option enables a simple container subsystem that
+   exports useful debugging information about the containers
+   framework
+
+   Say N if unsure
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP && CONTAINERS
Index: container-2.6.22-rc6-mm1/kernel/Makefile
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/kernel/Makefile
+++ container-2.6.22-rc6-mm1/kernel/Makefile
@@ -38,6 +38,7 @@ obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CONTAINERS) += container.o
+obj-$(CONFIG_CONTAINER_DEBUG) += container_debug.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o
Index: container-2.6.22-rc6-mm1/kernel/container_debug.c
 ===================================================================
--- /dev/null
+++ container-2.6.22-rc6-mm1/kernel/container_debug.c
@@ -0,0 +1,97 @@
+/*
+ * kernel/ccontainer_debug.c - Example container subsystem that
```

```
+ * exposes debug info
+ *
+ * Copyright (C) Google Inc, 2007
+ *
+ * Developed by Paul Menage (menage@google.com)
+ *
+ */
+
+#include <linux/container.h>
+#include <linux/fs.h>
+#include <linux/slab.h>
+#include <linux/rcupdate.h>
+
+#include <asm/atomic.h>
+
+static struct container_subsys_state *debug_create(struct container_subsys *ss,
+         struct container *cont)
+{
+ struct container_subsys_state *css = kzalloc(sizeof(*css), GFP_KERNEL);
+
+ if (!css)
+  return ERR_PTR(-ENOMEM);
+
+ return css;
+}
+
+static void debug_destroy(struct container_subsys *ss, struct container *cont)
+{
+ kfree(cont->subsys[debug_subsys_id]);
+}
+
+static u64 container_refcount_read(struct container *cont, struct cftype *cft)
+{
+ return atomic_read(&cont->count);
+}
+
+static u64 taskcount_read(struct container *cont, struct cftype *cft)
+{
+ u64 count;
+
+ container_lock();
+ count = container_task_count(cont);
+ container_unlock();
+ return count;
+}
+
+static u64 current_css_group_read(struct container *cont, struct cftype *cft)
+{
```

```
+ return (u64)(long)current->containers;
+}
+
+static u64 current_css_group_refcount_read(struct container *cont,
+        struct cftype *cft)
+{
+ u64 count;
+
+ rcu_read_lock();
+ count = atomic_read(&current->containers->ref.refcount);
+ rcu_read_unlock();
+ return count;
+}
+
+static struct cftype files[] =  {
+ {
+  .name = "container_refcount",
+  .read_uint = container_refcount_read,
+ },
+ {
+  .name = "taskcount",
+  .read_uint = taskcount_read,
+ },
+
+ {
+  .name = "current_css_group",
+  .read_uint = current_css_group_read,
+ },
+
+ {
+  .name = "current_css_group_refcount",
+  .read_uint = current_css_group_refcount_read,
+ },
+};
+
+static int debug_populate(struct container_subsys *ss, struct container *cont)
+{
+ return container_add_files(cont, ss, files, ARRAY_SIZE(files));
+}
+
+struct container_subsys debug_subsys = {
+ .name = "debug",
+ .create = debug_create,
+ .destroy = debug_destroy,
+ .populate = debug_populate,
+ .subsys_id = debug_subsys_id,
+};
```

--

Subject: Re: [PATCH 02/10] Task Containers(V11): Add tasks file interface
Posted by James Morris on Fri, 20 Jul 2007 20:20:10 GMT
View Forum Message <> Reply to Message

On Fri, 20 Jul 2007, menage@google.com wrote:

```
> +/*
> + * 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;
```

I wonder if we should allow CAP_SYS_ADMIN to do this, too.

--
James Morris
<jmorris@namei.org>

Subject: Re: [PATCH 02/10] Task Containers(V11): Add tasks file interface
Posted by Paul Menage on Fri, 20 Jul 2007 21:25:41 GMT
View Forum Message <> Reply to Message

On 7/20/07, James Morris <jmorris@namei.org> wrote:
> > + if ((current->euid) && (current->euid != tsk->uid)
> > + && (current->euid != tsk->suid)) {
> > + put_task_struct(tsk);
> > + return -EACCES;
>
> I wonder if we should allow CAP_SYS_ADMIN to do this, too.
>

Yes, I ought to replace "current->euid" with "capable(CAP_SYS_ADMIN)".

Thanks,

Paul

Subject: Re: [PATCH 07/10] Task Containers(V11): Automatic userspace notification of idle containers
Posted by serue on Mon, 23 Jul 2007 17:41:02 GMT
View Forum Message <> Reply to Message

Quoting menage@google.com (menage@google.com):
> This patch adds the following files to the container filesystem:
>
> notify_on_release - configures/reports whether the container subsystem should
> attempt to run a release script when this container becomes unused
>
> release_agent - configures/reports the release agent to be used for
> this hierarchy (top level in each hierarchy only)
>
> releasable - reports whether this container would have been auto-released if
> notify_on_release was true and a release agent was configured (mainly useful
> for debugging)
>
> To avoid locking issues, invoking the userspace release agent is done via a
> workqueue task; containers that need to have their release agents invoked by
> the workqueue task are linked on to a list.

Hi Paul,

my tree may be a bit crufty, but I had to #include <linux/kmod.h> in
order for this to compile on s390.

thanks,
-serge

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

> ---
>
> include/linux/container.h | 11 -
> kernel/container.c | 425 ++-----
> 2 files changed, 393 insertions(+), 43 deletions(-)
>
> Index: container-2.6.22-rc6-mm1/include/linux/container.h
> ========== =======
> --- container-2.6.22-rc6-mm1.orig/include/linux/container.h
> +++ container-2.6.22-rc6-mm1/include/linux/container.h
> @@ -77,10 +77,11 @@ static inline void css_get(struct contai
>   * css_get()
>   */
>
> +extern void __css_put(struct container_subsys_state *css);
> static inline void css_put(struct container_subsys_state *css)
> {
>   if (!test_bit(CSS_ROOT, &css->flags))
> - atomic_dec(&css->refcnt);
> + __css_put(css);
> }
>
> struct container {
> @@ -112,6 +113,13 @@ struct container {
>   * tasks in this container. Protected by css_group_lock
>   */
>   struct list_head css_groups;
> +
> + /*
> + * Linked list running through all containers that can
> + * potentially be reaped by the release agent. Protected by
> + * release_list_lock
> + */
> + struct list_head release_list;
> };
>
> /* A css_group is a structure holding pointers to a set of
> @@ -285,7 +293,6 @@ struct task_struct *container_iter_next(
>   struct container_iter *it);
> void container_iter_end(struct container *cont, struct container_iter *it);
>
> -
> #else /* !CONFIG_CONTAINERS */
>
> static inline int container_init_early(void) { return 0; }
> Index: container-2.6.22-rc6-mm1/kernel/container.c
> ========== =======
> --- container-2.6.22-rc6-mm1.orig/kernel/container.c

```
> +++ container-2.6.22-rc6-mm1/kernel/container.c
> @@ -44,6 +44,8 @@
>  #include <linux/sort.h>
>  #include <asm/atomic.h>
>
> +static DEFINE_MUTEX(container_mutex);
> +
>  /* Generate an array of container subsystem pointers */
>  #define SUBSYS(_x) &_x ## _subsys,
>
> @@ -82,6 +84,13 @@ struct containerfs_root {
>
>   /* Hierarchy-specific flags */
>   unsigned long flags;
> +
> + /* The path to use for release notifications. No locking
> +  * between setting and use - so if userspace updates this
> +  * while subcontainers exist, you could miss a
> +  * notification. We ensure that it's always a valid
> +  * NUL-terminated string */
> + char release_agent_path[PATH_MAX];
>  };
>
>
> @@ -109,7 +118,13 @@ static int need_forkexit_callback;
>
>  /* bits in struct container flags field */
>  enum {
> + /* Container is dead */
>   CONT_REMOVED,
> + /* Container has previously had a child container or a task,
> +  * but no longer (only if CONT_NOTIFY_ON_RELEASE is set) */
> + CONT_RELEASABLE,
> + /* Container requires release notifications to userspace */
> + CONT_NOTIFY_ON_RELEASE,
>  };
>
>  /* convenient tests for these bits */
> @@ -123,6 +138,19 @@ enum {
>   ROOT_NOPREFIX, /* mounted subsystems have no named prefix */
>  };
>
> +inline int container_is_releasable(const struct container *cont)
> +{
> + const int bits =
> +  (1 << CONT_RELEASABLE) |
> +  (1 << CONT_NOTIFY_ON_RELEASE);
> + return (cont->flags & bits) == bits;
```

> +}
> +
> +inline int notify_on_release(const struct container *cont)
> +{
> + return test_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
> +}
> +
> /*
>   * for_each_subsys() allows you to iterate on each subsystem attached to
>   * an active hierarchy
> @@ -134,6 +162,14 @@ list_for_each_entry(_ss, &_root->subsys_
> #define for_each_root(_root) \
> list_for_each_entry(_root, &roots, root_list)
>
> +/* the list of containers eligible for automatic release. Protected by
> + * release_list_lock */
> +static LIST_HEAD(release_list);
> +static DEFINE_SPINLOCK(release_list_lock);
> +static void container_release_agent(struct work_struct *work);
> +static DECLARE_WORK(release_agent_work, container_release_agent);
> +static void check_for_release(struct container *cont);
> +
> /* Link structure for associating css_group objects with containers */
> struct cg_container_link {
>   /*
> @@ -188,11 +224,8 @@ static int use_task_css_group_links;
> /*
>   * unlink a css_group from the list and free it
>   */
> -static void release_css_group(struct kref *k)
> +static void unlink_css_group(struct css_group *cg)
> {
> - struct css_group *cg = container_of(k, struct css_group, ref);
> - int i;
> -
>   write_lock(&css_group_lock);
>   list_del(&cg->list);
>   css_group_count--;
> @@ -205,11 +238,39 @@ static void release_css_group(struct kre
>   kfree(link);
>   }
>   write_unlock(&css_group_lock);
> - for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++)
> - atomic_dec(&cg->subsys[i]->container->count);
> +}
> +
> +static void __release_css_group(struct kref *k, int taskexit)
> +{

```
> + int i;
> + struct css_group *cg = container_of(k, struct css_group, ref);
> +
> + unlink_css_group(cg);
> +
> + rcu_read_lock();
> + for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
> +  struct container *cont = cg->subsys[i]->container;
> +  if (atomic_dec_and_test(&cont->count) &&
> +      notify_on_release(cont)) {
> +   if (taskexit)
> +    set_bit(CONT_RELEASABLE, &cont->flags);
> +   check_for_release(cont);
> +  }
> + }
> + rcu_read_unlock();
>   kfree(cg);
>  }
>
> +static void release_css_group(struct kref *k)
> +{
> + __release_css_group(k, 0);
> +}
> +
> +static void release_css_group_taskexit(struct kref *k)
> +{
> + __release_css_group(k, 1);
> +}
> +
>  /*
>   * refcounted get/put for css_group objects
>   */
> @@ -223,6 +284,11 @@ static inline void put_css_group(struct
>   kref_put(&cg->ref, release_css_group);
>  }
>
> +static inline void put_css_group_taskexit(struct css_group *cg)
> +{
> + kref_put(&cg->ref, release_css_group_taskexit);
> +}
> +
>  /*
>   * find_existing_css_group() is a helper for
>   * find_css_group(), and checks to see whether an existing
> @@ -464,8 +530,6 @@ static struct css_group *find_css_group(
>   * update of a tasks container pointer by attach_task()
>   */
>
```

```
> -static DEFINE_MUTEX(container_mutex);
> -
>  /**
>   * container_lock - lock out any changes to container structures
>   *
> @@ -524,6 +588,13 @@ static void container_diput(struct dentr
>   if (S_ISDIR(inode->i_mode)) {
>    struct container *cont = dentry->d_fsdata;
>    BUG_ON(!(container_is_removed(cont)));
> +  /* It's possible for external users to be holding css
> +   * reference counts on a container; css_put() needs to
> +   * be able to access the container after decrementing
> +   * the reference count in order to know if it needs to
> +   * queue the container to be handled by the release
> +   * agent */
> +  synchronize_rcu();
>    kfree(cont);
>   }
>   iput(inode);
> @@ -668,6 +739,8 @@ static int container_show_options(struct
>    seq_printf(seq, ",%s", ss->name);
>   if (test_bit(ROOT_NOPREFIX, &root->flags))
>    seq_puts(seq, ",noprefix");
> + if (strlen(root->release_agent_path))
> +  seq_printf(seq, ",release_agent=%s", root->release_agent_path);
>   mutex_unlock(&container_mutex);
>   return 0;
>  }
> @@ -675,6 +748,7 @@ static int container_show_options(struct
>  struct container_sb_opts {
>   unsigned long subsys_bits;
>   unsigned long flags;
> + char *release_agent;
>  };
>
>  /* Convert a hierarchy specifier into a bitmask of subsystems and
> @@ -686,6 +760,7 @@ static int parse_containerfs_options(cha
>
>   opts->subsys_bits = 0;
>   opts->flags = 0;
> + opts->release_agent = NULL;
>
>   while ((token = strsep(&o, ",")) != NULL) {
>    if (!*token)
> @@ -694,6 +769,15 @@ static int parse_containerfs_options(cha
>     opts->subsys_bits = (1 << CONTAINER_SUBSYS_COUNT) - 1;
>    } else if (!strcmp(token, "noprefix")) {
>     set_bit(ROOT_NOPREFIX, &opts->flags);
```

```
> +  } else if (!strncmp(token, "release_agent=", 14)) {
> +   /* Specifying two release agents is forbidden */
> +   if (opts->release_agent)
> +    return -EINVAL;
> +   opts->release_agent = kzalloc(PATH_MAX, GFP_KERNEL);
> +   if (!opts->release_agent)
> +    return -ENOMEM;
> +   strncpy(opts->release_agent, token + 14, PATH_MAX - 1);
> +   opts->release_agent[PATH_MAX - 1] = 0;
>    } else {
>     struct container_subsys *ss;
>     int i;
> @@ -743,7 +827,11 @@ static int container_remount(struct supe
>   if (!ret)
>    container_populate_dir(cont);
>
> + if (opts.release_agent)
> +  strcpy(root->release_agent_path, opts.release_agent);
>   out_unlock:
> + if (opts.release_agent)
> +  kfree(opts.release_agent);
>   mutex_unlock(&container_mutex);
>   mutex_unlock(&cont->dentry->d_inode->i_mutex);
>   return ret;
> @@ -767,6 +855,7 @@ static void init_container_root(struct c
>   INIT_LIST_HEAD(&cont->sibling);
>   INIT_LIST_HEAD(&cont->children);
>   INIT_LIST_HEAD(&cont->css_groups);
> + INIT_LIST_HEAD(&cont->release_list);
>  }
>
>  static int container_test_super(struct super_block *sb, void *data)
> @@ -841,8 +930,11 @@ static int container_get_sb(struct file_
>
>   /* First find the desired set of subsystems */
>   ret = parse_containerfs_options(data, &opts);
> - if (ret)
> + if (ret) {
> +  if (opts.release_agent)
> +   kfree(opts.release_agent);
>    return ret;
> + }
>
>   root = kzalloc(sizeof(*root), GFP_KERNEL);
>   if (!root)
> @@ -851,6 +943,10 @@ static int container_get_sb(struct file_
>   init_container_root(root);
>   root->subsys_bits = opts.subsys_bits;
```

```
>   root->flags = opts.flags;
> + if (opts.release_agent) {
> +  strcpy(root->release_agent_path, opts.release_agent);
> +  kfree(opts.release_agent);
> + }
>
>   sb = sget(fs_type, container_test_super, container_set_super, root);
>
> @@ -1125,7 +1221,7 @@ static int attach_task(struct container
>     ss->attach(ss, cont, oldcont, tsk);
>    }
>   }
> -
> + set_bit(CONT_RELEASABLE, &oldcont->flags);
>   synchronize_rcu();
>   put_css_group(cg);
>   return 0;
> @@ -1175,6 +1271,9 @@ enum container_filetype {
>   FILE_ROOT,
>   FILE_DIR,
>   FILE_TASKLIST,
> + FILE_NOTIFY_ON_RELEASE,
> + FILE_RELEASABLE,
> + FILE_RELEASE_AGENT,
>  };
>
>  static ssize_t container_common_file_write(struct container *cont,
> @@ -1212,6 +1311,32 @@ static ssize_t container_common_file_wri
>   case FILE_TASKLIST:
>    retval = attach_task_by_pid(cont, buffer);
>    break;
> + case FILE_NOTIFY_ON_RELEASE:
> +  clear_bit(CONT_RELEASABLE, &cont->flags);
> +  if (simple_strtoul(buffer, NULL, 10) != 0)
> +   set_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
> +  else
> +   clear_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
> +  break;
> + case FILE_RELEASE_AGENT:
> + {
> +  struct containerfs_root *root = cont->root;
> +  /* Strip trailing newline */
> +  if (nbytes && (buffer[nbytes-1] == '\n')) {
> +   buffer[nbytes-1] = 0;
> +  }
> +  if (nbytes < sizeof(root->release_agent_path)) {
> +   /* We never write anything other than '\0'
> +    * into the last char of release_agent_path,
```

```
> +    * so it always remains a NUL-terminated
> +    * string */
> +   strncpy(root->release_agent_path, buffer, nbytes);
> +   root->release_agent_path[nbytes] = 0;
> +  } else {
> +   retval = -ENOSPC;
> +  }
> +  break;
> + }
>   default:
>    retval = -EINVAL;
>    goto out2;
> @@ -1252,6 +1377,49 @@ static ssize_t container_read_uint(struc
>   return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
>  }
>
> +static ssize_t container_common_file_read(struct container *cont,
> +       struct cftype *cft,
> +       struct file *file,
> +       char __user *buf,
> +       size_t nbytes, loff_t *ppos)
> +{
> + enum container_filetype type = cft->private;
> + char *page;
> + ssize_t retval = 0;
> + char *s;
> +
> + if (!(page = (char *)__get_free_page(GFP_KERNEL)))
> +  return -ENOMEM;
> +
> + s = page;
> +
> + switch (type) {
> + case FILE_RELEASE_AGENT:
> + {
> +  struct containerfs_root *root;
> +  size_t n;
> +  mutex_lock(&container_mutex);
> +  root = cont->root;
> +  n = strnlen(root->release_agent_path,
> +       sizeof(root->release_agent_path));
> +  n = min(n, (size_t) PAGE_SIZE);
> +  strncpy(s, root->release_agent_path, n);
> +  mutex_unlock(&container_mutex);
> +  s += n;
> +  break;
> + }
> + default:
```

> + retval = -EINVAL;
> + goto out;
> + }
> + *s++ = '\n';
> +
> + retval = simple_read_from_buffer(buf, nbytes, ppos, page, s - page);
> +out:
> + free_page((unsigned long)page);
> + return retval;
> +}
> +
> static ssize_t container_file_read(struct file *file, char __user *buf,
>   size_t nbytes, loff_t *ppos)
> {
> @@ -1667,16 +1835,49 @@ static int container_tasks_release(struc
>   return 0;
> }
>
> +static u64 container_read_notify_on_release(struct container *cont,
> +   struct cftype *cft)
> +{
> + return notify_on_release(cont);
> +}
> +
> +static u64 container_read_releasable(struct container *cont, struct cftype *cft)
> +{
> + return test_bit(CONT_RELEASABLE, &cont->flags);
> +}
> +
> /*
>   * 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,
> +static struct cftype files[] = {
> + {
> + .name = "tasks",
> + .open = container_tasks_open,
> + .read = container_tasks_read,
> + .write = container_common_file_write,
> + .release = container_tasks_release,
> + .private = FILE_TASKLIST,
> + },
> +
> + {
> + .name = "notify_on_release",

```
> +  .read_uint = container_read_notify_on_release,
> +  .write = container_common_file_write,
> +  .private = FILE_NOTIFY_ON_RELEASE,
> + },
> +
> + {
> +  .name = "releasable",
> +  .read_uint = container_read_releasable,
> +  .private = FILE_RELEASABLE,
> + }
> +};
> +
> +static struct cftype cft_release_agent = {
> + .name = "release_agent",
> + .read = container_common_file_read,
>    .write = container_common_file_write,
> - .release = container_tasks_release,
> - .private = FILE_TASKLIST,
> + .private = FILE_RELEASE_AGENT,
>   };
>
>   static int container_populate_dir(struct container *cont)
> @@ -1687,10 +1888,15 @@ static int container_populate_dir(struct
>    /* First clear out any existing files */
>    container_clear_directory(cont->dentry);
>
> - err = container_add_file(cont, NULL, &cft_tasks);
> + err = container_add_files(cont, NULL, files, ARRAY_SIZE(files));
>    if (err < 0)
>     return err;
>
> + if (cont == cont->top_container) {
> +  if ((err = container_add_file(cont, NULL, &cft_release_agent)) < 0)
> +   return err;
> + }
> +
>    for_each_subsys(cont->root, ss) {
>     if (ss->populate && (err = ss->populate(ss, cont)) < 0)
>      return err;
> @@ -1747,6 +1953,7 @@ static long container_create(struct cont
>    INIT_LIST_HEAD(&cont->sibling);
>    INIT_LIST_HEAD(&cont->children);
>    INIT_LIST_HEAD(&cont->css_groups);
> + INIT_LIST_HEAD(&cont->release_list);
>
>    cont->parent = parent;
>    cont->root = parent->root;
> @@ -1808,6 +2015,38 @@ static int container_mkdir(struct inode
```

> return container_create(c_parent, dentry, mode | S_IFDIR);
> }
>
> +static inline int container_has_css_refs(struct container *cont)
> +{
> + /* Check the reference count on each subsystem. Since we
> + * already established that there are no tasks in the
> + * container, if the css refcount is also 0, then there should
> + * be no outstanding references, so the subsystem is safe to
> + * destroy. We scan across all subsystems rather than using
> + * the per-hierarchy linked list of mounted subsystems since
> + * we can be called via check_for_release() with no
> + * synchronization other than RCU, and the subsystem linked
> + * list isn't RCU-safe */
> + int i;
> + for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
> + struct container_subsys *ss = subsys[i];
> + struct container_subsys_state *css;
> + /* Skip subsystems not in this hierarchy */
> + if (ss->root != cont->root)
> +   continue;
> + css = cont->subsys[ss->subsys_id];
> + /* When called from check_for_release() it's possible
> +   * that by this point the container has been removed
> +   * and the css deleted. But a false-positive doesn't
> +   * matter, since it can only happen if the container
> +   * has been deleted and hence no longer needs the
> +   * release agent to be called anyway. */
> + if (css && atomic_read(&css->refcnt)) {
> +   return 1;
> + }
> + }
> + return 0;
> +}
> +
> static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)
> {
>   struct container *cont = dentry->d_fsdata;
> @@ -1816,7 +2055,6 @@ static int container_rmdir(struct inode
>   struct container_subsys *ss;
>   struct super_block *sb;
>   struct containerfs_root *root;
> - int css_busy = 0;
>
>   /* the vfs holds both inode->i_mutex already */
>
> @@ -1834,20 +2072,7 @@ static int container_rmdir(struct inode
>   root = cont->root;

```
>   sb = root->sb;
>
> - /* Check the reference count on each subsystem. Since we
> -  * already established that there are no tasks in the
> -  * container, if the css refcount is also 0, then there should
> -  * be no outstanding references, so the subsystem is safe to
> -  * destroy */
> - for_each_subsys(root, ss) {
> -  struct container_subsys_state *css;
> -  css = cont->subsys[ss->subsys_id];
> -  if (atomic_read(&css->refcnt)) {
> -   css_busy = 1;
> -   break;
> -  }
> - }
> - if (css_busy) {
> + if (container_has_css_refs(cont)) {
>    mutex_unlock(&container_mutex);
>    return -EBUSY;
>   }
> @@ -1857,7 +2082,11 @@ static int container_rmdir(struct inode
>     ss->destroy(ss, cont);
>   }
>
> + spin_lock(&release_list_lock);
>   set_bit(CONT_REMOVED, &cont->flags);
> + if (!list_empty(&cont->release_list))
> +  list_del(&cont->release_list);
> + spin_unlock(&release_list_lock);
>   /* delete my sibling from parent->children */
>   list_del(&cont->sibling);
>   spin_lock(&cont->dentry->d_lock);
> @@ -1869,6 +2098,9 @@ static int container_rmdir(struct inode
>   dput(d);
>   root->number_of_containers--;
>
> + set_bit(CONT_RELEASABLE, &parent->flags);
> + check_for_release(parent);
> +
>   mutex_unlock(&container_mutex);
>   /* Drop the active superblock reference that we took when we
>    * created the container */
> @@ -1906,15 +2138,15 @@ static void container_init_subsys(struct
>    /* If this subsystem requested that it be notified with fork
>     * events, we should send it one now for every process in the
>     * system */
> -  if (ss->fork) {
> -   struct task_struct *g, *p;
```

```
> + if (ss->fork) {
> +  struct task_struct *g, *p;
>
> -   read_lock(&tasklist_lock);
> -   do_each_thread(g, p) {
> -    ss->fork(ss, p);
> -   } while_each_thread(g, p);
> -   read_unlock(&tasklist_lock);
> -  }
> +  read_lock(&tasklist_lock);
> +  do_each_thread(g, p) {
> +   ss->fork(ss, p);
> +  } while_each_thread(g, p);
> +  read_unlock(&tasklist_lock);
> + }
>
>   need_forkexit_callback |= ss->fork || ss->exit;
>
> @@ -2241,7 +2473,7 @@ void container_exit(struct task_struct *
>   tsk->containers = &init_css_group;
>   task_unlock(tsk);
>   if (cg)
> -  put_css_group(cg);
> +  put_css_group_taskexit(cg);
>  }
>
>  /**
> @@ -2352,7 +2584,10 @@ int container_clone(struct task_struct *
>
>   out_release:
>   mutex_unlock(&inode->i_mutex);
> +
> + mutex_lock(&container_mutex);
>   put_css_group(cg);
> + mutex_unlock(&container_mutex);
>   deactivate_super(parent->root->sb);
>   return ret;
>  }
> @@ -2382,3 +2617,111 @@ int container_is_descendant(const struct
>   ret = (cont == target);
>   return ret;
>  }
> +
> +static void check_for_release(struct container *cont)
> +{
> + /* All of these checks rely on RCU to keep the container
> +  * structure alive */
> + if (container_is_releasable(cont) && !atomic_read(&cont->count)
```

```
> +     && list_empty(&cont->children) && !container_has_css_refs(cont)) {
> +  /* Container is currently removeable. If it's not
> +   * already queued for a userspace notification, queue
> +   * it now */
> +  int need_schedule_work = 0;
> +  spin_lock(&release_list_lock);
> +  if (!container_is_removed(cont) &&
> +      list_empty(&cont->release_list)) {
> +   list_add(&cont->release_list, &release_list);
> +   need_schedule_work = 1;
> +  }
> +  spin_unlock(&release_list_lock);
> +  if (need_schedule_work)
> +   schedule_work(&release_agent_work);
> + }
> +}
> +
> +void __css_put(struct container_subsys_state *css)
> +{
> + struct container *cont = css->container;
> + rcu_read_lock();
> + if (atomic_dec_and_test(&css->refcnt) && notify_on_release(cont)) {
> +  set_bit(CONT_RELEASABLE, &cont->flags);
> +  check_for_release(cont);
> + }
> + rcu_read_unlock();
> +}
> +
> +/*
> + * Notify userspace when a container is released, by running the
> + * configured release agent with the name of the container (path
> + * relative to the root of container file system) as the argument.
> + *
> + * Most likely, this user command will try to rmdir this container.
> + *
> + * This races with the possibility that some other task will be
> + * attached to this container before it is removed, or that some other
> + * user task will 'mkdir' a child container of this container.  That's ok.
> + * The presumed 'rmdir' will fail quietly if this container is no longer
> + * unused, and this container will be reprieved from its death sentence,
> + * to continue to serve a useful existence.  Next time it's released,
> + * we will get notified again, if it still has 'notify_on_release' set.
> + *
> + * The final arg to call_usermodehelper() is UMH_WAIT_EXEC, which
> + * means only wait until the task is successfully execve()'d.  The
> + * separate release agent task is forked by call_usermodehelper(),
> + * then control in this thread returns here, without waiting for the
> + * release agent task.  We don't bother to wait because the caller of
```

```
> + * this routine has no use for the exit status of the release agent
> + * task, so no sense holding our caller up for that.
> + *
> + */
> +
> +static void container_release_agent(struct work_struct *work)
> +{
> + BUG_ON(work != &release_agent_work);
> + mutex_lock(&container_mutex);
> + spin_lock(&release_list_lock);
> + while (!list_empty(&release_list)) {
> +  char *argv[3], *envp[3];
> +  int i;
> +  char *pathbuf;
> +  struct container *cont = list_entry(release_list.next,
> +         struct container,
> +         release_list);
> +  list_del_init(&cont->release_list);
> +  spin_unlock(&release_list_lock);
> +  pathbuf = kmalloc(PAGE_SIZE, GFP_KERNEL);
> +  if (!pathbuf) {
> +   spin_lock(&release_list_lock);
> +   continue;
> +  }
> +
> +  if (container_path(cont, pathbuf, PAGE_SIZE) < 0) {
> +   kfree(pathbuf);
> +   spin_lock(&release_list_lock);
> +   continue;
> +  }
> +
> +  i = 0;
> +  argv[i++] = cont->root->release_agent_path;
> +  argv[i++] = (char *)pathbuf;
> +  argv[i] = NULL;
> +
> +  i = 0;
> +  /* minimal command environment */
> +  envp[i++] = "HOME=/";
> +  envp[i++] = "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
> +  envp[i] = NULL;
> +
> +  /* Drop the lock while we invoke the usermode helper,
> +   * since the exec could involve hitting disk and hence
> +   * be a slow process */
> +  mutex_unlock(&container_mutex);
> +  call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
> +  kfree(pathbuf);
```

> + mutex_lock(&container_mutex);
> + spin_lock(&release_list_lock);
> + }
> + spin_unlock(&release_list_lock);
> + mutex_unlock(&container_mutex);
> +}
>
> --

Subject: Re: [PATCH 01/10] Task Containers(V11): Basic task container framework
Posted by yamamoto on Fri, 27 Jul 2007 05:31:19 GMT
View Forum Message <> Reply to Message

> +Other fields in the container_subsys object include:

> +- hierarchy: an index indicating which hierarchy, if any, this
> + subsystem is currently attached to. If this is -1, then the
> + subsystem is not attached to any hierarchy, and all tasks should be
> + considered to be members of the subsystem's top_container. It should
> + be initialized to -1.

stale info?

> +struct container {

> + struct containerfs_root *root;
> + struct container *top_container;
> +};

can cont->top_container be different from than &cont->root.top_container?

YAMAMOTO Takashi

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

---

Subject: Re: [PATCH 01/10] Task Containers(V11): Basic task container framework
Posted by Paul Menage on Sun, 29 Jul 2007 17:03:49 GMT
View Forum Message <> Reply to Message

On 7/26/07, YAMAMOTO Takashi <yamamoto@valinux.co.jp> wrote:
> > +Other fields in the container_subsys object include:
>
> > +- hierarchy: an index indicating which hierarchy, if any, this

> > + subsystem is currently attached to. If this is -1, then the
> > + subsystem is not attached to any hierarchy, and all tasks should be
> > + considered to be members of the subsystem's top_container. It should
> > + be initialized to -1.
>
> stale info?

Yes, I think so. I really need to do a proper pass over
Documentation/containers.txt one more time following a bunch of recent
changes.

>
> > +struct container {
>
> > +   struct containerfs_root *root;
> > +   struct container *top_container;
> > +};
>
> can cont->top_container be different from than &cont->root.top_container?

No, I guess it can't. And there are only a couple of places using
cont->top_container now, so I should probably remove it.

Thanks,

Paul

Subject: Re: [PATCH 01/10] Task Containers(V11): Basic task container framework
Posted by yamamoto on Mon, 30 Jul 2007 04:00:00 GMT
View Forum Message <> Reply to Message

> +extern void container_init_smp(void);

> +static inline void container_init_smp(void) {}

stale prototypes?

YAMAMOTO Takashi
