Subject: [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:07 GMT
View Forum Message <> Reply to Message

--

This is an update to my multi-hierarchy multi-subsystem generic
process containers patch. Changes since V8 (April 6th) include:

- The patchset has been rebased over 2.6.21-rc7-mm1

- The patchset has been restructured based on feedback; more
  functionality is now split out into separate patches where
  practical.

- The container_group structure has been renamed css_group since this
  is more descriptive of its true function

- Added a simplified file registration interface, and a simple
  interface for the common operation of returning a single number to
  userspace from a container control file

- Added a simple "debug" subsystem that is both an example of how to
  use the container system and a useful debugging tool for checking
  reference counts, etc.

Still TODO:

- decide whether "Containers" is an acceptable name for the system
given its usage by some other development groups, or whether something
else (ProcessSets? ResourceGroups? TaskGroups?) would be better

- decide whether merging css_group and nsproxy is desirable

- 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 back support for the "release agent" functionality

- lots more testing

- define standards for container file names

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.

Already existing in the kernel is the cpuset subsystem; this has a process grouping mechanism that is mature, tested, and well documented (particularly with regards to synchronization rules).

This patchset extracts the process grouping code from cpusets into a generic container system, and makes the cpusets code a client of the container system, along with a couple of simple example subsystems.

The patch set is structured as follows:

1) Basic container framework - filesystem and tracking structures

2) Simple CPU Accounting example subsystem

3) Support for the "tasks" control file

4) Hooks for fork() and exit()

5) Support for the container_clone() operation

6) Add /proc reporting interface

7) Make cpusets a container subsystem

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

9) Simple container debugging subsystem

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

- the userspace APIs are (somewhat) normalised

- it's easier to test out 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

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

Subject: [PATCH 1/9] Containers (V9): Basic container framework
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:08 GMT
View Forum Message <> Reply to Message

This patch adds the main 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 | 524 ++++++++++++++++++
 include/linux/container.h | 198 ++++++
 include/linux/container_subsys.h | 10
 include/linux/sched.h | 34 +
 init/Kconfig | 3
 init/main.c | 3
 kernel/Makefile | 1
 kernel/container.c | 1151 +++++++++++++++++++++++++++++++++++++++
 8 files changed, 1923 insertions(+), 1 deletion(-)

Index: container-2.6.21-rc7-mm1/Documentation/containers.txt
 == =======
--- /dev/null
+++ container-2.6.21-rc7-mm1/Documentation/containers.txt
@@ -0,0 +1,524 @@
+   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 may 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 $$ > 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.
+
+int create(struct container *cont)
+LL=container_mutex
+

+Called to create a subsystem state object for a container. The
+subsystem should set its subsystem pointer for the passed container,
+returning 0 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-2.6.21-rc7-mm1/include/linux/container.h
 ========================================================== =======
--- /dev/null
+++ container-2.6.21-rc7-mm1/include/linux/container.h
@@ -0,0 +1,198 @@
+#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>
+
+#ifdef CONFIG_CONTAINERS
+
+extern int container_init_early(void);
+extern int container_init(void);
+extern void container_init_smp(void);
+
+extern struct file_operations proc_container_operations;
+
+extern void container_lock(void);
+extern void container_unlock(void);
+
+struct containerfs_root;
+
+/* 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;
+};
+
+/*
+ * Call css_get() to hold a reference on the container;
+ *
+ */
+
+static inline void css_get(struct container_subsys_state *css)
+{
+ 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)
+{
+ 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
+ */
+
+struct inode;
+#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);
+ 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, 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, 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 {
+ int (*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-2.6.21-rc7-mm1/include/linux/container_subsys.h
 ===================================================================
--- /dev/null
+++ container-2.6.21-rc7-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-2.6.21-rc7-mm1/include/linux/sched.h
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/include/linux/sched.h
+++ container-2.6.21-rc7-mm1/include/linux/sched.h
@@ -820,6 +820,34 @@ struct uts_namespace;

 struct prio_array;

+#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 */
  struct thread_info *thread_info;
@@ -1072,6 +1100,9 @@ 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;
@@ -1509,7 +1540,8 @@ 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-2.6.21-rc7-mm1/init/Kconfig
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/init/Kconfig
+++ container-2.6.21-rc7-mm1/init/Kconfig
@@ -288,6 +288,9 @@ config IKCONFIG_PROC
    This option enables access to the kernel configuration file
    through /proc/config.gz.

+config CONTAINERS
+ bool
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
Index: container-2.6.21-rc7-mm1/init/main.c
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/init/main.c
+++ container-2.6.21-rc7-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>
@@ -499,6 +500,7 @@ asmlinkage void __init start_kernel(void
  char * command_line;
  extern struct kernel_param __start___param[], __stop___param[];
```

```
+ container_init_early();
  smp_setup_processor_id();

  /*
@@ -624,6 +626,7 @@ 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-2.6.21-rc7-mm1/kernel/Makefile
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/Makefile
+++ container-2.6.21-rc7-mm1/kernel/Makefile
@@ -36,6 +36,7 @@ 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-2.6.21-rc7-mm1/kernel/container.c
 ===================================================================
--- /dev/null
+++ container-2.6.21-rc7-mm1/kernel/container.c
@@ -0,0 +1,1151 @@
+/*
+ *  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/cpu.h>
+#include <linux/cpumask.h>
+#include <linux/container.h>
+#include <linux/err.h>
+#include <linux/errno.h>
+#include <linux/file.h>
+#include <linux/fs.h>
+#include <linux/init.h>
+#include <linux/interrupt.h>
+#include <linux/kernel.h>
+#include <linux/kmod.h>
+#include <linux/list.h>
+#include <linux/mempolicy.h>
+#include <linux/mm.h>
+#include <linux/module.h>
+#include <linux/mount.h>
+#include <linux/namei.h>
+#include <linux/pagemap.h>
+#include <linux/proc_fs.h>
+#include <linux/rcupdate.h>
+#include <linux/sched.h>
+#include <linux/seq_file.h>
+#include <linux/security.h>
+#include <linux/slab.h>
+#include <linux/smp_lock.h>
+#include <linux/spinlock.h>
+#include <linux/stat.h>
+#include <linux/string.h>
+#include <linux/time.h>
+#include <linux/backing-dev.h>
+#include <linux/sort.h>
+
+#include <asm/uaccess.h>
+#include <asm/atomic.h>
+#include <linux/mutex.h>
+
+#define CONTAINER_SUPER_MAGIC  0x27e0eb
+
+/* 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 attached to this hierarchy */
+ unsigned long 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;
+};
+
+
+/* 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 = 0;
+
```

```
+/* bits in struct container flags field */
+typedef enum {
+ CONT_REMOVED,
+} container_flagbits_t;
+
+/* convenient tests for these bits */
+inline int container_is_removed(const struct container *cont)
+{
+ return test_bit(CONT_REMOVED, &cont->flags);
+}
+
+/* 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 backing_dev_info container_backing_dev_info = {
+ .ra_pages = 0,  /* No readahead */
+ .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
+};
+
+static struct inode *container_new_inode(mode_t mode, struct super_block *sb)
+{
+ struct inode *inode = new_inode(sb);
+
+ 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_operations container_dops = {
+ .d_iput = container_diput,
+};
+
+static struct dentry *container_get_dentry(struct dentry *parent,
```

```
+        const char *name)
+{
+ struct dentry *d = lookup_one_len(name, parent, strlen(name));
+ 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->subsys_bits & ~final_bits;
+ added_bits = final_bits & ~root->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 erro r 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 = final_bits;
+ synchronize_rcu();
+
+ return 0;
+}
+
+/*
+ * Release the last use of a hierarchy.  Will never be called when
+ * there are active subcontainers since each subcontainer bumps the
+ * value of sb->s_active.
+ */
+
+static void container_put_super(struct super_block *sb) {
+
+ struct containerfs_root *root = sb->s_fs_info;
+ struct container *cont = &root->top_container;
+ int ret;
+
+ root->sb = NULL;
+ sb->s_fs_info = NULL;
+
+ mutex_lock(&container_mutex);
+
+ BUG_ON(root->number_of_containers != 1);
+ BUG_ON(!list_empty(&cont->children));
+ BUG_ON(!list_empty(&cont->sibling));
```

```
+ BUG_ON(!root->subsys_bits);
+
+ /* Rebind all subsystems back to the default hierarchy */
+ ret = rebind_subsystems(root, 0);
+ BUG_ON(ret);
+
+ kfree(root);
+ mutex_unlock(&container_mutex);
+}
+
+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;
+ for_each_subsys(root, ss) {
+  seq_printf(seq, ",%s", ss->name);
+ }
+ return 0;
+}
+
+/* Convert a hierarchy specifier into a bitmask. LL=container_mutex */
+static int parse_containerfs_options(char *opts, unsigned long *bits)
+{
+ char *token, *o = opts ?: "all";
+
+ *bits = 0;
+
+ while ((token = strsep(&o, ",")) != NULL) {
+  if (!*token)
+   return -EINVAL;
+  if (!strcmp(token, "all")) {
+   *bits = (1 << CONTAINER_SUBSYS_COUNT) - 1;
+  } else {
+   struct container_subsys *ss;
+   int i;
+   for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+    ss = subsys[i];
+    if (!strcmp(token, ss->name)) {
+     *bits |= 1 << i;
+     break;
+    }
+   }
+   if (i == CONTAINER_SUBSYS_COUNT)
+    return -ENOENT;
+  }
+ }
+
+ /* We can't have an empty hierarchy */
```

```
+ if (!*bits)
+  return -EINVAL;
+
+ return 0;
+}
+
+static int container_remount(struct super_block *sb, int *flags, char *data)
+{
+ int ret = 0;
+ unsigned long subsys_bits;
+ struct containerfs_root *root = sb->s_fs_info;
+ struct container *cont = &root->top_container;
+
+ mutex_lock(&cont->dentry->d_inode->i_mutex);
+ mutex_lock(&container_mutex);
+
+ /* See what subsystems are wanted */
+ ret = parse_containerfs_options(data, &subsys_bits);
+ if (ret)
+  goto out_unlock;
+
+ ret = rebind_subsystems(root, 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,
+ .put_super = container_put_super,
+ .show_options = container_show_options,
+ .remount_fs = container_remount,
+};
+
+static int container_fill_super(struct super_block *sb, void *options,
+    int unused_silent)
+{
+ struct inode *inode;
+ struct dentry *root;
+ struct containerfs_root *hroot = options;
+
```

```
+ sb->s_blocksize = PAGE_CACHE_SIZE;
+ sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
+ sb->s_magic = CONTAINER_SUPER_MAGIC;
+ sb->s_op = &container_ops;
+
+ inode = container_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR, sb);
+ 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);
+
+ root = d_alloc_root(inode);
+ if (!root) {
+  iput(inode);
+  return -ENOMEM;
+ }
+ sb->s_root = root;
+ root->d_fsdata = &hroot->top_container;
+ hroot->top_container.dentry = root;
+
+ sb->s_fs_info = hroot;
+ hroot->sb = sb;
+
+ return 0;
+}
+
+static void init_container_root(struct containerfs_root *root) {
+ struct container *cont = &root->top_container;
+ INIT_LIST_HEAD(&root->subsys_list);
+ root->number_of_containers = 1;
+ cont->root = root;
+ cont->top_container = cont;
+ INIT_LIST_HEAD(&cont->sibling);
+ INIT_LIST_HEAD(&cont->children);
+ list_add(&root->root_list, &roots);
+}
+
+static int container_get_sb(struct file_system_type *fs_type,
+    int flags, const char *unused_dev_name,
+    void *data, struct vfsmount *mnt)
+{
+ unsigned long subsys_bits = 0;
+ int ret = 0;
+ struct containerfs_root *root = NULL;
```

```
+ int use_existing = 0;
+
+ mutex_lock(&container_mutex);
+
+ /* First find the desired set of resource controllers */
+ ret = parse_containerfs_options(data, &subsys_bits);
+ if (ret)
+  goto out_unlock;
+
+ /* See if we already have a hierarchy containing this set */
+
+ for_each_root(root) {
+  /* We match - use this hieracrchy */
+  if (root->subsys_bits == subsys_bits) {
+   use_existing = 1;
+   break;
+  }
+  /* We clash - fail */
+  if (root->subsys_bits & subsys_bits) {
+   ret = -EBUSY;
+   goto out_unlock;
+  }
+ }
+
+ if (!use_existing) {
+  /* We need a new root */
+  root = kzalloc(sizeof(*root), GFP_KERNEL);
+  if (!root) {
+   ret = -ENOMEM;
+   goto out_unlock;
+  }
+  init_container_root(root);
+ }
+
+ if (!root->sb) {
+  /* We need a new superblock for this container combination */
+  struct container *cont = &root->top_container;
+
+  BUG_ON(root->subsys_bits);
+  ret = get_sb_nodev(fs_type, flags, root,
+        container_fill_super, mnt);
+  if (ret)
+   goto out_unlock;
+
+  BUG_ON(!list_empty(&cont->sibling));
+  BUG_ON(!list_empty(&cont->children));
+  BUG_ON(root->number_of_containers != 1);
+
```

```
+  ret = rebind_subsystems(root, subsys_bits);
+
+  /* It's safe to nest i_mutex inside container_mutex in
+   * this case, since no-one else can be accessing this
+   * directory yet */
+  mutex_lock(&cont->dentry->d_inode->i_mutex);
+  container_populate_dir(cont);
+  mutex_unlock(&cont->dentry->d_inode->i_mutex);
+  BUG_ON(ret);
+
+ } else {
+  /* Reuse the existing superblock */
+  ret = simple_set_mnt(mnt, root->sb);
+  if (!ret)
+   atomic_inc(&root->sb->s_active);
+ }
+
+ out_unlock:
+ mutex_unlock(&container_mutex);
+ return ret;
+}
+
+static struct file_system_type container_fs_type = {
+ .name = "container",
+ .get_sb = container_get_sb,
+ .kill_sb = kill_litter_super,
+};
+
+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;
+}
+
+/*
+ * Call 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;
+}
+
+static inline 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;
+}
+
+/* The various types of files and directories in a container file system */
+
+typedef enum {
+ FILE_ROOT,
+ FILE_DIR,
+ FILE_TASKLIST,
+} container_filetype_t;
+
+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", 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, const struct cftype *cft)
+{
+ struct dentry *dir = cont->dentry;
+ struct dentry *dentry;
+ int error;
+
+ BUG_ON(!mutex_is_locked(&dir->d_inode->i_mutex));
+ dentry = container_get_dentry(dir, cft->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, const struct cftype cft[],
+   int count)
+{
+ int i, err;
+ for (i = 0; i < count; i++) {
+  if ((err = container_add_file(cont, &cft[i])))
+   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 *ss,
+          struct container *cont)
+{
+ struct container_subsys_state *css = cont->subsys[ss->subsys_id];
+ css->container = cont;
+ atomic_set(&css->refcnt, 0);
+}
+
+/*
+ * 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) {
+  err = ss->create(ss, cont);
+  if (err) goto err_destroy;
+  init_container_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) {
+ int retval;
+ 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;
+ retval = ss->create(ss, dummytop);
+ BUG_ON(retval);
+ BUG_ON(!dummytop->subsys[ss->subsys_id]);
+ init_container_css(ss, dummytop);
+ css = dummytop->subsys[ss->subsys_id];
+
+ /* 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);
+
+ 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;
+}

--
```

Subject: [PATCH 2/9] Containers (V9): Example CPU accounting subsystem
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:09 GMT
View Forum Message <> Reply to Message

This example demonstrates how to use the generic container subsystem
for a simple resource tracker that counts, for the processes in a
container, the total CPU time used and the %CPU used in the last
complete 10 second interval.

Portions contributed by Balbir Singh <balbir@in.ibm.com>

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

---
 include/linux/container_subsys.h | 6 +

```
 include/linux/cpu_acct.h        |   14 ++
 init/Kconfig                    |    7 +
 kernel/Makefile                 |    1
 kernel/cpu_acct.c               |  185 ++++++++++++++++++++++++++++++++++++++++
 kernel/sched.c                  |   14 ++
 6 files changed, 224 insertions(+), 3 deletions(-)

Index: container-2.6.21-rc7-mm1/include/linux/container_subsys.h
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/include/linux/container_subsys .h
+++ container-2.6.21-rc7-mm1/include/linux/container_subsys.h
@@ -7,4 +7,10 @@

 /* */

+#ifdef CONFIG_CONTAINER_CPUACCT
+SUBSYS(cpuacct)
+#endif
+
+/* */
+
 /* */
Index: container-2.6.21-rc7-mm1/include/linux/cpu_acct.h
 ===================================================================
--- /dev/null
+++ container-2.6.21-rc7-mm1/include/linux/cpu_acct.h
@@ -0,0 +1,14 @@
+
+#ifndef _LINUX_CPU_ACCT_H
+#define _LINUX_CPU_ACCT_H
+
+#include <linux/container.h>
+#include <asm/cputime.h>
+
+#ifdef CONFIG_CONTAINER_CPUACCT
+extern void cpuacct_charge(struct task_struct *, cputime_t cputime);
+#else
+static void inline cpuacct_charge(struct task_struct *p, cputime_t cputime) {}
+#endif
+
+#endif
Index: container-2.6.21-rc7-mm1/init/Kconfig
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/init/Kconfig
+++ container-2.6.21-rc7-mm1/init/Kconfig
@@ -322,6 +322,13 @@ config SYSFS_DEPRECATED
   If you are using a distro that was released in 2006 or later,
   it should be safe to say N here.
```

```
+config CONTAINER_CPUACCT
+ bool "Simple CPU accounting container subsystem"
+ select CONTAINERS
+ help
+   Provides a simple Resource Controller for monitoring the
+   total CPU consumed by the tasks in a container
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
Index: container-2.6.21-rc7-mm1/kernel/Makefile
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/Makefile
+++ container-2.6.21-rc7-mm1/kernel/Makefile
@@ -38,6 +38,7 @@ obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
+obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
 obj-$(CONFIG_AUDIT) += audit.o auditfilter.o
Index: container-2.6.21-rc7-mm1/kernel/cpu_acct.c
 ===================================================================
--- /dev/null
+++ container-2.6.21-rc7-mm1/kernel/cpu_acct.c
@@ -0,0 +1,185 @@
+/*
+ * kernel/cpu_acct.c - CPU accounting container subsystem
+ *
+ * Copyright (C) Google Inc, 2006
+ *
+ * Developed by Paul Menage (menage@google.com) and Balbir Singh
+ * (balbir@in.ibm.com)
+ *
+ */
+
+/*
+ * Container subsystem for reporting total CPU usage of tasks in a
+ * container, along with percentage load over a time interval
+ */
+
+#include <linux/module.h>
+#include <linux/container.h>
+#include <linux/fs.h>
+#include <asm/div64.h>
+
```

```
+struct cpuacct {
+ struct container_subsys_state css;
+ spinlock_t lock;
+ /* total time used by this class */
+ cputime64_t time;
+
+ /* time when next load calculation occurs */
+ u64 next_interval_check;
+
+ /* time used in current period */
+ cputime64_t current_interval_time;
+
+ /* time used in last period */
+ cputime64_t last_interval_time;
+};
+
+struct container_subsys cpuacct_subsys;
+
+static inline struct cpuacct *container_ca(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, cpuacct_subsys_id),
+       struct cpuacct, css);
+}
+
+static inline struct cpuacct *task_ca(struct task_struct *task)
+{
+ return container_of(task_subsys_state(task, cpuacct_subsys_id),
+       struct cpuacct, css);
+}
+
+#define INTERVAL (HZ * 10)
+
+static inline u64 next_interval_boundary(u64 now) {
+ /* calculate the next interval boundary beyond the
+  * current time */
+ do_div(now, INTERVAL);
+ return (now + 1) * INTERVAL;
+}
+
+static int cpuacct_create(struct container_subsys *ss, struct container *cont)
+{
+ struct cpuacct *ca = kzalloc(sizeof(*ca), GFP_KERNEL);
+ if (!ca)
+  return -ENOMEM;
+ spin_lock_init(&ca->lock);
+ ca->next_interval_check = next_interval_boundary(get_jiffies_64());
+ cont->subsys[cpuacct_subsys_id] = &ca->css;
+ return 0;
```

```
+}
+
+static void cpuacct_destroy(struct container_subsys *ss,
+       struct container *cont)
+{
+ kfree(container_ca(cont));
+}
+
+/* Lazily update the load calculation if necessary. Called with ca locked */
+static void cpuusage_update(struct cpuacct *ca)
+{
+ u64 now = get_jiffies_64();
+ /* If we're not due for an update, return */
+ if (ca->next_interval_check > now)
+  return;
+
+ if (ca->next_interval_check <= (now - INTERVAL)) {
+  /* If it's been more than an interval since the last
+   * check, then catch up - the last interval must have
+   * been zero load */
+  ca->last_interval_time = 0;
+  ca->next_interval_check = next_interval_boundary(now);
+ } else {
+  /* If a steal takes the last interval time negative,
+   * then we just ignore it */
+  if ((s64)ca->current_interval_time > 0) {
+   ca->last_interval_time = ca->current_interval_time;
+  } else {
+   ca->last_interval_time = 0;
+  }
+  ca->next_interval_check += INTERVAL;
+ }
+ ca->current_interval_time = 0;
+}
+
+static u64 cpuusage_read(struct container *cont,
+     struct cftype *cft)
+{
+ struct cpuacct *ca = container_ca(cont);
+ u64 time;
+
+ spin_lock_irq(&ca->lock);
+ cpuusage_update(ca);
+ time = cputime64_to_jiffies64(ca->time);
+ spin_unlock_irq(&ca->lock);
+
+ /* Convert 64-bit jiffies to seconds */
+ time *= 1000;
```

```
+ do_div(time, HZ);
+ return time;
+}
+
+static u64 load_read(struct container *cont,
+       struct cftype *cft)
+{
+ struct cpuacct *ca = container_ca(cont);
+ u64 time;
+
+ /* Find the time used in the previous interval */
+ spin_lock_irq(&ca->lock);
+ cpuusage_update(ca);
+ time = cputime64_to_jiffies64(ca->last_interval_time);
+ spin_unlock_irq(&ca->lock);
+
+ /* Convert time to a percentage, to give the load in the
+  * previous period */
+ time *= 100;
+ do_div(time, INTERVAL);
+
+ return time;
+}
+
+static struct cftype files[] = {
+ {
+  .name = "cpuacct.usage",
+  .read_uint = cpuusage_read,
+ },
+ {
+  .name = "cpuacct.load",
+  .read_uint = load_read,
+ }
+};
+
+static int cpuacct_populate(struct container_subsys *ss,
+       struct container *cont)
+{
+ return container_add_files(cont, files, ARRAY_SIZE(files));
+}
+
+void cpuacct_charge(struct task_struct *task, cputime_t cputime)
+{
+
+ struct cpuacct *ca;
+ unsigned long flags;
+
+ if (!cpuacct_subsys.active)
```

```
+  return;
+ rcu_read_lock();
+ ca = task_ca(task);
+ if (ca) {
+  spin_lock_irqsave(&ca->lock, flags);
+  cpuusage_update(ca);
+  ca->time = cputime64_add(ca->time, cputime);
+  ca->current_interval_time =
+   cputime64_add(ca->current_interval_time, cputime);
+  spin_unlock_irqrestore(&ca->lock, flags);
+ }
+ rcu_read_unlock();
+}
+
+struct container_subsys cpuacct_subsys = {
+ .name = "cpuacct",
+ .create = cpuacct_create,
+ .destroy = cpuacct_destroy,
+ .populate = cpuacct_populate,
+ .subsys_id = cpuacct_subsys_id,
+};
Index: container-2.6.21-rc7-mm1/kernel/sched.c
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/sched.c
+++ container-2.6.21-rc7-mm1/kernel/sched.c
@@ -56,6 +56,7 @@
 #include <linux/kprobes.h>
 #include <linux/delayacct.h>
 #include <linux/reciprocal_div.h>
+#include <linux/cpu_acct.h>

 #include <asm/tlb.h>
 #include <asm/unistd.h>
@@ -3328,9 +3329,13 @@ void account_user_time(struct task_struc
 {
  struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
  cputime64_t tmp;
+ struct rq *rq = this_rq();

  p->utime = cputime_add(p->utime, cputime);

+ if (p != rq->idle)
+  cpuacct_charge(p, cputime);
+
  /* Add user time to cpustat. */
  tmp = cputime_to_cputime64(cputime);
  if (TASK_NICE(p) > 0)
@@ -3360,9 +3365,10 @@ void account_system_time(struct task_str
```

```
   cpustat->irq = cputime64_add(cpustat->irq, tmp);
  else if (softirq_count())
   cpustat->softirq = cputime64_add(cpustat->softirq, tmp);
- else if (p != rq->idle)
+ else if (p != rq->idle) {
   cpustat->system = cputime64_add(cpustat->system, tmp);
- else if (atomic_read(&rq->nr_iowait) > 0)
+  cpuacct_charge(p, cputime);
+ } else if (atomic_read(&rq->nr_iowait) > 0)
   cpustat->iowait = cputime64_add(cpustat->iowait, tmp);
  else
   cpustat->idle = cputime64_add(cpustat->idle, tmp);
@@ -3387,8 +3393,10 @@ void account_steal_time(struct task_stru
    cpustat->iowait = cputime64_add(cpustat->iowait, tmp);
   else
    cpustat->idle = cputime64_add(cpustat->idle, tmp);
- } else
+ } else {
   cpustat->steal = cputime64_add(cpustat->steal, tmp);
+  cpuacct_charge(p, -tmp);
+ }
 }

 /*

--
```

Subject: [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:10 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 |    2
 kernel/container.c        |  344 ++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 346 insertions(+)

Index: container-2.6.21-rc7-mm1/include/linux/container.h
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/include/linux/container.h
+++ container-2.6.21-rc7-mm1/include/linux/container.h
```

```
@@ -128,6 +128,8 @@ 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);
+
 /* Return true if the container is a descendant of the current container */
 int container_is_descendant(const struct container *cont);

Index: container-2.6.21-rc7-mm1/kernel/container.c
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/container.c
+++ container-2.6.21-rc7-mm1/kernel/container.c
@@ -676,6 +676,111 @@ static inline void get_first_subsys(cons
   *subsys_id = test_ss->subsys_id;
 }

+/*
+ * Attach task 'tsk' to container 'cont'
+ *
+ * Call holding container_mutex.  May take task_lock of
+ * the task 'pid' during call.
+ */
+
+static int attach_task(struct container *cont, struct task_struct *tsk)
+{
+ int retval = 0;
+ struct container_subsys *ss;
+ struct container *oldcont;
+ struct css_group *cg = &tsk->containers;
+ struct containerfs_root *root = cont->root;
+ int i;
+
+ int subsys_id;
+ get_first_subsys(cont, NULL, &subsys_id);
+
+ /* Nothing to do if the task is already in that container */
+ oldcont = task_container(tsk, subsys_id);
+ if (cont == oldcont)
+  return 0;
+
+ for_each_subsys(root, ss) {
+  if (ss->can_attach) {
+   retval = ss->can_attach(ss, cont, tsk);
+   if (retval) {
+    return retval;
+   }
+  }
```

```
+ }
+
+ task_lock(tsk);
+ if (tsk->flags & PF_EXITING) {
+  task_unlock(tsk);
+  return -ESRCH;
+ }
+ /* Update the css_group pointers for the subsystems in this
+  * hierarchy */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  if (root->subsys_bits & (1ull << i)) {
+   /* Subsystem is in this hierarchy. So we want
+    * the subsystem state from the new
+    * container. Transfer the refcount from the
+    * old to the new */
+   atomic_inc(&cont->count);
+   atomic_dec(&cg->subsys[i]->container->count);
+   rcu_assign_pointer(cg->subsys[i], cont->subsys[i]);
+  }
+ }
+ task_unlock(tsk);
+
+ for_each_subsys(root, ss) {
+  if (ss->attach) {
+   ss->attach(ss, cont, oldcont, tsk);
+  }
+ }
+
+ synchronize_rcu();
+ return 0;
+}
+
+/*
+ * Attach task with pid 'pid' to container 'cont'. Call with
+ * container_mutex, may take task_lock of task
+ *
+ */
+
+static int attach_task_by_pid(struct container *cont, char *pidbuf)
+{
+ pid_t pid;
+ struct task_struct *tsk;
+ int ret;
+
+ if (sscanf(pidbuf, "%d", &pid) != 1)
+  return -EIO;
+
+ if (pid) {
```

```
+  read_lock(&tasklist_lock);
+
+  tsk = find_task_by_pid(pid);
+  if (!tsk || tsk->flags & PF_EXITING) {
+   read_unlock(&tasklist_lock);
+   return -ESRCH;
+  }
+
+  get_task_struct(tsk);
+  read_unlock(&tasklist_lock);
+
+  if ((current->euid) && (current->euid != tsk->uid)
+      && (current->euid != tsk->suid)) {
+   put_task_struct(tsk);
+   return -EACCES;
+  }
+ } else {
+  tsk = current;
+  get_task_struct(tsk);
+ }
+
+ ret = attach_task(cont, tsk);
+ put_task_struct(tsk);
+ return ret;
+}
+
 /* The various types of files and directories in a container file system */

 typedef enum {
@@ -684,6 +789,54 @@ typedef enum {
  FILE_TASKLIST,
 } container_filetype_t;

+static ssize_t container_common_file_write(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       const char __user *userbuf,
+       size_t nbytes, loff_t *unused_ppos)
+{
+ container_filetype_t type = cft->private;
+ char *buffer;
+ int retval = 0;
+
+ if (nbytes >= PATH_MAX)
+  return -E2BIG;
+
+ /* +1 for nul-terminator */
+ if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
```

```
+  return -ENOMEM;
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+  retval = -EFAULT;
+  goto out1;
+ }
+ buffer[nbytes] = 0; /* nul-terminate */
+
+ mutex_lock(&container_mutex);
+
+ if (container_is_removed(cont)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch (type) {
+ case FILE_TASKLIST:
+  retval = attach_task_by_pid(cont, buffer);
+  break;
+ default:
+  retval = -EINVAL;
+  goto out2;
+ }
+
+ if (retval == 0)
+  retval = nbytes;
+out2:
+ mutex_unlock(&container_mutex);
+out1:
+ kfree(buffer);
+ return retval;
+}
+
 static ssize_t container_file_write(struct file *file, const char __user *buf,
       size_t nbytes, loff_t *ppos)
 {
@@ -872,6 +1025,194 @@ 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. */
+
+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);
+
+ read_lock(&tasklist_lock);
+ do_each_thread(g, p) {
+  if (task_subsys_state(p, subsys_id) == css)
+   count ++;
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+ return count;
+}
+
+/*
+ * Stuff for reading the 'tasks' file.
+ *
+ * Reading this file can return large amounts of data if a container has
+ * *lots* of attached tasks. So it may need several calls to read(),
+ * but we cannot guarantee that the information we produce is correct
+ * unless we produce it entirely atomically.
+ *
+ * Upon tasks file open(), a struct ctr_struct is allocated, that
+ * will have a pointer to an array (also allocated here).  The struct
+ * ctr_struct * is stored in file->private_data.  Its resources will
+ * be freed by release() when the file is closed.  The array is used
+ * to sprintf the PIDs and then used by read().
+ */
+
+/* containers_tasks_read array */
+
+struct ctr_struct {
+ char *buf;
+ int bufsz;
+};
+
+/*
+ * Load into 'pidarray' up to 'npids' of the tasks using container
+ * 'cont'.  Return actual number of pids loaded.  No need to
+ * task_lock(p) when reading out p->container, since we're in an RCU
+ * read section, so the css_group can't go away, and is
+ * immutable after creation.
+ */
+static int pid_array_load(pid_t *pidarray, int npids, struct container *cont)
+{
+ int n = 0;
+ struct task_struct *g, *p;
+ struct container_subsys_state *css;
+ int subsys_id;
```

```
+ get_first_subsys(cont, &css, &subsys_id);
+ rcu_read_lock();
+ read_lock(&tasklist_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:
+ read_unlock(&tasklist_lock);
+ 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);
+ 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);
+ 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;
+
+ if (*ppos + nbytes > ctr->bufsz)
+  nbytes = ctr->bufsz - *ppos;
+ if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
+  return -EFAULT;
+ *ppos += nbytes;
+ return nbytes;
+}
+
+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;
@@ -880,6 +1221,9 @@ static int container_populate_dir(struct
  /* First clear out any existing files */
  container_clear_directory(cont->dentry);

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

```
    return err;

--
```

Subject: [PATCH 4/9] Containers (V9): Add fork/exit hooks
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:11 GMT
View Forum Message <> Reply to Message

This patch adds the necessary hooks to the fork() and exit() paths to
ensure that new children inherit their parent's container assignments,
and that exiting processes release reference counts on their
containers.

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

```
---
 include/linux/container.h |    6 ++
 kernel/container.c        |  126 ++++++++++++++++++++++++++++++++++++++++++++++
 kernel/exit.c             |    2
 kernel/fork.c             |   14 ++++-
 4 files changed, 146 insertions(+), 2 deletions(-)

Index: container-2.6.21-rc7-mm1/kernel/exit.c
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/kernel/exit.c
+++ container-2.6.21-rc7-mm1/kernel/exit.c
@@ -32,6 +32,7 @@
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
 #include <linux/posix-timers.h>
@@ -939,6 +940,7 @@ fastcall NORET_TYPE void do_exit(long co
   __exit_fs(tsk);
   exit_thread();
   cpuset_exit(tsk);
+ container_exit(tsk, 1);
   exit_keys(tsk);

   if (group_dead && tsk->signal->leader)
Index: container-2.6.21-rc7-mm1/kernel/fork.c
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/kernel/fork.c
+++ container-2.6.21-rc7-mm1/kernel/fork.c
@@ -30,6 +30,7 @@
```

```
 #include <linux/capability.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/security.h>
 #include <linux/swap.h>
 #include <linux/syscalls.h>
@@ -962,6 +963,7 @@ static struct task_struct *copy_process(
 {
  int retval;
  struct task_struct *p = NULL;
+ int container_callbacks_done = 0;

  if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
   return ERR_PTR(-EINVAL);
@@ -1061,12 +1063,13 @@ static struct task_struct *copy_process(
  p->io_wait = NULL;
  p->audit_context = NULL;
  cpuset_fork(p);
+ container_fork(p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
   if (IS_ERR(p->mempolicy)) {
    retval = PTR_ERR(p->mempolicy);
    p->mempolicy = NULL;
-   goto bad_fork_cleanup_cpuset;
+   goto bad_fork_cleanup_container;
   }
  mpol_fix_fork_child_flag(p);
 #endif
@@ -1176,6 +1179,12 @@ static struct task_struct *copy_process(
  /* Perform scheduler related setup. Assign this task to a CPU. */
  sched_fork(p, clone_flags);

+ /* Now that the task is set up, run container callbacks if
+  * necessary. We need to run them before the task is visible
+  * on the tasklist. */
+ container_fork_callbacks(p);
+ container_callbacks_done = 1;
+
  /* Need tasklist lock for parent etc handling! */
  write_lock_irq(&tasklist_lock);

@@ -1298,9 +1307,10 @@ bad_fork_cleanup_security:
 bad_fork_cleanup_policy:
 #ifdef CONFIG_NUMA
  mpol_free(p->mempolicy);
-bad_fork_cleanup_cpuset:
```

```
+bad_fork_cleanup_container:
 #endif
  cpuset_exit(p);
+ container_exit(p, container_callbacks_done);
  delayacct_tsk_free(p);
  if (p->binfmt)
   module_put(p->binfmt->module);
Index: container-2.6.21-rc7-mm1/include/linux/container.h
 ===================================================== =======
--- container-2.6.21-rc7-mm1.orig/include/linux/container.h
+++ container-2.6.21-rc7-mm1/include/linux/container.h
@@ -18,6 +18,9 @@
 extern int container_init_early(void);
 extern int container_init(void);
 extern void container_init_smp(void);
+extern void container_fork(struct task_struct *p);
+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;

@@ -191,6 +194,9 @@ int container_path(const struct containe
 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_fork(struct task_struct *p) {}
+static inline void container_fork_callbacks(struct task_struct *p) {}
+static inline void container_exit(struct task_struct *p, int callbacks) {}

 static inline void container_lock(void) {}
 static inline void container_unlock(void) {}
Index: container-2.6.21-rc7-mm1/kernel/container.c
 ===================================================== =======
--- container-2.6.21-rc7-mm1.orig/kernel/container.c
+++ container-2.6.21-rc7-mm1/kernel/container.c
@@ -132,6 +132,34 @@ 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
+ * subsystems moved between hierarchies while we were doing so.
+ *
+ * Possible TODO: decide at boot time based on the number of
```

```
+ * registered subsystems and the number of CPUs or NUMA nodes whether
+ * 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) {
+ int i;
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  atomic_inc(&cg->subsys[i]->container->count);
+ }
+}
+
+static void put_css_group(struct css_group *cg) {
+ int i;
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  atomic_dec(&cg->subsys[i]->container->count);
+ }
+}
+
 /*
  * There is one global container mutex. We also require taking
  * task_lock() when dereferencing a task's container subsys pointers.
@@ -1493,3 +1521,101 @@ int __init container_init(void)
 out:
  return err;
 }
+
+/**
+ * container_fork - attach newly forked task to its parents container.
+ * @tsk: pointer to task_struct of forking parent process.
+ *
+ * Description: A task inherits its parent's container at fork().
+ *
+ * A pointer to the shared css_group was automatically copied in
+ * 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.
+ *
+ * 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();
+}
+
+/**
+ * container_fork_callbacks - called on a new task very soon before
+ * adding it to the tasklist. No need to take any locks since no-one
+ * can be operating on this task
+ **/
+
+void container_fork_callbacks(struct task_struct *child)
+{
+ if (need_forkexit_callback) {
+  int i;
+  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+   struct container_subsys *ss = subsys[i];
+   if (ss->fork) {
+    ss->fork(ss, child);
+   }
+  }
+ }
+}
+
+/**
+ * container_exit - detach container from exiting task
+ * @tsk: pointer to task_struct of exiting process
+ *
+ * Description: Detach container from @tsk and release it.
+ *
+ * Note that containers marked notify_on_release force every task in
+ * them to take the global container_mutex mutex when exiting.
+ * This could impact scaling on very large systems.  Be reluctant to
+ * use notify_on_release containers where very high task exit scaling
+ * is required on large systems.
+ *
+ * the_top_container_hack:
+ *
+ *    Set the exiting tasks container to the root container (top_container).
+ *
+ *    We call container_exit() while the task is still competent to
+ *    handle notify_on_release(), then leave the task attached to the
+ *    root container in each hierarchy for the remainder of its exit.
+ *
+ *    To do this properly, we would increment the reference count on
+ *    top_container, and near the very end of the kernel/exit.c do_exit()
+ *    code we would add a second container function call, to drop that
+ *    reference.  This would just create an unnecessary hot spot on
+ *    the top_container reference count, to no avail.
```

```
+ *
+ *    Normally, holding a reference to a container without bumping its
+ *    count is unsafe.   The container could go away, or someone could
+ *    attach us to a different container, decrementing the count on
+ *    the first container that we never incremented.  But in this case,
+ *    top_container isn't going away, and either task has PF_EXITING set,
+ *    which wards off any attach_task() attempts, or task is a failed
+ *    fork, never visible to attach_task.
+ *
+ **/
+
+void container_exit(struct task_struct *tsk, int run_callbacks)
+{
+ int i;
+ if (run_callbacks && need_forkexit_callback) {
+  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+   struct container_subsys *ss = subsys[i];
+   if (ss->exit) {
+    ss->exit(ss, tsk);
+   }
+  }
+ }
+ /* Reassign the task to the init_css_group. */
+ task_lock(tsk);
+ put_css_group(&tsk->containers);
+ tsk->containers = init_task.containers;
+ task_unlock(tsk);
+}

--
```

Subject: [PATCH 6/9] Containers (V9): Add procfs interface
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:13 GMT
View Forum Message <> Reply to Message

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 ++
 kernel/container.c |  128 +++++++++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 135 insertions(+)
```

```
Index: container-2.6.21-rc7-mm1/fs/proc/base.c
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/fs/proc/base.c
+++ container-2.6.21-rc7-mm1/fs/proc/base.c
@@ -68,6 +68,7 @@
 #include <linux/security.h>
 #include <linux/ptrace.h>
 #include <linux/seccomp.h>
+#include <linux/container.h>
 #include <linux/cpuset.h>
 #include <linux/audit.h>
 #include <linux/poll.h>
@@ -1980,6 +1981,9 @@ 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
@@ -2270,6 +2274,9 @@ 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-2.6.21-rc7-mm1/kernel/container.c
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/container.c
+++ container-2.6.21-rc7-mm1/kernel/container.c
@@ -247,6 +247,7 @@ 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;
+struct file_operations proc_containerstats_operations;

 static struct backing_dev_info container_backing_dev_info = {
  .ra_pages = 0,  /* No readahead */
@@ -1507,6 +1508,7 @@ 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];
@@ -1518,10 +1520,136 @@ 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->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);
+}
+
+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.

--
```

Subject: [PATCH 7/9] Containers (V9): Make cpusets a client of containers
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:14 GMT
View Forum Message <> Reply to Message

This patch removes the filesystem support logic from the cpusets
system and makes cpusets a container subsystem

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

 Documentation/cpusets.txt | 91 +--
 fs/proc/base.c | 4
 include/linux/container_subsys.h | 6
 include/linux/cpuset.h | 12
 include/linux/mempolicy.h | 12
 include/linux/sched.h | 3
 init/Kconfig | 6
 kernel/cpuset.c | 1146 +++++++----------------------------------
 kernel/exit.c | 2
 kernel/fork.c | 3
 mm/mempolicy.c | 2
 11 files changed, 236 insertions(+), 1051 deletions(-)

Index: container-2.6.21-rc7-mm1/Documentation/cpusets.txt
 === =======
--- container-2.6.21-rc7-mm1.orig/Documentation/cpusets.txt
+++ container-2.6.21-rc7-mm1/Documentation/cpusets.txt
@@ -7,6 +7,7 @@ Written by Simon.Derr@bull.net
 Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
 Modified by Paul Jackson <pj@sgi.com>
 Modified by Christoph Lameter <clameter@sgi.com>
+Modified by Paul Menage <menage@google.com>

 CONTENTS:
 =========
@@ -16,10 +17,9 @@ CONTENTS:
   1.2 Why are cpusets needed ?
   1.3 How are cpusets implemented ?
   1.4 What are exclusive cpusets ?
- 1.5 What does notify_on_release do ?
- 1.6 What is memory_pressure ?
- 1.7 What is memory spread ?
- 1.8 How do I use cpusets ?
+ 1.5 What is memory_pressure ?
+ 1.6 What is memory spread ?
+ 1.7 How do I use cpusets ?
 2. Usage Examples and Syntax
   2.1 Basic Usage
   2.2 Adding/removing cpus

@@ -43,18 +43,19 @@ hierarchy visible in a virtual file syst
 hooks, beyond what is already present, required to manage dynamic
 job placement on large systems.

-Each task has a pointer to a cpuset. Multiple tasks may reference
-the same cpuset. Requests by a task, using the sched_setaffinity(2)
-system call to include CPUs in its CPU affinity mask, and using the
-mbind(2) and set_mempolicy(2) system calls to include Memory Nodes
-in its memory policy, are both filtered through that tasks cpuset,
-filtering out any CPUs or Memory Nodes not in that cpuset. The
-scheduler will not schedule a task on a CPU that is not allowed in
-its cpus_allowed vector, and the kernel page allocator will not
-allocate a page on a node that is not allowed in the requesting tasks
-mems_allowed vector.
+Cpusets use the generic container subsystem described in
+Documentation/container.txt.

-User level code may create and destroy cpusets by name in the cpuset
+Requests by a task, using the sched_setaffinity(2) system call to
+include CPUs in its CPU affinity mask, and using the mbind(2) and
+set_mempolicy(2) system calls to include Memory Nodes in its memory
+policy, are both filtered through that tasks cpuset, filtering out any
+CPUs or Memory Nodes not in that cpuset. The scheduler will not
+schedule a task on a CPU that is not allowed in its cpus_allowed
+vector, and the kernel page allocator will not allocate a page on a
+node that is not allowed in the requesting tasks mems_allowed vector.
+
+User level code may create and destroy cpusets by name in the container
 virtual file system, manage the attributes and permissions of these
 cpusets and which CPUs and Memory Nodes are assigned to each cpuset,
 specify and query to which cpuset a task is assigned, and list the
@@ -114,7 +115,7 @@ Cpusets extends these two mechanisms as
   - Cpusets are sets of allowed CPUs and Memory Nodes, known to the
     kernel.
   - Each task in the system is attached to a cpuset, via a pointer
-    in the task structure to a reference counted cpuset structure.
+   in the task structure to a reference counted container structure.
   - Calls to sched_setaffinity are filtered to just those CPUs
     allowed in that tasks cpuset.
   - Calls to mbind and set_mempolicy are filtered to just
@@ -144,15 +145,10 @@ into the rest of the kernel, none in per
   - in page_alloc.c, to restrict memory to allowed nodes.
   - in vmscan.c, to restrict page recovery to the current cpuset.

-In addition a new file system, of type "cpuset" may be mounted,
-typically at /dev/cpuset, to enable browsing and modifying the cpusets
-presently known to the kernel. No new system calls are added for
-cpusets - all support for querying and modifying cpusets is via

-this cpuset file system.
-
-Each task under /proc has an added file named 'cpuset', displaying
-the cpuset name, as the path relative to the root of the cpuset file
-system.
+You should mount the "container" filesystem type in order to enable
+browsing and modifying the cpusets presently known to the kernel. No
+new system calls are added for cpusets - all support for querying and
+modifying cpusets is via this cpuset file system.

 The /proc/<pid>/status file for each task has two added lines,
 displaying the tasks cpus_allowed (on which CPUs it may be scheduled)
@@ -162,16 +158,15 @@ in the format seen in the following exam
   Cpus_allowed: ffffffff,ffffffff,ffffffff,ffffffff
   Mems_allowed: ffffffff,ffffffff

-Each cpuset is represented by a directory in the cpuset file system
-containing the following files describing that cpuset:
+Each cpuset is represented by a directory in the container file system
+containing (on top of the standard container files) the following
+files describing that cpuset:

  - cpus: list of CPUs in that cpuset
  - mems: list of Memory Nodes in that cpuset
  - memory_migrate flag: if set, move pages to cpusets nodes
  - cpu_exclusive flag: is cpu placement exclusive?
  - mem_exclusive flag: is memory placement exclusive?
- - tasks: list of tasks (by pid) attached to that cpuset
- - notify_on_release flag: run /sbin/cpuset_release_agent on exit?
  - memory_pressure: measure of how much paging pressure in cpuset

 In addition, the root cpuset only has the following file:
@@ -236,21 +231,7 @@ such as requests from interrupt handlers
 outside even a mem_exclusive cpuset.

-1.5 What does notify_on_release do ?

-
-If the notify_on_release flag is enabled (1) in a cpuset, then whenever
-the last task in the cpuset leaves (exits or attaches to some other
-cpuset) and the last child cpuset of that cpuset is removed, then
-the kernel runs the command /sbin/cpuset_release_agent, supplying the
-pathname (relative to the mount point of the cpuset file system) of the
-abandoned cpuset. This enables automatic removal of abandoned cpusets.
-The default value of notify_on_release in the root cpuset at system
-boot is disabled (0). The default value of other cpusets at creation
-is the current value of their parents notify_on_release setting.

-
-
-1.6 What is memory_pressure ?
+1.5 What is memory_pressure ?

 The memory_pressure of a cpuset provides a simple per-cpuset metric
 of the rate that the tasks in a cpuset are attempting to free up in
@@ -307,7 +288,7 @@ the tasks in the cpuset, in units of rec
 times 1000.

-1.7 What is memory spread ?
+1.6 What is memory spread ?

 There are two boolean flag files per cpuset that control where the
 kernel allocates pages for the file system buffers and related in
@@ -378,7 +359,7 @@ data set, the memory allocation across t
 can become very uneven.

-1.8 How do I use cpusets ?
+1.7 How do I use cpusets ?

 In order to minimize the impact of cpusets on critical kernel
@@ -468,7 +449,7 @@ than stress the kernel.
 To start a new job that is to be contained within a cpuset, the steps are:

   1) mkdir /dev/cpuset
- 2) mount -t cpuset none /dev/cpuset
+ 2) mount -t container -ocpuset cpuset /dev/cpuset
   3) Create the new cpuset by doing mkdir's and write's (or echo's) in
      the /dev/cpuset virtual file system.
   4) Start a task that will be the "founding father" of the new job.
@@ -480,7 +461,7 @@ For example, the following sequence of c
 named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
 and then start a subshell 'sh' in that cpuset:

- mount -t cpuset none /dev/cpuset
+ mount -t container -ocpuset cpuset /dev/cpuset
    cd /dev/cpuset
    mkdir Charlie
    cd Charlie
@@ -512,7 +493,7 @@ Creating, modifying, using the cpusets c
 virtual filesystem.

 To mount it, type:
-# mount -t cpuset none /dev/cpuset

```
+# mount -t container -o cpuset cpuset /dev/cpuset

 Then under /dev/cpuset you can find a tree that corresponds to the
 tree of the cpusets in the system. For instance, /dev/cpuset
@@ -555,6 +536,18 @@ To remove a cpuset, just use rmdir:
 This will fail if the cpuset is in use (has cpusets inside, or has
 processes attached).

+Note that for legacy reasons, the "cpuset" filesystem exists as a
+wrapper around the container filesystem.
+
+The command
+
+mount -t cpuset X /dev/cpuset
+
+is equivalent to
+
+mount -t container -ocpuset X /dev/cpuset
+echo "/sbin/cpuset_release_agent" > /dev/cpuset/release_agent
+
 2.2 Adding/removing cpus
 ------------------------

Index: container-2.6.21-rc7-mm1/include/linux/cpuset.h
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/include/linux/cpuset.h
+++ container-2.6.21-rc7-mm1/include/linux/cpuset.h
@@ -11,6 +11,7 @@
 #include <linux/sched.h>
 #include <linux/cpumask.h>
 #include <linux/nodemask.h>
+#include <linux/container.h>

 #ifdef CONFIG_CPUSETS

@@ -19,8 +20,6 @@ extern int number_of_cpusets; /* How man
 extern int cpuset_init_early(void);
 extern int cpuset_init(void);
 extern void cpuset_init_smp(void);
-extern void cpuset_fork(struct task_struct *p);
-extern void cpuset_exit(struct task_struct *p);
 extern cpumask_t cpuset_cpus_allowed(struct task_struct *p);
 extern nodemask_t cpuset_mems_allowed(struct task_struct *p);
 #define cpuset_current_mems_allowed (current->mems_allowed)
@@ -75,13 +74,13 @@ static inline int cpuset_do_slab_mem_spr

 extern void cpuset_track_online_nodes(void);
```

```
+extern int current_cpuset_is_being_rebound(void);
+
 #else /* !CONFIG_CPUSETS */

 static inline int cpuset_init_early(void) { return 0; }
 static inline int cpuset_init(void) { return 0; }
 static inline void cpuset_init_smp(void) {}
-static inline void cpuset_fork(struct task_struct *p) {}
-static inline void cpuset_exit(struct task_struct *p) {}

 static inline cpumask_t cpuset_cpus_allowed(struct task_struct *p)
 {
@@ -146,6 +145,11 @@ static inline int cpuset_do_slab_mem_spr

 static inline void cpuset_track_online_nodes(void) {}

+static inline int current_cpuset_is_being_rebound(void)
+{
+ return 0;
+}
+
 #endif /* !CONFIG_CPUSETS */

 #endif /* _LINUX_CPUSET_H */
Index: container-2.6.21-rc7-mm1/include/linux/mempolicy.h
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/include/linux/mempolicy.h
+++ container-2.6.21-rc7-mm1/include/linux/mempolicy.h
@@ -148,14 +148,6 @@ extern void mpol_rebind_task(struct task
      const nodemask_t *new);
 extern void mpol_rebind_mm(struct mm_struct *mm, nodemask_t *new);
 extern void mpol_fix_fork_child_flag(struct task_struct *p);
-#define set_cpuset_being_rebound(x) (cpuset_being_rebound = (x))
-
-#ifdef CONFIG_CPUSETS
-#define current_cpuset_is_being_rebound() \
-    (cpuset_being_rebound == current->cpuset)
-#else
-#define current_cpuset_is_being_rebound() 0
-#endif

 extern struct mempolicy default_policy;
 extern struct zonelist *huge_zonelist(struct vm_area_struct *vma,
@@ -173,8 +165,6 @@ static inline void check_highest_zone(en
 int do_migrate_pages(struct mm_struct *mm,
  const nodemask_t *from_nodes, const nodemask_t *to_nodes, int flags);

-extern void *cpuset_being_rebound; /* Trigger mpol_copy vma rebind */
```

```
-
 #else

 struct mempolicy {};
@@ -253,8 +243,6 @@ static inline void mpol_fix_fork_child_f
 {
 }

-#define set_cpuset_being_rebound(x) do {} while (0)
-
 static inline struct zonelist *huge_zonelist(struct vm_area_struct *vma,
   unsigned long addr, gfp_t gfp_flags)
 {
Index: container-2.6.21-rc7-mm1/include/linux/sched.h
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/include/linux/sched.h
+++ container-2.6.21-rc7-mm1/include/linux/sched.h
@@ -772,8 +772,6 @@ static inline int above_background_load(
 }

 struct io_context;   /* See blkdev.h */
-struct cpuset;
-
 #define NGROUPS_SMALL  32
 #define NGROUPS_PER_BLOCK ((int)(PAGE_SIZE / sizeof(gid_t)))
 struct group_info {
@@ -1095,7 +1093,6 @@ struct task_struct {
  short il_next;
 #endif
 #ifdef CONFIG_CPUSETS
- struct cpuset *cpuset;
  nodemask_t mems_allowed;
  int cpuset_mems_generation;
  int cpuset_mem_spread_rotor;
Index: container-2.6.21-rc7-mm1/init/Kconfig
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/init/Kconfig
+++ container-2.6.21-rc7-mm1/init/Kconfig
@@ -294,6 +294,7 @@ config CONTAINERS
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
+ select CONTAINERS
  help
    This option will let you create and manage CPUSETs which
    allow dynamically partitioning a system into sets of CPUs and
@@ -329,6 +330,11 @@ config CONTAINER_CPUACCT
    Provides a simple Resource Controller for monitoring the
```

```
    total CPU consumed by the tasks in a container

+config PROC_PID_CPUSET
+ bool "Include legacy /proc/<pid>/cpuset file"
+ depends on CPUSETS
+ default y
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
Index: container-2.6.21-rc7-mm1/kernel/cpuset.c
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/cpuset.c
+++ container-2.6.21-rc7-mm1/kernel/cpuset.c
@@ -5,6 +5,7 @@
  *
  *  Copyright (C) 2003 BULL SA.
  *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *  Copyright (C) 2006 Google, Inc
  *
  *  Portions derived from Patrick Mochel's sysfs code.
  *  sysfs is Copyright (c) 2001-3 Patrick Mochel
@@ -12,6 +13,7 @@
  *  2003-10-10 Written by Simon Derr.
  *  2003-10-22 Updates by Stephen Hemminger.
  *  2004 May-July Rework by Paul Jackson.
+ *  2006 Rework by Paul Menage to use generic containers
  *
  *  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
@@ -53,8 +55,6 @@
 #include <asm/atomic.h>
 #include <linux/mutex.h>

-#define CPUSET_SUPER_MAGIC  0x27e0eb
-
 /*
  * Tracks how many cpusets are currently defined in system.
  * When there is only one cpuset (the root cpuset) we can
@@ -62,6 +62,10 @@
  */
 int number_of_cpusets __read_mostly;

+/* Retrieve the cpuset from a container */
+struct container_subsys cpuset_subsys;
+struct cpuset;
+
 /* See "Frequency meter" comments, below. */
```

```
 struct fmeter {
@@ -72,24 +76,13 @@ struct fmeter {
 };

 struct cpuset {
+ struct container_subsys_state css;
+
  unsigned long flags;  /* "unsigned long" so bitops work */
  cpumask_t cpus_allowed;  /* CPUs allowed to tasks in cpuset */
  nodemask_t mems_allowed; /* Memory Nodes allowed to tasks */

- /*
-  * Count is atomic so can incr (fork) or decr (exit) without a lock.
-  */
- atomic_t count;   /* count tasks using this cpuset */
-
- /*
-  * We link our 'sibling' struct into our parents 'children'.
-  * Our children link their 'sibling' into our 'children'.
-  */
- struct list_head sibling; /* my parents children */
- struct list_head children; /* my children */
-
  struct cpuset *parent;  /* my parent */
- struct dentry *dentry;  /* cpuset fs entry */

  /*
   * Copy of global cpuset_mems_generation as of the most
@@ -100,13 +93,32 @@ struct cpuset {
  struct fmeter fmeter;  /* memory_pressure filter */
 };

+/* Update the cpuset for a container */
+static inline void set_container_cs(struct container *cont, struct cpuset *cs)
+{
+ cont->subsys[cpuset_subsys_id] = &cs->css;
+}
+
+/* Retrieve the cpuset for a container */
+static inline struct cpuset *container_cs(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, cpuset_subsys_id),
+       struct cpuset, css);
+}
+
+/* Retrieve the cpuset for a task */
+static inline struct cpuset *task_cs(struct task_struct *task)
```

```
+{
+ return container_of(task_subsys_state(task, cpuset_subsys_id),
+       struct cpuset, css);
+}
+
+
 /* bits in struct cpuset flags field */
 typedef enum {
  CS_CPU_EXCLUSIVE,
  CS_MEM_EXCLUSIVE,
  CS_MEMORY_MIGRATE,
- CS_REMOVED,
- CS_NOTIFY_ON_RELEASE,
  CS_SPREAD_PAGE,
  CS_SPREAD_SLAB,
 } cpuset_flagbits_t;
@@ -122,16 +134,6 @@ static inline int is_mem_exclusive(const
  return test_bit(CS_MEM_EXCLUSIVE, &cs->flags);
 }

-static inline int is_removed(const struct cpuset *cs)
-{
- return test_bit(CS_REMOVED, &cs->flags);
-}
-
-static inline int notify_on_release(const struct cpuset *cs)
-{
- return test_bit(CS_NOTIFY_ON_RELEASE, &cs->flags);
-}
-
 static inline int is_memory_migrate(const struct cpuset *cs)
 {
  return test_bit(CS_MEMORY_MIGRATE, &cs->flags);
@@ -172,14 +174,8 @@ static struct cpuset top_cpuset = {
  .flags = ((1 << CS_CPU_EXCLUSIVE) | (1 << CS_MEM_EXCLUSIVE)),
  .cpus_allowed = CPU_MASK_ALL,
  .mems_allowed = NODE_MASK_ALL,
- .count = ATOMIC_INIT(0),
- .sibling = LIST_HEAD_INIT(top_cpuset.sibling),
- .children = LIST_HEAD_INIT(top_cpuset.children),
 };

-static struct vfsmount *cpuset_mount;
-static struct super_block *cpuset_sb;
-
 /*
  * We have two global cpuset mutexes below.  They can nest.
  * It is ok to first take manage_mutex, then nest callback_mutex.  We also
```

```
@@ -263,297 +259,31 @@ static struct super_block *cpuset_sb;
  * the routine cpuset_update_task_memory_state().
  */

-static DEFINE_MUTEX(manage_mutex);
 static DEFINE_MUTEX(callback_mutex);

-/*
- * A couple of forward declarations required, due to cyclic reference loop:
- *  cpuset_mkdir -> cpuset_create -> cpuset_populate_dir -> cpuset_add_file
- *  -> cpuset_create_file -> cpuset_dir_inode_operations -> cpuset_mkdir.
- */
-
-static int cpuset_mkdir(struct inode *dir, struct dentry *dentry, int mode);
-static int cpuset_rmdir(struct inode *unused_dir, struct dentry *dentry);
-
-static struct backing_dev_info cpuset_backing_dev_info = {
- .ra_pages = 0,  /* No readahead */
- .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
-};
-
-static struct inode *cpuset_new_inode(mode_t mode)
-{
- struct inode *inode = new_inode(cpuset_sb);
-
- 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 = &cpuset_backing_dev_info;
- }
- return inode;
-}
-
-static void cpuset_diput(struct dentry *dentry, struct inode *inode)
-{
- /* is dentry a directory ? if so, kfree() associated cpuset */
- if (S_ISDIR(inode->i_mode)) {
-  struct cpuset *cs = dentry->d_fsdata;
-  BUG_ON(!(is_removed(cs)));
-  kfree(cs);
- }
- iput(inode);
-}
-
-static struct dentry_operations cpuset_dops = {
```

```
- .d_iput = cpuset_diput,
-};
-
-static struct dentry *cpuset_get_dentry(struct dentry *parent, const char *name)
-{
- struct dentry *d = lookup_one_len(name, parent, strlen(name));
- if (!IS_ERR(d))
-  d->d_op = &cpuset_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);
-}
-
-/*
- * NOTE : the dentry must have been dget()'ed
- */
-static void cpuset_d_remove_dir(struct dentry *dentry)
-{
- struct list_head *node;
-
- 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) {
-   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;
- }
- list_del_init(&dentry->d_u.d_child);
- spin_unlock(&dcache_lock);
- remove_dir(dentry);
-}
-
-static struct super_operations cpuset_ops = {
```

```
- .statfs = simple_statfs,
- .drop_inode = generic_delete_inode,
-};
-
-static int cpuset_fill_super(struct super_block *sb, void *unused_data,
-       int unused_silent)
-{
- struct inode *inode;
- struct dentry *root;
-
- sb->s_blocksize = PAGE_CACHE_SIZE;
- sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
- sb->s_magic = CPUSET_SUPER_MAGIC;
- sb->s_op = &cpuset_ops;
- cpuset_sb = sb;
-
- inode = cpuset_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR);
- if (inode) {
-  inode->i_op = &simple_dir_inode_operations;
-  inode->i_fop = &simple_dir_operations;
-  /* directories start off with i_nlink == 2 (for "." entry) */
-  inc_nlink(inode);
- } else {
-  return -ENOMEM;
- }
-
- root = d_alloc_root(inode);
- if (!root) {
-  iput(inode);
-  return -ENOMEM;
- }
- sb->s_root = root;
- return 0;
-}
-
+/* This is ugly, but preserves the userspace API for existing cpuset
+ * users. If someone tries to mount the "cpuset" filesystem, we
+ * silently switch it to mount "container" instead */
 static int cpuset_get_sb(struct file_system_type *fs_type,
     int flags, const char *unused_dev_name,
     void *data, struct vfsmount *mnt)
 {
- return get_sb_single(fs_type, flags, data, cpuset_fill_super, mnt);
+ struct file_system_type *container_fs = get_fs_type("container");
+ int ret = -ENODEV;
+ if (container_fs) {
+  ret = container_fs->get_sb(container_fs, flags,
+        unused_dev_name,
```

```
+        "cpuset", mnt);
+  put_filesystem(container_fs);
+ }
+ return ret;
 }

 static struct file_system_type cpuset_fs_type = {
  .name = "cpuset",
  .get_sb = cpuset_get_sb,
- .kill_sb = kill_litter_super,
-};
-
-/* struct cftype:
- *
- * The files in the cpuset 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 cpuset to use in file->f_path.dentry->d_parent->d_fsdata
- * - the 'cftype' of the file is file->f_path.dentry->d_fsdata
- */
-
-struct cftype {
- char *name;
- int private;
- int (*open) (struct inode *inode, struct file *file);
- ssize_t (*read) (struct file *file, char __user *buf, size_t nbytes,
-       loff_t *ppos);
- int (*write) (struct file *file, const char __user *buf, size_t nbytes,
-       loff_t *ppos);
- int (*release) (struct inode *inode, struct file *file);
 };

-static inline struct cpuset *__d_cs(struct dentry *dentry)
-{
- return dentry->d_fsdata;
-}
-
-static inline struct cftype *__d_cft(struct dentry *dentry)
-{
- return dentry->d_fsdata;
-}
-
-/*
- * Call with manage_mutex held.  Writes path of cpuset into buf.
```

```
- * Returns 0 on success, -errno on error.
- */
-
-static int cpuset_path(const struct cpuset *cs, char *buf, int buflen)
-{
- char *start;
-
- start = buf + buflen;
-
- *--start = '\0';
- for (;;) {
-  int len = cs->dentry->d_name.len;
-  if ((start -= len) < buf)
-   return -ENAMETOOLONG;
-  memcpy(start, cs->dentry->d_name.name, len);
-  cs = cs->parent;
-  if (!cs)
-   break;
-  if (!cs->parent)
-   continue;
-  if (--start < buf)
-   return -ENAMETOOLONG;
-  *start = '/';
- }
- memmove(buf, start, buf + buflen - start);
- return 0;
-}
-
-/*
- * Notify userspace when a cpuset is released, by running
- * /sbin/cpuset_release_agent with the name of the cpuset (path
- * relative to the root of cpuset file system) as the argument.
- *
- * Most likely, this user command will try to rmdir this cpuset.
- *
- * This races with the possibility that some other task will be
- * attached to this cpuset before it is removed, or that some other
- * user task will 'mkdir' a child cpuset of this cpuset.  That's ok.
- * The presumed 'rmdir' will fail quietly if this cpuset is no longer
- * unused, and this cpuset 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 0, which means don't
- * wait.  The separate /sbin/cpuset_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 /sbin/cpuset_release_agent task, so no sense holding
- * our caller up for that.
- *
- * When we had only one cpuset mutex, we had to call this
- * without holding it, to avoid deadlock when call_usermodehelper()
- * allocated memory.  With two locks, we could now call this while
- * holding manage_mutex, but we still don't, so as to minimize
- * the time manage_mutex is held.
- */
-
-static void cpuset_release_agent(const char *pathbuf)
-{
- char *argv[3], *envp[3];
- int i;
-
- if (!pathbuf)
-  return;
-
- i = 0;
- argv[i++] = "/sbin/cpuset_release_agent";
- 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;
-
- call_usermodehelper(argv[0], argv, envp, 0);
- kfree(pathbuf);
-}
-
-/*
- * Either cs->count of using tasks transitioned to zero, or the
- * cs->children list of child cpusets just became empty.  If this
- * cs is notify_on_release() and now both the user count is zero and
- * the list of children is empty, prepare cpuset path in a kmalloc'd
- * buffer, to be returned via ppathbuf, so that the caller can invoke
- * cpuset_release_agent() with it later on, once manage_mutex is dropped.
- * Call here with manage_mutex held.
- *
- * This check_for_release() routine is responsible for kmalloc'ing
- * pathbuf.  The above cpuset_release_agent() is responsible for
- * kfree'ing pathbuf.  The caller of these routines is responsible
- * for providing a pathbuf pointer, initialized to NULL, then
- * calling check_for_release() with manage_mutex held and the address
- * of the pathbuf pointer, then dropping manage_mutex, then calling
```

```
- * cpuset_release_agent() with pathbuf, as set by check_for_release().
- */
-
-static void check_for_release(struct cpuset *cs, char **ppathbuf)
-{
- if (notify_on_release(cs) && atomic_read(&cs->count) == 0 &&
-     list_empty(&cs->children)) {
-  char *buf;
-
-  buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
-  if (!buf)
-   return;
-  if (cpuset_path(cs, buf, PAGE_SIZE) < 0)
-   kfree(buf);
-  else
-   *ppathbuf = buf;
- }
-}
-
 /*
  * Return in *pmask the portion of a cpusets's cpus_allowed that
  * are online.  If none are online, walk up the cpuset hierarchy
@@ -651,20 +381,19 @@ void cpuset_update_task_memory_state(voi
  struct task_struct *tsk = current;
  struct cpuset *cs;

- if (tsk->cpuset == &top_cpuset) {
+ if (task_cs(tsk) == &top_cpuset) {
   /* Don't need rcu for top_cpuset.  It's never freed. */
   my_cpusets_mem_gen = top_cpuset.mems_generation;
  } else {
   rcu_read_lock();
-  cs = rcu_dereference(tsk->cpuset);
-  my_cpusets_mem_gen = cs->mems_generation;
+  my_cpusets_mem_gen = task_cs(current)->mems_generation;
   rcu_read_unlock();
  }

  if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
   mutex_lock(&callback_mutex);
   task_lock(tsk);
-  cs = tsk->cpuset; /* Maybe changed when task not locked */
+  cs = task_cs(tsk); /* Maybe changed when task not locked */
   guarantee_online_mems(cs, &tsk->mems_allowed);
   tsk->cpuset_mems_generation = cs->mems_generation;
   if (is_spread_page(cs))
@@ -719,11 +448,12 @@ static int is_cpuset_subset(const struct
```

```
 static int validate_change(const struct cpuset *cur, const struct cpuset *trial)
 {
+ struct container *cont;
  struct cpuset *c, *par;

  /* Each of our child cpusets must be a subset of us */
- list_for_each_entry(c, &cur->children, sibling) {
-  if (!is_cpuset_subset(c, trial))
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+  if (!is_cpuset_subset(container_cs(cont), trial))
    return -EBUSY;
  }

@@ -738,7 +468,8 @@ static int validate_change(const struct
   return -EACCES;

  /* If either I or some sibling (!= me) is exclusive, we can't overlap */
- list_for_each_entry(c, &par->children, sibling) {
+ list_for_each_entry(cont, &par->css.container->children, sibling) {
+  c = container_cs(cont);
   if ((is_cpu_exclusive(trial) || is_cpu_exclusive(c)) &&
       c != cur &&
       cpus_intersects(trial->cpus_allowed, c->cpus_allowed))
@@ -826,7 +557,7 @@ static void cpuset_migrate_mm(struct mm_
  do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);

  mutex_lock(&callback_mutex);
- guarantee_online_mems(tsk->cpuset, &tsk->mems_allowed);
+ guarantee_online_mems(task_cs(tsk),&tsk->mems_allowed);
  mutex_unlock(&callback_mutex);
 }

@@ -844,6 +575,8 @@ static void cpuset_migrate_mm(struct mm_
  * their mempolicies to the cpusets new mems_allowed.
  */

+static void *cpuset_being_rebound;
+
 static int update_nodemask(struct cpuset *cs, char *buf)
 {
  struct cpuset trialcs;
@@ -854,12 +587,14 @@ static int update_nodemask(struct cpuset
  int migrate;
  int fudge;
  int retval;
+ struct container *cont;

  /* top_cpuset.mems_allowed tracks node_online_map; it's read-only */
```

```
  if (cs == &top_cpuset)
   return -EACCES;

  trialcs = *cs;
+ cont = cs->css.container;
  retval = nodelist_parse(buf, trialcs.mems_allowed);
  if (retval < 0)
   goto done;
@@ -882,7 +617,7 @@ static int update_nodemask(struct cpuset
  cs->mems_generation = cpuset_mems_generation++;
  mutex_unlock(&callback_mutex);

- set_cpuset_being_rebound(cs);  /* causes mpol_copy() rebind */
+ cpuset_being_rebound = cs;  /* causes mpol_copy() rebind */

  fudge = 10;    /* spare mmarray[] slots */
  fudge += cpus_weight(cs->cpus_allowed); /* imagine one fork-bomb/cpu */
@@ -896,13 +631,13 @@ static int update_nodemask(struct cpuset
   * enough mmarray[] w/o using GFP_ATOMIC.
   */
  while (1) {
-  ntasks = atomic_read(&cs->count); /* guess */
+  ntasks = container_task_count(cs->css.container);  /* guess */
   ntasks += fudge;
   mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
   if (!mmarray)
    goto done;
   write_lock_irq(&tasklist_lock);  /* block fork */
-  if (atomic_read(&cs->count) <= ntasks)
+  if (container_task_count(cs->css.container) <= ntasks)
    break;    /* got enough */
   write_unlock_irq(&tasklist_lock); /* try again */
   kfree(mmarray);
@@ -919,7 +654,7 @@ static int update_nodemask(struct cpuset
     "Cpuset mempolicy rebind incomplete.\n");
    continue;
   }
-  if (p->cpuset != cs)
+  if (task_cs(p) != cs)
    continue;
   mm = get_task_mm(p);
   if (!mm)
@@ -953,12 +688,17 @@ static int update_nodemask(struct cpuset

  /* We're done rebinding vma's to this cpusets new mems_allowed. */
  kfree(mmarray);
- set_cpuset_being_rebound(NULL);
+ cpuset_being_rebound = NULL;
```

```
  retval = 0;
 done:
  return retval;
 }

+int current_cpuset_is_being_rebound(void)
+{
+ return task_cs(current) == cpuset_being_rebound;
+}
+
 /*
  * Call with manage_mutex held.
  */
@@ -1105,85 +845,34 @@ static int fmeter_getrate(struct fmeter
  return val;
 }

-/*
- * Attack task specified by pid in 'pidbuf' to cpuset 'cs', possibly
- * writing the path of the old cpuset in 'ppathbuf' if it needs to be
- * notified on release.
- *
- * Call holding manage_mutex.  May take callback_mutex and task_lock of
- * the task 'pid' during call.
- */
-
-static int attach_task(struct cpuset *cs, char *pidbuf, char **ppathbuf)
+int cpuset_can_attach(struct container_subsys *ss,
+        struct container *cont, struct task_struct *tsk)
 {
- pid_t pid;
- struct task_struct *tsk;
- struct cpuset *oldcs;
- cpumask_t cpus;
- nodemask_t from, to;
- struct mm_struct *mm;
- int retval;
+ struct cpuset *cs = container_cs(cont);

- if (sscanf(pidbuf, "%d", &pid) != 1)
-  return -EIO;
  if (cpus_empty(cs->cpus_allowed) || nodes_empty(cs->mems_allowed))
   return -ENOSPC;

- if (pid) {
-  read_lock(&tasklist_lock);
-
-  tsk = find_task_by_pid(pid);
```

```
-  if (!tsk || tsk->flags & PF_EXITING) {
-   read_unlock(&tasklist_lock);
-   return -ESRCH;
-  }
-
-  get_task_struct(tsk);
-  read_unlock(&tasklist_lock);
-
-  if ((current->euid) && (current->euid != tsk->uid)
-      && (current->euid != tsk->suid)) {
-   put_task_struct(tsk);
-   return -EACCES;
-  }
- } else {
-  tsk = current;
-  get_task_struct(tsk);
- }
+ return security_task_setscheduler(tsk, 0, NULL);
+}

- retval = security_task_setscheduler(tsk, 0, NULL);
- if (retval) {
-  put_task_struct(tsk);
-  return retval;
- }
+void cpuset_attach(struct container_subsys *ss,
+      struct container *cont, struct container *oldcont,
+      struct task_struct *tsk)
+{
+ cpumask_t cpus;
+ nodemask_t from, to;
+ struct mm_struct *mm;
+ struct cpuset *cs = container_cs(cont);
+ struct cpuset *oldcs = container_cs(oldcont);

  mutex_lock(&callback_mutex);
-
- task_lock(tsk);
- oldcs = tsk->cpuset;
- /*
-  * After getting 'oldcs' cpuset ptr, be sure still not exiting.
-  * If 'oldcs' might be the top_cpuset due to the_top_cpuset_hack
-  * then fail this attach_task(), to avoid breaking top_cpuset.count.
-  */
- if (tsk->flags & PF_EXITING) {
-  task_unlock(tsk);
-  mutex_unlock(&callback_mutex);
-  put_task_struct(tsk);
```

```
-  return -ESRCH;
- }
- atomic_inc(&cs->count);
- rcu_assign_pointer(tsk->cpuset, cs);
- task_unlock(tsk);
-
  guarantee_online_cpus(cs, &cpus);
  set_cpus_allowed(tsk, cpus);
+ mutex_unlock(&callback_mutex);

  from = oldcs->mems_allowed;
  to = cs->mems_allowed;
-
- mutex_unlock(&callback_mutex);
-
  mm = get_task_mm(tsk);
  if (mm) {
   mpol_rebind_mm(mm, &to);
@@ -1192,40 +881,31 @@ static int attach_task(struct cpuset *cs
   mmput(mm);
  }

- put_task_struct(tsk);
- synchronize_rcu();
- if (atomic_dec_and_test(&oldcs->count))
-  check_for_release(oldcs, ppathbuf);
- return 0;
 }

 /* The various types of files and directories in a cpuset file system */

 typedef enum {
- FILE_ROOT,
- FILE_DIR,
  FILE_MEMORY_MIGRATE,
  FILE_CPULIST,
  FILE_MEMLIST,
  FILE_CPU_EXCLUSIVE,
  FILE_MEM_EXCLUSIVE,
- FILE_NOTIFY_ON_RELEASE,
  FILE_MEMORY_PRESSURE_ENABLED,
  FILE_MEMORY_PRESSURE,
  FILE_SPREAD_PAGE,
  FILE_SPREAD_SLAB,
- FILE_TASKLIST,
 } cpuset_filetype_t;

-static ssize_t cpuset_common_file_write(struct file *file,
```

```
+static ssize_t cpuset_common_file_write(struct container *cont,
+     struct cftype *cft,
+     struct file *file,
      const char __user *userbuf,
      size_t nbytes, loff_t *unused_ppos)
 {
- struct cpuset *cs = __d_cs(file->f_path.dentry->d_parent);
- struct cftype *cft = __d_cft(file->f_path.dentry);
+ struct cpuset *cs = container_cs(cont);
  cpuset_filetype_t type = cft->private;
  char *buffer;
- char *pathbuf = NULL;
  int retval = 0;

  /* Crude upper limit on largest legitimate cpulist user might write. */
@@ -1242,9 +922,9 @@ static ssize_t cpuset_common_file_write(
  }
  buffer[nbytes] = 0; /* nul-terminate */

- mutex_lock(&manage_mutex);
+ container_lock();

- if (is_removed(cs)) {
+ if (container_is_removed(cont)) {
   retval = -ENODEV;
   goto out2;
  }
@@ -1262,9 +942,6 @@ static ssize_t cpuset_common_file_write(
  case FILE_MEM_EXCLUSIVE:
   retval = update_flag(CS_MEM_EXCLUSIVE, cs, buffer);
   break;
- case FILE_NOTIFY_ON_RELEASE:
-  retval = update_flag(CS_NOTIFY_ON_RELEASE, cs, buffer);
-  break;
  case FILE_MEMORY_MIGRATE:
   retval = update_flag(CS_MEMORY_MIGRATE, cs, buffer);
   break;
@@ -1282,9 +959,6 @@ static ssize_t cpuset_common_file_write(
   retval = update_flag(CS_SPREAD_SLAB, cs, buffer);
   cs->mems_generation = cpuset_mems_generation++;
   break;
- case FILE_TASKLIST:
-  retval = attach_task(cs, buffer, &pathbuf);
-  break;
  default:
   retval = -EINVAL;
   goto out2;
@@ -1293,30 +967,12 @@ static ssize_t cpuset_common_file_write(
```

```
  if (retval == 0)
   retval = nbytes;
 out2:
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
+ container_unlock();
 out1:
  kfree(buffer);
  return retval;
 }

-static ssize_t cpuset_file_write(struct file *file, const char __user *buf,
-      size_t nbytes, loff_t *ppos)
-{
- ssize_t retval = 0;
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (!cft)
-  return -ENODEV;
-
- /* special function ? */
- if (cft->write)
-  retval = cft->write(file, buf, nbytes, ppos);
- else
-  retval = cpuset_common_file_write(file, buf, nbytes, ppos);
-
- return retval;
-}
-
 /*
  * These ascii lists should be read in a single call, by using a user
  * buffer large enough to hold the entire map.  If read in smaller
@@ -1351,11 +1007,13 @@ static int cpuset_sprintf_memlist(char *
  return nodelist_scnprintf(page, PAGE_SIZE, mask);
 }

-static ssize_t cpuset_common_file_read(struct file *file, char __user *buf,
-    size_t nbytes, loff_t *ppos)
+static ssize_t cpuset_common_file_read(struct container *cont,
+           struct cftype *cft,
+           struct file *file,
+           char __user *buf,
+           size_t nbytes, loff_t *ppos)
 {
- struct cftype *cft = __d_cft(file->f_path.dentry);
- struct cpuset *cs = __d_cs(file->f_path.dentry->d_parent);
+ struct cpuset *cs = container_cs(cont);
  cpuset_filetype_t type = cft->private;
  char *page;
```

```
  ssize_t retval = 0;
@@ -1379,9 +1037,6 @@ static ssize_t cpuset_common_file_read(s
  case FILE_MEM_EXCLUSIVE:
   *s++ = is_mem_exclusive(cs) ? '1' : '0';
   break;
- case FILE_NOTIFY_ON_RELEASE:
-  *s++ = notify_on_release(cs) ? '1' : '0';
-  break;
  case FILE_MEMORY_MIGRATE:
   *s++ = is_memory_migrate(cs) ? '1' : '0';
   break;
@@ -1409,391 +1064,100 @@ out:
  return retval;
 }

-static ssize_t cpuset_file_read(struct file *file, char __user *buf, size_t nbytes,
-        loff_t *ppos)
-{
- ssize_t retval = 0;
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (!cft)
-  return -ENODEV;
-
- /* special function ? */
- if (cft->read)
-  retval = cft->read(file, buf, nbytes, ppos);
- else
-  retval = cpuset_common_file_read(file, buf, nbytes, ppos);
-
- return retval;
-}
-
-static int cpuset_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_path.dentry);
- if (!cft)
-  return -ENODEV;
- if (cft->open)
-  err = cft->open(inode, file);
- else
-  err = 0;
```

```
-
- return err;
-}
-
-static int cpuset_file_release(struct inode *inode, struct file *file)
-{
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (cft->release)
-  return cft->release(inode, file);
- return 0;
-}
-
-/*
- * cpuset_rename - Only allow simple rename of directories in place.
- */
-static int cpuset_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 const struct file_operations cpuset_file_operations = {
- .read = cpuset_file_read,
- .write = cpuset_file_write,
- .llseek = generic_file_llseek,
- .open = cpuset_file_open,
- .release = cpuset_file_release,
-};
-
-static const struct inode_operations cpuset_dir_inode_operations = {
- .lookup = simple_lookup,
- .mkdir = cpuset_mkdir,
- .rmdir = cpuset_rmdir,
- .rename = cpuset_rename,
-};
-
-static int cpuset_create_file(struct dentry *dentry, int mode)
-{
- struct inode *inode;
-
- if (!dentry)
-  return -ENOENT;
```

```
- if (dentry->d_inode)
-  return -EEXIST;
-
- inode = cpuset_new_inode(mode);
- if (!inode)
-  return -ENOMEM;
-
- if (S_ISDIR(mode)) {
-  inode->i_op = &cpuset_dir_inode_operations;
-  inode->i_fop = &simple_dir_operations;
-
-  /* start off with i_nlink == 2 (for "." entry) */
-  inc_nlink(inode);
- } else if (S_ISREG(mode)) {
-  inode->i_size = 0;
-  inode->i_fop = &cpuset_file_operations;
- }
-
- d_instantiate(dentry, inode);
- dget(dentry); /* Extra count - pin the dentry in core */
- return 0;
-}
-
-/*
- * cpuset_create_dir - create a directory for an object.
- * cs: the cpuset 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 cpuset directory. Will be copied.
- * mode: mode to set on new directory.
- */
-
-static int cpuset_create_dir(struct cpuset *cs, const char *name, int mode)
-{
- struct dentry *dentry = NULL;
- struct dentry *parent;
- int error = 0;
-
- parent = cs->parent->dentry;
- dentry = cpuset_get_dentry(parent, name);
- if (IS_ERR(dentry))
-  return PTR_ERR(dentry);
- error = cpuset_create_file(dentry, S_IFDIR | mode);
- if (!error) {
-  dentry->d_fsdata = cs;
-  inc_nlink(parent->d_inode);
-  cs->dentry = dentry;
- }
```

```
- dput(dentry);
-
- return error;
-}
-
-static int cpuset_add_file(struct dentry *dir, const struct cftype *cft)
-{
- struct dentry *dentry;
- int error;
-
- mutex_lock(&dir->d_inode->i_mutex);
- dentry = cpuset_get_dentry(dir, cft->name);
- if (!IS_ERR(dentry)) {
-  error = cpuset_create_file(dentry, 0644 | S_IFREG);
-  if (!error)
-   dentry->d_fsdata = (void *)cft;
-  dput(dentry);
- } else
-  error = PTR_ERR(dentry);
- mutex_unlock(&dir->d_inode->i_mutex);
- return error;
-}
-
-/*
- * Stuff for reading the 'tasks' file.
- *
- * Reading this file can return large amounts of data if a cpuset 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().
- */
-
-/* cpusets_tasks_read array */
-
-struct ctr_struct {
- char *buf;
- int bufsz;
-};
-
-/*
- * Load into 'pidarray' up to 'npids' of the tasks using cpuset 'cs'.
- * Return actual number of pids loaded.  No need to task_lock(p)
```

```
- * when reading out p->cpuset, as we don't really care if it changes
- * on the next cycle, and we are not going to try to dereference it.
- */
-static int pid_array_load(pid_t *pidarray, int npids, struct cpuset *cs)
-{
- int n = 0;
- struct task_struct *g, *p;
-
- read_lock(&tasklist_lock);
-
- do_each_thread(g, p) {
-  if (p->cpuset == cs) {
-   pidarray[n++] = p->pid;
-   if (unlikely(n == npids))
-    goto array_full;
-  }
- } while_each_thread(g, p);
-
-array_full:
- read_unlock(&tasklist_lock);
- 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 cpuset being opened.
- *
- * Does not require any specific cpuset mutexes, and does not take any.
```

```
- */
-static int cpuset_tasks_open(struct inode *unused, struct file *file)
-{
- struct cpuset *cs = __d_cs(file->f_path.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 cpuset 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 cpuset users didn't
-  * show up until sometime later on.
-  */
- npids = atomic_read(&cs->count);
- pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
- if (!pidarray)
-  goto err1;
-
- npids = pid_array_load(pidarray, npids, cs);
- 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);
- file->private_data = ctr;
- return 0;
-
-err2:
- kfree(pidarray);
-err1:
- kfree(ctr);
-err0:
- return -ENOMEM;
-}
```

```
-
-static ssize_t cpuset_tasks_read(struct file *file, char __user *buf,
-      size_t nbytes, loff_t *ppos)
-{
- struct ctr_struct *ctr = file->private_data;

- if (*ppos + nbytes > ctr->bufsz)
-  nbytes = ctr->bufsz - *ppos;
- if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
-  return -EFAULT;
- *ppos += nbytes;
- return nbytes;
-}

-static int cpuset_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 = cpuset_tasks_open,
- .read = cpuset_tasks_read,
- .release = cpuset_tasks_release,
- .private = FILE_TASKLIST,
-};
-
 static struct cftype cft_cpus = {
  .name = "cpus",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_CPULIST,
 };

 static struct cftype cft_mems = {
  .name = "mems",
+ .read = cpuset_common_file_read,
```

```
+ .write = cpuset_common_file_write,
  .private = FILE_MEMLIST,
 };

 static struct cftype cft_cpu_exclusive = {
  .name = "cpu_exclusive",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_CPU_EXCLUSIVE,
 };

 static struct cftype cft_mem_exclusive = {
  .name = "mem_exclusive",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_MEM_EXCLUSIVE,
 };

-static struct cftype cft_notify_on_release = {
- .name = "notify_on_release",
- .private = FILE_NOTIFY_ON_RELEASE,
-};
-
 static struct cftype cft_memory_migrate = {
  .name = "memory_migrate",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_MEMORY_MIGRATE,
 };

 static struct cftype cft_memory_pressure_enabled = {
  .name = "memory_pressure_enabled",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_MEMORY_PRESSURE_ENABLED,
 };

 static struct cftype cft_memory_pressure = {
  .name = "memory_pressure",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_MEMORY_PRESSURE,
 };

 static struct cftype cft_spread_page = {
  .name = "memory_spread_page",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
```

```
  .private = FILE_SPREAD_PAGE,
 };

 static struct cftype cft_spread_slab = {
  .name = "memory_spread_slab",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_SPREAD_SLAB,
 };

-static int cpuset_populate_dir(struct dentry *cs_dentry)
+int cpuset_populate(struct container_subsys *ss, struct container *cont)
 {
  int err;

- if ((err = cpuset_add_file(cs_dentry, &cft_cpus)) < 0)
-  return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_mems)) < 0)
+ if ((err = container_add_file(cont, &cft_cpus)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_cpu_exclusive)) < 0)
+ if ((err = container_add_file(cont, &cft_mems)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_mem_exclusive)) < 0)
+ if ((err = container_add_file(cont, &cft_cpu_exclusive)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_notify_on_release)) < 0)
+ if ((err = container_add_file(cont, &cft_mem_exclusive)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_memory_migrate)) < 0)
+ if ((err = container_add_file(cont, &cft_memory_migrate)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_memory_pressure)) < 0)
+ if ((err = container_add_file(cont, &cft_memory_pressure)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_spread_page)) < 0)
+ if ((err = container_add_file(cont, &cft_spread_page)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_spread_slab)) < 0)
-  return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_tasks)) < 0)
+ if ((err = container_add_file(cont, &cft_spread_slab)) < 0)
   return err;
+ /* memory_pressure_enabled is in root cpuset only */
+ if (err == 0 && !cont->parent)
+  err = container_add_file(cont, &cft_memory_pressure_enabled);
  return 0;
 }
```

```
@@ -1806,106 +1170,61 @@ static int cpuset_populate_dir(struct de
  * Must be called with the mutex on the parent inode held
  */

-static long cpuset_create(struct cpuset *parent, const char *name, int mode)
+int cpuset_create(struct container_subsys *ss, struct container *cont)
 {
  struct cpuset *cs;
- int err;
+ struct cpuset *parent;

+ if (!cont->parent) {
+  /* This is early initialization for the top container */
+  set_container_cs(cont, &top_cpuset);
+  top_cpuset.css.container = cont;
+  top_cpuset.mems_generation = cpuset_mems_generation++;
+  return 0;
+ }
+ parent = container_cs(cont->parent);
  cs = kmalloc(sizeof(*cs), GFP_KERNEL);
  if (!cs)
   return -ENOMEM;

- mutex_lock(&manage_mutex);
  cpuset_update_task_memory_state();
  cs->flags = 0;
- if (notify_on_release(parent))
-  set_bit(CS_NOTIFY_ON_RELEASE, &cs->flags);
  if (is_spread_page(parent))
   set_bit(CS_SPREAD_PAGE, &cs->flags);
  if (is_spread_slab(parent))
   set_bit(CS_SPREAD_SLAB, &cs->flags);
  cs->cpus_allowed = CPU_MASK_NONE;
  cs->mems_allowed = NODE_MASK_NONE;
- atomic_set(&cs->count, 0);
- INIT_LIST_HEAD(&cs->sibling);
- INIT_LIST_HEAD(&cs->children);
  cs->mems_generation = cpuset_mems_generation++;
  fmeter_init(&cs->fmeter);

  cs->parent = parent;
-
- mutex_lock(&callback_mutex);
- list_add(&cs->sibling, &cs->parent->children);
+ set_container_cs(cont, cs);
+ cs->css.container = cont;
  number_of_cpusets++;
```

```
- mutex_unlock(&callback_mutex);
-
- err = cpuset_create_dir(cs, name, mode);
- if (err < 0)
-  goto err;
-
- /*
-  * Release manage_mutex before cpuset_populate_dir() because it
-  * will down() this new directory's i_mutex and if we race with
-  * another mkdir, we might deadlock.
-  */
- mutex_unlock(&manage_mutex);
-
- err = cpuset_populate_dir(cs->dentry);
- /* If err < 0, we have a half-filled directory - oh well ;) */
  return 0;
-err:
- list_del(&cs->sibling);
- mutex_unlock(&manage_mutex);
- kfree(cs);
- return err;
 }

-static int cpuset_mkdir(struct inode *dir, struct dentry *dentry, int mode)
+void cpuset_destroy(struct container_subsys *ss, struct container *cont)
 {
- struct cpuset *c_parent = dentry->d_parent->d_fsdata;
-
- /* the vfs holds inode->i_mutex already */
- return cpuset_create(c_parent, dentry->d_name.name, mode | S_IFDIR);
-}
-
-static int cpuset_rmdir(struct inode *unused_dir, struct dentry *dentry)
-{
- struct cpuset *cs = dentry->d_fsdata;
- struct dentry *d;
- struct cpuset *parent;
- char *pathbuf = NULL;
-
- /* the vfs holds both inode->i_mutex already */
+ struct cpuset *cs = container_cs(cont);

- mutex_lock(&manage_mutex);
  cpuset_update_task_memory_state();
- if (atomic_read(&cs->count) > 0) {
-  mutex_unlock(&manage_mutex);
-  return -EBUSY;
- }
```

```
- if (!list_empty(&cs->children)) {
-  mutex_unlock(&manage_mutex);
-  return -EBUSY;
- }
- parent = cs->parent;
- mutex_lock(&callback_mutex);
- set_bit(CS_REMOVED, &cs->flags);
- list_del(&cs->sibling); /* delete my sibling from parent->children */
- spin_lock(&cs->dentry->d_lock);
- d = dget(cs->dentry);
- cs->dentry = NULL;
- spin_unlock(&d->d_lock);
- cpuset_d_remove_dir(d);
- dput(d);
  number_of_cpusets--;
- mutex_unlock(&callback_mutex);
- if (list_empty(&parent->children))
-  check_for_release(parent, &pathbuf);
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
- return 0;
+ kfree(cs);
 }

+struct container_subsys cpuset_subsys = {
+ .name = "cpuset",
+ .create = cpuset_create,
+ .destroy  = cpuset_destroy,
+ .can_attach = cpuset_can_attach,
+ .attach = cpuset_attach,
+ .populate = cpuset_populate,
+ .subsys_id = cpuset_subsys_id,
+ .early_init = 1,
+};
+
 /*
  * cpuset_init_early - just enough so that the calls to
  * cpuset_update_task_memory_state() in early init code
@@ -1914,13 +1233,11 @@ static int cpuset_rmdir(struct inode *un

 int __init cpuset_init_early(void)
 {
- struct task_struct *tsk = current;
-
- tsk->cpuset = &top_cpuset;
- tsk->cpuset->mems_generation = cpuset_mems_generation++;
+ top_cpuset.mems_generation = cpuset_mems_generation++;
  return 0;
```

```
 }

+
 /**
  * cpuset_init - initialize cpusets at system boot
  *
@@ -1929,8 +1246,7 @@ int __init cpuset_init_early(void)

 int __init cpuset_init(void)
 {
- struct dentry *root;
- int err;
+ int err = 0;

  top_cpuset.cpus_allowed = CPU_MASK_ALL;
  top_cpuset.mems_allowed = NODE_MASK_ALL;
@@ -1938,30 +1254,12 @@ int __init cpuset_init(void)
  fmeter_init(&top_cpuset.fmeter);
  top_cpuset.mems_generation = cpuset_mems_generation++;

- init_task.cpuset = &top_cpuset;
-
  err = register_filesystem(&cpuset_fs_type);
  if (err < 0)
-  goto out;
- cpuset_mount = kern_mount(&cpuset_fs_type);
- if (IS_ERR(cpuset_mount)) {
-  printk(KERN_ERR "cpuset: could not mount!\n");
-  err = PTR_ERR(cpuset_mount);
-  cpuset_mount = NULL;
-  goto out;
- }
- root = cpuset_mount->mnt_sb->s_root;
- root->d_fsdata = &top_cpuset;
- inc_nlink(root->d_inode);
- top_cpuset.dentry = root;
- root->d_inode->i_op = &cpuset_dir_inode_operations;
+  return err;
+
  number_of_cpusets = 1;
- err = cpuset_populate_dir(root);
- /* memory_pressure_enabled is in root cpuset only */
- if (err == 0)
-  err = cpuset_add_file(root, &cft_memory_pressure_enabled);
-out:
- return err;
+ return 0;
 }
```

```
 /*
@@ -1987,10 +1285,12 @@ out:

 static void guarantee_online_cpus_mems_in_subtree(const struct cpuset *cur)
 {
+ struct container *cont;
  struct cpuset *c;

  /* Each of our child cpusets mems must be online */
- list_for_each_entry(c, &cur->children, sibling) {
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+  c = container_cs(cont);
   guarantee_online_cpus_mems_in_subtree(c);
   if (!cpus_empty(c->cpus_allowed))
    guarantee_online_cpus(c, &c->cpus_allowed);
@@ -2017,7 +1317,7 @@ static void guarantee_online_cpus_mems_i

 static void common_cpu_mem_hotplug_unplug(void)
 {
- mutex_lock(&manage_mutex);
+ container_lock();
  mutex_lock(&callback_mutex);

  guarantee_online_cpus_mems_in_subtree(&top_cpuset);
@@ -2025,7 +1325,7 @@ static void common_cpu_mem_hotplug_unplu
  top_cpuset.mems_allowed = node_online_map;

  mutex_unlock(&callback_mutex);
- mutex_unlock(&manage_mutex);
+ container_unlock();
 }

 /*
@@ -2073,109 +1373,7 @@ void __init cpuset_init_smp(void)
 }

 /**
- * cpuset_fork - attach newly forked task to its parents cpuset.
- * @tsk: pointer to task_struct of forking parent process.
- *
- * Description: A task inherits its parent's cpuset at fork().
- *
- * A pointer to the shared cpuset was automatically copied in fork.c
- * by dup_task_struct().  However, we ignore that copy, since it was
- * not made under the protection of task_lock(), so might no longer be
- * a valid cpuset pointer.  attach_task() might have already changed
- * current->cpuset, allowing the previously referenced cpuset to
```

```
- * be removed and freed.  Instead, we task_lock(current) and copy
- * its present value of current->cpuset for our freshly forked child.
- *
- * At the point that cpuset_fork() is called, 'current' is the parent
- * task, and the passed argument 'child' points to the child task.
- **/
-
-void cpuset_fork(struct task_struct *child)
-{
- task_lock(current);
- child->cpuset = current->cpuset;
- atomic_inc(&child->cpuset->count);
- task_unlock(current);
-}
-
-/**
- * cpuset_exit - detach cpuset from exiting task
- * @tsk: pointer to task_struct of exiting process
- *
- * Description: Detach cpuset from @tsk and release it.
- *
- * Note that cpusets marked notify_on_release force every task in
- * them to take the global manage_mutex mutex when exiting.
- * This could impact scaling on very large systems.  Be reluctant to
- * use notify_on_release cpusets where very high task exit scaling
- * is required on large systems.
- *
- * Don't even think about derefencing 'cs' after the cpuset use count
- * goes to zero, except inside a critical section guarded by manage_mutex
- * or callback_mutex.   Otherwise a zero cpuset use count is a license to
- * any other task to nuke the cpuset immediately, via cpuset_rmdir().
- *
- * This routine has to take manage_mutex, not callback_mutex, because
- * it is holding that mutex while calling check_for_release(),
- * which calls kmalloc(), so can't be called holding callback_mutex().
- *
- * the_top_cpuset_hack:
- *
- *    Set the exiting tasks cpuset to the root cpuset (top_cpuset).
- *
- *    Don't leave a task unable to allocate memory, as that is an
- *    accident waiting to happen should someone add a callout in
- *    do_exit() after the cpuset_exit() call that might allocate.
- *    If a task tries to allocate memory with an invalid cpuset,
- *    it will oops in cpuset_update_task_memory_state().
- *
- *    We call cpuset_exit() while the task is still competent to
- *    handle notify_on_release(), then leave the task attached to
```

```
- *    the root cpuset (top_cpuset) for the remainder of its exit.
- *
- *    To do this properly, we would increment the reference count on
- *    top_cpuset, and near the very end of the kernel/exit.c do_exit()
- *    code we would add a second cpuset function call, to drop that
- *    reference.  This would just create an unnecessary hot spot on
- *    the top_cpuset reference count, to no avail.
- *
- *    Normally, holding a reference to a cpuset without bumping its
- *    count is unsafe.   The cpuset could go away, or someone could
- *    attach us to a different cpuset, decrementing the count on
- *    the first cpuset that we never incremented.  But in this case,
- *    top_cpuset isn't going away, and either task has PF_EXITING set,
- *    which wards off any attach_task() attempts, or task is a failed
- *    fork, never visible to attach_task.
- *
- *    Another way to do this would be to set the cpuset pointer
- *    to NULL here, and check in cpuset_update_task_memory_state()
- *    for a NULL pointer.  This hack avoids that NULL check, for no
- *    cost (other than this way too long comment ;).
- **/

-void cpuset_exit(struct task_struct *tsk)
-{
- struct cpuset *cs;
-
- task_lock(current);
- cs = tsk->cpuset;
- tsk->cpuset = &top_cpuset; /* the_top_cpuset_hack - see above */
- task_unlock(current);
-
- if (notify_on_release(cs)) {
-  char *pathbuf = NULL;
-
-  mutex_lock(&manage_mutex);
-  if (atomic_dec_and_test(&cs->count))
-   check_for_release(cs, &pathbuf);
-  mutex_unlock(&manage_mutex);
-  cpuset_release_agent(pathbuf);
- } else {
-  atomic_dec(&cs->count);
- }
-}
-
-/**
  * cpuset_cpus_allowed - return cpus_allowed mask from a tasks cpuset.
  * @tsk: pointer to task_struct from which to obtain cpuset->cpus_allowed.
  *
```

```
@@ -2191,7 +1389,7 @@ cpumask_t cpuset_cpus_allowed(struct tas

  mutex_lock(&callback_mutex);
  task_lock(tsk);
- guarantee_online_cpus(tsk->cpuset, &mask);
+ guarantee_online_cpus(task_cs(tsk), &mask);
  task_unlock(tsk);
  mutex_unlock(&callback_mutex);

@@ -2219,7 +1417,7 @@ nodemask_t cpuset_mems_allowed(struct ta

  mutex_lock(&callback_mutex);
  task_lock(tsk);
- guarantee_online_mems(tsk->cpuset, &mask);
+ guarantee_online_mems(task_cs(tsk), &mask);
  task_unlock(tsk);
  mutex_unlock(&callback_mutex);

@@ -2350,7 +1548,7 @@ int __cpuset_zone_allowed_softwall(struc
  mutex_lock(&callback_mutex);

  task_lock(current);
- cs = nearest_exclusive_ancestor(current->cpuset);
+ cs = nearest_exclusive_ancestor(task_cs(current));
  task_unlock(current);

  allowed = node_isset(node, cs->mems_allowed);
@@ -2487,7 +1685,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock(current);
   goto done;
  }
- cs1 = nearest_exclusive_ancestor(current->cpuset);
+ cs1 = nearest_exclusive_ancestor(task_cs(current));
  task_unlock(current);

  task_lock((struct task_struct *)p);
@@ -2495,7 +1693,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock((struct task_struct *)p);
   goto done;
  }
- cs2 = nearest_exclusive_ancestor(p->cpuset);
+ cs2 = nearest_exclusive_ancestor(task_cs((struct task_struct *)p));
  task_unlock((struct task_struct *)p);

  overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
@@ -2531,14 +1729,12 @@ int cpuset_memory_pressure_enabled __rea

 void __cpuset_memory_pressure_bump(void)
```

```
 {
- struct cpuset *cs;
-
  task_lock(current);
- cs = current->cpuset;
- fmeter_markevent(&cs->fmeter);
+ fmeter_markevent(&task_cs(current)->fmeter);
  task_unlock(current);
 }

+#ifdef CONFIG_PROC_PID_CPUSET
 /*
  * proc_cpuset_show()
  *  - Print tasks cpuset path into seq_file.
@@ -2555,6 +1751,7 @@ static int proc_cpuset_show(struct seq_f
  struct pid *pid;
  struct task_struct *tsk;
  char *buf;
+ struct container_subsys_state *css;
  int retval;

  retval = -ENOMEM;
@@ -2569,15 +1766,15 @@ static int proc_cpuset_show(struct seq_f
   goto out_free;

  retval = -EINVAL;
- mutex_lock(&manage_mutex);
-
- retval = cpuset_path(tsk->cpuset, buf, PAGE_SIZE);
+ container_lock();
+ css = task_subsys_state(tsk, cpuset_subsys_id);
+ retval = container_path(css->container, buf, PAGE_SIZE);
  if (retval < 0)
   goto out_unlock;
  seq_puts(m, buf);
  seq_putc(m, '\n');
 out_unlock:
- mutex_unlock(&manage_mutex);
+ container_unlock();
  put_task_struct(tsk);
 out_free:
  kfree(buf);
@@ -2597,6 +1794,7 @@ const struct file_operations proc_cpuset
  .llseek  = seq_lseek,
  .release = single_release,
 };
+#endif /* CONFIG_PROC_PID_CPUSET */
```

```
 /* Display task cpus_allowed, mems_allowed in /proc/<pid>/status file. */
 char *cpuset_task_status_allowed(struct task_struct *task, char *buffer)
Index: container-2.6.21-rc7-mm1/kernel/exit.c
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/exit.c
+++ container-2.6.21-rc7-mm1/kernel/exit.c
@@ -31,7 +31,6 @@
 #include <linux/mempolicy.h>
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
-#include <linux/cpuset.h>
 #include <linux/container.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
@@ -939,7 +938,6 @@ fastcall NORET_TYPE void do_exit(long co
  __exit_files(tsk);
  __exit_fs(tsk);
  exit_thread();
- cpuset_exit(tsk);
  container_exit(tsk, 1);
  exit_keys(tsk);

Index: container-2.6.21-rc7-mm1/kernel/fork.c
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/fork.c
+++ container-2.6.21-rc7-mm1/kernel/fork.c
@@ -29,7 +29,6 @@
 #include <linux/nsproxy.h>
 #include <linux/capability.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
 #include <linux/container.h>
 #include <linux/security.h>
 #include <linux/swap.h>
@@ -1062,7 +1061,6 @@ static struct task_struct *copy_process(
  p->io_context = NULL;
  p->io_wait = NULL;
  p->audit_context = NULL;
- cpuset_fork(p);
  container_fork(p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
@@ -1309,7 +1307,6 @@ bad_fork_cleanup_policy:
  mpol_free(p->mempolicy);
 bad_fork_cleanup_container:
 #endif
- cpuset_exit(p);
  container_exit(p, container_callbacks_done);
```

```
  delayacct_tsk_free(p);
  if (p->binfmt)
Index: container-2.6.21-rc7-mm1/mm/mempolicy.c
 ===================================================== =======
--- container-2.6.21-rc7-mm1.orig/mm/mempolicy.c
+++ container-2.6.21-rc7-mm1/mm/mempolicy.c
@@ -1310,7 +1310,6 @@ EXPORT_SYMBOL(alloc_pages_current);
  * keeps mempolicies cpuset relative after its cpuset moves.  See
  * further kernel/cpuset.c update_nodemask().
  */
-void *cpuset_being_rebound;

 /* Slow path of a mempolicy copy */
 struct mempolicy *__mpol_copy(struct mempolicy *old)
@@ -1909,4 +1908,3 @@ out:
   m->version = (vma != priv->tail_vma) ? vma->vm_start : 0;
  return 0;
 }
-
Index: container-2.6.21-rc7-mm1/fs/proc/base.c
 ===================================================== =======
--- container-2.6.21-rc7-mm1.orig/fs/proc/base.c
+++ container-2.6.21-rc7-mm1/fs/proc/base.c
@@ -1978,7 +1978,7 @@ static const struct pid_entry tgid_base_
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat",  S_IRUGO, pid_schedstat),
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",     S_IRUGO, cpuset),
 #endif
 #ifdef CONFIG_CONTAINERS
@@ -2271,7 +2271,7 @@ static const struct pid_entry tid_base_s
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat", S_IRUGO, pid_schedstat),
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",    S_IRUGO, cpuset),
 #endif
 #ifdef CONFIG_CONTAINERS
Index: container-2.6.21-rc7-mm1/include/linux/container_subsys.h
 ===================================================== =======
---  container-2.6.21-rc7-mm1.orig/include/linux/container_subsys .h
+++ container-2.6.21-rc7-mm1/include/linux/container_subsys.h
@@ -13,4 +13,10 @@ SUBSYS(cpuacct)

 /* */
```

+#ifdef CONFIG_CPUSETS
+SUBSYS(cpuset)
+#endif
+
+/* */
+
 /* */

--

Subject: [PATCH 8/9] Containers (V9): Share css_group arrays between tasks with same container memberships
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:15 GMT
View Forum Message <> Reply to Message

This patch replaces the struct css_group embedded in task_struct with a pointer; all tasks with the same set of memberships across all hierarchies will share a css_group object.

The css_group used by init isn't refcounted, since it can't ever be freed; this speeds up fork/exit for any systems that have containers compiled in but haven't actually created any containers other than the default one.

With more than one registered subsystem, this reduces the number of atomic inc/dec operations required when tasks fork/exit;

Assuming that many tasks share the same container assignments, this reduces overall space usage and keeps the size of the task_struct down (only one pointer added to task_struct compared to a non-containers kernel).

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

---
 include/linux/container.h | 35 ++++++
 include/linux/sched.h | 30 -----
 kernel/container.c | 248 ++++++++++++++++++++++++++++++++++++++---------
 3 files changed, 238 insertions(+), 75 deletions(-)

Index: container-2.6.21-rc7-mm1/include/linux/container.h
 ========== =======
--- container-2.6.21-rc7-mm1.orig/include/linux/container.h
+++ container-2.6.21-rc7-mm1/include/linux/container.h
@@ -29,6 +29,14 @@ extern void container_unlock(void);

```
 struct containerfs_root;

+/* 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
@@ -87,6 +95,31 @@ struct container {
  struct container *top_container;
 };

+/* 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 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 */
+ struct list_head list;
+
+ /* 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];
+
+};
+
 /* struct cftype:
  *
  * The files in the container filesystem mostly have a very simple read/write
@@ -178,7 +211,7 @@ 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,
Index: container-2.6.21-rc7-mm1/include/linux/sched.h
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/include/linux/sched.h
+++ container-2.6.21-rc7-mm1/include/linux/sched.h
@@ -818,34 +818,6 @@ struct uts_namespace;

 struct prio_array;

-#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 */
  struct thread_info *thread_info;
@@ -1098,7 +1070,7 @@ struct task_struct {
  int cpuset_mem_spread_rotor;
 #endif
```

```
 #ifdef CONFIG_CONTAINERS
- struct css_group containers;
+ struct css_group *containers;
 #endif
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
Index: container-2.6.21-rc7-mm1/kernel/container.c
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/kernel/container.c
+++ container-2.6.21-rc7-mm1/kernel/container.c
@@ -132,12 +132,29 @@ 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
+/* 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;
+
+/*
+ * This lock nests inside tasklist lock, which can be taken by
+ * interrupts, and hence always needs to be taken with
+ * spin_lock_irq*
+ */
+static DEFINE_SPINLOCK(css_group_lock);
+static int css_group_count;
+
+/* 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 +163,140 @@ list_for_each_entry(_root, &roots, root_
  * 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);
+ unsigned long flags;
  int i;
+ spin_lock_irqsave(&css_group_lock, flags);
+ list_del(&cg->list);
+ css_group_count--;
+ spin_unlock_irqrestore(&css_group_lock, flags);
  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) {
+ if (cg != &init_css_group)
+  kref_get(&cg->ref);
+}
+
+static inline void put_css_group(struct css_group *cg) {
+ if (cg != &init_css_group)
+  kref_put(&cg->ref, release_css_group);
 }

-static void put_css_group(struct css_group *cg) {
+/*
+ * 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;
+ 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++) {
-  atomic_dec(&cg->subsys[i]->container->count);
+  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, oldcg->subsys, sizeof(oldcg->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;
+}
+
+/*
+ * 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;
+
+ /* First see if we already have a container group that matches
+  * the desired set */
+ spin_lock_irq(&css_group_lock);
+ res = find_existing_css_group(oldcg, cont, template);
+ if (res)
+  get_css_group(res);
+ spin_unlock_irq(&css_group_lock);
+
+ if (res)
+  return res;
+
+ res = kmalloc(sizeof(*res), GFP_KERNEL);
+ if (!res)
+  return NULL;
+
+ kref_init(&res->ref);
+ /* Copy the set of subsystem state objects generated in
+  * find_existing_css_group() */
+ memcpy(res->subsys, template, sizeof(res->subsys));
+ /* Add reference counts from the new css_group */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  atomic_inc(&res->subsys[i]->container->count);
  }
+
+ /* Link this container group into the list */
+ spin_lock_irq(&css_group_lock);
+ list_add(&res->list, &init_css_group.list);
+ css_group_count++;
+ spin_unlock_irq(&css_group_lock);
+
+ return res;
 }

 /*
@@ -717,9 +856,9 @@ 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);
@@ -738,24 +877,20 @@ 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);

  for_each_subsys(root, ss) {
@@ -765,6 +900,7 @@ static int attach_task(struct container
  }
```

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

@@ -1431,8 +1567,9 @@ static int container_rmdir(struct inode

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

  /* Create the top container state for this subsystem */
@@ -1443,26 +1580,32 @@ static void container_init_subsys(struct
  init_container_css(ss, dummytop);
  css = dummytop->subsys[ss->subsys_id];

- /* 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. */
+ spin_lock_irqsave(&css_group_lock, flags);
+ 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);
+ spin_unlock_irqrestore(&css_group_lock, flags);

   /* 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;

@@ -1478,7 +1621,12 @@ 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);
+ css_group_count = 1;
  init_container_root(&rootnode);
+ init_task.containers = &init_css_group;

  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
   struct container_subsys *ss = subsys[i];
@@ -1635,6 +1783,7 @@ 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;
 }
@@ -1671,7 +1820,7 @@ void container_fork(struct task_struct *
 {
  rcu_read_lock();
  child->containers = rcu_dereference(current->containers);
- get_css_group(&child->containers);
+ get_css_group(child->containers);
  rcu_read_unlock();
 }

@@ -1733,6 +1882,7 @@ void container_fork_callbacks(struct tas
 void container_exit(struct task_struct *tsk, int run_callbacks)
```

```
 {
  int i;
+ struct css_group *cg = NULL;
  if (run_callbacks && need_forkexit_callback) {
   for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
    struct container_subsys *ss = subsys[i];
@@ -1743,9 +1893,13 @@ void container_exit(struct task_struct *
  }
  /* Reassign the task to the init_css_group. */
  task_lock(tsk);
- put_css_group(&tsk->containers);
- tsk->containers = init_task.containers;
+ if (tsk->containers != &init_css_group) {
+  cg = tsk->containers;
+  tsk->containers = &init_css_group;
+ }
  task_unlock(tsk);
+ if (cg)
+  put_css_group(cg);
 }

 static atomic_t namecnt;
@@ -1783,11 +1937,13 @@ 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);
  /* 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 */
@@ -1832,6 +1988,7 @@ 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
@@ -1849,6 +2006,7 @@ int container_clone(struct task_struct *

  out_release:
```

```
  mutex_unlock(&inode->i_mutex);
+ put_css_group(cg);
  deactivate_super(parent->root->sb);
  return ret;
 }

--
```

Subject: [PATCH 9/9] Containers (V9): Simple debug info subsystem
Posted by Paul Menage on Fri, 27 Apr 2007 10:46:16 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         |   89 +++++++++++++++++++++++++++++++++++++++
 4 files changed, 104 insertions(+)

Index: container-2.6.21-rc7-mm1/include/linux/container_subsys.h
 ===================================================================== =======
---  container-2.6.21-rc7-mm1.orig/include/linux/container_subsys .h
+++ container-2.6.21-rc7-mm1/include/linux/container_subsys.h
@@ -19,4 +19,8 @@ SUBSYS(cpuset)

 /* */

+#ifdef CONFIG_CONTAINER_DEBUG
+SUBSYS(debug)
+#endif
+
 /* */
Index: container-2.6.21-rc7-mm1/init/Kconfig
 ===================================================================== =======
--- container-2.6.21-rc7-mm1.orig/init/Kconfig
+++ container-2.6.21-rc7-mm1/init/Kconfig
@@ -291,6 +291,16 @@ config IKCONFIG_PROC
 config CONTAINERS
  bool

+config CONTAINER_DEBUG
+ bool "Example debug container subsystem"
```

```
+ select 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
Index: container-2.6.21-rc7-mm1/kernel/container_debug.c
 ===================================================================
--- /dev/null
+++ container-2.6.21-rc7-mm1/kernel/container_debug.c
@@ -0,0 +1,89 @@
+/*
+ * 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>
+
+static int debug_create(struct container_subsys *ss, struct container *cont)
+{
+ struct container_subsys_state *css = kzalloc(sizeof(*css), GFP_KERNEL);
+ if (!css)
+  return -ENOMEM;
+ cont->subsys[debug_subsys_id] = css;
+ return 0;
+}
+
+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) 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 = "debug.container_refcount",
+  .read_uint = container_refcount_read,
+ },
+ {
+  .name = "debug.taskcount",
+  .read_uint = taskcount_read,
+ },
+
+ {
+  .name = "debug.current_css_group",
+  .read_uint = current_css_group_read,
+ },
+
+ {
+  .name = "debug.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, 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,
+};
Index: container-2.6.21-rc7-mm1/kernel/Makefile
 ===================================================================
--- container-2.6.21-rc7-mm1.orig/kernel/Makefile
+++ container-2.6.21-rc7-mm1/kernel/Makefile
@@ -37,6 +37,7 @@ 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

--
```

Subject: Re: [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Paul Jackson on Sun, 29 Apr 2007 01:47:44 GMT
View Forum Message <> Reply to Message

> - decide whether "Containers" is an acceptable name for the system
> given its usage by some other development groups, or whether something
> else (ProcessSets? ResourceGroups? TaskGroups?) would be better

I place in nomination:

 tasksets

However this would conflict with the taskset utility in Robert Loves
schedutils package. The 'taskset' command "is used to set or retrieve
the CPU affinity of a running process given its PID or to launch a new
COMMAND with a given CPU affinity."

The various alternatives you listed all have the advantage of stating
both that we have a container/group/set/collection/... of something,
and -what- that something might be - process/resource/task/...

"tasksets" is the shortest spelling I could think of for such a compond
"Something-Collection" form.

If the conflict with the schedutils utility is a concern, then the
next best alternative would be:

taskgroups

This is the next shortest way to spell this "Something-Collection" form.

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

Subject: Re: [ckrm-tech] [PATCH 1/9] Containers (V9): Basic container framework
Posted by Paul Jackson on Sun, 29 Apr 2007 03:12:29 GMT
View Forum Message <> Reply to Message

```
+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", val);
```

This sprintf is giving me the following compiler warnings:

kernel/container.c: In function 'container_read_uint':
kernel/container.c:1025: warning: format '%llu' expects type 'long long unsigned int', but argument 3 has type 'u64'
kernel/container.c:1025: warning: format '%llu' expects type 'long long unsigned int', but argument 3 has type 'u64'

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

Subject: Re: [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Paul Jackson on Sun, 29 Apr 2007 09:37:21 GMT
View Forum Message <> Reply to Message

I'm afraid that this patch set doesn't do cpusets very well.

It builds and boots and mounts the cpuset file system ok.
But trying to write the 'mems' file hangs the system hard.

I had to test it against 2.6.21-rc7-mm2, because I can't boot
2.6.21-rc7-mm1, due to the 'bad_page' problem that I noted in an
earlier post this evening on lkml.

These container patches seemed to apply ok against 2.6.21-rc7-mm2,
and built and booted ok. I built with an sn2_defconfig configuration,
having the following CONTAINER and CPUSET settings:

```
CONFIG_CONTAINERS=y
CONFIG_CONTAINER_DEBUG=y
CONFIG_CPUSETS=y
CONFIG_CONTAINER_CPUACCT=y
CONFIG_PROC_PID_CPUSET=y
# CONFIG_ACPI_CONTAINER is not set
```

I could mount the cpuset file system on /dev/cpuset just fine.

Then I invoked the following commands:

```
# cd /dev/cpuset
# mkdir foo
# cd foo
# echo 0-3 > cpus
# echo 0-1 > mems
```

At that point, the system hangs. Reproduced three times, on two boots.
I never get a shell prompt back from that second echo. I have to hit
Reset. The three different hangs were done with the following three
different values:

```
echo 0-3 > mems
echo 0-1 > mems
echo 1 > mems
```

On that last one, "echo 1 > mems", I did not do the echo to cpus first.

The test system had 8 cpus, numbered 0-7, and 4 mems, numbered 0-3.

--

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

Subject: Re: [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Srivatsa Vaddagiri on Mon, 30 Apr 2007 17:04:45 GMT
View Forum Message <> Reply to Message

On Sun, Apr 29, 2007 at 02:37:21AM -0700, Paul Jackson wrote:
> It builds and boots and mounts the cpuset file system ok.
> But trying to write the 'mems' file hangs the system hard.

Basically we are attempting a read_lock(&tasklist_lock) in
container_task_count() after taking write_lock_irq(&tasklist_lock) in
update_nodemask()!

This patch seems to fix the prb for me:

Fix write_lock() followed by read_lock() bug by introducing a 2nd
argument to be passed into container_task_count. Other choice is to
introduce a lock and unlocked versions of container_task_count() ..

Signed-off-by : Srivatsa Vaddagiri <vatsa@in.ibm.com>

```
diff -puN include/linux/container.h~lock_fix include/linux/container.h
--- linux-2.6.21-rc7/include/linux/container.h~lock_fix 2007-04-30 21:56:10.000000000 +0530
+++ linux-2.6.21-rc7-vatsa/include/linux/container.h 2007-04-30 21:56:32.000000000 +0530
@@ -164,7 +164,7 @@ 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);
+int container_task_count(const struct container *cont, int take_lock);

 /* Return true if the container is a descendant of the current container */
 int container_is_descendant(const struct container *cont);
diff -puN kernel/container.c~lock_fix kernel/container.c
--- linux-2.6.21-rc7/kernel/container.c~lock_fix 2007-04-30 21:56:10.000000000 +0530
+++ linux-2.6.21-rc7-vatsa/kernel/container.c 2007-04-30 22:06:45.000000000 +0530
@@ -1193,21 +1193,25 @@ int container_add_files(struct container
 /* 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. */
+ * that referenced it. 'take_lock' indicates whether caller has locked
+ * tasklist or not.
+ */
```

```
-int container_task_count(const struct container *cont) {
+int container_task_count(const struct container *cont, int take_lock) {
  int count = 0;
  struct task_struct *g, *p;
  struct container_subsys_state *css;
  int subsys_id;
  get_first_subsys(cont, &css, &subsys_id);

- read_lock(&tasklist_lock);
+ if (take_lock)
+  read_lock(&tasklist_lock);
  do_each_thread(g, p) {
   if (task_subsys_state(p, subsys_id) == css)
    count ++;
  } while_each_thread(g, p);
- read_unlock(&tasklist_lock);
+ if (take_lock)
+  read_unlock(&tasklist_lock);
  return count;
 }

@@ -1311,7 +1315,7 @@ static int container_tasks_open(struct i
   * caller from the case that the additional container users didn't
   * show up until sometime later on.
   */
- npids = container_task_count(cont);
+ npids = container_task_count(cont, 1);
  pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
  if (!pidarray)
   goto err1;
diff -puN kernel/cpuset.c~lock_fix kernel/cpuset.c
--- linux-2.6.21-rc7/kernel/cpuset.c~lock_fix 2007-04-30 21:56:10.000000000 +0530
+++ linux-2.6.21-rc7-vatsa/kernel/cpuset.c 2007-04-30 21:59:25.000000000 +0530
@@ -631,13 +631,13 @@ static int update_nodemask(struct cpuset
   * enough mmarray[] w/o using GFP_ATOMIC.
   */
  while (1) {
-  ntasks = container_task_count(cs->css.container);  /* guess */
+  ntasks = container_task_count(cs->css.container, 1); /* guess */
   ntasks += fudge;
   mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
   if (!mmarray)
    goto done;
   write_lock_irq(&tasklist_lock);  /* block fork */
-  if (container_task_count(cs->css.container) <= ntasks)
+  if (container_task_count(cs->css.container, 0) <= ntasks)
    break;    /* got enough */
   write_unlock_irq(&tasklist_lock); /* try again */
```

```
   kfree(mmarray);
diff -puN kernel/container_debug.c~lock_fix kernel/container_debug.c
--- linux-2.6.21-rc7/kernel/container_debug.c~lock_fix 2007-04-30 21:58:54.000000000 +0530
+++ linux-2.6.21-rc7-vatsa/kernel/container_debug.c 2007-04-30 21:59:04.000000000 +0530
@@ -34,7 +34,7 @@ static u64 taskcount_read(struct contain
 {
  u64 count;
  container_lock();
- count = container_task_count(cont);
+ count = container_task_count(cont, 1);
  container_unlock();
  return count;
 }
_


--
Regards,
vatsa
```

Subject: Re: [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Paul Menage on Mon, 30 Apr 2007 17:09:38 GMT
View Forum Message <> Reply to Message

On 4/30/07, Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:
> On Sun, Apr 29, 2007 at 02:37:21AM -0700, Paul Jackson wrote:
> > It builds and boots and mounts the cpuset file system ok.
> > But trying to write the 'mems' file hangs the system hard.
>
> Basically we are attempting a read_lock(&tasklist_lock) in
> container_task_count() after taking write_lock_irq(&tasklist_lock) in
> update_nodemask()!

Paul, is there any reason why we need to do a write_lock() on
tasklist_lock if we're just trying to block fork, or is it just
historical accident? Wouldn't it be fine to do a read_lock()?

Paul

Subject: Re: [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Christoph Hellwig on Mon, 30 Apr 2007 17:23:52 GMT
View Forum Message <> Reply to Message

On Mon, Apr 30, 2007 at 10:42:25PM +0530, Srivatsa Vaddagiri wrote:
> On Sun, Apr 29, 2007 at 02:37:21AM -0700, Paul Jackson wrote:

> > It builds and boots and mounts the cpuset file system ok.
> > But trying to write the 'mems' file hangs the system hard.
>
> Basically we are attempting a read_lock(&tasklist_lock) in
> container_task_count() after taking write_lock_irq(&tasklist_lock) in
> update_nodemask()!
>
> This patch seems to fix the prb for me:
>
>
> Fix write_lock() followed by read_lock() bug by introducing a 2nd
> argument to be passed into container_task_count. Other choice is to
> introduce a lock and unlocked versions of container_task_count() ..
>
> Signed-off-by : Srivatsa Vaddagiri <vatsa@in.ibm.com>

> -int container_task_count(const struct container *cont) {
> +int container_task_count(const struct container *cont, int take_lock) {
> int count = 0;
> struct task_struct *g, *p;
> struct container_subsys_state *css;
> int subsys_id;
> get_first_subsys(cont, &css, &subsys_id);
>
> - read_lock(&tasklist_lock);
> + if (take_lock)
> + read_lock(&tasklist_lock);
> do_each_thread(g, p) {
> if (task_subsys_state(p, subsys_id) == css)
> count ++;
> } while_each_thread(g, p);
> - read_unlock(&tasklist_lock);
> + if (take_lock)
> + read_unlock(&tasklist_lock);
> return count;

Umm, no - please naje two versions with and without the lock. Also
Please fix up the codingstyle, the { belongs onto a line of it's own.

Subject: Re: [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Srivatsa Vaddagiri on Mon, 30 Apr 2007 17:59:03 GMT
View Forum Message <> Reply to Message

On Mon, Apr 30, 2007 at 10:09:38AM -0700, Paul Menage wrote:
> Paul, is there any reason why we need to do a write_lock() on
> tasklist_lock if we're just trying to block fork, or is it just
> historical accident? Wouldn't it be fine to do a read_lock()?

Good point ..read_lock() will probably suffice in update_nodemask which
means we don't need the patch I sent earlier.

Paul (Jackson),
 This made me see another race in update_nodemask vs fork:

Lets say cpuset CS1 has only one task T1 to begin with.

update_nodemask(CS1)   T1 in do_fork()
 CPU0   CPU1
 == =

    cpuset_fork();
    mpol_copy();

ntasks = atomic_read(&cs->count);
[ntasks = 2, accounting new born child T2]
cs->mems_allowed = something;
set_cpuset_being_rebound()

write/read_lock(tasklist_lock);

do_each_thread {

 /* Finds only T1 */

 mmarray[] = ..

} while_each_thread();

write/read_unlock(tasklist_lock);

    write_lock(tasklist_lock);

    /* Add T2, child of T1 to tasklist */

    write_unlock(tasklist_lock);

for (i = 0; i < n; i++) {

 mpol_rebind_mm(..);

}

In this for loop, we migrate only T1's ->mm. T2's->mm isn't migrated
AFAICS.

Is that fine?

--
Regards,
vatsa

Subject: Re: [ckrm-tech] [PATCH 0/9] Containers (V9): Generic Process Containers
Posted by Srivatsa Vaddagiri on Mon, 30 Apr 2007 18:09:06 GMT
View Forum Message <> Reply to Message

On Mon, Apr 30, 2007 at 06:23:52PM +0100, Christoph Hellwig wrote:
> Umm, no - please naje two versions with and without the lock.

Ok this patch may not be necessary (given Paul's observation that
read_lock will suffice in update_nodemask).

> Also Please fix up the codingstyle, the { belongs onto a line of it's own.

Paul, I suspect that there are a number of Codingstyle related changes you
may need to address in your next version (including the one pointed by
Christoph)

--
Regards,
vatsa

Subject: Re: [PATCH 1/9] Containers (V9): Basic container framework
Posted by Balbir Singh on Tue, 01 May 2007 17:40:52 GMT
View Forum Message <> Reply to Message

menage@google.com wrote:
> This patch adds the main containers framework - the container
> filesystem, and the basic structures for tracking membership and
> associating subsystem state objects to tasks.

[snip]

> +*** notify_on_release is disabled in the current patch set. It may be

> +*** reactivated in a future patch in a less-intrusive manner
> +

Won't this break user space tools for cpusets?

[snip]

> +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().
> +

Hmm.. looks like a documentation error. Both mutex's are obtained through
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()
> +

container_mutex() and task_lock() can be used for changing the pointer?
Could you please explain this a bit further.

[snip]

> +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.

We needed the equivalent of container_remove_file() to be called
if container_add_file() failed.

[snip]

> +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;
> +};

Can't we derive the top_container from containerfs_root?

> +
> +/* 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
> + */
> +
> +struct inode;
> +#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);
> + u64 (*read_uint) (struct container *cont, struct cftype *cft);

Is this a new callback, a specialization of the read() callback?

> + 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);
> +};
> +

[snip]

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [PATCH 1/9] Containers (V9): Basic container framework
Posted by Paul Menage on Tue, 01 May 2007 17:46:10 GMT
View Forum Message <> Reply to Message

On 5/1/07, Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
> menage@google.com wrote:
> > This patch adds the main containers framework - the container
> > filesystem, and the basic structures for tracking membership and
> > associating subsystem state objects to tasks.
>
> [snip]
>
> > +*** notify_on_release is disabled in the current patch set. It may be
> > +*** reactivated in a future patch in a less-intrusive manner
> > +
>
> Won't this break user space tools for cpusets?

Yes, so it's a must-fix before this gets anywhere near a real distribution.

>
> [snip]
>
> > +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().
> > +
>
> Hmm.. looks like a documentation error. Both mutex's are obtained through
> container_lock/container_unlock ?

The second half of that sentence is obsolete and should have been deleted.

>
> > +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()
> > +
>
> container_mutex() and task_lock() can be used for changing the pointer?

No, these are all for read operations. (Actually, this is a bit of
documentation that's bit-rotted - there's no longer a per-task
"container" pointer). I'll update this.

For write operations, only the container system should be modifying
those pointers (under the protection of both container_mutex and
alloc_lock).

>
> We needed the equivalent of container_remove_file() to be called
> if container_add_file() failed.
>

Yes, this is some incomplete behaviour that I inherited from cpusets.
Needs tidying up.

>
> Can't we derive the top_container from containerfs_root?

Yes, we could for the cost of an extra dereference. Not sure it's a
big deal either way.

> > +	ssize_t (*read) (struct container *cont, struct cftype *cft,
> > +			struct file *file,
> > +			char __user *buf, size_t nbytes, loff_t *ppos);
> > +	u64 (*read_uint) (struct container *cont, struct cftype *cft);
>
> Is this a new callback, a specialization of the read() callback?

Yes. It's to simplify the common case of reporting a number in a

control file. (Not yet well documented :-( )

Paul

Subject: Re: [PATCH 2/9] Containers (V9): Example CPU accounting subsystem
Posted by Balbir Singh on Tue, 01 May 2007 17:52:03 GMT
View Forum Message <> Reply to Message

menage@google.com wrote:
> +
> +/* Lazily update the load calculation if necessary. Called with ca locked */
> +static void cpuusage_update(struct cpuacct *ca)
> +{
> + u64 now = get_jiffies_64();
> + /* If we're not due for an update, return */
> + if (ca->next_interval_check > now)
> + return;
> +
> + if (ca->next_interval_check <= (now - INTERVAL)) {

These two conditions seem a little confusing.

If ca->next_interval_check > (now - INTERVAL), the else part
is executed, but if ca->next_interval_check > (now - INTERVAL)
then ca->next_interval_check > now, which implies we return
and never enter the else part. It's been quite sometime since
I looked at this code, so I might have gotten it wrong.
I see a load of 0% on my powerpc box. I think it is because
last_interval_time is always 0, I'll debug further

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Balbir Singh on Tue, 01 May 2007 18:12:33 GMT
View Forum Message <> Reply to Message

> +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) {
> +  read_lock(&tasklist_lock);
```

You could just use rcu_read_lock() and rcu_read_unlock() instead
of read_lock(&tasklist_lock) and read_unlock(&tasklist_lock).

```
> +
> +  tsk = find_task_by_pid(pid);
> +  if (!tsk || tsk->flags & PF_EXITING) {
> +   read_unlock(&tasklist_lock);
> +   return -ESRCH;
> +  }
> +
> +  get_task_struct(tsk);
> +  read_unlock(&tasklist_lock);
> +
> +  if ((current->euid) && (current->euid != tsk->uid)
> +      && (current->euid != tsk->suid)) {
> +   put_task_struct(tsk);
> +   return -EACCES;
> +  }
> + } else {
> +  tsk = current;
> +  get_task_struct(tsk);
> + }
> +
> + ret = attach_task(cont, tsk);
> + put_task_struct(tsk);
> + return ret;
> +}
> +
>  /* The various types of files and directories in a container file system */
>
>  typedef enum {
> @@ -684,6 +789,54 @@ typedef enum {
>   FILE_TASKLIST,
>  } container_filetype_t;
>
> +static ssize_t container_common_file_write(struct container *cont,
> +        struct cftype *cft,
> +        struct file *file,
> +        const char __user *userbuf,
> +        size_t nbytes, loff_t *unused_ppos)
```

```
> +{
> + container_filetype_t type = cft->private;
> + char *buffer;
> + int retval = 0;
> +
> + if (nbytes >= PATH_MAX)
> +  return -E2BIG;
> +
> + /* +1 for nul-terminator */
> + if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
> +  return -ENOMEM;
> +
> + if (copy_from_user(buffer, userbuf, nbytes)) {
> +  retval = -EFAULT;
> +  goto out1;
> + }
> + buffer[nbytes] = 0; /* nul-terminate */
> +
> + mutex_lock(&container_mutex);
> +
> + if (container_is_removed(cont)) {
> +  retval = -ENODEV;
> +  goto out2;
> + }
```

Can't we make this check prior to kmalloc() and copy_from_user()?

```
> +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);
> +
> + read_lock(&tasklist_lock);
```

Can be replaced with rcu_read_lock() and rcu_read_unlock()

```
> + do_each_thread(g, p) {
> +  if (task_subsys_state(p, subsys_id) == css)
> +   count ++;
> + } while_each_thread(g, p);
> + read_unlock(&tasklist_lock);
> + return count;
> +}
> +
```

```
> +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();
> + read_lock(&tasklist_lock);
```

The read_lock() and read_unlock() are redundant

```
> +
> + 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:
> + read_unlock(&tasklist_lock);
> + rcu_read_unlock();
> + return n;
> +}
> +
```

[snip]

```
> +static int container_tasks_open(struct inode *unused, struct file *file)
> +{
> + struct container *cont = __d_cont(file->f_dentry->d_parent);
> + struct ctr_struct *ctr;
> + pid_t *pidarray;
> + int npids;
> + char c;
> +
> + if (!(file->f_mode & FMODE_READ))
> +  return 0;
> +
> + ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
> + if (!ctr)
> +  goto err0;
> +
> + /*
> +  * If container gets more users after we read count, we won't have
> +  * enough space - tough.  This race is indistinguishable to the
> +  * caller from the case that the additional container users didn't
```

```
> +  * show up until sometime later on.
> +  */
> + npids = container_task_count(cont);
> + pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
> + if (!pidarray)
> +  goto err1;
> +
> + npids = pid_array_load(pidarray, npids, cont);
> + sort(pidarray, npids, sizeof(pid_t), 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);
> + file->private_data = ctr;
> + return 0;
> +
> +err2:
> + kfree(pidarray);
> +err1:
> + kfree(ctr);
> +err0:
> + return -ENOMEM;
> +}
> +
```

Any chance we could get a per-container task list? It will
help subsystem writers as well. Alternatively, subsystems
could use the attach_task() callback to track all tasks,
but a per-container list will avoid duplication.

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [ckrm-tech] [PATCH 1/9] Containers (V9): Basic container framework
Posted by Paul Jackson on Tue, 01 May 2007 18:40:30 GMT
View Forum Message <> Reply to Message

[[I have bcc'd one or more batch scheduler experts on this post.
They will know who they are, and should be aware that they are
not listed in the public cc list of this message. - pj]]

Balbir Singh, responding to Paul Menage's Container patch set on lkml, wrote:
>
> > +*** notify_on_release is disabled in the current patch set. It may be
> > +*** reactivated in a future patch in a less-intrusive manner
> > +
>
> Won't this break user space tools for cpusets?

Yes - disabling notify_on_release would definitely break some important
uses of cpusets. This feature must be reactivated somehow before I'll
sign up for putting this patch set in the main line.

Actually, after I posted a few days ago in another lkml post:
http://lkml.org/lkml/2007/4/29/66

that just the simplest cpuset command:
mount -t cpuset cpuset /dev/cpuset
mkdir /dev/cpuset/foo
echo 0 > /dev/cpuset/foo/mems

caused an immediate kernel deadlock (Srivatsa has proposed a fix), it
is pretty clear that this container patch set is not getting the cpuset
testing it will need for acceptance. That's partly my fault.

The batch scheduler folks, such as the variants of PBS, LSF and SGE are
major user of cpusets on NUMA hardware.

This container based replacement for cpusets isn't ready for the main
line until at least one of those schedulers can run through one of
their test suites. I hesitate to even acknowledge this, as I might be
the only person in a position to make this happen, and my time
available to contribute to this patch set has been less than I would
like.

But if it looks like we have all the pieces in place to base cpusets
on containers, with no known regressions in cpuset capability, then
we must find a way to ensure that one of these batch schedulers, using
cpusets on a NUMA box, still works.

--

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

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Paul Menage on Tue, 01 May 2007 20:37:24 GMT
View Forum Message <> Reply to Message

On 5/1/07, Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
> > + if (container_is_removed(cont)) {
> > + retval = -ENODEV;
> > + goto out2;
> > + }
>
> Can't we make this check prior to kmalloc() and copy_from_user()?

We could but I'm not sure what it would buy us - we'd be optimizing
for the case that essentially never occurs.

>
>
>
> > +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);
> > +
> > + read_lock(&tasklist_lock);
>
> Can be replaced with rcu_read_lock() and rcu_read_unlock()

Are you sure about that? I see many users of
do_each_thread()/while_each_thread() taking a lock on tasklist_lock,
and only one (fs/binfmt_elf.c) that's clearly relying on an RCU
critical sections. Documentation?

>
> Any chance we could get a per-container task list? It will
> help subsystem writers as well.

It would be possible, yes - but we probably wouldn't want the overhead
(additional ref counts and list manipulations on every fork/exit) of
it on by default. We could make it a config option that particular
subsystems could select.

I guess the question is how useful is this really, compared to just
doing a do_each_thread() and seeing which tasks are in the container?
Certainly that's a non-trivial operation, but in what circumstances is
it really necessary to do it?

Paul

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Srivatsa Vaddagiri on Wed, 02 May 2007 03:17:56 GMT
View Forum Message <> Reply to Message

On Tue, May 01, 2007 at 01:37:24PM -0700, Paul Menage wrote:
> > Any chance we could get a per-container task list? It will
> > help subsystem writers as well.
>
> It would be possible, yes - but we probably wouldn't want the overhead
> (additional ref counts and list manipulations on every fork/exit) of
> it on by default. We could make it a config option that particular
> subsystems could select.
>
> I guess the question is how useful is this really, compared to just
> doing a do_each_thread() and seeing which tasks are in the container?
> Certainly that's a non-trivial operation, but in what circumstances is
> it really necessary to do it?

For the CPU controller I was working on, (a fast access to) such a list would have been valuable. Basically each task has a weight associated with it (p->load_weight) which is made to depend upon its class limit. Whenever the class limit changes, we need to go and change all its member task's ->load_weight value.

If you don't maintain the per-container task list, I guess I could still work around it, by either:

- Walk the task table and find relevant members, OR better perhaps
- Move p->load_weight to a class structure

--
Regards,
vatsa

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Paul Menage on Wed, 02 May 2007 03:25:35 GMT
View Forum Message <> Reply to Message

On 5/1/07, Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:
>
> For the CPU controller I was working on, (a fast access to) such a list would
> have been valuable. Basically each task has a weight associated with it
> (p->load_weight) which is made to depend upon its class limit. Whenever
> the class limit changes, we need to go and change all its member task's
> ->load_weight value.
>

> If you don't maintain the per-container task list, I guess I could still
> work around it, by either:
>
> - Walk the task table and find relevant members

That doesn't seem like a terrible solution to me, unless you expect
the class limit to be changing incredibly frequently.

If we had multiple subsystems that needed to walk the container member
list on a fast-path operation (e.g. to make a scheduling decision)
that would be a good reason to maintain such a list.

> perhaps
> - Move p->load_weight to a class structure

Sounds like a good idea if you can do it - but if it's per-process,
how would it fit in the class structure?

Paul

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Srivatsa Vaddagiri on Wed, 02 May 2007 03:38:34 GMT
View Forum Message <> Reply to Message

On Tue, May 01, 2007 at 08:25:35PM -0700, Paul Menage wrote:
> > - Walk the task table and find relevant members
>
> That doesn't seem like a terrible solution to me, unless you expect
> the class limit to be changing incredibly frequently.

yeah i agree. Group limit(s) should not be changing so frequently.

> > perhaps
> > - Move p->load_weight to a class structure
>
> Sounds like a good idea if you can do it - but if it's per-process,
> how would it fit in the class structure?

p->load_weight essentially depends on two things:

 - nice value or static priority (which is per process, already present
   in task_struct)
 - class limit (which is per class)

So in theory we can eliminate the load_weight field in task_struct and
compute it at runtime from the above two fields, although it will be
slightly inefficient I guess to compute the value every time a task is

added to the runqueue. If that is not desirable, then we can stick with
option 1 (walk task list and change member task's->load_weight upon class
limit change).

--
Regards,
vatsa

Subject: Re: [ckrm-tech] [PATCH 1/9] Containers (V9): Basic container framework
Posted by Balbir Singh on Wed, 02 May 2007 03:44:01 GMT
View Forum Message <> Reply to Message

Paul Jackson wrote:
> [[I have bcc'd one or more batch scheduler experts on this post.
> They will know who they are, and should be aware that they are
> not listed in the public cc list of this message. - pj]]
>
> Balbir Singh, responding to Paul Menage's Container patch set on lkml, wrote:
>>> +*** notify_on_release is disabled in the current patch set. It may be
>>> +*** reactivated in a future patch in a less-intrusive manner
>>> +
>> Won't this break user space tools for cpusets?
>
>
> Yes - disabling notify_on_release would definitely break some important
> uses of cpusets. This feature must be reactivated somehow before I'll
> sign up for putting this patch set in the main line.
>
> Actually, after I posted a few days ago in another lkml post:
> http://lkml.org/lkml/2007/4/29/66
>
> that just the simplest cpuset command:
> mount -t cpuset cpuset /dev/cpuset
> mkdir /dev/cpuset/foo
> echo 0 > /dev/cpuset/foo/mems
>
> caused an immediate kernel deadlock (Srivatsa has proposed a fix), it
> is pretty clear that this container patch set is not getting the cpuset
> testing it will need for acceptance. That's partly my fault.
>
> The batch scheduler folks, such as the variants of PBS, LSF and SGE are
> major user of cpusets on NUMA hardware.
>
> This container based replacement for cpusets isn't ready for the main
> line until at least one of those schedulers can run through one of
> their test suites. I hesitate to even acknowledge this, as I might be
> the only person in a position to make this happen, and my time

> available to contribute to this patch set has been less than I would
> like.
>
> But if it looks like we have all the pieces in place to base cpusets
> on containers, with no known regressions in cpuset capability, then
> we must find a way to ensure that one of these batch schedulers, using
> cpusets on a NUMA box, still works.
>

Would it be possible to extract those test cases and integrate them
with a testing framework like LTP? Do you have any regression test
suite for cpusets that can be made available publicly so that
any changes to cpusets can be validated?

The reason I ask for the test suite is that I suspect that the
container framework will evolve further and a reliable testing
mechanism would be extremely useful.

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Balbir Singh on Wed, 02 May 2007 03:58:01 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> On 5/1/07, Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
>> > +   if (container_is_removed(cont)) {
>> > +   retval = -ENODEV;
>> > +   goto out2;
>> > +   }
>>
>> Can't we make this check prior to kmalloc() and copy_from_user()?
>
> We could but I'm not sure what it would buy us - we'd be optimizing
> for the case that essentially never occurs.
>

I am not sure about the never occurs part of it, because we check
for the condition, so it could occur. I agree, it is a premature
optimization and could wait a little longer before going in.

>>
>>

>>
>> > +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);
>> > +
>> > +	read_lock(&tasklist_lock);
>>
>> Can be replaced with rcu_read_lock() and rcu_read_unlock()
>
> Are you sure about that? I see many users of
> do_each_thread()/while_each_thread() taking a lock on tasklist_lock,
> and only one (fs/binfmt_elf.c) that's clearly relying on an RCU
> critical sections. Documentation?
>

I suspect they are all pending conversions to be made.
Eric is the expert on this. Meanwhile here's a couple of
pointers. Quoting from the second URL

"We don't need the tasklist_lock to safely iterate through processes
anymore."

http://www.linuxjournal.com/article/6993 (please see incremental use
of RCU) and
 http://kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2
.6.17/2.6.17-mm2/broken-out/proc-remove-tasklist_lock-from-p roc_pid_readdir.patch

>>
>> Any chance we could get a per-container task list? It will
>> help subsystem writers as well.
>
> It would be possible, yes - but we probably wouldn't want the overhead
> (additional ref counts and list manipulations on every fork/exit) of
> it on by default. We could make it a config option that particular
> subsystems could select.
>
> I guess the question is how useful is this really, compared to just
> doing a do_each_thread() and seeing which tasks are in the container?
> Certainly that's a non-trivial operation, but in what circumstances is
> it really necessary to do it?
>
> Paul

--

Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [ckrm-tech] [PATCH 1/9] Containers (V9): Basic container framework
Posted by Paul Jackson on Wed, 02 May 2007 06:12:54 GMT
View Forum Message <> Reply to Message

Balbir wrote:
> Would it be possible to extract those test cases and integrate them
> with a testing framework like LTP? Do you have any regression test
> suite for cpusets that can be made available publicly so that
> any changes to cpusets can be validated?

There are essentially two sorts of cpuset regression tests of interest.

I have one such test, and the batch scheduler developers have various
tests of their batch schedulers.

1) Testing batch schedulers against cpusets:

> I doubt that the batch scheduler developers would be able to
> extract a cpuset test from their tests, or be able to share it if
> they did. Their tests tend to be large tests of batch schedulers,
> and only incidentally test cpusets -- if we break cpusets,
> in sometimes even subtle ways that they happen to depend on,
> we break them.
>
> Sometimes there is no way to guess exactly what sorts of changes
> will break their code; we'll just have to schedule at least one
> run through one or more of them that rely heavily on cpusets
> before a change as big as rebasing cpusets on containers is
> reasonably safe. This test cycle won't be all that easy, so I'd
> wait until we are pretty close to what we think should be taken
> into the mainline kernel.
>
> I suppose I will have to be the one co-ordinating this test,
> as I am the only one I know with a presence in both camps.
>
> Once this test is done, from then forward, if we break them,
> we'll just have to deal with it as we do now, when the breakage
> shows up well down stream from the main kernel tree, at the point
> that a major batch scheduler release runs into a major distribution
> release containing the breakage. There is no practical way that I
> can see, as an ongoing basis, to continue testing for such breakage
> with every minor change to cpuset related code in the kernel. Any

breakage found this way is dealt with by changes in user level code.

Once again, I have bcc'd one or more developers of batch schedulers,
so they can see what nonsense I am spouting about them now ;).

2) Testing cpusets with a specific test.

There I can do better. Attached is the cpuset regression test I
use. It requires at least 4 cpus and 2 memory nodes to do anything
useful. It is copyright by SGI, released under GPL license.

This regression test is the primary cpuset test upon which I
relied during the development of cpusets, and continue to rely.
Except for one subtle race condition in the test itself, it has
not changed in the last two to three years.

This test requires no user level code not found in an ordinary
distro. It does require the taskset and numactl commands,
for the purposes of testing certain interactions with them.
It assumes that there are not other cpusets currently setup in
the system that happen to conflict with the ones it creates.

See further comments within the test script itself.

--

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

File Attachments

1) cpuset_test, downloaded 297 times

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Balbir Singh on Tue, 08 May 2007 14:51:04 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> On 5/1/07, Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:
>> For the CPU controller I was working on, (a fast access to) such a list would
>> have been valuable. Basically each task has a weight associated with it
>> (p->load_weight) which is made to depend upon its class limit. Whenever
>> the class limit changes, we need to go and change all its member task's
>> ->load_weight value.
>>
>> If you don't maintain the per-container task list, I guess I could still
>> work around it, by either:
>>

>> - Walk the task table and find relevant members
>
> That doesn't seem like a terrible solution to me, unless you expect
> the class limit to be changing incredibly frequently.
>
> If we had multiple subsystems that needed to walk the container member
> list on a fast-path operation (e.g. to make a scheduling decision)
> that would be a good reason to maintain such a list.
>

I now have a use case for maintaining a per-container task list.
I am trying to build a per-container stats similar to taskstats.
I intend to support container accounting of

1. Tasks running
2. Tasks stopped
3. Tasks un-interruptible
4. Tasks blocked on IO
5. Tasks sleeping

This would provide statistics similar to the patch that Pavel had sent out.

I faced the following problems while trying to implement this feature

1. There is no easy way to get a list of all tasks belonging to a container
   (we need to walk all threads)
2. There is no concept of a container identifier. When a user issues a command
   to extract statistics, the only unique container identifier is the container
   path, which means that we need to do a path lookup to determine the dentry
   for the container (which gets quite ugly with all the string manipulation)

   Adding a container id, will make it easier to find a container and return
   statistics belonging to the container.

If we get these two features added to the current patches, it will make the
infrastructure more "developer" and eventually more user friendly :-)

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [ckrm-tech] [PATCH 1/9] Containers (V9): Basic container framework
Posted by Balbir Singh on Thu, 10 May 2007 04:09:55 GMT

Paul Jackson wrote:
> Balbir wrote:

>
> 1) Testing batch schedulers against cpusets:
>
> I doubt that the batch scheduler developers would be able to
> extract a cpuset test from their tests, or be able to share it if
> they did. Their tests tend to be large tests of batch schedulers,
> and only incidentally test cpusets -- if we break cpusets,
> in sometimes even subtle ways that they happen to depend on,
> we break them.
>
> Sometimes there is no way to guess exactly what sorts of changes
> will break their code; we'll just have to schedule at least one
> run through one or more of them that rely heavily on cpusets
> before a change as big as rebasing cpusets on containers is
> reasonably safe. This test cycle won't be all that easy, so I'd
> wait until we are pretty close to what we think should be taken
> into the mainline kernel.
>
> I suppose I will have to be the one co-ordinating this test,
> as I am the only one I know with a presence in both camps.
>
> Once this test is done, from then forward, if we break them,
> we'll just have to deal with it as we do now, when the breakage
> shows up well down stream from the main kernel tree, at the point
> that a major batch scheduler release runs into a major distribution
> release containing the breakage. There is no practical way that I
> can see, as an ongoing basis, to continue testing for such breakage
> with every minor change to cpuset related code in the kernel. Any
> breakage found this way is dealt with by changes in user level code.
>
> Once again, I have bcc'd one or more developers of batch schedulers,
> so they can see what nonsense I am spouting about them now ;).
>

That sounds reasonable to me

> 2) Testing cpusets with a specific test.
>
> There I can do better. Attached is the cpuset regression test I
> use. It requires at least 4 cpus and 2 memory nodes to do anything
> useful. It is copyright by SGI, released under GPL license.
>
> This regression test is the primary cpuset test upon which I
> relied during the development of cpusets, and continue to rely.

> Except for one subtle race condition in the test itself, it has
> not changed in the last two to three years.
>
> This test requires no user level code not found in an ordinary
> distro. It does require the taskset and numactl commands,
> for the purposes of testing certain interactions with them.
> It assumes that there are not other cpusets currently setup in
> the system that happen to conflict with the ones it creates.
>
> See further comments within the test script itself.
>

Thanks for the script. Would you like to contribute this script to
LTP for wider availability and testing?

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [ckrm-tech] [PATCH 1/9] Containers (V9): Basic container framework
Posted by Paul Jackson on Thu, 10 May 2007 04:47:06 GMT
View Forum Message <> Reply to Message

Balbir wrote:
> Thanks for the script. Would you like to contribute this script to
> LTP for wider availability and testing?

Sounds good - though I'm a tad lazy, and don't know how to contribute
this to the LTP.

You're welcome to contribute it yourself, if that's easier for you,
or to point me in the direction of some information on this.

Since it's GPL, you can republish it, under that license.

... Most likely however, it will take a little tweaking to make this
test work well for LTP. Most automated test environments have some
requirements on the behaviour of each test, and it would be surprising
if this test happen already to conform to LTP's requirements.

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

Subject: Re: [ckrm-tech] [PATCH 1/9] Containers (V9): Basic container framework
Posted by Balbir Singh on Thu, 10 May 2007 04:49:09 GMT
View Forum Message <> Reply to Message

Paul Jackson wrote:
> Balbir wrote:
>> Thanks for the script. Would you like to contribute this script to
>> LTP for wider availability and testing?
>
> Sounds good - though I'm a tad lazy, and don't know how to contribute
> this to the LTP.
>
> You're welcome to contribute it yourself, if that's easier for you,
> or to point me in the direction of some information on this.
>
> Since it's GPL, you can republish it, under that license.
>
> ... Most likely however, it will take a little tweaking to make this
> test work well for LTP. Most automated test environments have some
> requirements on the behaviour of each test, and it would be surprising
> if this test happen already to conform to LTP's requirements.
>

I'll figure out more, once I am comfortable with the script and the
setup environment, I'll try and get this into LTP with a GPL
license.

--
Thanks,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Paul Menage on Thu, 10 May 2007 21:21:39 GMT
View Forum Message <> Reply to Message

On 5/8/07, Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
>
> I now have a use case for maintaining a per-container task list.
> I am trying to build a per-container stats similar to taskstats.
> I intend to support container accounting of
>
> 1. Tasks running
> 2. Tasks stopped
> 3. Tasks un-interruptible

> 4. Tasks blocked on IO
> 5. Tasks sleeping
>
> This would provide statistics similar to the patch that Pavel had sent out.
>
> I faced the following problems while trying to implement this feature
>
> 1. There is no easy way to get a list of all tasks belonging to a container
> (we need to walk all threads)

Well, walking the taks list is pretty easy - but yes, it could become
inefficient when there are many small containers in use.

I've got some ideas for a way of tracking this specifically for
containers with subsystems that want this, while avoiding the overhead
for subsystems that don't really need it. I'll try to add them to the
next patchset.

> 2. There is no concept of a container identifier. When a user issues a command
> to extract statistics, the only unique container identifier is the container
> path, which means that we need to do a path lookup to determine the dentry
> for the container (which gets quite ugly with all the string manipulation)

We could just cache the container path permanently in the container,
and invalidate it if any of its parents gets renamed. (I imagine this
happens almost never.)

>
> Adding a container id, will make it easier to find a container and return
> statistics belonging to the container.

Not unreasonable, but there are a few questions that would have to be answered:

- how is the container id picked? Like a pid, or user-defined? Or some
kind of string?

- how would it be exposed to userspace? A generic control file
provided by the container filesystem in all container directories?

- can you give a more concrete example of how this would actually be
useful? For your container stats, it seems that just reading a control
file in the container's directory would give you the stats that you
want, and userspace already knows the container's name/id since it
opened the control file.

Paul

Subject: Re: [ckrm-tech] [PATCH 3/9] Containers (V9): Add tasks file interface
Posted by Balbir Singh on Fri, 11 May 2007 02:31:30 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> On 5/8/07, Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
>>
>> I now have a use case for maintaining a per-container task list.
>> I am trying to build a per-container stats similar to taskstats.
>> I intend to support container accounting of
>>
>> 1. Tasks running
>> 2. Tasks stopped
>> 3. Tasks un-interruptible
>> 4. Tasks blocked on IO
>> 5. Tasks sleeping
>>
>> This would provide statistics similar to the patch that Pavel had sent
>> out.
>>
>> I faced the following problems while trying to implement this feature
>>
>> 1. There is no easy way to get a list of all tasks belonging to a
>> container
>>   (we need to walk all threads)
>
> Well, walking the taks list is pretty easy - but yes, it could become
> inefficient when there are many small containers in use.
>
> I've got some ideas for a way of tracking this specifically for
> containers with subsystems that want this, while avoiding the overhead
> for subsystems that don't really need it. I'll try to add them to the
> next patchset.

Super!

>
>> 2. There is no concept of a container identifier. When a user issues a
>> command
>>   to extract statistics, the only unique container identifier is the
>> container
>>   path, which means that we need to do a path lookup to determine the
>> dentry
>>   for the container (which gets quite ugly with all the string
>> manipulation)
>
> We could just cache the container path permanently in the container,
> and invalidate it if any of its parents gets renamed. (I imagine this
> happens almost never.)

>

Here's what I have so far, I cache the mount point of the container
and add the container path to it. I'm now stuck examining tasks,
while walking through a bunch of tasks, there is no easy way of
knowing the container path of the task without walking all subsystems
and then extracting the containers absolute path.

>>
>> Adding a container id, will make it easier to find a container and
>> return
>> statistics belonging to the container.
>
> Not unreasonable, but there are a few questions that would have to be
> answered:
>
> - how is the container id picked? Like a pid, or user-defined? Or some
> kind of string?
>

I was planning on using a hierarchical scheme, top 8 bits for
the container hierarchy and bottom 24 for a unique id. The id
is automatically selected. Once we know the container id, we'll
need a more efficient mechanism to map the id to the container.

> - how would it be exposed to userspace? A generic control file
> provided by the container filesystem in all container directories?
>

A file in all container directories is an option

> - can you give a more concrete example of how this would actually be
> useful? For your container stats, it seems that just reading a control
> file in the container's directory would give you the stats that you
> want, and userspace already knows the container's name/id since it
> opened the control file.
>

Sure, the plan is to build a containerstats interface like taskstats.
In taskstats, we exchange data between user space and kernel space
using genetlink sockets. We have a push and pull mechanism for statistics.

> Paul

--
 Warm Regards,

Balbir Singh
Linux Technology Center
IBM, ISTL