
Subject: [PATCH 01/10] Containers(V10): Basic container framework
Posted by [Paul Menage](#) on Tue, 29 May 2007 13:01:05 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 | 1155 ++++++
8 files changed, 1927 insertions(+), 1 deletion(-)

Index: container-2.6.22-rc2-mm1/Documentation/containers.txt

```
=====
--- /dev/null
+++ container-2.6.22-rc2-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 hierarchies 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 cpuset may add additional files in each container dir

+

+New containers are created using the mkdir system call or shell command. The properties of a container, such as its flags, are modified by writing to the appropriate file in that containers directory, as listed above.

+

+The named hierarchical structure of nested containers allows partitioning a large system into nested, dynamically changeable, "soft-partitions".

+

+The attachment of each task, automatically inherited at fork by any children of that task, to a container allows organizing the work load on a system into related sets of tasks. A task may be re-attached to any other container, if allowed by the permissions on the necessary container file system directories.

+

+When a task is moved from one container to another, it gets a new css_group pointer - if there's an already existing css_group with the desired collection of containers then that group is reused, else a new css_group is allocated. Note that the current implementation uses a linear search to locate an appropriate existing css_group, so isn't very efficient. A future version will use a hash table for better performance.

+

+The use of a Linux virtual file system (vfs) to represent the container hierarchy provides for a familiar permission and name space for containers, with a minimum of additional kernel code.

+

+1.4 What does notify_on_release do ?

+-----

+

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

+*** reactivated in a future patch in a less-intrusive manner
+
+If the notify_on_release flag is enabled (1) in a container, then
+whenever the last task in the container leaves (exits or attaches to
+some other container) and the last child container of that container
+is removed, then the kernel runs the command specified by the contents
+of the "release_agent" file in that hierarchy's root directory,
+supplying the pathname (relative to the mount point of the container
+file system) of the abandoned container. This enables automatic
+removal of abandoned containers. The default value of
+notify_on_release in the root container at system boot is disabled
+(0). The default value of other containers at creation is the current
+value of their parents notify_on_release setting. The default value of
+a container hierarchy's release_agent path is empty.

+
+1.5 How do I use containers ?

+-----

+
+To start a new job that is to be contained within a container, using
+the "cpuset" container subsystem, the steps are something like:

- +
+ 1) mkdir /dev/container
+ 2) mount -t container -ocpuset cpuset /dev/container
+ 3) Create the new container by doing mkdir's and write's (or echo's) in
+ the /dev/container virtual file system.
+ 4) Start a task that will be the "founding father" of the new job.
+ 5) Attach that task to the new container by writing its pid to the
+ /dev/container tasks file for that container.
+ 6) fork, exec or clone the job tasks from this founding father task.

+
+For example, the following sequence of commands will setup a container
+named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
+and then start a subshell 'sh' in that container:

+
+ mount -t container cpuset -ocpuset /dev/container
+ cd /dev/container
+ mkdir Charlie
+ cd Charlie
+ /bin/echo \$\$ > 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 with 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_subsys *ss, struct task_struct *task)
+LL=callback_mutex, maybe read_lock(tasklist_lock)
+
+Called when a task is forked into a container. Also called during registration for all existing tasks.
+
+void exit(struct container_subsys *ss, struct task_struct *task)
+LL=callback_mutex
+
+Called during task exit
+
+int populate(struct container_subsys *ss, struct container *cont)
+LL=none
+
+Called after creation of a container to allow a subsystem to populate the container directory with file entries. The subsystem should make calls to container_add_file() with objects of type cftype (see include/linux/container.h for details). Note that although this method can return an error code, the error code is currently not

+always handled well.

+

+void bind(struct container_subsys *ss, struct container *root)

+LL=callback_mutex

+

+Called when a container subsystem is rebound to a different hierarchy

+and root container. Currently this will only involve movement between

+the default hierarchy (which never has sub-containers) and a hierarchy

+that is being created/destroyed (and hence has no sub-containers).

+

+4. Questions

+=====

+

+Q: what's up with this '/bin/echo' ?

+A: bash's builtin 'echo' command does not check calls to write() against

+ errors. If you use it in the container file system, you won't be

+ able to tell whether a command succeeded or failed.

+

+Q: When I attach processes, only the first of the line gets really attached !

+A: We can only return one error code per call to write(). So you should also

+ put only ONE pid.

+

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

=====

--- /dev/null

+++ container-2.6.22-rc2-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 subsystem_id;
+ int active;
+ int early_init;
+#define MAX_CONTAINER_TYPE_NAMELEN 32
+ const char *name;
+
+ /* Protected by RCU */
+ struct containerfs_root *root;
+
+ struct list_head sibling;
+
+ void *private;
+};
+
+#define SUBSYS(_x) extern struct container_subsys _x ## _subsys;
+#include <linux/container_subsys.h>
+#undef SUBSYS

```

```

+
+static inline struct container_subsys_state *container_subsys_state(
+ struct container *cont, int subsys_id)
+{
+ return cont->subsys[subsys_id];
+}
+
+static inline struct container_subsys_state *task_subsys_state(
+ struct task_struct *task, int subsys_id)
+{
+ return rcu_dereference(task->containers.subsys[subsys_id]);
+}
+
+static inline struct container* task_container(struct task_struct *task,
+      int subsys_id)
+{
+ return task_subsys_state(task, subsys_id)->container;
+}
+
+int container_path(const struct container *cont, char *buf, int buflen);
+
+#else /* !CONFIG_CONTAINERS */
+
+static inline int container_init_early(void) { return 0; }
+static inline int container_init(void) { return 0; }
+static inline void container_init_smp(void) {}
+
+static inline void container_lock(void) {}
+static inline void container_unlock(void) {}
+
+#endif /* !CONFIG_CONTAINERS */
+
+#endif /* _LINUX_CONTAINER_H */
Index: container-2.6.22-rc2-mm1/include/linux/container_subsys.h
=====
--- /dev/null
+++ container-2.6.22-rc2-mm1/include/linux/container_subsys.h
@@ -0,0 +1,10 @@
+/* Add subsystem definitions of the form SUBSYS(<name>) in this
+ * file. Surround each one by a line of comment markers so that
+ * patches don't collide
+ */
+
+/* */
+
+/* */
+
+/* */

```


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

```
=====
--- container-2.6.22-rc2-mm1.orig/include/linux/sched.h
+++ container-2.6.22-rc2-mm1/include/linux/sched.h
@@ -851,6 +851,34 @@ struct sched_class {
    void (*task_new) (struct rq *rq, struct task_struct *p);
};

#ifdef CONFIG_CONTAINERS
+
+#define SUBSYS(_x) _x ## _subsys_id,
+enum container_subsys_id {
+#include <linux/container_subsys.h>
+ CONTAINER_SUBSYS_COUNT
+};
+#undef SUBSYS
+
+/* A css_group is a structure holding pointers to a set of
+ * container_subsys_state objects.
+ */
+
+struct css_group {
+
+ /* Set of subsystem states, one for each subsystem. NULL for
+ * subsystems that aren't part of this hierarchy. These
+ * pointers reduce the number of dereferences required to get
+ * from a task to its state for a given container, but result
+ * in increased space usage if tasks are in wildly different
+ * groupings across different hierarchies. This array is
+ * immutable after creation */
+ struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
+
+};
+
+#endif /* CONFIG_CONTAINERS */
+
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
@@ -1107,6 +1135,9 @@ 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;
@@ -1553,7 +1584,8 @@ static inline int thread_group_empty(str
/*
 * Protects ->fs, ->files, ->mm, ->group_info, ->comm, keyring
 * subscriptions and synchronises with wait4(). Also used in procfs. Also
- * pins the final release of task.io_context. Also protects ->cpuset.
+ * pins the final release of task.io_context. Also protects ->cpuset and
+ * ->container.subsys[].
 *
 * Nests both inside and outside of read_lock(&tasklist_lock).
 * It must not be nested with write_lock_irq(&tasklist_lock),

```

Index: container-2.6.22-rc2-mm1/init/Kconfig

```

=====
--- container-2.6.22-rc2-mm1.orig/init/Kconfig
+++ container-2.6.22-rc2-mm1/init/Kconfig
@@ -303,6 +303,9 @@ config LOG_BUF_SHIFT
     13 => 8 KB
     12 => 4 KB

```

+config CONTAINERS

+ bool

+

config CPUSETS

bool "Cpuset support"

depends on SMP

Index: container-2.6.22-rc2-mm1/init/main.c

```

=====
--- container-2.6.22-rc2-mm1.orig/init/main.c
+++ container-2.6.22-rc2-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-2.6.22-rc2-mm1/kernel/Makefile
=====
--- container-2.6.22-rc2-mm1.orig/kernel/Makefile
+++ container-2.6.22-rc2-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-2.6.22-rc2-mm1/kernel/container.c
=====
--- /dev/null
+++ container-2.6.22-rc2-mm1/kernel/container.c
@@ -0,0 +1,1155 @@
+/*
+ * 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_mutex, 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's 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 error handling, so it's being left until
+ * later */
+ if (!list_empty(&cont->children)) {
+ return -EBUSY;
+ }
+
+ /* Process each subsystem */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ struct container_subsys *ss = subsys[i];
+ unsigned long bit = 1UL << i;
+ if (bit & added_bits) {
+ /* We're binding this subsystem to this hierarchy */
+ BUG_ON(cont->subsys[i]);
+ BUG_ON(!dummytop->subsys[i]);
+ BUG_ON(dummytop->subsys[i]->container != dummytop);
+ cont->subsys[i] = dummytop->subsys[i];
+ cont->subsys[i]->container = cont;
+ list_add(&ss->sibling, &root->subsys_list);
+ rcu_assign_pointer(ss->root, root);
+ if (ss->bind)
+ ss->bind(ss, cont);
+ } else if (bit & removed_bits) {
+ /* We're removing this subsystem */
+ BUG_ON(cont->subsys[i] != dummytop->subsys[i]);
+ BUG_ON(cont->subsys[i]->container != cont);

```

```

+ if (ss->bind)
+   ss->bind(ss, dummytop);
+   dummytop->subsys[i]->container = dummytop;
+   cont->subsys[i] = NULL;
+   rcu_assign_pointer(subsys[i]->root, &rootnode);
+   list_del(&ss->sibling);
+ } else if (bit & final_bits) {
+   /* Subsystem state should already exist */
+   BUG_ON(!cont->subsys[i]);
+ } else {
+   /* Subsystem state shouldn't exist */
+   BUG_ON(cont->subsys[i]);
+ }
+ }
+ root->subsys_bits = 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);
+
+   list_del(&root->root_list);

```

```

+ 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\n", (unsigned long long) val);
+ return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
+}
+
+static ssize_t container_file_read(struct file *file, char __user *buf,
+    size_t nbytes, loff_t *ppos)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct container *cont = __d_cont(file->f_dentry->d_parent);
+ if (!cft)
+ return -ENODEV;
+
+ if (cft->read)
+ return cft->read(cont, cft, file, buf, nbytes, ppos);
+ if (cft->read_uint)
+ return container_read_uint(cont, cft, file, buf, nbytes, ppos);
+ return -EINVAL;
+}
+
+static int container_file_open(struct inode *inode, struct file *file)
+{
+ int err;
+ struct cftype *cft;
+
+ err = generic_file_open(inode, file);
+ if (err)
+ return err;
+
+ cft = __d_cft(file->f_dentry);
+ if (!cft)
+ return -ENODEV;
+ if (cft->open)
+ err = cft->open(inode, file);
+ else
+ err = 0;
+
+ return err;

```

```

+}
+
+static int container_file_release(struct inode *inode, struct file *file)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ if (cft->release)
+ return cft->release(inode, file);
+ return 0;
+}
+
+/*
+ * container_rename - Only allow simple rename of directories in place.
+ */
+static int container_rename(struct inode *old_dir, struct dentry *old_dentry,
+ struct inode *new_dir, struct dentry *new_dentry)
+{
+ if (!S_ISDIR(old_dentry->d_inode->i_mode))
+ return -ENOTDIR;
+ if (new_dentry->d_inode)
+ return -EEXIST;
+ if (old_dir != new_dir)
+ return -EIO;
+ return simple_rename(old_dir, old_dentry, new_dir, new_dentry);
+}
+
+static struct file_operations container_file_operations = {
+ .read = container_file_read,
+ .write = container_file_write,
+ .llseek = generic_file_llseek,
+ .open = container_file_open,
+ .release = container_file_release,
+};
+
+static struct inode_operations container_dir_inode_operations = {
+ .lookup = simple_lookup,
+ .mkdir = container_mkdir,
+ .rmdir = container_rmdir,
+ .rename = container_rename,
+};
+
+static int container_create_file(struct dentry *dentry, int mode, struct super_block *sb)
+{
+ struct inode *inode;
+
+ if (!dentry)
+ return -ENOENT;
+ if (dentry->d_inode)
+ return -EEXIST;

```

```

+
+ inode = container_new_inode(mode, sb);
+ if (!inode)
+ return -ENOMEM;
+
+ if (S_ISDIR(mode)) {
+ inode->i_op = &container_dir_inode_operations;
+ inode->i_fop = &simple_dir_operations;
+
+ /* start off with i_nlink == 2 (for "." entry) */
+ inc_nlink(inode);
+
+ /* start with the directory inode held, so that we can
+ * populate it without racing with another mkdir */
+ mutex_lock(&inode->i_mutex);
+ } else if (S_ISREG(mode)) {
+ inode->i_size = 0;
+ inode->i_fop = &container_file_operations;
+ }
+
+ d_instantiate(dentry, inode);
+ dget(dentry); /* Extra count - pin the dentry in core */
+ return 0;
+}
+
+/*
+ * container_create_dir - create a directory for an object.
+ * cont: the container we create the directory for.
+ * It must have a valid ->parent field
+ * And we are going to fill its ->dentry field.
+ * name: The name to give to the container directory. Will be copied.
+ * mode: mode to set on new directory.
+ */
+
+static int container_create_dir(struct container *cont, struct dentry *dentry,
+ int mode)
+{
+ struct dentry *parent;
+ int error = 0;
+
+ parent = cont->parent->dentry;
+ if (IS_ERR(parent))
+ return PTR_ERR(parent);
+ 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;
+}

--
```
