
Subject: [PATCH 3/6] containers: Add generic multi-subsystem API to containers
Posted by [Paul Menage](#) on Fri, 22 Dec 2006 14:14:45 GMT

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

This patch removes all cpuset-specific knowledge from the container system, replacing it with a generic API that can be used by multiple subsystems. Cpusets is adapted to be a container subsystem.

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

```
---
Documentation/containers.txt | 273 ++++++++---
Documentation/cpusets.txt   | 20
include/linux/container.h   | 137 ++++++
include/linux/cpuset.h      | 16
include/linux/mempolicy.h   | 12
include/linux/sched.h       | 4
init/Kconfig                | 12
kernel/container.c          | 903 ++++++-----
kernel/cpuset.c             | 173 +++++---
mm/mempolicy.c              | 2
10 files changed, 1215 insertions(+), 337 deletions(-)
```

Index: container-2.6.20-rc1/include/linux/container.h

```
=====
--- container-2.6.20-rc1.orig/include/linux/container.h
+++ container-2.6.20-rc1/include/linux/container.h
@@ -14,8 +14,6 @@
```

```
#ifdef CONFIG_CONTAINERS
```

```
-extern int number_of_containers; /* How many containers are defined in system? */
```

```
-
```

```
extern int container_init_early(void);
extern int container_init(void);
extern void container_init_smp(void);
@@ -30,6 +28,68 @@ extern void container_unlock(void);
extern void container_manage_lock(void);
extern void container_manage_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() */
+ spinlock_t refcnt_lock;
+ atomic_t refcnt;
+};
+
+/*
+ * Call css_get() to hold a reference on the container; following a
+ * return of 0, this container subsystem state object is guaranteed
+ * not to be destroyed until css_put() is called on it. A non-zero
+ * return code indicates that a reference could not be taken.
+ *
+ */
+
+static inline int css_get(struct container_subsys_state *css)
+{
+ int retval = 0;
+ unsigned long flags;
+ /* Synchronize with container_rmdir() */
+ spin_lock_irqsave(&css->refcnt_lock, flags);
+ if (atomic_read(&css->refcnt) >= 0) {
+ /* Container is still alive */
+ atomic_inc(&css->refcnt);
+ } else {
+ /* Container removal is in progress */
+ retval = -EINVAL;
+ }
+ spin_unlock_irqrestore(&css->refcnt_lock, flags);
+ return retval;
+}
+
+/*
+ * If you are holding current->alloc_lock then it's impossible for you
+ * to be moved out of your container, and hence it's impossible for
+ * your container to be destroyed. Therefore doing a simple
+ * atomic_inc() on a css is safe.
+ */
+
+static inline void css_get_current(struct container_subsys_state *css)
+{
+ atomic_inc(&css->refcnt);
+}
+
+/*
+ * css_put() should be called to release a reference taken by
+ * css_get() or css_get_current()

```

```

+ */
+
+static inline void css_put(struct container_subsys_state *css) {
+ atomic_dec(&css->refcnt);
+}
+
struct container {
    unsigned long flags; /* "unsigned long" so bitops work */

@@ -46,11 +106,15 @@ struct container {
    struct list_head children; /* my children */

    struct container *parent; /* my parent */
- struct dentry *dentry; /* container fs entry */
+ struct dentry *dentry; /* container fs entry */
+
+ /* Private pointers for each registered subsystem */
+ struct container_subsys_state *subsys[CONFIG_MAX_CONTAINER_SUBSYS];
+
+ int hierarchy;

#ifdef CONFIG_CPUSETS
- struct cpuset *cpuset;
#endif
+ struct containerfs_root *root;
+ struct container *top_container;
};

/* struct cftype:
@@ -67,8 +131,11 @@ struct container {
    */

    struct inode;
#define MAX_CFTYPE_NAME 64
    struct cftype {
- char *name;
+ /* 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,
@@ -83,7 +150,61 @@ struct cftype {
    @@ -83,7 +150,61 @@ struct cftype {
        int container_add_file(struct container *cont, const struct cftype *cft);

        int container_is_removed(const struct container *cont);
- void container_set_release_agent_path(const char *path);
+

```

```

+int container_path(const struct container *cont, char *buf, int buflen);
+
+/* 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 (*post_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);
+
+ int subsys_id;
+#define MAX_CONTAINER_TYPE_NAMELEN 32
+ const char *name;
+
+ /* Protected by RCU */
+ int hierarchy;
+
+ struct list_head sibling;
+};
+
+int container_register_subsys(struct container_subsys *subsys);
+void container_set_release_agent_path(struct container_subsys *ss,
+   const char *path);
+
+static inline struct container_subsys_state *container_subsys_state(
+ struct container *cont,
+ struct container_subsys *ss)
+{
+ return cont->subsys[ss->subsys_id];
+}
+
+static inline struct container* task_container(struct task_struct *task,
+   struct container_subsys *ss)
+{
+ return rcu_dereference(task->container[ss->hierarchy]);
+}
+

```

```
+static inline struct container_subsys_state *task_subsys_state(
+ struct task_struct *task,
+ struct container_subsys *ss)
+{
+ return container_subsys_state(task_container(task, ss), ss);
+}
```

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

Index: container-2.6.20-rc1/include/linux/cpuset.h

```
=====
```

```
--- container-2.6.20-rc1.orig/include/linux/cpuset.h
```

```
+++ container-2.6.20-rc1/include/linux/cpuset.h
```

```
@@ -70,16 +70,7 @@ static inline int cpuset_do_slab_mem_spr
```

```
extern void cpuset_track_online_nodes(void);
```

```
-extern int cpuset_can_attach_task(struct container *cont,
- struct task_struct *tsk);
```

```
-extern void cpuset_attach_task(struct container *cont,
- struct task_struct *tsk);
```

```
-extern void cpuset_post_attach_task(struct container *cont,
- struct container *oldcont,
- struct task_struct *tsk);
```

```
-extern int cpuset_populate_dir(struct container *cont);
```

```
-extern int cpuset_create(struct container *cont);
```

```
-extern void cpuset_destroy(struct container *cont);
```

```
+extern int current_cpuset_is_being_rebound(void);
```

```
#else /* !CONFIG_CPUSETS */
```

```
@@ -147,6 +138,11 @@ 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-2.6.20-rc1/kernel/container.c

```
=====
```

```
--- container-2.6.20-rc1.orig/kernel/container.c
```

```
+++ container-2.6.20-rc1/kernel/container.c
```

```
@@ -55,7 +55,6 @@
```

```

#include <linux/time.h>
#include <linux/backing-dev.h>
#include <linux/sort.h>
#include <linux/cpuset.h>

#include <asm/uaccess.h>
#include <asm/atomic.h>
@@ -63,12 +62,58 @@

#define CONTAINER_SUPER_MAGIC 0x27e0eb

-/*
- * Tracks how many containers are currently defined in system.
- * When there is only one container (the root container) we can
- * short circuit some hooks.
+static struct container_subsys *subsys[CONFIG_MAX_CONTAINER_SUBSYS];
+static int subsys_count = 0;
+
+/* 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;
+
+ /* The path to use for release notifications. No locking
+ * between setting and use - so if userspace updates this
+ * while subcontainers exist, you could miss a
+ * notification. We ensure that it's always a valid
+ * NUL-terminated string */
+ char release_agent_path[PATH_MAX];
+};
+
+/* The set of hierarchies in use. Hierarchy 0 is the "dummy
+ * container", 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[CONFIG_MAX_CONTAINER_HIERARCHIES];
+
+/* dummytop is a shorthand for the dummy hierarchy's top container */
+#define dummytop (&rootnode[0].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 take locks in the fork/exit path if none of the
+ * subsystems need to be called.
+ *
+ * It is protected via RCU, with the invariant that a process in an
+ * rcu_read_lock() section will never see this as 0 if there are
+ * actually registered subsystems with a fork or exit
+ * handler. (Sometimes it may be 1 without there being any registered
+ * subsystems with such a handler, but such periods are safe and of
+ * short duration).
+ */
-int number_of_containers __read_mostly;
+static int need_forkexit_callback = 0;

/* bits in struct container flags field */
typedef enum {
@@ -87,27 +132,18 @@ static inline int notify_on_release(const
    return test_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
}

-static struct container top_container = {
- .count = ATOMIC_INIT(0),
- .sibling = LIST_HEAD_INIT(top_container.sibling),
- .children = LIST_HEAD_INIT(top_container.children),
-};
+#define for_each_subsys(_hierarchy, _ss) list_for_each_entry(_ss,
&rootnode[_hierarchy].subsys_list, sibling)

-/* The path to use for release notifications. No locking between
- * setting and use - so if userspace updates this while subcontainers
- * exist, you could miss a notification */
-static char release_agent_path[PATH_MAX] = "/sbin/container_release_agent";
-
-void container_set_release_agent_path(const char *path)
+void container_set_release_agent_path(struct container_subsys *ss,
+    const char *path)
{
+ struct containerfs_root *root;
+ container_manage_lock();
- strcpy(release_agent_path, path);
+ root = &rootnode[ss->hierarchy];

```

```

+ strcpy(root->release_agent_path, path);
+ container_manage_unlock();
+ }

-static struct vfsmount *container_mount;
-static struct super_block *container_sb;
-
/*
 * We have two global container mutexes below. They can nest.
 * It is ok to first take manage_mutex, then nest callback_mutex. We also
@@ -189,11 +225,71 @@ static struct super_block *container_sb;
 * update of a tasks container pointer by attach_task() and the
 * access of task->container->mems_generation via that pointer in
 * the routine container_update_task_memory_state().
+ *
+ * Some container subsystems and other external code also use these
+ * mutexes, exposed through the container_lock()/container_unlock()
+ * and container_manage_lock()/container_manage_unlock() functions.
+ *
+ * E.g. the out of memory (OOM) code needs to prevent containers from
+ * being changed while it scans the tasklist looking for a task in an
+ * overlapping container. The tasklist_lock is a spinlock, so must be
+ * taken inside callback_mutex.
+ *
+ * Some container subsystems (including cpusets) also use
+ * callback_mutex as a primary lock for synchronizing access to
+ * subsystem state. Deciding on best practices of when to use
+ * fine-grained locks vs container_lock()/container_unlock() is still
+ * a TODO.
+ *
+ */

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

+/**
+ * container_lock - lock out any changes to container structures
+ *
+ */
+
+void container_lock(void)
+{
+ mutex_lock(&callback_mutex);
+}
+
+/**
+ * container_unlock - release lock on container changes
+ *

```

```

+ * Undo the lock taken in a previous container_lock() call.
+ */
+
+void container_unlock(void)
+{
+ mutex_unlock(&callback_mutex);
+}
+
+/**
+ * container_manage_lock() - lock out anyone else considering making
+ * changes to container structures. This is a more heavy-weight lock
+ * than the callback_mutex taken by container_lock() */
+
+void container_manage_lock(void)
+{
+ mutex_lock(&manage_mutex);
+}
+
+/**
+ * container_manage_unlock
+ *
+ * Undo the lock taken in a previous container_manage_lock() call.
+ */
+
+void container_manage_unlock(void)
+{
+ mutex_unlock(&manage_mutex);
+}
+
+
+
+/*
+ * A couple of forward declarations required, due to cyclic reference loop:
+ * container_mkdir -> container_create -> container_populate_dir -> container_add_file
+ @@ -202,15 +298,18 @@ static DEFINE_MUTEX(callback_mutex);
+
+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 file_operations proc_containerstats_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)

```

```

+static struct inode *container_new_inode(mode_t mode, struct super_block *sb)
{
- struct inode *inode = new_inode(container_sb);
+ struct inode *inode = new_inode(sb);

    if (inode) {
        inode->i_mode = mode;
@@ -282,32 +381,102 @@ static void container_d_remove_dir(struct
    remove_dir(dentry);
    }

+/*
+ * 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;
+ int hierarchy = root->top_container.hierarchy;
+ int i;
+ struct container *cont = &root->top_container;
+ struct task_struct *g, *p;
+
+ root->sb = NULL;
+ sb->s_fs_info = NULL;
+
+ mutex_lock(&manage_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);
+
+ mutex_lock(&callback_mutex);
+
+ /* Remove all tasks from this container hierarchy */
+ read_lock(&tasklist_lock);
+ do_each_thread(g, p) {
+     task_lock(p);
+     BUG_ON(!p->container[hierarchy]);
+     BUG_ON(p->container[hierarchy] != cont);
+     rcu_assign_pointer(p->container[hierarchy], NULL);
+     task_unlock(p);
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+ atomic_set(&cont->count, 1);

```

```

+
+ /* Remove all subsystems from this hierarchy */
+ for (i = 0; i < subsys_count; i++) {
+   if (root->subsys_bits & (1 << i)) {
+     struct container_subsys *ss = subsys[i];
+     BUG_ON(cont->subsys[i] != dummytop->subsys[i]);
+     BUG_ON(cont->subsys[i]->container != cont);
+     dummytop->subsys[i]->container = dummytop;
+     cont->subsys[i] = NULL;
+     rcu_assign_pointer(subsys[i]->hierarchy, 0);
+     list_del(&ss->sibling);
+   } else {
+     BUG_ON(cont->subsys[i]);
+   }
+ }
+ root->subsys_bits = 0;
+ mutex_unlock(&callback_mutex);
+ synchronize_rcu();
+
+ mutex_unlock(&manage_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->top_container.hierarchy, ss) {
+   seq_printf(seq, "%s", ss->name);
+ }
+ return 0;
+}
+
+static struct super_operations container_ops = {
+   .statfs = simple_statfs,
+   .drop_inode = generic_delete_inode,
+   .put_super = container_put_super,
+   .show_options = container_show_options,
+ };

-static int container_fill_super(struct super_block *sb, void *unused_data,
-    int unused_silent)
+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;
- container_sb = sb;

- inode = container_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) */
- inode->i_nlink++;
- } else {
+ 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) {
@@ -315,6 +484,13 @@ static int container_fill_super(struct s
return -ENOMEM;
}
sb->s_root = root;
+ root->d_fsdata = &hroot->top_container;
+ hroot->top_container.dentry = root;
+
+ strcpy(hroot->release_agent_path, "");
+ sb->s_fs_info = hroot;
+ hroot->sb = sb;
+
return 0;
}

@@ -322,7 +498,130 @@ static int container_get_sb(struct file_
int flags, const char *unused_dev_name,
void *data, struct vfsmount *mnt)
{
- return get_sb_single(fs_type, flags, data, container_fill_super, mnt);
+ int i;
+ struct container_subsys *ss;
+ char *token, *o = data ? "all";
+ unsigned long subsys_bits = 0;

```

```

+ int ret = 0;
+ struct containerfs_root *root = NULL;
+ int hierarchy;
+
+ mutex_lock(&manage_mutex);
+
+ /* First find the desired set of resource controllers */
+ while ((token = strsep(&o, ",")) != NULL) {
+   if (!*token) {
+     ret = -EINVAL;
+     goto out_unlock;
+   }
+   if (!strcmp(token, "all")) {
+     subsys_bits = (1 << subsys_count) - 1;
+   } else {
+     for (i = 0; i < subsys_count; i++) {
+       ss = subsys[i];
+       if (!strcmp(token, ss->name)) {
+         subsys_bits |= 1 << i;
+         break;
+       }
+     }
+     if (i == subsys_count) {
+       ret = -ENOENT;
+       goto out_unlock;
+     }
+   }
+ }
+
+ /* See if we already have a hierarchy containing this set */
+
+ for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+   root = &rootnode[i];
+   /* We match - use this hierarchy */
+   if (root->subsys_bits == subsys_bits) break;
+   /* We clash - fail */
+   if (root->subsys_bits & subsys_bits) {
+     ret = -EBUSY;
+     goto out_unlock;
+   }
+ }
+
+ if (i == CONFIG_MAX_CONTAINER_HIERARCHIES) {
+   /* No existing hierarchy matched this set - but we
+    * know that all the subsystems are free */
+   for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+     root = &rootnode[i];
+     if (!root->sb && !root->subsys_bits) break;

```

```

+ }
+ }
+
+ if (i == CONFIG_MAX_CONTAINER_HIERARCHIES) {
+   ret = -ENOSPC;
+   goto out_unlock;
+ }
+
+ hierarchy = i;
+
+ if (!root->sb) {
+   /* We need a new superblock for this container combination */
+   struct container *cont = &root->top_container;
+   struct container_subsys *ss;
+   struct task_struct *p, *g;
+
+   BUG_ON(root->subsys_bits);
+   root->subsys_bits = 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);
+
+   mutex_lock(&callback_mutex);
+
+   /* Add all tasks into this container hierarchy */
+   atomic_set(&cont->count, 1);
+   read_lock(&tasklist_lock);
+   do_each_thread(g, p) {
+     task_lock(p);
+     BUG_ON(p->container[hierarchy]);
+     rcu_assign_pointer(p->container[hierarchy], cont);
+     if (!(p->flags & PF_EXITING)) {
+       atomic_inc(&cont->count);
+     }
+     task_unlock(p);
+   } while_each_thread(g, p);
+   read_unlock(&tasklist_lock);
+
+   /* Move all the relevant subsystems into the hierarchy. */
+   for (i = 0; i < subsys_count; i++) {
+     if (!(subsys_bits & (1 << i))) continue;
+
+     ss = subsys[i];

```

```

+
+ BUG_ON(cont->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(subsys[i]->hierarchy,
+     hierarchy);
+ }
+ mutex_unlock(&callback_mutex);
+ synchronize_rcu();
+
+ container_populate_dir(cont);
+
+ } else {
+ /* Reuse the existing superblock */
+ ret = simple_set_mnt(mnt, root->sb);
+ if (!ret)
+     atomic_inc(&root->sb->s_active);
+ }
+
+ out_unlock:
+ mutex_unlock(&manage_mutex);
+ return ret;
+ }

static struct file_system_type container_fs_type = {
@@ -401,7 +700,7 @@ int container_path(const struct containe
 * the time manage_mutex is held.
 */

-static void container_release_agent(const char *pathbuf)
+static void container_release_agent(int hierarchy, const char *pathbuf)
{
    char *argv[3], *envp[3];
    int i;
@@ -410,7 +709,7 @@ static void container_release_agent(const
    return;

    i = 0;
- argv[i++] = release_agent_path;
+ argv[i++] = rootnode[hierarchy].release_agent_path;
    argv[i++] = (char *)pathbuf;
    argv[i] = NULL;

@@ -487,7 +786,7 @@ static int update_flag(container_flagbit

```

```

/*
- * Attach task specified by pid in 'pidbuf' to container 'cont', possibly
+ * Attach task specified by pid in 'pidbuf' to container 'cont', possibly
  * writing the path of the old container in 'ppathbuf' if it needs to be
  * notified on release.
  *
@@ -501,6 +800,8 @@ static int attach_task(struct container
    struct task_struct *tsk;
    struct container *oldcont;
    int retval = 0;
+ struct container_subsys *ss;
+ int h = cont->hierarchy;

    if (sscanf(pidbuf, "%d", &pid) != 1)
        return -EIO;
@@ -527,37 +828,45 @@ static int attach_task(struct container
    get_task_struct(tsk);
}

-#ifdef CONFIG_CPUSETS
- retval = cpuset_can_attach_task(cont, tsk);
-#endif
- if (retval) {
-     put_task_struct(tsk);
-     return retval;
+ for_each_subsys(h, ss) {
+     if (ss->can_attach) {
+         retval = ss->can_attach(ss, cont, tsk);
+         if (retval) {
+             put_task_struct(tsk);
+             return retval;
+         }
+     }
+ }

    mutex_lock(&callback_mutex);

    task_lock(tsk);
- oldcont = tsk->container;
+ oldcont = tsk->container[h];
    if (!oldcont) {
        task_unlock(tsk);
        mutex_unlock(&callback_mutex);
        put_task_struct(tsk);
        return -ESRCH;
    }
+     BUG_ON(oldcont == dummytop);
+

```

```

    atomic_inc(&cont->count);
- rcu_assign_pointer(tsk->container, cont);
+ rcu_assign_pointer(tsk->container[h], cont);
    task_unlock(tsk);

-#ifdef CONFIG_CPUSETS
- cpuset_attach_task(cont, tsk);
-#endif
+ for_each_subsys(h, ss) {
+   if (ss->attach) {
+     ss->attach(ss, cont, oldcont, tsk);
+   }
+ }

    mutex_unlock(&callback_mutex);

-#ifdef CONFIG_CPUSETS
- cpuset_post_attach_task(cont, oldcont, tsk);
-#endif
+ for_each_subsys(h, ss) {
+   if (ss->post_attach) {
+     ss->post_attach(ss, cont, oldcont, tsk);
+   }
+ }

    put_task_struct(tsk);
    synchronize_rcu();
@@ -616,13 +925,14 @@ static ssize_t container_common_file_wri
    break;
    case FILE_RELEASE_AGENT:
    {
-   if (nbytes < sizeof(release_agent_path)) {
+   struct containerfs_root *root = &rootnode[cont->hierarchy];
+   if (nbytes < sizeof(root->release_agent_path)) {
        /* We never write anything other than '\0'
         * into the last char of release_agent_path,
         * so it always remains a NUL-terminated
         * string */
-   strncpy(release_agent_path, buffer, nbytes);
-   release_agent_path[nbytes] = 0;
+   strncpy(root->release_agent_path, buffer, nbytes);
+   root->release_agent_path[nbytes] = 0;
    } else {
        retval = -ENOSPC;
    }
}
@@ -637,7 +947,7 @@ static ssize_t container_common_file_wri
    retval = nbytes;
    out2:

```

```

    mutex_unlock(&manage_mutex);
- container_release_agent(pathbuf);
+ container_release_agent(cont->hierarchy, pathbuf);
out1:
    kfree(buffer);
    return retval;
@@ -683,11 +993,14 @@ static ssize_t container_common_file_rea
    break;
    case FILE_RELEASE_AGENT:
    {
+ struct containerfs_root *root;
    size_t n;
    container_manage_lock();
- n = strlen(release_agent_path, sizeof(release_agent_path));
+ root = &rootnode[cont->hierarchy];
+ n = strlen(root->release_agent_path,
+ sizeof(root->release_agent_path));
    n = min(n, (size_t) PAGE_SIZE);
- strncpy(s, release_agent_path, n);
+ strncpy(s, root->release_agent_path, n);
    container_manage_unlock();
    s += n;
    break;
@@ -780,7 +1093,7 @@ static struct inode_operations container
    .rename = container_rename,
};

-static int container_create_file(struct dentry *dentry, int mode)
+static int container_create_file(struct dentry *dentry, int mode, struct super_block *sb)
{
    struct inode *inode;

@@ -789,7 +1102,7 @@ static int container_create_file(struct
    if (dentry->d_inode)
        return -EEXIST;

- inode = container_new_inode(mode);
+ inode = container_new_inode(mode, sb);
    if (!inode)
        return -ENOMEM;

@@ -798,7 +1111,7 @@ static int container_create_file(struct
    inode->i_fop = &simple_dir_operations;

    /* start off with i_nlink == 2 (for "." entry) */
- inode->i_nlink++;
+ inc_nlink(inode);
    } else if (S_ISREG(mode)) {

```

```

inode->i_size = 0;
inode->i_fop = &container_file_operations;
@@ -828,10 +1141,10 @@ static int container_create_dir(struct c
    dentry = container_get_dentry(parent, name);
    if (IS_ERR(dentry))
        return PTR_ERR(dentry);
- error = container_create_file(dentry, S_IFDIR | mode);
+ error = container_create_file(dentry, S_IFDIR | mode, cont->root->sb);
    if (!error) {
        dentry->d_fsdata = cont;
- parent->d_inode->i_nlink++;
+ inc_nlink(parent->d_inode);
        cont->dentry = dentry;
    }
    dput(dentry);
@@ -848,7 +1161,7 @@ int container_add_file(struct container
    mutex_lock(&dir->d_inode->i_mutex);
    dentry = container_get_dentry(dir, cft->name);
    if (!IS_ERR(dentry)) {
- error = container_create_file(dentry, 0644 | S_IFREG);
+ error = container_create_file(dentry, 0644 | S_IFREG, cont->root->sb);
        if (!error)
            dentry->d_fsdata = (void *)cft;
        dput(dentry);
@@ -894,7 +1207,7 @@ static int pid_array_load(pid_t *pidarra
    read_lock(&tasklist_lock);

    do_each_thread(g, p) {
- if (p->container == cont) {
+ if (p->container[cont->hierarchy] == cont) {
        pidarray[n++] = p->pid;
        if (unlikely(n == npids))
            goto array_full;
@@ -1037,21 +1350,33 @@ static struct cftype cft_release_agent =
static int container_populate_dir(struct container *cont)
{
    int err;
+ struct container_subsys *ss;

    if ((err = container_add_file(cont, &cft_notify_on_release)) < 0)
        return err;
    if ((err = container_add_file(cont, &cft_tasks)) < 0)
        return err;
- if ((cont == &top_container) &&
+ if ((cont == &top_container) &&
+ if ((cont == cont->top_container) &&
        (err = container_add_file(cont, &cft_release_agent)) < 0)
        return err;
-#ifdef CONFIG_CPUSETS

```

```

- if ((err = cpuset_populate_dir(cont)) < 0)
- return err;
-#endif
+
+ for_each_subsys(cont->hierarchy, 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;
+ spin_lock_init(&css->refcnt_lock);
+ atomic_set(&css->refcnt, 0);
+}
+
+/*
+ * container_create - create a container
+ * parent: container that will be parent of the new container.
@@ -1064,13 +1389,24 @@ static int container_populate_dir(struct
static long container_create(struct container *parent, const char *name, int mode)
{
+ struct container *cont;
- int err;
+ struct containerfs_root *root = parent->root;
+ int err = 0;
+ struct container_subsys *ss;
+ struct super_block *sb = root->sb;

- cont = kmalloc(sizeof(*cont), GFP_KERNEL);
+ 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 manage_mutex, since the sb can't
+ * disappear while someone has an open control file on the
+ * fs */
+ atomic_inc(&sb->s_active);
+
+ mutex_lock(&manage_mutex);
+

```

```

cont->flags = 0;
if (notify_on_release(parent))
    set_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
@@ -1079,16 +1415,19 @@ static long container_create(struct cont
    INIT_LIST_HEAD(&cont->children);

    cont->parent = parent;
-
-#ifdef CONFIG_CPUSETS
- err = cpuset_create(cont);
- if (err)
-     goto err_unlock_free;
-#endif
+ cont->root = parent->root;
+ cont->hierarchy = parent->hierarchy;
+ cont->top_container = parent->top_container;
+
+ for_each_subsys(cont->hierarchy, ss) {
+     err = ss->create(ss, cont);
+     if (err) goto err_destroy;
+     init_container_css(ss, cont);
+ }

    mutex_lock(&callback_mutex);
    list_add(&cont->sibling, &cont->parent->children);
- number_of_containers++;
+ root->number_of_containers++;
    mutex_unlock(&callback_mutex);

    err = container_create_dir(cont, name, mode);
@@ -1107,15 +1446,23 @@ static long container_create(struct cont
    return 0;

err_remove:
-#ifdef CONFIG_CPUSETS
- cpuset_destroy(cont);
-#endif
+
+     mutex_lock(&callback_mutex);
+     list_del(&cont->sibling);
+     number_of_containers--;
+     root->number_of_containers--;
+     mutex_unlock(&callback_mutex);
- err_unlock_free:
+
+ err_destroy:
+
+ for_each_subsys(cont->hierarchy, ss) {

```

```

+ if (cont->subsys[ss->subsys_id])
+ ss->destroy(ss, cont);
+ }
+
+ mutex_unlock(&manage_mutex);
+
+ deactivate_super(sb);
+
+ kfree(cont);
+ return err;
+ }
@@ -1128,23 +1475,18 @@ static int container_mkdir(struct inode
+ return container_create(c_parent, dentry->d_name.name, mode | S_IFDIR);
+ }

-/*
- * Locking note on the strange update_flag() call below:
- *
- * If the container being removed is marked cpu_exclusive, then simulate
- * turning cpu_exclusive off, which will call update_cpu_domains().
- * The lock_cpu_hotplug() call in update_cpu_domains() must not be
- * made while holding callback_mutex. Elsewhere the kernel nests
- * callback_mutex inside lock_cpu_hotplug() calls. So the reverse
- * nesting would risk an ABBA deadlock.
- */
-
static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)
{
+ struct container *cont = dentry->d_fsdata;
+ struct dentry *d;
+ struct container *parent;
+ char *pathbuf = NULL;
+ struct container_subsys *ss;
+ struct super_block *sb;
+ struct containerfs_root *root;
+ unsigned long flags;
+ int css_busy = 0;
+ int hierarchy;

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

@@ -1157,7 +1499,41 @@ static int container_rmdir(struct inode
+ mutex_unlock(&manage_mutex);
+ return -EBUSY;
+ }
+
+ hierarchy = cont->hierarchy;
+ parent = cont->parent;

```

```

+ root = cont->root;
+ sb = root->sb;
+
+ local_irq_save(flags);
+ /* Check each container, locking the refcnt lock and testing
+  * the refcnt. This will lock out any calls to css_get() */
+ for_each_subsys(hierarchy, ss) {
+ struct container_subsys_state *css;
+ css = cont->subsys[ss->subsys_id];
+ spin_lock(&css->refcnt_lock);
+ css_busy += atomic_read(&css->refcnt);
+ }
+ /* Go through and release all the locks; if we weren't busy,
+  * the set the refcount to -1 to prevent css_get() from adding
+  * a refcount */
+ for_each_subsys(hierarchy, ss) {
+ struct container_subsys_state *css;
+ css = cont->subsys[ss->subsys_id];
+ if (!css_busy) atomic_dec(&css->refcnt);
+ spin_unlock(&css->refcnt_lock);
+ }
+ local_irq_restore(flags);
+ if (css_busy) {
+ mutex_unlock(&manage_mutex);
+ return -EBUSY;
+ }
+
+ for_each_subsys(hierarchy, ss) {
+ if (cont->subsys[ss->subsys_id])
+ ss->destroy(ss, cont);
+ }
+
+ mutex_lock(&callback_mutex);
+ set_bit(CONT_REMOVED, &cont->flags);
+ list_del(&cont->sibling); /* delete my sibling from parent->children */
@@ -1165,67 +1541,143 @@ static int container_rmdir(struct inode
d = dget(cont->dentry);
cont->dentry = NULL;
spin_unlock(&d->d_lock);
+
+ container_d_remove_dir(d);
+ dput(d);
- number_of_containers--;
+ root->number_of_containers--;
+ mutex_unlock(&callback_mutex);
-#ifdef CONFIG_CPUSETS
- cpuset_destroy(cont);
-#endif

```

```

+
+ if (list_empty(&parent->children))
+   check_for_release(parent, &pathbuf);
+
+   mutex_unlock(&manage_mutex);
- container_release_agent(pathbuf);
+ /* Drop the active superblock reference that we took when we
+  * created the container */
+ deactivate_super(sb);
+ container_release_agent(hierarchy, pathbuf);
+   return 0;
+ }

-/*
- * container_init_early - probably not needed yet, but will be needed
- * once cpusets are hooked into this code
- */
+
+/**
+ * container_init_early - initialize containers at system boot
+ *
+ * Description: Initialize the container housekeeping structures
+ **/

int __init container_init_early(void)
{
- struct task_struct *tsk = current;
+ int i;
+
+ for (i = 0; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+   struct containerfs_root *root = &rootnode[i];
+   struct container *cont = &root->top_container;
+   INIT_LIST_HEAD(&root->subsys_list);
+   root->number_of_containers = 1;
+
+   cont->root = root;
+   cont->hierarchy = i;
+   INIT_LIST_HEAD(&cont->sibling);
+   INIT_LIST_HEAD(&cont->children);
+   cont->top_container = cont;
+   atomic_set(&cont->count, 1);
+ }
+ init_task.container[0] = &rootnode[0].top_container;

- tsk->container = &top_container;
+   return 0;
+ }

```

```

/**
- * container_init - initialize containers at system boot
- *
- * Description: Initialize top_container and the container internal file system,
+ * container_init - register container filesystem and /proc file
**/

```

```

int __init container_init(void)
{
- struct dentry *root;
  int err;
-
- init_task.container = &top_container;
+ struct proc_dir_entry *entry;

  err = register_filesystem(&container_fs_type);
  if (err < 0)
    goto out;
- container_mount = kern_mount(&container_fs_type);
- if (IS_ERR(container_mount)) {
-   printk(KERN_ERR "container: could not mount!\n");
-   err = PTR_ERR(container_mount);
-   container_mount = NULL;
-   goto out;
- }
- root = container_mount->mnt_sb->s_root;
- root->d_fsdata = &top_container;
- root->d_inode->i_nlink++;
- top_container.dentry = root;
- root->d_inode->i_op = &container_dir_inode_operations;
- number_of_containers = 1;
- err = container_populate_dir(&top_container);
+
+ entry = create_proc_entry("containers", 0, NULL);
+ if (entry)
+   entry->proc_fops = &proc_containerstats_operations;
+
  out:
  return err;
}

```

```

+int container_register_subsys(struct container_subsys *new_subsys) {
+ int retval = 0;
+ int i;
+
+ BUG_ON(new_subsys->hierarchy);
+ mutex_lock(&manage_mutex);
+ if (subsys_count == CONFIG_MAX_CONTAINER_SUBSYS) {

```



```

+ read_lock(&tasklist_lock);
+ do_each_thread(g, p) {
+   new_subsys->fork(new_subsys, p);
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+ }
+
+ subsys[subsys_count++] = new_subsys;
+ mutex_unlock(&callback_mutex);
+ out:
+ mutex_unlock(&manage_mutex);
+ return retval;
+
+}
+
+/**
+ * container_fork - attach newly forked task to its parents container.
+ * @tsk: pointer to task_struct of forking parent process.
@@ -1246,10 +1698,39 @@ out:

void container_fork(struct task_struct *child)
{
+ int i, need_callback;
+
+ rcu_read_lock();
+ /* need_forkexit_callback will be true if we might need to do
+  * a callback. If so then switch from RCU to mutex locking */
+ need_callback = rcu_dereference(need_forkexit_callback);
+ if (need_callback) {
+   rcu_read_unlock();
+   mutex_lock(&callback_mutex);
+ }
+ task_lock(current);
- child->container = current->container;
- atomic_inc(&child->container->count);
+   /* Add the child task to the same container as the parent in
+   * each hierarchy. Skip hierarchy 0 since it's permanent */
+ for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+   struct container *cont = current->container[i];
+   if (!cont) continue;
+   child->container[i] = cont;
+   atomic_inc(&cont->count);
+ }
+ if (need_callback) {
+   for (i = 0; i < subsys_count; i++) {
+     struct container_subsys *ss = subsys[i];
+     if (ss->fork) {
+       ss->fork(ss, child);

```

```

+ }
+ }
+ }
    task_unlock(current);
+ if (need_callback) {
+ mutex_unlock(&callback_mutex);
+ } else {
+ rcu_read_unlock();
+ }
}

/**
@@ -1314,71 +1795,49 @@ void container_fork(struct task_struct *
void container_exit(struct task_struct *tsk)
{
    struct container *cont;
-
- cont = tsk->container;
- tsk->container = &top_container; /* the_top_container_hack - see above */
-
- if (notify_on_release(cont)) {
-     char *pathbuf = NULL;
-
-     mutex_lock(&manage_mutex);
-     if (atomic_dec_and_test(&cont->count))
-         check_for_release(cont, &pathbuf);
-     mutex_unlock(&manage_mutex);
-     container_release_agent(pathbuf);
+ int i;
+ rcu_read_lock();
+ if (rcu_dereference(need_forkexit_callback)) {
+     rcu_read_unlock();
+     mutex_lock(&callback_mutex);
+     for (i = 0; i < subsys_count; i++) {
+         struct container_subsys *ss = subsys[i];
+         if (ss->exit) {
+             ss->exit(ss, tsk);
+         }
+     }
+     mutex_unlock(&callback_mutex);
+ } else {
-     atomic_dec(&cont->count);
+ rcu_read_unlock();
+ }
-}
-
-/**
- * container_lock - lock out any changes to container structures

```

```

- *
- * The out of memory (oom) code needs to mutex_lock containers
- * from being changed while it scans the tasklist looking for a
- * task in an overlapping container. Expose callback_mutex via this
- * container_lock() routine, so the oom code can lock it, before
- * locking the task list. The tasklist_lock is a spinlock, so
- * must be taken inside callback_mutex.
- */
-
-void container_lock(void)
-{
- mutex_lock(&callback_mutex);
-}
-
-/**
- * container_unlock - release lock on container changes
- *
- * Undo the lock taken in a previous container_lock() call.
- */
-
-void container_unlock(void)
-{
- mutex_unlock(&callback_mutex);
-}
-
-void container_manage_lock(void)
-{
- mutex_lock(&manage_mutex);
-}
-
-/**
- * container_manage_unlock - release lock on container changes
- *
- * Undo the lock taken in a previous container_manage_lock() call.
- */
-
-void container_manage_unlock(void)
-{
- mutex_unlock(&manage_mutex);
+ /* For each hierarchy, remove the task from its current
+ * container in that hierarchy and attach it to the top
+ * container instead. Skip hierarchy 0 since it never has
+ * subcontainers */
+ for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+ cont = tsk->container[i];
+ if (!cont) continue;
+ /* the_top_container_hack - see above */
+ tsk->container[i] = cont->top_container;

```

```

+ if (notify_on_release(cont)) {
+   char *pathbuf = NULL;
+   int hierarchy;
+   mutex_lock(&manage_mutex);
+   hierarchy = cont->hierarchy;
+   if (atomic_dec_and_test(&cont->count))
+     check_for_release(cont, &pathbuf);
+   mutex_unlock(&manage_mutex);
+   container_release_agent(hierarchy, pathbuf);
+ } else {
+   atomic_dec(&cont->count);
+ }
+ }
+ }
+ }

-
-
/*
 * proc_container_show()
- * - Print tasks container path into seq_file.
+ * - 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,
@@ -1387,12 +1846,15 @@ void container_manage_unlock(void)
 *   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;
+ int i;

    retval = -ENOMEM;
    buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
@@ -1405,14 +1867,26 @@ static int proc_container_show(struct se
    if (!tsk)
        goto out_free;

- retval = -EINVAL;
+ retval = 0;
+
+   mutex_lock(&manage_mutex);

```

```

- retval = container_path(tsk->container, buf, PAGE_SIZE);
- if (retval < 0)
- goto out_unlock;
- seq_puts(m, buf);
- seq_putc(m, '\n');
+ for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+ struct containerfs_root *root = &rootnode[i];
+ struct container_subsys *ss;
+ int count = 0;
+ /* Skip this hierarchy if it has no active subsystems */
+ if (!root->subsys_bits) continue;
+ for_each_subsys(i, ss) {
+ seq_printf(m, "%s%s", count++ ? ", " : "", ss->name);
+ }
+ seq_putc(m, ':');
+ retval = container_path(tsk->container[i], buf, PAGE_SIZE);
+ if (retval < 0)
+ goto out_unlock;
+ seq_puts(m, buf);
+ seq_putc(m, '\n');
+ }
out_unlock:
mutex_unlock(&manage_mutex);
put_task_struct(tsk);
@@ -1434,3 +1908,48 @@ struct file_operations proc_container_op
.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;
+ mutex_lock(&manage_mutex);
+ seq_puts(m, "Hierarchies:\n");
+ for (i = 0; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+ struct containerfs_root *root = &rootnode[i];
+ struct container_subsys *ss;
+ int first = 1;
+ seq_printf(m, "%d: topcount=%d bits=%lx containers=%d (",
+ i, atomic_read(&root->top_container.count),
+ root->subsys_bits, root->number_of_containers);
+ for_each_subsys(i, ss) {
+ seq_printf(m, "%s%s", first ? "" : ", ", ss->name);
+ first = false;
+ }
+ seq_putc(m, '\n');

```

```
+ 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 < subsys_count; i++) {
+ struct container_subsys *ss = subsys[i];
+ seq_printf(m, "%d: name=%s hierarchy=%d\n",
+ i, ss->name, ss->hierarchy);
+ }
+ mutex_unlock(&manage_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,
+};
```

Index: container-2.6.20-rc1/kernel/cpuset.c

=====

--- container-2.6.20-rc1.orig/kernel/cpuset.c

+++ container-2.6.20-rc1/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
```

@@ -61,6 +63,10 @@

```
*/
```

```

int number_of_cpusets __read_mostly;

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

struct fmeter {
@@ -71,11 +77,12 @@ 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 */

- struct container *container; /* Task container */
+ struct cpuset *parent; /* my parent */

+/*
@@ -87,6 +94,26 @@ 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.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),
+ struct cpuset, css);
+}
+
+/* Retrieve the cpuset for a task */
+static inline struct cpuset *task_cs(struct task_struct *task)
+{
+ return container_cs(task_container(task, &cpuset_subsys));
+}
+
+/* bits in struct cpuset flags field */

```

```

typedef enum {
    CS_CPU_EXCLUSIVE,
@@ -158,12 +185,16 @@ static int cpuset_get_sb(struct file_sys
{
    struct file_system_type *container_fs = get_fs_type("container");
    int ret = -ENODEV;
- container_set_release_agent_path("/sbin/cpuset_release_agent ");
    if (container_fs) {
        ret = container_fs->get_sb(container_fs, flags,
            unused_dev_name,
-        data, mnt);
+        "cpuset", mnt);
        put_filesystem(container_fs);
+    if (!ret) {
+    container_set_release_agent_path(
+    &cpuset_subsys,
+    "/sbin/cpuset_release_agent");
+    }
    }
    return ret;
}
@@ -270,20 +301,19 @@ void cpuset_update_task_memory_state(voi
struct task_struct *tsk = current;
struct cpuset *cs;

- if (tsk->container->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->container->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) {
    container_lock();
    task_lock(tsk);
- cs = tsk->container->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))
@@ -342,9 +372,8 @@ static int validate_change(const struct
struct cpuset *c, *par;

```

```

/* Each of our child cpusets must be a subset of us */
- list_for_each_entry(cont, &cur->container->children, sibling) {
- c = cont->cpuset;
- 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;
}

@@ -359,8 +388,8 @@ 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(cont, &par->container->children, sibling) {
- c = cont->cpuset;
+ 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))
@@ -402,8 +431,8 @@ static void update_cpu_domains(struct cp
    * children
    */
    pspan = par->cpus_allowed;
- list_for_each_entry(cont, &par->container->children, sibling) {
- c = cont->cpuset;
+ list_for_each_entry(cont, &par->css.container->children, sibling) {
+ c = container_cs(cont);
    if (is_cpu_exclusive(c))
        cpus_andnot(pspan, pspan, c->cpus_allowed);
}

@@ -420,8 +449,8 @@ static void update_cpu_domains(struct cp
    * Get all cpus from current cpuset's cpus_allowed not part
    * of exclusive children
    */
- list_for_each_entry(cont, &cur->container->children, sibling) {
- c = cont->cpuset;
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+ c = container_cs(cont);
    if (is_cpu_exclusive(c))
        cpus_andnot(cspan, cspan, c->cpus_allowed);
}

@@ -509,7 +538,7 @@ static void cpuset_migrate_mm(struct mm_
    do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);

    container_lock();
- guarantee_online_mems(tsk->container->cpuset, &tsk->mems_allowed);
+ guarantee_online_mems(task_cs(tsk), &tsk->mems_allowed);

```

```

    container_unlock();
}

@@ -527,6 +556,8 @@ 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;
@@ -544,7 +575,7 @@ static int update_nodemask(struct cpuset
    return -EACCES;

    trialcs = *cs;
-    cont = cs->container;
+    cont = cs->css.container;
    retval = nodelist_parse(buf, trialcs.mems_allowed);
    if (retval < 0)
        goto done;
@@ -567,7 +598,7 @@ static int update_nodemask(struct cpuset
    cs->mems_generation = cpuset_mems_generation++;
    container_unlock();

-    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 */
@@ -581,13 +612,13 @@ static int update_nodemask(struct cpuset
    * enough mmarray[] w/o using GFP_ATOMIC.
    */
    while (1) {
-        ntasks = atomic_read(&cs->container->count); /* guess */
+        ntasks = atomic_read(&cs->css.container->count); /* 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->container->count) <= ntasks)
+        if (atomic_read(&cs->css.container->count) <= ntasks)
            break; /* got enough */
        write_unlock_irq(&tasklist_lock); /* try again */
        kfree(mmarray);
@@ -604,7 +635,7 @@ static int update_nodemask(struct cpuset
    "Cpuset mempolicy rebind incomplete.\n");

```

```

    continue;
}
- if (p->container != cont)
+ if (task_cs(p) != cs)
    continue;
    mm = get_task_mm(p);
    if (!mm)
@@ -638,12 +669,17 @@ 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.
 */
@@ -794,9 +830,10 @@ static int fmeter_getrate(struct fmeter
    return val;
}

-int cpuset_can_attach_task(struct container *cont, struct task_struct *tsk)
+int cpuset_can_attach(struct container_subsys *ss,
+    struct container *cont, struct task_struct *tsk)
{
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);

    if (cpus_empty(cs->cpus_allowed) || nodes_empty(cs->mems_allowed))
        return -ENOSPC;
@@ -804,22 +841,23 @@ int cpuset_can_attach_task(struct contai
    return security_task_setscheduler(tsk, 0, NULL);
}

-void cpuset_attach_task(struct container *cont, struct task_struct *tsk)
+void cpuset_attach(struct container_subsys *ss, struct container *cont,
+    struct container *old_cont, struct task_struct *tsk)
{
    cpumask_t cpus;

```

```

- struct cpuset *cs = cont->cpuset;
- guarantee_online_cpus(cs, &cpus);
+ guarantee_online_cpus(container_cs(cont), &cpus);
  set_cpus_allowed(tsk, cpus);
}

-void cpuset_post_attach_task(struct container *cont,
-    struct container *oldcont,
-    struct task_struct *tsk)
+void cpuset_post_attach(struct container_subsys *ss,
+    struct container *cont,
+    struct container *oldcont,
+    struct task_struct *tsk)
{
    nodemask_t from, to;
    struct mm_struct *mm;
- struct cpuset *cs = cont->cpuset;
- struct cpuset *oldcs = oldcont->cpuset;
+ struct cpuset *cs = container_cs(cont);
+ struct cpuset *oldcs = container_cs(oldcont);

    from = oldcs->mems_allowed;
    to = cs->mems_allowed;
@@ -853,7 +891,7 @@ static ssize_t cpuset_common_file_write(
    const char __user *userbuf,
    size_t nbytes, loff_t *unused_ppos)
{
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);
    cpuset_filetype_t type = cft->private;
    char *buffer;
    int retval = 0;
@@ -963,7 +1001,7 @@ static ssize_t cpuset_common_file_read(s
    char __user *buf,
    size_t nbytes, loff_t *ppos)
{
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);
    cpuset_filetype_t type = cft->private;
    char *page;
    ssize_t retval = 0;
@@ -1081,7 +1119,7 @@ static struct cftype cft_spread_slab = {
    .private = FILE_SPREAD_SLAB,
};

-int cpuset_populate_dir(struct container *cont)
+int cpuset_populate(struct container_subsys *ss, struct container *cont)
{

```

```

int err;

@@ -1116,11 +1154,19 @@ int cpuset_populate_dir(struct container
 * Must be called with the mutex on the parent inode held
 */

-int cpuset_create(struct container *cont)
+int cpuset_create(struct container_subsys *ss, struct container *cont)
{
    struct cpuset *cs;
- struct cpuset *parent = cont->parent->cpuset;
+ 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;
@@ -1137,8 +1183,8 @@ int cpuset_create(struct container *cont
    fmeter_init(&cs->fmeter);

    cs->parent = parent;
- cont->cpuset = cs;
- cs->container = cont;
+ set_container_cs(cont, cs);
+ cs->css.container = cont;
    number_of_cpuset++;
    return 0;
}
@@ -1154,9 +1200,9 @@ int cpuset_create(struct container *cont
 * nesting would risk an ABBA deadlock.
 */

-void cpuset_destroy(struct container *cont)
+void cpuset_destroy(struct container_subsys *ss, struct container *cont)
{
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);

    cpuset_update_task_memory_state();
    if (is_cpu_exclusive(cs)) {
@@ -1164,8 +1210,20 @@ void cpuset_destroy(struct container *co

```

```

    BUG_ON(retval);
}
number_of_cpuset--;
+ kfree(cs);
}

+static struct container_subsys cpuset_subsys = {
+ .name = "cpuset",
+ .create = cpuset_create,
+ .destroy = cpuset_destroy,
+ .can_attach = cpuset_can_attach,
+ .attach = cpuset_attach,
+ .post_attach = cpuset_post_attach,
+ .populate = cpuset_populate,
+ .subsys_id = -1,
+};
+
/*
 * cpuset_init_early - just enough so that the calls to
 * cpuset_update_task_memory_state() in early init code
@@ -1174,13 +1232,13 @@ void cpuset_destroy(struct container *co

int __init cpuset_init_early(void)
{
- struct container *cont = current->container;
- cont->cpuset = &top_cpuset;
- top_cpuset.container = cont;
- cont->cpuset->mems_generation = cpuset_mems_generation++;
+ if (container_register_subsys(&cpuset_subsys) < 0)
+ panic("Couldn't register cpuset subsystem");
+ top_cpuset.mems_generation = cpuset_mems_generation++;
    return 0;
}

+
/**
 * cpuset_init - initialize cpusets at system boot
 *
@@ -1190,6 +1248,7 @@ int __init cpuset_init_early(void)
int __init cpuset_init(void)
{
    int err = 0;
+
    top_cpuset.cpus_allowed = CPU_MASK_ALL;
    top_cpuset.mems_allowed = NODE_MASK_ALL;

@@ -1231,8 +1290,8 @@ static void guarantee_online_cpus_mems_i
    struct cpuset *c;

```

```

/* Each of our child cpusets mems must be online */
- list_for_each_entry(cont, &cur->container->children, sibling) {
- c = cont->cpuset;
+ 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);
@@ -1330,7 +1389,7 @@ cpumask_t cpuset_cpus_allowed(struct tas

  container_lock();
  task_lock(tsk);
- guarantee_online_cpus(tsk->container->cpuset, &mask);
+ guarantee_online_cpus(task_cs(tsk), &mask);
  task_unlock(tsk);
  container_unlock();

@@ -1358,7 +1417,7 @@ nodemask_t cpuset_mems_allowed(struct ta

  container_lock();
  task_lock(tsk);
- guarantee_online_mems(tsk->container->cpuset, &mask);
+ guarantee_online_mems(task_cs(tsk), &mask);
  task_unlock(tsk);
  container_unlock();

@@ -1479,7 +1538,7 @@ int __cpuset_zone_allowed_softwall(struc
  container_lock();

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

  allowed = node_isset(node, cs->mems_allowed);
@@ -1581,7 +1640,7 @@ int cpuset_excl_nodes_overlap(const stru
  task_unlock(current);
  goto done;
}
- cs1 = nearest_exclusive_ancestor(current->container->cpuset);
+ cs1 = nearest_exclusive_ancestor(task_cs(current));
  task_unlock(current);

  task_lock((struct task_struct *)p);
@@ -1589,7 +1648,7 @@ int cpuset_excl_nodes_overlap(const stru
  task_unlock((struct task_struct *)p);
  goto done;

```

```

}
- cs2 = nearest_exclusive_ancestor(p->container->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);
@@ -1625,11 +1684,8 @@ int cpuset_memory_pressure_enabled __read

```

```

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

```

```

@@ -1666,7 +1722,8 @@ static int proc_cpuset_show(struct seq_file
  retval = -EINVAL;
  container_manage_lock();

```

```

- retval = container_path(tsk->container, buf, PAGE_SIZE);
+ retval = container_path(tsk->container[cpuset_subsys.hierarchy],
+   buf, PAGE_SIZE);
  if (retval < 0)
    goto out_unlock;
  seq_puts(m, buf);

```

Index: container-2.6.20-rc1/Documentation/containers.txt

=====

--- container-2.6.20-rc1.orig/Documentation/containers.txt

+++ container-2.6.20-rc1/Documentation/containers.txt

@@ -21,8 +21,11 @@ CONTENTS:

2. Usage Examples and Syntax

2.1 Basic Usage

2.2 Attaching processes

-3. Questions

-4. Contact

+3. Kernel API

+ 3.1 Overview

+ 3.2 Synchronization

+ 3.3 Subsystem API

+4. Questions

1. Containers

=====

@@ -30,14 +33,18 @@ CONTENTS:

1.1 What are containers ?

-Containers provide a mechanism for aggregating sets of tasks, and all their children, into hierarchical groups.

-

-Each task has a pointer to a container. Multiple tasks may reference the same container. User level code may create and destroy containers by name in the container virtual file system, specify and query to which container a task is assigned, and list the task pids assigned to a container.

+Containers provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups. A container associates a set of tasks with a set of parameters for one or more "subsystems" (typically resource controllers).

+

+At any one time there may be up to CONFIG_MAX_CONTAINER_HIERARCHIES active hierarchies of task containers. Each task has a pointer to a container in each active hierarchy. Multiple tasks may reference (i.e. be members of) the same container. 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.

On their own, the only use for containers is for simple job tracking. The intention is that other subsystems, such as cpusets (see @@ -51,9 +58,9 @@ resources which processes in a container There are multiple efforts to provide process aggregations in the Linux kernel, mainly for resource tracking purposes. Such efforts include cpusets, CKRM/ResGroups, and UserBeanCounters. These all require the basic notion of a grouping of processes, with newly forked processes ending in the same group (container) as their parent process.

+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 @@ -67,27 +74,46 @@ desired.

Containers extends the kernel as follows:

- - Each task in the system is attached to a container, via a pointer in the task structure to a reference counted container structure.
- - The hierarchy of containers can be mounted at /dev/container (or elsewhere), for browsing and manipulation from user space.
- + - Each task in the system has set of reference-counted container

- + pointers, one for each active hierarchy
- + - 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 container at system boot.
- - in fork and exit, to attach and detach a task from its container.
-
- In addition a new file system, of type "container" may be mounted, typically at /dev/container, to enable browsing and modifying the containers presently known to the kernel. No new system calls are added for containers - all support for querying and modifying containers is via this container file system.
- + - in init/main.c, to initialize the root containers at system boot.
- + - in fork and exit, to attach and detach a task from its containers.
- Each task under /proc has an added file named 'container', displaying the container name, as the path relative to the root of the container file system.
- +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 created, 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.
- +
- +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:

@@ -119,23 +145,27 @@ for containers, with a minimum of additi

1.4 What does notify_on_release do ?

-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 /sbin/container_release_agent, 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.
+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, the steps are:
+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 container /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.
- @@ -147,7 +177,7 @@ 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 container:

```
- mount -t container none /dev/container
+ mount -t container cpuset -ocpuset /dev/container
  cd /dev/container
  mkdir Charlie
  cd Charlie
@@ -157,11 +187,6 @@ and then start a subshell 'sh' in that c
  # The next line should display '/Charlie'
  cat /proc/self/container
```

-In the future, a C library interface to containers will likely be available. For now, the only way to query or modify containers is via the container file system, using the various cd, mkdir, echo, cat, rmdir commands from the shell, or their equivalent from C.

2. Usage Examples and Syntax

=====

@@ -171,8 +196,15 @@ rmdir commands from the shell, or their
Creating, modifying, using the containers can be done through the container virtual filesystem.

```
-To mount it, type:
-# mount -t container none /dev/container
+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
```

Then under /dev/container you can find a tree that corresponds to the tree of the containers in the system. For instance, /dev/container
@@ -187,7 +219,8 @@ Now you want to do something with this c

```
In this directory you can find several files:
# ls
-notify_on_release tasks
+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
@@ -214,8 +247,150 @@ If you have several tasks to attach, you
...
# /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 are two global mutexes used by the container system. The first is the `manage_mutex`, which should be taken by anything that wants to modify a container; The second is the `callback_mutex`, which should be taken by holders of the `manage_mutex` at the point when they actually make changes, and by callbacks from lower-level subsystems that want to ensure that no container changes occur. Note that memory allocations cannot be made while holding `callback_mutex`.

+

+The `callback_mutex` nests inside the `manage_mutex`.

+

+In general, the pattern of use is:

+

```

+1) take manage_mutex
+2) verify that the change is valid and do any necessary allocations\
+3) take callback_mutex
+4) make changes
+5) release callback_mutex
+6) release manage_mutex
+
+See kernel/container.c for more details.
+
+Subsystems can take/release the manage_mutex via the functions
+container_manage_lock()/container_manage_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 manage_mutex
+- while holding callback_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 call container_register_subsys() with a pointer
+to its subsystem object. This will store the new subsystem id in the
+subsystem subsys_id field and return 0, or a negative error. There's
+currently no facility for deregistering a subsystem nor for
+registering a subsystem after any containers (other than the default
+"top_container") have been created.
+
+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=manage_mutex
+
+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.
+
+void destroy(struct container *cont)
+LL=manage_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=manage_mutex

+

+Called prior to moving a task into a container; if the subsystem
+returns an error, this will abort the attach operation. Note that
+this isn't called on a fork.

+

+void attach(struct container_subsys *ss, struct container *cont,
+ struct container *old_cont, struct task_struct *task)

+LL=manage_mutex & callback_mutex

+

+Called during the attach operation. The subsystem should do any
+necessary work that can be accomplished without memory allocations or
+sleeping.

+

+void post_attach(struct container_subsys *ss, struct container *cont,
+ struct container *old_cont, struct task_struct *task)

+LL=manage_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). Called during
+container_register_subsys() to populate the root container. Note that
+although this method can return an error code, the error code is
+currently not always handled well.

+

-3. Questions

+4. Questions

=====

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

Index: container-2.6.20-rc1/include/linux/mempolicy.h

=====

--- container-2.6.20-rc1.orig/include/linux/mempolicy.h

+++ container-2.6.20-rc1/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->container->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)

{

Index: container-2.6.20-rc1/include/linux/sched.h

=====

--- container-2.6.20-rc1.orig/include/linux/sched.h

+++ container-2.6.20-rc1/include/linux/sched.h

@@ -1030,7 +1030,7 @@ struct task_struct {
int cpuset_mem_spread_rotor;

```

#endif
#ifdef CONFIG_CONTAINERS
- struct container *container;
+ struct container *container[CONFIG_MAX_CONTAINER_HIERARCHIES];
#endif
    struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
@@ -1469,7 +1469,7 @@ 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 ->container.
+ * pins the final release of task.io_context. Also protects ->container[].
 *
 * Nests both inside and outside of read_lock(&tasklist_lock).
 * It must not be nested with write_lock_irq(&tasklist_lock),
Index: container-2.6.20-rc1/mm/mempolicy.c
=====
--- container-2.6.20-rc1.orig/mm/mempolicy.c
+++ container-2.6.20-rc1/mm/mempolicy.c
@@ -1309,7 +1309,6 @@ 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)
@@ -1908,4 +1907,3 @@ out:
    m->version = (vma != priv->tail_vma) ? vma->vm_start : 0;
    return 0;
}
-
Index: container-2.6.20-rc1/init/Kconfig
=====
--- container-2.6.20-rc1.orig/init/Kconfig
+++ container-2.6.20-rc1/init/Kconfig
@@ -241,6 +241,18 @@ config IKCONFIG_PROC
config CONTAINERS
    bool

+config MAX_CONTAINER_SUBSYS
+    int "Number of container subsystems to support"
+    depends on CONTAINERS
+    range 1 255
+    default 8
+
+config MAX_CONTAINER_HIERARCHIES

```

- + int "Number of container hierarchies to support"
- + depends on CONTAINERS
- + range 2 255
- + default 4
- +

```
config CPUSSETS
    bool "Cpuset support"
    depends on SMP
```

Index: container-2.6.20-rc1/Documentation/cpusets.txt

=====

--- container-2.6.20-rc1.orig/Documentation/cpusets.txt

+++ container-2.6.20-rc1/Documentation/cpusets.txt

@@ -466,7 +466,7 @@ 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 container 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.
- @@ -478,7 +478,7 @@ 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 container none /dev/cpuset
+ mount -t container -ocpuset cpuset /dev/cpuset
  cd /dev/cpuset
  mkdir Charlie
  cd Charlie
@@ -488,7 +488,7 @@ and then start a subshell 'sh' in that c
  sh
  # The subshell 'sh' is now running in cpuset Charlie
  # The next line should display '/Charlie'
- cat /proc/self/container
+ cat /proc/self/cpuset
```

In the future, a C library interface to cpusets will likely be available. For now, the only way to query or modify cpusets is

@@ -510,7 +510,7 @@ Creating, modifying, using the cpusets c virtual filesystem.

To mount it, type:

```
-# mount -t container 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

@@ -550,6 +550,18 @@ 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

--
