

---

Subject: [PATCH 3/7] Containers (V8): Add generic multi-subsystem API to containers

Posted by [Paul Menage](#) on Fri, 06 Apr 2007 23:32:24 GMT

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

---

This patch removes all cpuset-specific knowlege 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 | 415 ++++++-----
Documentation/cpusets.txt   | 20
include/linux/container.h   | 156 +++-
include/linux/container_subsys.h | 14
include/linux/cpuset.h      | 14
include/linux/mempolicy.h   | 12
include/linux/sched.h       | 4
kernel/container.c          | 1465 ++++++-----
kernel/cpuset.c             | 161 +++-
kernel/exit.c               | 2
kernel/fork.c               | 9
mm/mempolicy.c              | 2
12 files changed, 1712 insertions(+), 562 deletions(-)
```

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

```
=====
--- container-2.6.20-new.orig/include/linux/container.h
+++ container-2.6.20-new/include/linux/container.h
@@ -9,33 +9,96 @@
 */

#include <linux/sched.h>
#include <linux/kref.h>
#include <linux/cpumask.h>
#include <linux/nodemask.h>

#ifdef CONFIG_CONTAINERS

extern int number_of_containers; /* How many containers are defined in system? */
#define SUBSYS(_x) _x ## _subsys_id,
enum container_subsys_id {
#include <linux/container_subsys.h>
+ CONTAINER_SUBSYS_COUNT
+};
+#undef SUBSYS
```

```

extern int container_init_early(void);
extern int container_init(void);
extern void container_init_smp(void);
extern void container_fork(struct task_struct *p);
-extern void container_exit(struct task_struct *p);
+extern void container_fork_callbacks(struct task_struct *p);
+extern void container_exit(struct task_struct *p, int run_callbacks);

extern struct file_operations proc_container_operations;

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;
+};
+
+/* A container_group is a structure holding pointers to a set of
+ * containers. This saves space in the task struct object and speeds
+ * up fork()/exit(), since a single inc/dec can bump the reference
+ * count on the entire container set for a task. */
+
+struct container_group {
+
+ /* Reference count */
+ struct kref ref;
+
+ /* List running through all container groups */
+ struct list_head list;
+
+ /* Set of subsystem states, one for each subsystem. NULL for
+ * subsystems that aren't part of this hierarchy. These
+ * pointers reduce the number of dereferences required to get
+ * from a task to its state for a given container, but result
+ * in increased space usage if tasks are in wildly different
+ * groupings across different hierarchies. This array is

```

```

+ * mostly immutable after creation - a newly registered
+ * subsystem can result in a pointer in this array
+ * transitioning from NULL to non-NULL */
+ struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
+
+};
+
+/*
+ * 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 is atomic so can incr (fork) or decr (exit) without a lock.
+  - */
+  - atomic_t count; /* count tasks using this container */
+  -
+  - /*
+   * We link our 'sibling' struct into our parent's 'children'.
+   * Our children link their 'sibling' into our 'children'.
+   */
+ @@ -43,11 +106,13 @@ 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 */
+
+ #ifdef CONFIG_CPUSETS
+  - struct cpuset *cpuset;
+ #endif

```

```

+ /* Private pointers for each registered subsystem */
+ struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
+
+ struct containerfs_root *root;
+ struct container *top_container;
};

/* struct cftype:
@@ -64,8 +129,11 @@ 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,
@@ -77,10 +145,72 @@ struct cftype {
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);

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);
+
+int container_task_count(const struct container *cont);
+
+/* Return true if the container is a descendant of the current container */
+int container_is_descendant(const struct container *cont);
+
+/* Container subsystem type. See Documentation/containers.txt for details */
+
+struct container_subsys {
+ int (*create)(struct container_subsys *ss,
+ struct container *cont);
+ void (*destroy)(struct container_subsys *ss, struct container *cont);
+ int (*can_attach)(struct container_subsys *ss,
+ struct container *cont, struct task_struct *tsk);
+ void (*attach)(struct container_subsys *ss, struct container *cont,

```

```

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

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

```

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

```

=====
--- container-2.6.20-new.orig/include/linux/cpuset.h
+++ container-2.6.20-new/include/linux/cpuset.h
@@ -74,14 +74,7 @@ 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 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 */

@@ -152,6 +145,11 @@ static inline int cpuset_do_slab_mem_spr

static inline void cpuset_track_online_nodes(void) {}

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

#endif /* _LINUX_CPUSSET_H */
Index: container-2.6.20-new/kernel/container.c
=====
--- container-2.6.20-new.orig/kernel/container.c
+++ container-2.6.20-new/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>
@@ -63,17 +62,59 @@

#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.
+/* 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.
 */
-int number_of_containers __read_mostly;
+static int need_forkexit_callback = 0;

/* bits in struct container flags field */
typedef enum {
    CONT_REMOVED,
- CONT_NOTIFY_ON_RELEASE,
} container_flagbits_t;

/* convenient tests for these bits */
@@ -82,31 +123,153 @@ inline int container_is_removed(const st
    return test_bit(CONT_REMOVED, &cont->flags);
}

-static inline int notify_on_release(const struct container *cont)
- {
- return test_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+/* for_each_subsys() allows you to iterate on each subsystem attached to
+ * an active hierarchy */
+#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)
+
+/* The default container group - used by init and its children prior
+ * to any hierarchies being mounted. It contains a pointer to the root
+ * state for each subsystem. Also used to anchor the list of container
+ * groups */
+static struct container_group init_container_group;
+
+/*
+ * This lock nests inside tasklist lock, which can be taken by
+ * interrupts, and hence always needs to be taken with
+ * spin_lock_irq*
+ */
+static DEFINE_SPINLOCK(container_group_lock);
+static int container_group_count;
+
+/*
+ * unlink a container_group from the list and free it
+ */
+static void release_container_group(struct kref *k) {
+ struct container_group *cg =

```



```

+ container_of(k, struct container_group, ref);
+ unsigned long flags;
+ spin_lock_irqsave(&container_group_lock, flags);
+ list_del(&cg->list);
+ container_group_count--;
+ spin_unlock_irqrestore(&container_group_lock, flags);
+ kfree(cg);
}

-static struct container top_container = {
- .count = ATOMIC_INIT(0),
- .sibling = LIST_HEAD_INIT(top_container.sibling),
- .children = LIST_HEAD_INIT(top_container.children),
-};
+/*
+ * refcounted get/put for container_group objects
+ */
+static inline void get_container_group(struct container_group *cg) {
+ kref_get(&cg->ref);
+}

-/* 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";
+static inline void put_container_group(struct container_group *cg) {
+ kref_put(&cg->ref, release_container_group);
+}
+
+/*
+ * find_existing_container_group() is a helper for
+ * find_container_group(), and checks to see whether an existing
+ * container_group is suitable. This currently walks a linked-list for
+ * simplicity; a later patch will use a hash table for better
+ * performance
+ *
+ * oldcg: the container group that we're using before the container
+ * transition
+ *
+ * cont: the container that we're moving into
+ *
+ * template: location in which to build the desired set of subsystem
+ * state objects for the new container group
+ */

-void container_set_release_agent_path(const char *path)
+static struct container_group *find_existing_container_group(
+ struct container_group *oldcg,

```

```

+ struct container *cont,
+ struct container_subsys_state *template[]
{
- container_lock();
- strcpy(release_agent_path, path);
- container_unlock();
+ int i;
+ struct containerfs_root *root = cont->root;
+ struct list_head *l = &init_container_group.list;
+
+ /* Built the set of subsystem state objects that we want to
+ * see in the new container_group */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ if (root->subsys_bits & (1ull << i)) {
+ /* Subsystem is in this hierarchy. So we want
+ * the subsystem state from the new
+ * container */
+ template[i] = cont->subsys[i];
+ } else {
+ /* Subsystem is not in this hierarchy, so we
+ * don't want to change the subsystem state */
+ template[i] = oldcg->subsys[i];
+ }
+ }
+
+ /* Look through existing container groups to find one to reuse */
+ do {
+ struct container_group *cg =
+ list_entry(l, struct container_group, list);
+
+ if (!memcmp(template, oldcg->subsys, sizeof(oldcg->subsys))) {
+ /* All subsystems matched */
+ return cg;
+ }
+ /* Try the next container group */
+ l = l->next;
+ } while (l != &init_container_group.list);
+
+ /* No existing container group matched */
+ return NULL;
+ }

-static struct vfsmount *container_mount;
-static struct super_block *container_sb;
+/*
+ * find_container_group() takes an existing container group and a
+ * container object, and returns a container_group object that's
+ * equivalent to the old group, but with the given container

```

```

+ * substituted into the appropriate hierarchy. Must be called with
+ * container_mutex held
+ */
+
+static struct container_group *find_container_group(
+ struct container_group *oldcg, struct container *cont)
+{
+ struct container_group *res;
+ struct container_subsys_state *template[CONTAINER_SUBSYS_COUNT];
+
+ /* First see if we already have a container group that matches
+ * the desired set */
+ spin_lock_irq(&container_group_lock);
+ res = find_existing_container_group(oldcg, cont, template);
+ if (res)
+ get_container_group(res);
+ spin_unlock_irq(&container_group_lock);
+
+ if (res)
+ return res;
+
+ res = kmalloc(sizeof(*res), GFP_KERNEL);
+ if (!res)
+ return NULL;
+
+ kref_init(&res->ref);
+ /* Copy the set of subsystem state objects generated in
+ * find_existing_container_group() */
+ memcpy(res->subsys, template, sizeof(res->subsys));
+
+ /* Link this container group into the list */
+ spin_lock_irq(&container_group_lock);
+ list_add(&res->list, &init_container_group.list);
+ container_group_count++;
+ spin_unlock_irq(&container_group_lock);
+
+ return res;
+}

/*
 * There is one global container mutexes. We also require taking
@@ -164,6 +327,27 @@ static struct super_block *container_sb;

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
@@ -172,15 +356,18 @@ static DEFINE_MUTEX(container_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;
@@ -208,7 +395,8 @@ static struct dentry_operations containe
    .d_iput = container_diput,
};

-static struct dentry *container_get_dentry(struct dentry *parent, const char *name)
+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))
@@ -225,19 +413,19 @@ static void remove_dir(struct dentry *d)
  dput(parent);
}

-/*
- * NOTE : the dentry must have been dget()'ed
- */
-static void container_d_remove_dir(struct dentry *dentry)
+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);
@@ -247,37 +435,221 @@ static void container_d_remove_dir(struct
  }
  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);
+

```

```

+ 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 *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) {
@@ -285,14 +657,101 @@ static int container_fill_super(struct s
        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)
{
- return get_sb_single(fs_type, flags, data, container_fill_super, 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 = {
@@ -341,134 +800,88 @@ int container_path(const struct containe
return 0;
}

-/*
- * Notify userspace when a container is released, by running
- * /sbin/container_release_agent with the name of the container (path
- * relative to the root of container file system) as the argument.
- *
- * Most likely, this user command will try to rmdir this container.
- *
- * This races with the possibility that some other task will be
- * attached to this container before it is removed, or that some other
- * user task will 'mkdir' a child container of this container. That's ok.
- * The presumed 'rmdir' will fail quietly if this container is no longer
- * unused, and this container will be reprieved from its death sentence,
- * to continue to serve a useful existence. Next time it's released,
- * we will get notified again, if it still has 'notify_on_release' set.
- *
- *

```

```

- * The final arg to call_usermodehelper() is 0, which means don't
- * wait. The separate /sbin/container_release_agent task is forked by
- * call_usermodehelper(), then control in this thread returns here,
- * without waiting for the release agent task. We don't bother to
- * wait because the caller of this routine has no use for the exit
- * status of the /sbin/container_release_agent task, so no sense holding
- * our caller up for that.
- *
- * When we had only one container mutex, we had to call this
- * without holding it, to avoid deadlock when call_usermodehelper()
- * allocated memory. With two locks, we could now call this while
- * holding container_mutex, but we still don't, so as to minimize
- * the time container_mutex is held.
- */
-
-static void container_release_agent(const char *pathbuf)
-{
- char *argv[3], *envp[3];
- int i;
-
- if (!pathbuf)
- return;
-
- i = 0;
- argv[i++] = release_agent_path;
- argv[i++] = (char *)pathbuf;
- argv[i] = NULL;
-
- i = 0;
- /* minimal command environment */
- envp[i++] = "HOME=/";
- envp[i++] = "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
- envp[i] = NULL;
-
- call_usermodehelper(argv[0], argv, envp, 0);
- kfree(pathbuf);
+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;
}

/*
- * Either cont->count of using tasks transitioned to zero, or the
- * cont->children list of child containers just became empty. If this
- * cont is notify_on_release() and now both the user count is zero and
- * the list of children is empty, prepare container path in a kmalloc'd
- * buffer, to be returned via ppathbuf, so that the caller can invoke
- * container_release_agent() with it later on, once container_mutex is dropped.
- * Call here with container_mutex held.
+ * Attach task 'tsk' to container 'cont'
*
- * This check_for_release() routine is responsible for kmalloc'ing
- * pathbuf. The above container_release_agent() is responsible for
- * kfree'ing pathbuf. The caller of these routines is responsible
- * for providing a pathbuf pointer, initialized to NULL, then
- * calling check_for_release() with container_mutex held and the address
- * of the pathbuf pointer, then dropping container_mutex, then calling
- * container_release_agent() with pathbuf, as set by check_for_release().
+ * Call holding container_mutex. May take task_lock of
+ * the task 'pid' during call.
*/

-static void check_for_release(struct container *cont, char **ppathbuf)
+static int attach_task(struct container *cont, struct task_struct *tsk)
{
- if (notify_on_release(cont) && atomic_read(&cont->count) == 0 &&
- list_empty(&cont->children)) {
- char *buf;
-
- buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
- if (!buf)
- return;
+ int retval = 0;
+ struct container_subsys *ss;
+ struct container_group *oldcg, *newcg;
+ struct container *oldcont;
+ struct containerfs_root *root = cont->root;

- if (container_path(cont, buf, PAGE_SIZE) < 0)
- kfree(buf);
- else
- *ppathbuf = buf;
- }
-}
+ int subsys_id;

```

```

+ get_first_subsys(cont, NULL, &subsys_id);

+ /* Nothing to do if the task is already in that container */
+ oldcont = tsk->containers->subsys[subsys_id]->container;
+ if (cont == oldcont)
+ return 0;

-/*
- * update_flag - read a 0 or a 1 in a file and update associated flag
- * bit: the bit to update (CONT_NOTIFY_ON_RELEASE)
- * cont: the container to update
- * buf: the buffer where we read the 0 or 1
- *
- * Call with container_mutex held.
- */
+ for_each_subsys(root, ss) {
+ if (ss->can_attach) {
+ retval = ss->can_attach(ss, cont, tsk);
+ if (retval) {
+ return retval;
+ }
+ }
+ }

-static int update_flag(container_flagbits_t bit, struct container *cont, char *buf)
-{
- int turning_on;
+ /* Locate or allocate a new container_group for this task,
+ * based on its final set of containers */
+ oldcg = tsk->containers;
+ newcg = find_container_group(oldcg, cont);
+ if (!newcg) {
+ return -ENOMEM;
+ }

- turning_on = (simple_strtoul(buf, NULL, 10) != 0);
+ task_lock(tsk);
+ rcu_assign_pointer(tsk->containers, newcg);
+ task_unlock(tsk);

- if (turning_on)
- set_bit(bit, &cont->flags);
- else
- clear_bit(bit, &cont->flags);
+ for_each_subsys(root, ss) {
+ if (ss->attach) {
+ ss->attach(ss, cont, oldcont, tsk);
+ }

```

```

+ }

+ synchronize_rcu();
+ put_container_group(oldcgroup);
  return 0;
}

-
/*
- * 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.
+ * Attach task with pid 'pid' to container 'cont'. Call with
+ * container_mutex, may take task_lock of task
  *
- * Call holding container_mutex. May take task_lock of the task 'pid'
- * during call.
  */

-static int attach_task(struct container *cont, char *pidbuf, char **ppathbuf)
+static int attach_task_by_pid(struct container *cont, char *pidbuf)
{
  pid_t pid;
  struct task_struct *tsk;
- struct container *oldcont;
- int retval = 0;
+ int ret;

  if (sscanf(pidbuf, "%d", &pid) != 1)
    return -EIO;
@@ -495,34 +908,9 @@ 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;
- }
-
- task_lock(tsk);
- oldcont = tsk->container;
- if (tsk->flags & PF_EXITING) {
-   task_unlock(tsk);
-   put_task_struct(tsk);
-   return -ESRCH;

```



```

- }
- atomic_inc(&cont->count);
- rcu_assign_pointer(tsk->container, cont);
- task_unlock(tsk);
-
-#ifdef CONFIG_CPUSETS
- cpuset_attach_task(cont, oldcont, tsk);
-#endif
-
+ ret = attach_task(cont, tsk);
  put_task_struct(tsk);
- synchronize_rcu();
- if (atomic_dec_and_test(&oldcont->count))
- check_for_release(oldcont, ppathbuf);
- return 0;
+ return ret;
}

/* The various types of files and directories in a container file system */
@@ -530,9 +918,7 @@ static int attach_task(struct container
typedef enum {
  FILE_ROOT,
  FILE_DIR,
- FILE_NOTIFY_ON_RELEASE,
  FILE_TASKLIST,
- FILE_RELEASE_AGENT,
} container_filetype_t;

static ssize_t container_common_file_write(struct container *cont,
@@ -543,7 +929,6 @@ static ssize_t container_common_file_wri
{
  container_filetype_t type = cft->private;
  char *buffer;
- char *pathbuf = NULL;
  int retval = 0;

  if (nbytes >= PATH_MAX)
@@ -567,26 +952,9 @@ static ssize_t container_common_file_wri
}

  switch (type) {
- case FILE_NOTIFY_ON_RELEASE:
-   retval = update_flag(CONT_NOTIFY_ON_RELEASE, cont, buffer);
-   break;
  case FILE_TASKLIST:
-   retval = attach_task(cont, buffer, &pathbuf);
+   retval = attach_task_by_pid(cont, buffer);
  break;

```

```

- case FILE_RELEASE_AGENT:
- {
- if (nbytes < sizeof(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;
- } else {
- retval = -ENOSPC;
- }
- break;
- }
default:
    retval = -EINVAL;
    goto out2;
@@ -596,7 +964,6 @@ static ssize_t container_common_file_wri
    retval = nbytes;
out2:
    mutex_unlock(&container_mutex);
- container_release_agent(pathbuf);
out1:
    kfree(buffer);
    return retval;
@@ -605,80 +972,27 @@ out1:
static ssize_t container_file_write(struct file *file, const char __user *buf,
    size_t nbytes, loff_t *ppos)
{
- ssize_t retval = 0;
    struct cftype *cft = __d_cft(file->f_dentry);
    struct container *cont = __d_cont(file->f_dentry->d_parent);
    if (!cft)
        return -ENODEV;
+ if (!cft->write)
+ return -EINVAL;

- /* special function ? */
- if (cft->write)
-     retval = cft->write(cont, cft, file, buf, nbytes, ppos);
- else
-     retval = -EINVAL;
-
- return retval;
+ return cft->write(cont, cft, file, buf, nbytes, ppos);
}

-static ssize_t container_common_file_read(struct container *cont,

```

```

- struct cftype *cft,
- struct file *file,
- char __user *buf,
- size_t nbytes, loff_t *ppos)
+static ssize_t container_file_read(struct file *file, char __user *buf,
+ size_t nbytes, loff_t *ppos)
{
- container_filetype_t type = cft->private;
- char *page;
- ssize_t retval = 0;
- char *s;
-
- if (!(page = (char *)__get_free_page(GFP_KERNEL)))
- return -ENOMEM;
-
- s = page;
-
- switch (type) {
- case FILE_NOTIFY_ON_RELEASE:
- *s++ = notify_on_release(cont) ? '1' : '0';
- break;
- case FILE_RELEASE_AGENT:
- {
- size_t n;
- container_lock();
- n = strlen(release_agent_path, sizeof(release_agent_path));
- n = min(n, (size_t) PAGE_SIZE);
- strncpy(s, release_agent_path, n);
- container_unlock();
- s += n;
- break;
- }
- default:
- retval = -EINVAL;
- goto out;
- }
- *s++ = '\n';
-
- retval = simple_read_from_buffer(buf, nbytes, ppos, page, s - page);
-out:
- free_page((unsigned long)page);
- return retval;
-}
-
-static ssize_t container_file_read(struct file *file, char __user *buf, size_t nbytes,
- loff_t *ppos)
- {
- ssize_t retval = 0;

```

```

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 -EINVAL;

- /* special function ? */
- if (cft->read)
- retval = cft->read(cont, cft, file, buf, nbytes, ppos);
- else
- retval = -EINVAL;
-
- return retval;
+ return cft->read(cont, cft, file, buf, nbytes, ppos);
}

static int container_file_open(struct inode *inode, struct file *file)
@@ -739,7 +1053,7 @@ 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;

@@ -748,7 +1062,7 @@ 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;

@@ -757,7 +1071,11 @@ 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);
+
+ /* 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;
@@ -777,20 +1095,19 @@ static int container_create_file(struct
 * mode: mode to set on new directory.
 */

```

```

-static int container_create_dir(struct container *cont, const char *name, int mode)
+static int container_create_dir(struct container *cont, struct dentry *dentry,
+ int mode)
{
- struct dentry *dentry = NULL;
  struct dentry *parent;
  int error = 0;

  parent = cont->parent->dentry;
- 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);

```

```

@@ -804,19 +1121,42 @@ int container_add_file(struct container
 struct dentry *dentry;
 int error;

```

```

- mutex_lock(&dir->d_inode->i_mutex);
+ 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);
+ 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);
- mutex_unlock(&dir->d_inode->i_mutex);
  return error;
}

```

```

+/* Count the number of tasks in a container. Could be made more
+ * time-efficient but less space-efficient with more linked lists
+ * running through each container and the container_group structures

```

```

+ * that referenced it. */
+
+int container_task_count(const struct container *cont) {
+ int count = 0;
+ struct list_head *l;
+ struct container_subsys_state *css;
+ int subsys_id;
+ get_first_subsys(cont, &css, &subsys_id);
+ spin_lock_irq(&container_group_lock);
+ l = &init_container_group.list;
+ do {
+ struct container_group *cg =
+ list_entry(l, struct container_group, list);
+ if (cg->subsys[subsys_id] == css)
+ count += atomic_read(&cg->ref.refcount);
+ l = l->next;
+ } while (l != &init_container_group.list);
+ spin_unlock_irq(&container_group_lock);
+ return count;
+}
+
+/*
+ * Stuff for reading the 'tasks' file.
+ *
@@ -840,20 +1180,24 @@ struct ctr_struct {
};

/*
- * Load into 'pidarray' up to 'npids' of the tasks using container 'cont'.
- * Return actual number of pids loaded. No need to task_lock(p)
- * when reading out p->container, as we don't really care if it changes
- * on the next cycle, and we are not going to try to dereference it.
+ * Load into 'pidarray' up to 'npids' of the tasks using container
+ * 'cont'. Return actual number of pids loaded. No need to
+ * task_lock(p) when reading out p->container, since we're in an RCU
+ * read section, so the container_group can't go away, and is
+ * immutable after creation.
+ */
static int pid_array_load(pid_t *pidarray, int npids, struct container *cont)
{
int n = 0;
struct task_struct *g, *p;
-
+ struct container_subsys_state *css;
+ int subsys_id;
+ get_first_subsys(cont, &css, &subsys_id);
+ rcu_read_lock();
read_lock(&tasklist_lock);

```

```

do_each_thread(g, p) {
- if (p->container == cont) {
+ if (p->containers->subsys[subsys_id] == css) {
    pidarray[n++] = pid_nr(task_pid(p));
    if (unlikely(n == npids))
        goto array_full;
@@ -862,6 +1206,7 @@ static int pid_array_load(pid_t *pidarra

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

@@ -912,7 +1257,7 @@ static int container_tasks_open(struct i
    * caller from the case that the additional container users didn't
    * show up until sometime later on.
    */
- npids = atomic_read(&cont->count);
+ npids = container_task_count(cont);
    pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
    if (!pidarray)
        goto err1;
@@ -979,38 +1324,33 @@ static struct cftype cft_tasks = {
    .private = FILE_TASKLIST,
};

-static struct cftype cft_notify_on_release = {
- .name = "notify_on_release",
- .read = container_common_file_read,
- .write = container_common_file_write,
- .private = FILE_NOTIFY_ON_RELEASE,
-};
-
-static struct cftype cft_release_agent = {
- .name = "release_agent",
- .read = container_common_file_read,
- .write = container_common_file_write,
- .private = FILE_RELEASE_AGENT,
-};
-
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);

- 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) &&
- (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->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.
@@ -1020,57 +1360,77 @@ static int container_populate_dir(struct
+ * Must be called with the mutex on the parent inode held
+ */

-static long container_create(struct container *parent, const char *name, int mode)
+static long container_create(struct container *parent, struct dentry *dentry,
+ 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 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;
- if (notify_on_release(parent))
- set_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
- atomic_set(&cont->count, 0);
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);

  cont->parent = parent;
+ cont->root = parent->root;
+ cont->top_container = parent->top_container;

-#ifdef CONFIG_CPUSETS
- err = cpuset_create(cont);
- if (err)
- goto err_unlock_free;
-#endif
+ 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);
- number_of_containers++;
+ root->number_of_containers++;

- err = container_create_dir(cont, name, mode);
+ err = container_create_dir(cont, dentry, mode);
  if (err < 0)
    goto err_remove;

- /*
- * Release container_mutex before container_populate_dir() because it
- * will down() this new directory's i_mutex and if we race with

```

```

- * another mkdir, we might deadlock.
- */
- mutex_unlock(&container_mutex);
+ /* 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:
-#ifdef CONFIG_CPUSETS
- cpuset_destroy(cont);
-#endif
+
    list_del(&cont->sibling);
- number_of_containers--;
- err_unlock_free:
+ 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;
}
@@ -1080,7 +1440,7 @@ static int container_mkdir(struct inode
    struct container *c_parent = dentry->d_parent->d_fsdata;

    /* the vfs holds inode->i_mutex already */
- return container_create(c_parent, dentry->d_name.name, mode | S_IFDIR);
+ return container_create(c_parent, dentry, mode | S_IFDIR);
}

static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)

```

```

@@ -1088,12 +1448,15 @@ static int container_rmdir(struct inode
    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;
+ int css_busy = 0;

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

    mutex_lock(&container_mutex);
- if (atomic_read(&cont->count) > 0) {
+ if (container_task_count(cont) > 0) {
    mutex_unlock(&container_mutex);
    return -EBUSY;
}
@@ -1101,70 +1464,281 @@ static int container_rmdir(struct inode
    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);
- list_del(&cont->sibling); /* delete my sibling from parent->children */
+ /* 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);
- number_of_containers--;
-#ifdef CONFIG_CPUSETS
- cpuset_destroy(cont);
-#endif
+ root->number_of_containers--;

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

-/*
- * container_init_early - probably not needed yet, but will be needed
- * once cpusets are hooked into this code
+static atomic_t namecnt;
+static void get_unused_name(char *buf) {
+ sprintf(buf, "node%d", atomic_inc_return(&namecnt));
+}
+
+/**
+ * container_clone - duplicate the current container in the hierarchy
+ * that the given subsystem is attached to, and move this task into
+ * the new child
+ */
+int container_clone(struct task_struct *tsk, struct container_subsys *subsys)
+{
+ struct dentry *dentry;
+ int ret = 0;
+ char nodename[32];
+ struct container *parent, *child;

```

```

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

```

```

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

```

```

+int container_is_descendant(const struct container *cont) {
+ int ret;
+ struct container *target;
+ int subsys_id;
+ get_first_subsys(cont, NULL, &subsys_id);
+ target = current->containers->subsys[subsys_id]->container;
+ while (cont != target && cont!= cont->top_container) {
+ cont = cont->parent;
+ }
+ ret = (cont == target);
+ return ret;
+}
+
+static void container_init_subsys(struct container_subsys *ss) {
+ int retval;
+ struct list_head *l;
+ 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);
+ init_container_css(ss, dummytop);
+
+ /* Update all container groups to contain a subsys
+  * pointer to this state - since the subsystem is
+  * newly registered, all tasks and hence all container
+  * groups are in the subsystem's top container. */
+ spin_lock_irq(&container_group_lock);
+ l = &init_container_group.list;
+ do {
+ struct container_group *cg =
+ list_entry(l, struct container_group, list);
+ cg->subsys[ss->subsys_id] = dummytop->subsys[ss->subsys_id];
+ l = l->next;
+ } while (l != &init_container_group.list);
+ spin_unlock_irq(&container_group_lock);
+
+ need_forkexit_callback |= ss->fork || ss->exit;
+
+ /* If this subsystem requested that it be notified with fork
+  * events, we should send it one now for every process in the
+  * system */
+ if (ss->fork) {
+ struct task_struct *g, *p;
+
+ read_lock(&tasklist_lock);
+ do_each_thread(g, p) {

```

```

+ ss->fork(ss, p);
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+ }
+ 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)
{
- struct task_struct *tsk = current;
+ int i;
+ kref_init(&init_container_group.ref);
+ get_container_group(&init_container_group);
+ INIT_LIST_HEAD(&init_container_group.list);
+ container_group_count = 1;
+ init_container_root(&rootnode);
+ init_task.containers = &init_container_group;
+
+ 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();
+ }

- tsk->container = &top_container;
+ if (ss->early_init)
+ container_init_subsys(ss);
+ }
  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, and
+ * initialize any subsystems that didn't request early init.
**/
```

```
int __init container_init(void)
{
- struct dentry *root;
  int err;
+ int i;
+
+ struct proc_dir_entry *entry;

- init_task.container = &top_container;
+ 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;
- 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;
}
@@ -1175,13 +1749,12 @@ out:
*
* Description: A task inherits its parent's container at fork().
*
```

- \* A pointer to the shared container was automatically copied in fork.c  
- \* by dup\_task\_struct(). However, we ignore that copy, since it was  
- \* not made under the protection of task\_lock(), so might no longer be  
- \* a valid container pointer. attach\_task() might have already changed  
- \* current->container, allowing the previously referenced container to  
- \* be removed and freed. Instead, we task\_lock(current) and copy  
- \* its present value of current->container for our freshly forked child.  
+ \* A pointer to the shared container\_group was automatically copied in  
+ \* fork.c by dup\_task\_struct(). However, we ignore that copy, since  
+ \* it was not made under the protection of RCU or callback mutex, so  
+ \* might no longer be a valid container pointer. attach\_task() might  
+ \* have already changed current->container, allowing the previously  
+ \* referenced container to be removed and freed.

\*  
\* At the point that container\_fork() is called, 'current' is the parent  
\* task, and the passed argument 'child' points to the child task.

@@ -1189,10 +1762,32 @@ out:

```
void container_fork(struct task_struct *child)
{
- task_lock(current);
- child->container = current->container;
- atomic_inc(&child->container->count);
- task_unlock(current);
+ struct container_group *cg;
+
+ rcu_read_lock();
+ cg = rcu_dereference(current->containers);
+ get_container_group(cg);
+ child->containers = cg;
+ rcu_read_unlock();
+}
+
+/**
+ * container_fork_callbacks - called on a new task very soon before
+ * adding it to the tasklist. No need to take any locks since no-one
+ * can be operating on this task
+ **/
+
+void container_fork_callbacks(struct task_struct *child)
+{
+ if (need_forkexit_callback) {
+ int i;
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ struct container_subsys *ss = subsys[i];
+ if (ss->fork) {
+ ss->fork(ss, child);
+ }
+ }
+ }
```

```

+ }
+ }
}

/**
@@ -1247,49 +1842,34 @@ void container_fork(struct task_struct *
 * cost (other than this way too long comment ;).
**/

-void container_exit(struct task_struct *tsk)
+void container_exit(struct task_struct *tsk, int run_callbacks)
{
- struct container *cont;
+ int i;
+ struct container_group *cg = NULL;

+ if (run_callbacks && need_forkexit_callback) {
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ struct container_subsys *ss = subsys[i];
+ if (ss->exit) {
+ ss->exit(ss, tsk);
+ }
+ }
+ }
+ /* Reassign the task to the init_container_group. */
task_lock(tsk);
- cont = tsk->container;
- tsk->container = &top_container; /* the_top_container_hack - see above */
- task_unlock(tsk);
-
- if (notify_on_release(cont)) {
- char *pathbuf = NULL;
-
- mutex_lock(&container_mutex);
- if (atomic_dec_and_test(&cont->count))
- check_for_release(cont, &pathbuf);
- mutex_unlock(&container_mutex);
- container_release_agent(pathbuf);
- } else {
- atomic_dec(&cont->count);
+ if (tsk->containers != &init_container_group) {
+ cg = tsk->containers;
+ tsk->containers = &init_container_group;
+ }
-}
-
-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);
+ task_unlock(tsk);
+ if (cg)
+ put_container_group(cg);
}

-
/*
 * 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,
@@ -1298,12 +1878,15 @@ void container_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;
+ struct containerfs_root *root;

    retval = -ENOMEM;
    buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
@@ -1316,14 +1899,30 @@ static int proc_container_show(struct se
    if (!tsk)
        goto out_free;

- retval = -EINVAL;
+ retval = 0;

```

```

+
+ mutex_lock(&container_mutex);

- retval = container_path(tsk->container, buf, PAGE_SIZE);
- if (retval < 0)
- goto out_unlock;
- seq_puts(m, buf);
- seq_putc(m, '\n');
+ for_each_root(root) {
+ struct container_subsys *ss;
+ struct container *cont;
+ int subsys_id;
+ int count = 0;
+ /* Skip this hierarchy if it has no active subsystems */
+ if (!root->subsys_bits) continue;
+ for_each_subsys(root, ss) {
+ seq_printf(m, "%s%s", count++ ? ", " : "", ss->name);
+ }
+ seq_putc(m, ':');
+ get_first_subsys(&root->top_container, NULL, &subsys_id);
+ cont = tsk->containers->subsys[subsys_id]->container;
+ retval = container_path(cont, buf, PAGE_SIZE);
+ if (retval < 0)
+ goto out_unlock;
+ seq_puts(m, buf);
+ seq_putc(m, '\n');
+ }
+
out_unlock:
+ mutex_unlock(&container_mutex);
+ put_task_struct(tsk);
@@ -1345,3 +1944,49 @@ struct file_operations proc_container_op
+ .lseek = seq_lseek,
+ .release = single_release,
+ };
+
+ /* Display information about each subsystem and each hierarchy */
+ static int proc_containerstats_show(struct seq_file *m, void *v)
+ {
+ int i;
+ struct containerfs_root *root;
+ mutex_lock(&container_mutex);
+ seq_puts(m, "Hierarchies:\n");
+ for_each_root(root) {
+ struct container_subsys *ss;
+ int first = 1;
+ seq_printf(m, "%p: bits=%lx containers=%d (", root,
+ root->subsys_bits, root->number_of_containers);

```

```

+ for_each_subsys(root, ss) {
+ seq_printf(m, "%s%s", first ? "" : ", ", ss->name);
+ first = false;
+ }
+ seq_putc(m, ');
+ if (root->sb) {
+ seq_printf(m, " s_active=%d",
+ atomic_read(&root->sb->s_active));
+ }
+ seq_putc(m, '\n');
+ }
+ seq_puts(m, "Subsystems:\n");
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ struct container_subsys *ss = subsys[i];
+ seq_printf(m, "%d: name=%s hierarchy=%p\n",
+ i, ss->name, ss->root);
+ }
+ seq_printf(m, "Container groups: %d\n", container_group_count);
+ mutex_unlock(&container_mutex);
+ return 0;
+}
+
+static int containerstats_open(struct inode *inode, struct file *file)
+{
+ return single_open(file, proc_containerstats_show, 0);
+}
+
+struct file_operations proc_containerstats_operations = {
+ .open = containerstats_open,
+ .read = seq_read,
+ .llseek = seq_lseek,
+ .release = single_release,
+};

```

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

```
=====
--- container-2.6.20-new.orig/kernel/cpuset.c
```

```
+++ container-2.6.20-new/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 */
+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 +94,27 @@ struct cpuset {
+ struct fmeter fmeter; /* memory_pressure filter */
};

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

```

```

+static inline struct cpuset *task_cs(struct task_struct *task)
+{
+ return container_of(task_subsys_state(task, cpuset_subsys_id),
+   struct cpuset, css);
+}
+
+
+/* bits in struct cpuset flags field */
typedef enum {
  CS_CPU_EXCLUSIVE,
@@ -243,11 +271,10 @@ 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);
  }
  return ret;
@@ -355,20 +382,19 @@ 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_cpuset_mem_gen = top_cpuset.mems_generation;
  } else {
    rcu_read_lock();
- cs = rcu_dereference(tsk->container->cpuset);
- my_cpuset_mem_gen = cs->mems_generation;
+ my_cpuset_mem_gen = task_cs(current)->mems_generation;
    rcu_read_unlock();
  }

  if (my_cpuset_mem_gen != tsk->cpuset_mems_generation) {
    mutex_lock(&callback_mutex);
    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))
@@ -427,9 +453,8 @@ 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;
}

@@ -444,8 +469,8 @@ 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))
@@ -487,8 +512,8 @@ 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);
}
@@ -505,8 +530,8 @@ 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);
}
@@ -594,7 +619,7 @@ static void cpuset_migrate_mm(struct mm_
    do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);

    mutex_lock(&callback_mutex);

```

```

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

@@ -612,6 +637,8 @@ 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;
@@ -629,7 +656,7 @@ 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;
@@ -652,7 +679,7 @@ static int update_nodemask(struct cpuset
  cs->mems_generation = cpuset_mems_generation++;
  mutex_unlock(&callback_mutex);

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

  fudge = 10; /* spare mmarray[] slots */
  fudge += cpus_weight(cs->cpus_allowed); /* imagine one fork-bomb/cpu */
@@ -666,13 +693,13 @@ static int update_nodemask(struct cpuset
 * enough mmarray[] w/o using GFP_ATOMIC.
 */
while (1) {
- ntasks = atomic_read(&cs->container->count); /* guess */
+ ntasks = container_task_count(cs->css.container); /* guess */
  ntasks += fudge;
  mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
  if (!mmarray)
    goto done;
  write_lock_irq(&tasklist_lock); /* block fork */
- if (atomic_read(&cs->container->count) <= ntasks)
+ if (container_task_count(cs->css.container) <= ntasks)
    break; /* got enough */
  write_unlock_irq(&tasklist_lock); /* try again */
  kfree(mmarray);

```

```

@@ -689,7 +716,7 @@ 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)
@@ -723,12 +750,17 @@ 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.
+ */
@@ -879,9 +911,10 @@ 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;
@@ -889,14 +922,15 @@ int cpuset_can_attach_task(struct contai
    return security_task_setscheduler(tsk, 0, NULL);
}

-void cpuset_attach_task(struct container *cont, struct container *oldcont,
-    struct task_struct *tsk)
+void cpuset_attach(struct container_subsys *ss,

```

```

+ struct container *cont, struct container *oldcont,
+ struct task_struct *tsk)
{
    cpumask_t cpus;
    nodemask_t from, to;
    struct mm_struct *mm;
- struct cpuset *cs = cont->cpuset;
- struct cpuset *oldcs = oldcont->cpuset;
+ struct cpuset *cs = container_cs(cont);
+ struct cpuset *oldcs = container_cs(oldcont);

    mutex_lock(&callback_mutex);
    guarantee_online_cpus(cs, &cpus);
@@ -935,7 +969,7 @@ 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;
@@ -1045,7 +1079,7 @@ 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;
@@ -1163,7 +1197,7 @@ 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;

@@ -1198,11 +1232,19 @@ 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;
@@ -1219,8 +1261,8 @@ 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_cpusets++;
  return 0;
}
@@ -1236,9 +1278,9 @@ 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)) {
@@ -1246,8 +1288,20 @@ void cpuset_destroy(struct container *co
  BUG_ON(retval);
  }
  number_of_cpusets--;
+ kfree(cs);
  }

+struct container_subsys cpuset_subsys = {
+ .name = "cpuset",
+ .create = cpuset_create,
+ .destroy = cpuset_destroy,

```

```

+ .can_attach = cpuset_can_attach,
+ .attach = cpuset_attach,
+ .populate = cpuset_populate,
+ .subsys_id = cpuset_subsys_id,
+ .early_init = 1,
+};
+
/*
 * cpuset_init_early - just enough so that the calls to
 * cpuset_update_task_memory_state() in early init code
@@ -1256,13 +1310,11 @@ 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++;
+ top_cpuset.mems_generation = cpuset_mems_generation++;
  return 0;
}

+
/**
 * cpuset_init - initialize cpusets at system boot
 *
@@ -1272,6 +1324,7 @@ 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;

@@ -1313,8 +1366,8 @@ 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);
@@ -1412,7 +1465,7 @@ cpumask_t cpuset_cpus_allowed(struct tas

  mutex_lock(&callback_mutex);

```

```

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

@@ -1440,7 +1493,7 @@ nodemask_t cpuset_mems_allowed(struct ta

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

@@ -1561,7 +1614,7 @@ int __cpuset_zone_allowed_softwall(struc
mutex_lock(&callback_mutex);

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);
@@ -1690,7 +1743,7 @@ 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);
@@ -1698,7 +1751,7 @@ 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);
@@ -1734,11 +1787,8 @@ int cpuset_memory_pressure_enabled __rea

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);
}
```

```
@@ -1759,6 +1809,7 @@ static int proc_cpuset_show(struct seq_f
struct pid *pid;
struct task_struct *tsk;
char *buf;
+ struct container_subsys_state *css;
int retval;
```

```
retval = -ENOMEM;
@@ -1774,8 +1825,8 @@ static int proc_cpuset_show(struct seq_f
```

```
retval = -EINVAL;
container_lock();
```

```
-
- retval = container_path(tsk->container, buf, PAGE_SIZE);
+ css = task_subsys_state(tsk, cpuset_subsys_id);
+ retval = container_path(css->container, buf, PAGE_SIZE);
if (retval < 0)
goto out_unlock;
seq_puts(m, buf);
```

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

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

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

```
@@ -3,7 +3,7 @@
```

Written by Paul Menage <menage@google.com> based on Documentation/cpusets.txt

-Original copyright in 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>

```
@@ -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

=====

@@ -30,30 +33,55 @@ CONTENTS:

##### 1.1 What are containers ?

-----

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

+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 up to CONFIG\_MAX\_CONTAINER\_HIERARCHIES active hierarchies of task containers. Each hierarchy is a partition of all tasks in the system.

-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.

+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, such as cpusets (see -Documentation/cpusets.txt) 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.

+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, 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.

+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 @@ -61,33 +89,130 @@ minimal impact on the system fast paths, 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:



+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 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 a reference-counted pointer to a
- + container\_group.
- +
- + - A container\_group contains a set of reference-counted pointers to
- + containers, one in each hierarchy in the system.
- +
- + - 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 init/main.c, to initialize the root containers and initial
- + container\_group at system boot.

-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 fork and exit, to attach and detach a task from its container\_group.

-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 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:

@@ -112,6 +237,14 @@ on a system into related sets of tasks.  
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  
+container\_group pointer - if there's an already existing  
+container\_group with the desired collection of containers then that  
+group is reused, else a new container\_group is allocated. Note that  
+the current implementation uses a linear search to locate an  
+appropriate existing container\_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.

@@ -119,23 +252,30 @@ 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.

+\*\*\* notify\_on\_release is disabled in the current patch set. It may be reactivated in a future patch in a less-intrusive manner

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

## 1.5 How do I use containers ?

-----

-To start a new job that is to be contained within a container, 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 +287,7 @@ 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 +297,6 @@ 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 +306,25 @@ 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 will all available subsystems, type:  
+# mount -t container xxx /dev/container  
+  
+The "xxx" is not interpreted by the container code, but will appear in /proc/mounts so may be any useful identifying string that you like.  
+  
+To mount a container hierarchy with just the cpuset and numtasks subsystems, type:  
+# mount -t container -o cpuset,numtasks hier1 /dev/container  
+  
+To change the set of subsystems bound to a mounted hierarchy, just remount with different options:  
+  
+# mount -o remount,cpuset,ns /dev/container  
+  
+Note that changing the set of subsystems is currently only supported when the hierarchy consists of a single (root) container. Supporting the ability to arbitrarily bind/unbind subsystems from an existing container hierarchy is intended to be implemented in the future.

Then under /dev/container you can find a tree that corresponds to the tree of the containers in the system. For instance, /dev/container @@ -187,7 +339,8 @@ 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
```

@@ -198,8 +351,10 @@ directory.

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).

+

+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

-----

@@ -214,8 +369,160 @@ 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 `container_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 `container_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 `container_mutex`.  
+  
+In general, the pattern of use is:  
+  
+1) take `container_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 `container_mutex`  
+  
+See `kernel/container.c` for more details.  
+  
+Subsystems can take/release the `container_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 `container_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=container\_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=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 & 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=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). 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.

+

+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).

-3. Questions

+4. Questions

=====

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

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

=====

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

+++ container-2.6.20-new/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-new/include/linux/sched.h
```

```
=====
```

```
--- container-2.6.20-new.orig/include/linux/sched.h
```

```
+++ container-2.6.20-new/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_group *containers;
```

```
#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-new/mm/mempolicy.c
```

```
=====
```

```
--- container-2.6.20-new.orig/mm/mempolicy.c
```

```
+++ container-2.6.20-new/mm/mempolicy.c
```

```
@@ -1313,7 +1313,6 @@ 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)
@@ -1912,4 +1911,3 @@ out:
    m->version = (vma != priv->tail_vma) ? vma->vm_start : 0;
    return 0;
}
-

```

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

```

=====
--- container-2.6.20-new.orig/Documentation/cpusets.txt
+++ container-2.6.20-new/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

-----

Index: container-2.6.20-new/kernel/fork.c

=====

--- container-2.6.20-new.orig/kernel/fork.c

+++ container-2.6.20-new/kernel/fork.c

@@ -958,6 +958,7 @@ static struct task\_struct \*copy\_process(  
{

{

int retval;

struct task\_struct \*p = NULL;

+ int container\_callbacks\_done = 0;

if ((clone\_flags & (CLONE\_NEWNS|CLONE\_FS)) == (CLONE\_NEWNS|CLONE\_FS))

return ERR\_PTR(-EINVAL);

@@ -1174,6 +1175,12 @@ static struct task\_struct \*copy\_process(  
/\* Perform scheduler related setup. Assign this task to a CPU. \*/

/\* Perform scheduler related setup. Assign this task to a CPU. \*/

sched\_fork(p, clone\_flags);

+ /\* Now that the task is set up, run container callbacks if

+ \* necessary. We need to run them before the task is visible

+ \* on the tasklist. \*/

+ container\_fork\_callbacks(p);

+ container\_callbacks\_done = 1;

+

/\* Need tasklist lock for parent etc handling! \*/

write\_lock\_irq(&tasklist\_lock);

@@ -1290,7 +1297,7 @@ bad\_fork\_cleanup\_policy:

