Subject: Re: [PATCH 01/10] Containers(V10): Basic container framework
Posted by Andrew Morton on Wed, 30 May 2007 07:15:52 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2007 06:01:05 -0700 menage@google.com wrote:

> This patch adds the main containers framework - the container
> filesystem, and the basic structures for tracking membership and
> associating subsystem state objects to tasks.
>
> ...
>
> --- /dev/null
> +++ container-2.6.22-rc2-mm1/include/linux/container_subsys.h
> @@ -0,0 +1,10 @@
> +/* Add subsystem definitions of the form SUBSYS(<name>) in this
> + * file. Surround each one by a line of comment markers so that
> + * patches don't collide
> + */
> +
> +/* */
> +
> +/* */
> +
> +/* */
> Index: container-2.6.22-rc2-mm1/include/linux/sched.h
> ===
> --- container-2.6.22-rc2-mm1.orig/include/linux/sched.h
> +++ container-2.6.22-rc2-mm1/include/linux/sched.h
> @@ -851,6 +851,34 @@ struct sched_class {
> void (*task_new) (struct rq *rq, struct task_struct *p);
> };
>
> +#ifdef CONFIG_CONTAINERS
> +
> +#define SUBSYS(_x) _x ## _subsys_id,
> +enum container_subsys_id {
> +#include <linux/container_subsys.h>
> + CONTAINER_SUBSYS_COUNT
> +};
> +#undef SUBSYS
> +
> +/* A css_group is a structure holding pointers to a set of
> + * container_subsys_state objects.
> + */
> +
> +struct css_group {
> +

> + /* Set of subsystem states, one for each subsystem. NULL for
> + * subsystems that aren't part of this hierarchy. These
> + * pointers reduce the number of dereferences required to get
> + * from a task to its state for a given container, but result
> + * in increased space usage if tasks are in wildly different
> + * groupings across different hierarchies. This array is
> + * immutable after creation */
> + struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
> +
> +};

hm, missing forward declaration of struct container_subsys_state, but it all
seems to work out (via nested include) once the patches are applied and the
config option is enableable.

>
> ...
>
> --- /dev/null
> +++ container-2.6.22-rc2-mm1/kernel/container.c
>
> ...
>
> +#include <linux/cpu.h>
> +#include <linux/cpumask.h>
> +#include <linux/container.h>
> +#include <linux/err.h>
> +#include <linux/errno.h>
> +#include <linux/file.h>
> +#include <linux/fs.h>
> +#include <linux/init.h>
> +#include <linux/interrupt.h>
> +#include <linux/kernel.h>
> +#include <linux/kmod.h>
> +#include <linux/list.h>
> +#include <linux/mempolicy.h>
> +#include <linux/mm.h>
> +#include <linux/module.h>
> +#include <linux/mount.h>
> +#include <linux/namei.h>
> +#include <linux/pagemap.h>
> +#include <linux/proc_fs.h>
> +#include <linux/rcupdate.h>
> +#include <linux/sched.h>
> +#include <linux/seq_file.h>
> +#include <linux/security.h>
> +#include <linux/slab.h>
> +#include <linux/smp_lock.h>

```
> +#include <linux/spinlock.h>
> +#include <linux/stat.h>
> +#include <linux/string.h>
> +#include <linux/time.h>
> +#include <linux/backing-dev.h>
> +#include <linux/sort.h>
> +
> +#include <asm/uaccess.h>
> +#include <asm/atomic.h>
> +#include <linux/mutex.h>
```

Holy cow, do we need all those?

```
> +typedef enum {
> + CONT_REMOVED,
> +} container_flagbits_t;
```

typedefs are verboten. Fortunately this one is never referred to - only the values are used, so we can delete it.

```
>
> ...
>
> +static void container_clear_directory(struct dentry *dentry)
> +{
> + struct list_head *node;
> + BUG_ON(!mutex_is_locked(&dentry->d_inode->i_mutex));
> + spin_lock(&dcache_lock);
> + node = dentry->d_subdirs.next;
> + while (node != &dentry->d_subdirs) {
> +  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
> +  list_del_init(node);
> +  if (d->d_inode) {
> +   /* This should never be called on a container
> +    * directory with child containers */
> +   BUG_ON(d->d_inode->i_mode & S_IFDIR);
> +   d = dget_locked(d);
> +   spin_unlock(&dcache_lock);
> +   d_delete(d);
> +   simple_unlink(dentry->d_inode, d);
> +   dput(d);
> +   spin_lock(&dcache_lock);
> +  }
> +  node = dentry->d_subdirs.next;
> + }
> + spin_unlock(&dcache_lock);
> +}
> +
```

```
> +/*
> + * NOTE : the dentry must have been dget()'ed
> + */
> +static void container_d_remove_dir(struct dentry *dentry)
> +{
> + container_clear_directory(dentry);
> +
> + spin_lock(&dcache_lock);
> + list_del_init(&dentry->d_u.d_child);
> + spin_unlock(&dcache_lock);
> + remove_dir(dentry);
> +}
```

Taking dcache_lock in here is unfortunate. A filesystem really shouldn't be playing with that lock.

But about 20 filesystems do so. Ho hum.

```
> +static int rebind_subsystems(struct containerfs_root *root,
> +          unsigned long final_bits)
```

The code's a bit short on comments.

```
> +{
> + unsigned long added_bits, removed_bits;
> + struct container *cont = &root->top_container;
> + int i;
> +
> + removed_bits = root->subsys_bits & ~final_bits;
> + added_bits = final_bits & ~root->subsys_bits;
> + /* Check that any added subsystems are currently free */
> + for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
> +  unsigned long long bit = 1ull << i;
> +  struct container_subsys *ss = subsys[i];
> +  if (!(bit & added_bits))
> +   continue;
> +  if (ss->root != &rootnode) {
> +   /* Subsystem isn't free */
> +   return -EBUSY;
> +  }
> + }
> +
> + /* Currently we don't handle adding/removing subsystems when
> +  * any subcontainers exist. This is theoretically supportable
> +  * but involves complex erro r handling, so it's being left until
> +  * later */
> + if (!list_empty(&cont->children)) {
> +  return -EBUSY;
```

```
> + }
> +
> + /* Process each subsystem */
> + for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
> +  struct container_subsys *ss = subsys[i];
> +  unsigned long bit = 1UL << i;
> +  if (bit & added_bits) {
> +   /* We're binding this subsystem to this hierarchy */
> +   BUG_ON(cont->subsys[i]);
> +   BUG_ON(!dummytop->subsys[i]);
> +   BUG_ON(dummytop->subsys[i]->container != dummytop);
> +   cont->subsys[i] = dummytop->subsys[i];
> +   cont->subsys[i]->container = cont;
> +   list_add(&ss->sibling, &root->subsys_list);
> +   rcu_assign_pointer(ss->root, root);
> +   if (ss->bind)
> +    ss->bind(ss, cont);
> +
> +  } else if (bit & removed_bits) {
> +   /* We're removing this subsystem */
> +   BUG_ON(cont->subsys[i] != dummytop->subsys[i]);
> +   BUG_ON(cont->subsys[i]->container != cont);
> +   if (ss->bind)
> +    ss->bind(ss, dummytop);
> +   dummytop->subsys[i]->container = dummytop;
> +   cont->subsys[i] = NULL;
> +   rcu_assign_pointer(subsys[i]->root, &rootnode);
> +   list_del(&ss->sibling);
> +  } else if (bit & final_bits) {
> +   /* Subsystem state should already exist */
> +   BUG_ON(!cont->subsys[i]);
> +  } else {
> +   /* Subsystem state shouldn't exist */
> +   BUG_ON(cont->subsys[i]);
> +  }
> + }
> + root->subsys_bits = final_bits;
> + synchronize_rcu();
> +
> + return 0;
> +}
>
> ...
>
> +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);

So container_mutex nests inside i_mutex. That mean that we'll get lockdep moaning if anyone does a __GFP_FS allocation inside container_mutex (some filesystems can take i_mutex on the ->writepage path, iirc).

Probably a false positive, we can cross that bridege if/when we come to it.

> + /* 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 int container_fill_super(struct super_block *sb, void *options,
> +   int unused_silent)
> +{
> + struct inode *inode;
> + struct dentry *root;
> + struct containerfs_root *hroot = options;
> +
> + sb->s_blocksize = PAGE_CACHE_SIZE;
> + sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
> + sb->s_magic = CONTAINER_SUPER_MAGIC;
> + sb->s_op = &container_ops;
> +
> + inode = container_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR, sb);
> + if (!inode)
> + return -ENOMEM;

```
> +
> + inode->i_op = &simple_dir_inode_operations;
> + inode->i_fop = &simple_dir_operations;
> + inode->i_op = &container_dir_inode_operations;
> + /* directories start off with i_nlink == 2 (for "." entry) */
> + inc_nlink(inode);
> +
> + root = d_alloc_root(inode);
> + if (!root) {
> +  iput(inode);
> +  return -ENOMEM;
```

I bet that iput() hasn't been tested ;)

People have hit unpleasant problems before now running iput() against
partially-constructed inodes.

```
> + }
> + 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 int container_get_sb(struct file_system_type *fs_type,
> +    int flags, const char *unused_dev_name,
> +    void *data, struct vfsmount *mnt)
> +{
> + unsigned long subsys_bits = 0;
> + int ret = 0;
> + struct containerfs_root *root = NULL;
> + int use_existing = 0;
> +
> + mutex_lock(&container_mutex);
> +
> + /* First find the desired set of resource controllers */
> + ret = parse_containerfs_options(data, &subsys_bits);
> + if (ret)
> +  goto out_unlock;
> +
> + /* See if we already have a hierarchy containing this set */
```

```
> +
> + for_each_root(root) {
> +  /* We match - use this hieracrchy */
> +  if (root->subsys_bits == subsys_bits) {
> +   use_existing = 1;
> +   break;
> +  }
> +  /* We clash - fail */
> +  if (root->subsys_bits & subsys_bits) {
> +   ret = -EBUSY;
> +   goto out_unlock;
> +  }
> + }
> +
> + if (!use_existing) {
> +  /* We need a new root */
> +  root = kzalloc(sizeof(*root), GFP_KERNEL);
> +  if (!root) {
> +   ret = -ENOMEM;
> +   goto out_unlock;
> +  }
> +  init_container_root(root);
> + }
> +
> + if (!root->sb) {
> +  /* We need a new superblock for this container combination */
> +  struct container *cont = &root->top_container;
> +
> +  BUG_ON(root->subsys_bits);
> +  ret = get_sb_nodev(fs_type, flags, root,
> +       container_fill_super, mnt);
> +  if (ret)
> +   goto out_unlock;
```

Did we just leak *root?

```
> +  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 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;
> +}

This ends up having several callers and its too large to inline.

> +/* The various types of files and directories in a container file system */
> +
> +typedef enum {
> + FILE_ROOT,
> + FILE_DIR,
> + FILE_TASKLIST,
> +} container_filetype_t;

Another typedef!

> +static struct file_operations container_file_operations = {
> + .read = container_file_read,
> + .write = container_file_write,
> + .llseek = generic_file_llseek,

> + .open = container_file_open,
> + .release = container_file_release,
> +};

Do we actually want to support lseek on these things?

If not we can leave this null and use nonseekable_open() in ->open.

> +static int container_create_file(struct dentry *dentry, int mode, struct super_block *sb)
> +{
> + struct inode *inode;
> +
> + if (!dentry)
> + return -ENOENT;
> + if (dentry->d_inode)
> + return -EEXIST;
> +
> + inode = container_new_inode(mode, sb);
> + if (!inode)
> + return -ENOMEM;
> +
> + if (S_ISDIR(mode)) {
> + inode->i_op = &container_dir_inode_operations;
> + inode->i_fop = &simple_dir_operations;
> +
> + /* start off with i_nlink == 2 (for "." entry) */
> + inc_nlink(inode);
> +
> + /* start with the directory inode held, so that we can
> +   * populate it without racing with another mkdir */
> + mutex_lock(&inode->i_mutex);
> + } else if (S_ISREG(mode)) {
> + inode->i_size = 0;
> + inode->i_fop = &container_file_operations;
> + }

The S_ISREG files have no ->i_ops?

> + d_instantiate(dentry, inode);
> + dget(dentry); /* Extra count - pin the dentry in core */
> + return 0;
> +}
> +
> +/*
> + * container_create_dir - create a directory for an object.
> + * cont: the container we create the directory for.
> + * It must have a valid ->parent field
> + * And we are going to fill its ->dentry field.

> + * name: The name to give to the container directory. Will be copied.
> + * mode: mode to set on new directory.
> + */

This comment is almost kernel-doc-like, but not quite.

There doesn't seem much point in converting it, as it'd be pretty much the
only kernel-doc think in there.

> +static int container_create_dir(struct container *cont, struct dentry *dentry,
> +   int mode)
> +{
> + struct dentry *parent;
> + int error = 0;
> +
> + parent = cont->parent->dentry;
> + if (IS_ERR(dentry))
> + return PTR_ERR(dentry);
> + error = container_create_file(dentry, S_IFDIR | mode, cont->root->sb);
> + if (!error) {
> + dentry->d_fsdata = cont;
> + inc_nlink(parent->d_inode);
> + cont->dentry = dentry;
> + }
> + dput(dentry);
> +
> + return error;
> +}
>
> ...
>
> +/*
> + * container_create - create a container
> + * parent: container that will be parent of the new container.
> + * name: name of the new container. Will be strcpy'ed.
> + * mode: mode to set on new inode
> + *
> + * Must be called with the mutex on the parent inode held
> + */

OK, there's another.