Subject: [PATCH 0/7] Containers (V8): Generic Process Containers
Posted by Paul Menage on Fri, 06 Apr 2007 23:32:21 GMT
View Forum Message <> Reply to Message

--

This is an update to my multi-hierarchy multi-subsystem generic
process containers patch. Changes since V7 (12th Feb) include:

- Removed the config-time choice of the number of supported
hierarchies - this is now completely dynamic; new hierarchies are
allocated on demand, and freed when no longer in use.

- Subsystems are now registered at compile-time in
linux/container_subsys.h. This allows for faster access to subsystem
state since the id is a compile-time constant, so there's only a
single extra pointer dereference compared to having a pointer directly
in the task_struct. It also avoids wasting space with unused subsystem
pointers.

- Removed the container pointers from container_group - this results
in a structure very similar to Srivatsa Vaddagiri's rcfs
approach. (RCFS uses the nsproxy object rather than the
container_group object; merging container_group and nsproxy would be
pretty straightforward if desired).

- Removed callback_mutex from container subsystem to be purely back in
the cpuset subsystem. Renamed manage_mutex to container_mutex.

- Condensed post_attach_task() into attach_task() now that
callback_mutex is purely within cpuset.c

- Simplified the container_subsys_state reference counting - stricter
rules on liveness make adding reference counts cheaper.

Still TODO:

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

- decide whether merging container_group and nsproxy is desirable

- add a hash-table based lookup for container_group objects.

- use seq_file properly in container tasks files (and also in
cpuset_attach_task) to avoid having to allocate a big array for all
the container's task pointers.

- add back support for the "release agent" functionality

- lots more testing

- define standards for container file names

Generic Process Containers
---

There have recently been various proposals floating around for
resource management/accounting and other task grouping subsystems in
the kernel, including ResGroups, User BeanCounters, NSProxy
containers, and others.  These all need the basic abstraction of being
able to group together multiple processes in an aggregate, in order to
track/limit the resources permitted to those processes, or control
other behaviour of the processes, and all implement this grouping in
different ways.

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

This patchset extracts the process grouping code from cpusets into a
generic container system, and makes the cpusets code a client of
the container system.

It also provides several example clients of the container system,
including ResGroups, BeanCounters and namespace proxy.

The change is implemented in three implementation patches, plus four example
subsystems that aren't necessarily intended to be merged as part of
this patch set, but demonstrate the applicability of the framework.

1) extract the process grouping code from cpusets into a standalone system

2) remove the process grouping code from cpusets and hook into the
  container system

3) convert the container system to present a generic multi-hierarchy
  API, and make cpusets a client of that API

4) example of a simple CPU accounting container subsystem. Useful as a
  boilerplate for people implementing their own subsystems.

5) example of implementing ResGroups and its numtasks controller over
  generic containers

6) example of implementing BeanCounters and its numfiles counter over
 generic containers

7) example of integrating the namespace isolation code (sys_unshare()
 or various clone flags) with generic containers, allowing virtual
 servers to take advantage of other resource control efforts.

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

- the userspace APIs are (somewhat) normalised

- it's easier to test out e.g. the ResGroups CPU controller in
 conjunction with the BeanCounters memory controller, or use either of
them as the resource-control portion of a virtual server system.

- the additional kernel footprint of any of the competing resource
 management systems is substantially reduced, since it doesn't need
 to provide process grouping/containment, hence improving their
 chances of getting into the kernel

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

---

## Subject: [PATCH 1/7] Containers (V8): Generic container system abstracted from cpusets code

Posted by Paul Menage on Fri, 06 Apr 2007 23:32:22 GMT

View Forum Message <> Reply to Message

This patch creates a generic process container system based on (and
parallel top) the cpusets code.  At a coarse level it was created by
copying kernel/cpuset.c, doing s/cpuset/container/g, and stripping out any
code that was cpuset-specific rather than applicable to any process
container subsystem.

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

```
---
 Documentation/containers.txt |  229 +++++++
 fs/proc/base.c               |    7
 include/linux/container.h    |   96 +++
 include/linux/sched.h        |    5
 init/Kconfig                 |    9
 init/main.c                  |    3
 kernel/Makefile              |    1
 kernel/container.c           | 1260 ++++++++++++++++++++++++++++++++++++++++++++++
 kernel/exit.c                |    2
```

```
 kernel/fork.c               |    3
 10 files changed, 1614 insertions(+), 1 deletion(-)

Index: container-2.6.20-new/fs/proc/base.c
 ===================================================================
--- container-2.6.20-new.orig/fs/proc/base.c
+++ container-2.6.20-new/fs/proc/base.c
@@ -68,6 +68,7 @@
 #include <linux/security.h>
 #include <linux/ptrace.h>
 #include <linux/seccomp.h>
+#include <linux/container.h>
 #include <linux/cpuset.h>
 #include <linux/audit.h>
 #include <linux/poll.h>
@@ -1870,6 +1871,9 @@ static struct pid_entry tgid_base_stuff[
 #ifdef CONFIG_CPUSETS
   REG("cpuset",     S_IRUGO, cpuset),
 #endif
+#ifdef CONFIG_CONTAINERS
+ REG("container",  S_IRUGO, container),
+#endif
   INF("oom_score",  S_IRUGO, oom_score),
   REG("oom_adj",    S_IRUGO|S_IWUSR, oom_adjust),
 #ifdef CONFIG_AUDITSYSCALL
@@ -2151,6 +2155,9 @@ static struct pid_entry tid_base_stuff[]
 #ifdef CONFIG_CPUSETS
   REG("cpuset",    S_IRUGO, cpuset),
 #endif
+#ifdef CONFIG_CONTAINERS
+ REG("container",  S_IRUGO, container),
+#endif
   INF("oom_score", S_IRUGO, oom_score),
   REG("oom_adj",   S_IRUGO|S_IWUSR, oom_adjust),
 #ifdef CONFIG_AUDITSYSCALL
Index: container-2.6.20-new/include/linux/container.h
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/include/linux/container.h
@@ -0,0 +1,96 @@
+#ifndef _LINUX_CONTAINER_H
+#define _LINUX_CONTAINER_H
+/*
+ *  container interface
+ *
+ *  Copyright (C) 2003 BULL SA
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
```

```
+ */
+
+#include <linux/sched.h>
+#include <linux/cpumask.h>
+#include <linux/nodemask.h>
+
+#ifdef CONFIG_CONTAINERS
+
+extern int number_of_containers; /* How many containers are defined in system? */
+
+extern int container_init_early(void);
+extern int container_init(void);
+extern void container_init_smp(void);
+extern void container_fork(struct task_struct *p);
+extern void container_exit(struct task_struct *p);
+
+extern struct file_operations proc_container_operations;
+
+extern void container_lock(void);
+extern void container_unlock(void);
+
+extern void container_manage_lock(void);
+extern void container_manage_unlock(void);
+
+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'.
+  */
+ struct list_head sibling; /* my parent's children */
+ struct list_head children; /* my children */
+
+ struct container *parent; /* my parent */
+ struct dentry *dentry;  /* container fs entry */
+};
+
+/* struct cftype:
+ *
+ * The files in the container filesystem mostly have a very simple read/write
+ * handling, some common function will take care of it. Nevertheless some cases
+ * (read tasks) are special and therefore I define this structure for every
```

```
+ * kind of file.
+ *
+ *
+ * When reading/writing to a file:
+ * - the container to use in file->f_dentry->d_parent->d_fsdata
+ * - the 'cftype' of the file is file->f_dentry->d_fsdata
+ */
+
+struct inode;
+struct cftype {
+ char *name;
+ int private;
+ int (*open) (struct inode *inode, struct file *file);
+ ssize_t (*read) (struct container *cont, struct cftype *cft,
+    struct file *file,
+    char __user *buf, size_t nbytes, loff_t *ppos);
+ ssize_t (*write) (struct container *cont, struct cftype *cft,
+     struct file *file,
+     const char __user *buf, size_t nbytes, loff_t *ppos);
+ int (*release) (struct inode *inode, struct file *file);
+};
+
+int container_add_file(struct container *cont, const struct cftype *cft);
+
+int container_is_removed(const struct container *cont);
+
+#else /* !CONFIG_CONTAINERS */
+
+static inline int container_init_early(void) { return 0; }
+static inline int container_init(void) { return 0; }
+static inline void container_init_smp(void) {}
+static inline void container_fork(struct task_struct *p) {}
+static inline void container_exit(struct task_struct *p) {}
+
+static inline void container_lock(void) {}
+static inline void container_unlock(void) {}
+
+#endif /* !CONFIG_CONTAINERS */
+
+#endif /* _LINUX_CONTAINER_H */
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
@@ -743,8 +743,8 @@ extern unsigned int max_cache_size;


 struct io_context;   /* See blkdev.h */
```

```
+struct container;
 struct cpuset;
-
 #define NGROUPS_SMALL  32
 #define NGROUPS_PER_BLOCK ((int)(PAGE_SIZE / sizeof(gid_t)))
 struct group_info {
@@ -1031,6 +1031,9 @@ struct task_struct {
  int cpuset_mems_generation;
  int cpuset_mem_spread_rotor;
 #endif
+#ifdef CONFIG_CONTAINERS
+ struct container *container;
+#endif
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
  struct compat_robust_list_head __user *compat_robust_list;
Index: container-2.6.20-new/init/Kconfig
 ===================================================================
--- container-2.6.20-new.orig/init/Kconfig
+++ container-2.6.20-new/init/Kconfig
@@ -238,6 +238,15 @@ config IKCONFIG_PROC
    This option enables access to the kernel configuration file
    through /proc/config.gz.

+config CONTAINERS
+ bool "Container support"
+ help
+   This option will let you create and manage process containers,
+   which can be used to aggregate multiple processes, e.g. for
+   the purposes of resource tracking.
+
+   Say N if unsure
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
Index: container-2.6.20-new/init/main.c
 ===================================================================
--- container-2.6.20-new.orig/init/main.c
+++ container-2.6.20-new/init/main.c
@@ -39,6 +39,7 @@
 #include <linux/writeback.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/efi.h>
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
```

```
@@ -485,6 +486,7 @@ asmlinkage void __init start_kernel(void
  char * command_line;
  extern struct kernel_param __start___param[], __stop___param[];

+ container_init_early();
  smp_setup_processor_id();

  /*
@@ -608,6 +610,7 @@ asmlinkage void __init start_kernel(void
 #ifdef CONFIG_PROC_FS
  proc_root_init();
 #endif
+ container_init();
  cpuset_init();
  taskstats_init_early();
  delayacct_init();
Index: container-2.6.20-new/kernel/container.c
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/container.c
@@ -0,0 +1,1260 @@
+/*
+ *  kernel/container.c
+ *
+ *  Generic process-grouping system.
+ *
+ *  Based originally on the cpuset system, extracted by Paul Menage
+ *  Copyright (C) 2006 Google, Inc
+ *
+ *  Copyright notices from the original cpuset code:
+ *  --------------------------------------------------
+ *  Copyright (C) 2003 BULL SA.
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ *  Portions derived from Patrick Mochel's sysfs code.
+ *  sysfs is Copyright (c) 2001-3 Patrick Mochel
+ *
+ *  2003-10-10 Written by Simon Derr.
+ *  2003-10-22 Updates by Stephen Hemminger.
+ *  2004 May-July Rework by Paul Jackson.
+ *  ---------------------------------------------------
+ *
+ *  This file is subject to the terms and conditions of the GNU General Public
+ *  License.  See the file COPYING in the main directory of the Linux
+ *  distribution for more details.
+ */
+
+#include <linux/cpu.h>
```

```
+#include <linux/cpumask.h>
+#include <linux/container.h>
+#include <linux/err.h>
+#include <linux/errno.h>
+#include <linux/file.h>
+#include <linux/fs.h>
+#include <linux/init.h>
+#include <linux/interrupt.h>
+#include <linux/kernel.h>
+#include <linux/kmod.h>
+#include <linux/list.h>
+#include <linux/mempolicy.h>
+#include <linux/mm.h>
+#include <linux/module.h>
+#include <linux/mount.h>
+#include <linux/namei.h>
+#include <linux/pagemap.h>
+#include <linux/proc_fs.h>
+#include <linux/rcupdate.h>
+#include <linux/sched.h>
+#include <linux/seq_file.h>
+#include <linux/security.h>
+#include <linux/slab.h>
+#include <linux/smp_lock.h>
+#include <linux/spinlock.h>
+#include <linux/stat.h>
+#include <linux/string.h>
+#include <linux/time.h>
+#include <linux/backing-dev.h>
+#include <linux/sort.h>
+
+#include <asm/uaccess.h>
+#include <asm/atomic.h>
+#include <linux/mutex.h>
+
+#define CONTAINER_SUPER_MAGIC  0x27e0eb
+
+/*
+ * Tracks how many containers are currently defined in system.
+ * When there is only one container (the root container) we can
+ * short circuit some hooks.
+ */
+int number_of_containers __read_mostly;
+
+/* bits in struct container flags field */
+typedef enum {
+ CONT_REMOVED,
+ CONT_NOTIFY_ON_RELEASE,
```

```
+} container_flagbits_t;
+
+/* convenient tests for these bits */
+inline int container_is_removed(const struct container *cont)
+{
+ 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);
+}
+
+static struct container top_container = {
+ .count = ATOMIC_INIT(0),
+ .sibling = LIST_HEAD_INIT(top_container.sibling),
+ .children = LIST_HEAD_INIT(top_container.children),
+};
+
+static struct vfsmount *container_mount;
+static struct super_block *container_sb;
+
+/*
+ * There is one global container mutexes. We also require taking
+ * task_lock() when dereferencing a tasks container pointer.  See "The
+ * task_lock() exception", at the end of this comment.
+ *
+ * A task must hold container_mutex to modify containers.
+ *
+ * Any task can increment and decrement the count field without lock.
+ * So in general, code holding container_mutex can't rely on the count
+ * field not changing.  However, if the count goes to zero, then only
+ * attach_task() can increment it again.  Because a count of zero
+ * means that no tasks are currently attached, therefore there is no
+ * way a task attached to that container can fork (the other way to
+ * increment the count).  So code holding container_mutex can safely
+ * assume that if the count is zero, it will stay zero. Similarly, if
+ * a task holds container_mutex on a container with zero count, it
+ * knows that the container won't be removed, as container_rmdir()
+ * needs that mutex.
+ *
+ * The container_common_file_write handler for operations that modify
+ * the container hierarchy holds container_mutex across the entire operation,
+ * single threading all such container modifications across the system.
+ *
+ * The fork and exit callbacks container_fork() and container_exit(), don't
+ * (usually) take container_mutex.  These are the two most performance
+ * critical pieces of code here.  The exception occurs on container_exit(),
```

```
+ * when a task in a notify_on_release container exits.  Then container_mutex
+ * is taken, and if the container count is zero, a usermode call made
+ * to /sbin/container_release_agent with the name of the container (path
+ * relative to the root of container file system) as the argument.
+ *
+ * A container can only be deleted if both its 'count' of using tasks
+ * is zero, and its list of 'children' containers is empty.  Since all
+ * tasks in the system use _some_ container, and since there is always at
+ * least one task in the system (init, pid == 1), therefore, top_container
+ * always has either children containers and/or using tasks.  So we don't
+ * need a special hack to ensure that top_container cannot be deleted.
+ *
+ * The task_lock() exception
+ *
+ * The need for this exception arises from the action of
+ * attach_task(), which overwrites one tasks container pointer with
+ * another.  It does so using container_mutexe, however there are
+ * several performance critical places that need to reference
+ * task->container without the expense of grabbing a system global
+ * mutex.  Therefore except as noted below, when dereferencing or, as
+ * in attach_task(), modifying a task'ss container pointer we use
+ * task_lock(), which acts on a spinlock (task->alloc_lock) already in
+ * the task_struct routinely used for such matters.
+ *
+ * P.S.  One more locking exception.  RCU is used to guard the
+ * update of a tasks container pointer by attach_task()
+ */
+
+static DEFINE_MUTEX(container_mutex);
+
+/*
+ * A couple of forward declarations required, due to cyclic reference loop:
+ *  container_mkdir -> container_create -> container_populate_dir -> container_add_file
+ *  -> container_create_file -> container_dir_inode_operations -> container_mkdir.
+ */
+
+static int container_mkdir(struct inode *dir, struct dentry *dentry, int mode);
+static int container_rmdir(struct inode *unused_dir, struct dentry *dentry);
+
+static struct backing_dev_info container_backing_dev_info = {
+ .ra_pages = 0,  /* No readahead */
+ .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
+};
+
+static struct inode *container_new_inode(mode_t mode)
+{
+ struct inode *inode = new_inode(container_sb);
+
```

```
+ if (inode) {
+  inode->i_mode = mode;
+  inode->i_uid = current->fsuid;
+  inode->i_gid = current->fsgid;
+  inode->i_blocks = 0;
+  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
+  inode->i_mapping->backing_dev_info = &container_backing_dev_info;
+ }
+ return inode;
+}
+
+static void container_diput(struct dentry *dentry, struct inode *inode)
+{
+ /* is dentry a directory ? if so, kfree() associated container */
+ if (S_ISDIR(inode->i_mode)) {
+  struct container *cont = dentry->d_fsdata;
+  BUG_ON(!(container_is_removed(cont)));
+  kfree(cont);
+ }
+ iput(inode);
+}
+
+static struct dentry_operations container_dops = {
+ .d_iput = container_diput,
+};
+
+static struct dentry *container_get_dentry(struct dentry *parent, const char *name)
+{
+ struct dentry *d = lookup_one_len(name, parent, strlen(name));
+ if (!IS_ERR(d))
+  d->d_op = &container_dops;
+ return d;
+}
+
+static void remove_dir(struct dentry *d)
+{
+ struct dentry *parent = dget(d->d_parent);
+
+ d_delete(d);
+ simple_rmdir(parent->d_inode, d);
+ dput(parent);
+}
+
+/*
+ * NOTE : the dentry must have been dget()'ed
+ */
+static void container_d_remove_dir(struct dentry *dentry)
+{
```

```
+ struct list_head *node;
+
+ spin_lock(&dcache_lock);
+ node = dentry->d_subdirs.next;
+ while (node != &dentry->d_subdirs) {
+  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
+  list_del_init(node);
+  if (d->d_inode) {
+   d = dget_locked(d);
+   spin_unlock(&dcache_lock);
+   d_delete(d);
+   simple_unlink(dentry->d_inode, d);
+   dput(d);
+   spin_lock(&dcache_lock);
+  }
+  node = dentry->d_subdirs.next;
+ }
+ list_del_init(&dentry->d_u.d_child);
+ spin_unlock(&dcache_lock);
+ remove_dir(dentry);
+}
+
+static struct super_operations container_ops = {
+ .statfs = simple_statfs,
+ .drop_inode = generic_delete_inode,
+};
+
+static int container_fill_super(struct super_block *sb, void *unused_data,
+       int unused_silent)
+{
+ struct inode *inode;
+ struct dentry *root;
+
+ sb->s_blocksize = PAGE_CACHE_SIZE;
+ sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
+ sb->s_magic = 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 {
+  return -ENOMEM;
+ }
```

```
+
+ root = d_alloc_root(inode);
+ if (!root) {
+  iput(inode);
+  return -ENOMEM;
+ }
+ sb->s_root = root;
+ 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)
+{
+ return get_sb_single(fs_type, flags, data, container_fill_super, mnt);
+}
+
+static struct file_system_type container_fs_type = {
+ .name = "container",
+ .get_sb = container_get_sb,
+ .kill_sb = kill_litter_super,
+};
+
+static inline struct container *__d_cont(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+static inline struct cftype *__d_cft(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+/*
+ * Call with container_mutex held.  Writes path of container into buf.
+ * Returns 0 on success, -errno on error.
+ */
+
+static int container_path(const struct container *cont, char *buf, int buflen)
+{
+ char *start;
+
+ start = buf + buflen;
+
+ *--start = '\0';
+ for (;;) {
+  int len = cont->dentry->d_name.len;
+  if ((start -= len) < buf)
```

```
+   return -ENAMETOOLONG;
+  memcpy(start, cont->dentry->d_name.name, len);
+  cont = cont->parent;
+  if (!cont)
+   break;
+  if (!cont->parent)
+   continue;
+  if (--start < buf)
+   return -ENAMETOOLONG;
+  *start = '/';
+ }
+ memmove(buf, start, buf + buflen - start);
+ return 0;
+}
+
+/*
+ * 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++] = "/sbin/container_release_agent";
+ argv[i++] = (char *)pathbuf;
+ argv[i] = NULL;
+
+ i = 0;
+ /* minimal command environment */
+ envp[i++] = "HOME=/";
+ envp[i++] = "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
+ envp[i] = NULL;
+
+ call_usermodehelper(argv[0], argv, envp, 0);
+ kfree(pathbuf);
+}
+
+/*
+ * Either 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.
+ *
+ * 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().
+ */
+
+static void check_for_release(struct container *cont, char **ppathbuf)
+{
+ 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;
+  if (container_path(cont, buf, PAGE_SIZE) < 0)
```

```
+   kfree(buf);
+  else
+   *ppathbuf = buf;
+ }
+}
+
+
+/*
+ * 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.
+ */
+
+static int update_flag(container_flagbits_t bit, struct container *cont, char *buf)
+{
+ int turning_on;
+
+ turning_on = (simple_strtoul(buf, NULL, 10) != 0);
+
+ if (turning_on)
+  set_bit(bit, &cont->flags);
+ else
+  clear_bit(bit, &cont->flags);
+
+ return 0;
+}
+
+
+/*
+ * Attack 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.
+ *
+ * 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)
+{
+ pid_t pid;
+ struct task_struct *tsk;
+ struct container *oldcont;
+ int retval;
+
+ if (sscanf(pidbuf, "%d", &pid) != 1)
```

```
+  return -EIO;
+
+ if (pid) {
+  read_lock(&tasklist_lock);
+
+  tsk = find_task_by_pid(pid);
+  if (!tsk || tsk->flags & PF_EXITING) {
+   read_unlock(&tasklist_lock);
+   return -ESRCH;
+  }
+
+  get_task_struct(tsk);
+  read_unlock(&tasklist_lock);
+
+  if ((current->euid) && (current->euid != tsk->uid)
+      && (current->euid != tsk->suid)) {
+   put_task_struct(tsk);
+   return -EACCES;
+  }
+ } else {
+  tsk = current;
+  get_task_struct(tsk);
+ }
+
+ retval = security_task_setscheduler(tsk, 0, NULL);
+ if (retval) {
+  put_task_struct(tsk);
+  return retval;
+ }
+
+ task_lock(tsk);
+ oldcont = tsk->container;
+ if (!oldcont) {
+  task_unlock(tsk);
+  put_task_struct(tsk);
+  return -ESRCH;
+ }
+ atomic_inc(&cont->count);
+ rcu_assign_pointer(tsk->container, cont);
+ task_unlock(tsk);
+
+ put_task_struct(tsk);
+ synchronize_rcu();
+ if (atomic_dec_and_test(&oldcont->count))
+  check_for_release(oldcont, ppathbuf);
+ return 0;
+}
+
```

```
+/* The various types of files and directories in a container file system */
+
+typedef enum {
+ FILE_ROOT,
+ FILE_DIR,
+ FILE_NOTIFY_ON_RELEASE,
+ FILE_TASKLIST,
+} container_filetype_t;
+
+static ssize_t container_common_file_write(struct container *cont,
+        struct cftype *cft,
+        struct file *file,
+        const char __user *userbuf,
+        size_t nbytes, loff_t *unused_ppos)
+{
+ container_filetype_t type = cft->private;
+ char *buffer;
+ char *pathbuf = NULL;
+ int retval = 0;
+
+ /* Crude upper limit on largest legitimate cpulist user might write. */
+ if (nbytes > 100 + 6 * NR_CPUS)
+  return -E2BIG;
+
+ /* +1 for nul-terminator */
+ if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
+  return -ENOMEM;
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+  retval = -EFAULT;
+  goto out1;
+ }
+ buffer[nbytes] = 0; /* nul-terminate */
+
+ mutex_lock(&container_mutex);
+
+ if (container_is_removed(cont)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch (type) {
+ case FILE_NOTIFY_ON_RELEASE:
+  retval = update_flag(CONT_NOTIFY_ON_RELEASE, cont, buffer);
+  break;
+ case FILE_TASKLIST:
+  retval = attach_task(cont, buffer, &pathbuf);
+  break;
```

```
+ default:
+  retval = -EINVAL;
+  goto out2;
+ }
+
+ if (retval == 0)
+  retval = nbytes;
+out2:
+ mutex_unlock(&container_mutex);
+ container_release_agent(pathbuf);
+out1:
+ kfree(buffer);
+ return retval;
+}
+
+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;
+
+ /* special function ? */
+ if (cft->write)
+  retval = cft->write(cont, cft, file, buf, nbytes, ppos);
+ else
+  retval = -EINVAL;
+
+ return retval;
+}
+
+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)
+{
+ 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;
+ 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;
+
+ /* special function ? */
+ if (cft->read)
+  retval = cft->read(cont, cft, file, buf, nbytes, ppos);
+ else
+  retval = -EINVAL;
+
+ return retval;
+}
+
+static int container_file_open(struct inode *inode, struct file *file)
+{
+ int err;
+ struct cftype *cft;
+
+ err = generic_file_open(inode, file);
+ if (err)
+  return err;
+
+ cft = __d_cft(file->f_dentry);
+ if (!cft)
+  return -ENODEV;
+ if (cft->open)
```

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

```
+
+ if (!dentry)
+  return -ENOENT;
+ if (dentry->d_inode)
+  return -EEXIST;
+
+ inode = container_new_inode(mode);
+ 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) */
+  inode->i_nlink++;
+ } else if (S_ISREG(mode)) {
+  inode->i_size = 0;
+  inode->i_fop = &container_file_operations;
+ }
+
+ d_instantiate(dentry, inode);
+ dget(dentry); /* Extra count - pin the dentry in core */
+ return 0;
+}
+
+/*
+ * container_create_dir - create a directory for an object.
+ * cont: the container we create the directory for.
+ *  It must have a valid ->parent field
+ *  And we are going to fill its ->dentry field.
+ * name: The name to give to the container directory. Will be copied.
+ * mode: mode to set on new directory.
+ */
+
+static int container_create_dir(struct container *cont, const char *name, 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);
+ if (!error) {
+  dentry->d_fsdata = cont;
```

```
+  parent->d_inode->i_nlink++;
+  cont->dentry = dentry;
+ }
+ dput(dentry);
+
+ return error;
+}
+
+int container_add_file(struct container *cont, const struct cftype *cft)
+{
+ struct dentry *dir = cont->dentry;
+ struct dentry *dentry;
+ int error;
+
+ mutex_lock(&dir->d_inode->i_mutex);
+ dentry = container_get_dentry(dir, cft->name);
+ if (!IS_ERR(dentry)) {
+  error = container_create_file(dentry, 0644 | S_IFREG);
+  if (!error)
+   dentry->d_fsdata = (void *)cft;
+  dput(dentry);
+ } else
+  error = PTR_ERR(dentry);
+ mutex_unlock(&dir->d_inode->i_mutex);
+ return error;
+}
+
+/*
+ * Stuff for reading the 'tasks' file.
+ *
+ * Reading this file can return large amounts of data if a container has
+ * *lots* of attached tasks. So it may need several calls to read(),
+ * but we cannot guarantee that the information we produce is correct
+ * unless we produce it entirely atomically.
+ *
+ * Upon tasks file open(), a struct ctr_struct is allocated, that
+ * will have a pointer to an array (also allocated here).  The struct
+ * ctr_struct * is stored in file->private_data.  Its resources will
+ * be freed by release() when the file is closed.  The array is used
+ * to sprintf the PIDs and then used by read().
+ */
+
+/* containers_tasks_read array */
+
+struct ctr_struct {
+ char *buf;
+ int bufsz;
+};
```

```
+
+/*
+ * Load into 'pidarray' up to 'npids' of the tasks using container 'cont'.
+ * Return actual number of pids loaded.  No need to task_lock(p)
+ * when reading out p->container, as we don't really care if it changes
+ * on the next cycle, and we are not going to try to dereference it.
+ */
+static int pid_array_load(pid_t *pidarray, int npids, struct container *cont)
+{
+ int n = 0;
+ struct task_struct *g, *p;
+
+ read_lock(&tasklist_lock);
+
+ do_each_thread(g, p) {
+  if (p->container == cont) {
+   pidarray[n++] = pid_nr(task_pid(p));
+   if (unlikely(n == npids))
+    goto array_full;
+  }
+ } while_each_thread(g, p);
+
+array_full:
+ read_unlock(&tasklist_lock);
+ return n;
+}
+
+static int cmppid(const void *a, const void *b)
+{
+ return *(pid_t *)a - *(pid_t *)b;
+}
+
+/*
+ * Convert array 'a' of 'npids' pid_t's to a string of newline separated
+ * decimal pids in 'buf'.  Don't write more than 'sz' chars, but return
+ * count 'cnt' of how many chars would be written if buf were large enough.
+ */
+static int pid_array_to_buf(char *buf, int sz, pid_t *a, int npids)
+{
+ int cnt = 0;
+ int i;
+
+ for (i = 0; i < npids; i++)
+  cnt += snprintf(buf + cnt, max(sz - cnt, 0), "%d\n", a[i]);
+ return cnt;
+}
+
+/*
```

```
+ * Handle an open on 'tasks' file.  Prepare a buffer listing the
+ * process id's of tasks currently attached to the container being opened.
+ *
+ * Does not require any specific container mutexes, and does not take any.
+ */
+static int container_tasks_open(struct inode *unused, struct file *file)
+{
+ struct container *cont = __d_cont(file->f_dentry->d_parent);
+ struct ctr_struct *ctr;
+ pid_t *pidarray;
+ int npids;
+ char c;
+
+ if (!(file->f_mode & FMODE_READ))
+  return 0;
+
+ ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
+ if (!ctr)
+  goto err0;
+
+ /*
+  * If container gets more users after we read count, we won't have
+  * enough space - tough.  This race is indistinguishable to the
+  * caller from the case that the additional container users didn't
+  * show up until sometime later on.
+  */
+ npids = atomic_read(&cont->count);
+ pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
+ if (!pidarray)
+  goto err1;
+
+ npids = pid_array_load(pidarray, npids, cont);
+ sort(pidarray, npids, sizeof(pid_t), cmppid, NULL);
+
+ /* Call pid_array_to_buf() twice, first just to get bufsz */
+ ctr->bufsz = pid_array_to_buf(&c, sizeof(c), pidarray, npids) + 1;
+ ctr->buf = kmalloc(ctr->bufsz, GFP_KERNEL);
+ if (!ctr->buf)
+  goto err2;
+ ctr->bufsz = pid_array_to_buf(ctr->buf, ctr->bufsz, pidarray, npids);
+
+ kfree(pidarray);
+ file->private_data = ctr;
+ return 0;
+
+err2:
+ kfree(pidarray);
+err1:
```

```
+ kfree(ctr);
+err0:
+ return -ENOMEM;
+}
+
+static ssize_t container_tasks_read(struct container *cont,
+        struct cftype *cft,
+        struct file *file, char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct ctr_struct *ctr = file->private_data;
+
+ if (*ppos + nbytes > ctr->bufsz)
+  nbytes = ctr->bufsz - *ppos;
+ if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
+  return -EFAULT;
+ *ppos += nbytes;
+ return nbytes;
+}
+
+static int container_tasks_release(struct inode *unused_inode, struct file *file)
+{
+ struct ctr_struct *ctr;
+
+ if (file->f_mode & FMODE_READ) {
+  ctr = file->private_data;
+  kfree(ctr->buf);
+  kfree(ctr);
+ }
+ return 0;
+}
+
+/*
+ * for the common functions, 'private' gives the type of file
+ */
+
+static struct cftype cft_tasks = {
+ .name = "tasks",
+ .open = container_tasks_open,
+ .read = container_tasks_read,
+ .write = container_common_file_write,
+ .release = container_tasks_release,
+ .private = FILE_TASKLIST,
+};
+
+static 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 int container_populate_dir(struct container *cont)
+{
+ int err;
+
+ if ((err = container_add_file(cont, &cft_notify_on_release)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_tasks)) < 0)
+  return err;
+ return 0;
+}
+
+/*
+ * container_create - create a container
+ * parent: container that will be parent of the new container.
+ * name:  name of the new container. Will be strcpy'ed.
+ * mode:  mode to set on new inode
+ *
+ * Must be called with the mutex on the parent inode held
+ */
+
+static long container_create(struct container *parent, const char *name, int mode)
+{
+ struct container *cont;
+ int err;
+
+ cont = kmalloc(sizeof(*cont), GFP_KERNEL);
+ if (!cont)
+  return -ENOMEM;
+
+ 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;
+
+ list_add(&cont->sibling, &cont->parent->children);
+ number_of_containers++;
+
+ err = container_create_dir(cont, name, 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);
+
+ err = container_populate_dir(cont);
+ /* If err < 0, we have a half-filled directory - oh well ;) */
+ return 0;
+
+ err_remove:
+ list_del(&cont->sibling);
+ number_of_containers--;
+
+ mutex_unlock(&container_mutex);
+ kfree(cont);
+ return err;
+}
+
+static int container_mkdir(struct inode *dir, struct dentry *dentry, int mode)
+{
+ struct container *c_parent = dentry->d_parent->d_fsdata;
+
+ /* the vfs holds inode->i_mutex already */
+ return container_create(c_parent, dentry->d_name.name, mode | S_IFDIR);
+}
+
+static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)
+{
+ struct container *cont = dentry->d_fsdata;
+ struct dentry *d;
+ struct container *parent;
+ char *pathbuf = NULL;
+
+ /* the vfs holds both inode->i_mutex already */
+
+ mutex_lock(&container_mutex);
+ if (atomic_read(&cont->count) > 0) {
+  mutex_unlock(&container_mutex);
+  return -EBUSY;
+ }
+ if (!list_empty(&cont->children)) {
+  mutex_unlock(&container_mutex);
+  return -EBUSY;
+ }
```

```
+ parent = cont->parent;
+ set_bit(CONT_REMOVED, &cont->flags);
+ list_del(&cont->sibling); /* delete my sibling from parent->children */
+ 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--;
+
+ if (list_empty(&parent->children))
+  check_for_release(parent, &pathbuf);
+ mutex_unlock(&container_mutex);
+ container_release_agent(pathbuf);
+ return 0;
+}
+
+/*
+ * container_init_early - probably not needed yet, but will be needed
+ * once cpusets are hooked into this code
+ */
+
+int __init container_init_early(void)
+{
+ struct task_struct *tsk = current;
+
+ tsk->container = &top_container;
+ return 0;
+}
+
+/**
+ * container_init - initialize containers at system boot
+ *
+ * Description: Initialize top_container and the container internal file system,
+ **/
+
+int __init container_init(void)
+{
+ struct dentry *root;
+ int err;
+
+ init_task.container = &top_container;
+
+ 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);
+out:
+ return err;
+}
+
+/**
+ * container_fork - attach newly forked task to its parents container.
+ * @tsk: pointer to task_struct of forking parent process.
+ *
+ * Description: A task inherits its parent's container at fork().
+ *
+ * A pointer to the shared 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.
+ *
+ * At the point that container_fork() is called, 'current' is the parent
+ * task, and the passed argument 'child' points to the child task.
+ **/
+
+void container_fork(struct task_struct *child)
+{
+ task_lock(current);
+ child->container = current->container;
+ atomic_inc(&child->container->count);
+ task_unlock(current);
+}
+
+/**
+ * container_exit - detach container from exiting task
+ * @tsk: pointer to task_struct of exiting process
+ *
+ * Description: Detach container from @tsk and release it.
```

```
+ *
+ * Note that containers marked notify_on_release force every task in
+ * them to take the global container_mutex mutex when exiting.
+ * This could impact scaling on very large systems.  Be reluctant to
+ * use notify_on_release containers where very high task exit scaling
+ * is required on large systems.
+ *
+ * Don't even think about derefencing 'cont' after the container use
+ * count goes to zero, except inside a critical section guarded by
+ * container_mutex.  Otherwise a zero container use count is a license
+ * to any other task to nuke the container immediately, via
+ * container_rmdir().
+ *
+ * We don't need to task_lock() this reference to tsk->container,
+ * because tsk is already marked PF_EXITING, so attach_task() won't
+ * mess with it, or task is a failed fork, never visible to attach_task.
+ *
+ * the_top_container_hack:
+ *
+ *    Set the exiting tasks container to the root container (top_container).
+ *
+ *    Don't leave a task unable to allocate memory, as that is an
+ *    accident waiting to happen should someone add a callout in
+ *    do_exit() after the container_exit() call that might allocate.
+ *    If a task tries to allocate memory with an invalid container,
+ *    it will oops in container_update_task_memory_state().
+ *
+ *    We call container_exit() while the task is still competent to
+ *    handle notify_on_release(), then leave the task attached to
+ *    the root container (top_container) for the remainder of its exit.
+ *
+ *    To do this properly, we would increment the reference count on
+ *    top_container, and near the very end of the kernel/exit.c do_exit()
+ *    code we would add a second container function call, to drop that
+ *    reference.  This would just create an unnecessary hot spot on
+ *    the top_container reference count, to no avail.
+ *
+ *    Normally, holding a reference to a container without bumping its
+ *    count is unsafe.   The container could go away, or someone could
+ *    attach us to a different container, decrementing the count on
+ *    the first container that we never incremented.  But in this case,
+ *    top_container isn't going away, and either task has PF_EXITING set,
+ *    which wards off any attach_task() attempts, or task is a failed
+ *    fork, never visible to attach_task.
+ *
+ *    Another way to do this would be to set the container pointer
+ *    to NULL here, and check in container_update_task_memory_state()
+ *    for a NULL pointer.  This hack avoids that NULL check, for no
```

```
+ *    cost (other than this way too long comment ;).
+ **/
+
+void container_exit(struct task_struct *tsk)
+{
+ struct container *cont;
+
+ cont = tsk->container;
+ tsk->container = &top_container; /* the_top_container_hack - see above */
+
+ if (notify_on_release(cont)) {
+  char *pathbuf = NULL;
+
+  mutex_lock(&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);
+ }
+}
+
+/*
+ * proc_container_show()
+ *  - Print tasks container path into seq_file.
+ *  - Used for /proc/<pid>/container.
+ *  - No need to task_lock(tsk) on this tsk->container reference, as it
+ *    doesn't really matter if tsk->container changes after we read it,
+ *    and we take container_mutex, keeping attach_task() from changing it
+ *    anyway.  No need to check that tsk->container != NULL, thanks to
+ *    the_top_container_hack in container_exit(), which sets an exiting tasks
+ *    container to top_container.
+ */
+static int proc_container_show(struct seq_file *m, void *v)
+{
+ struct pid *pid;
+ struct task_struct *tsk;
+ char *buf;
+ int retval;
+
+ retval = -ENOMEM;
+ buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
+ if (!buf)
+  goto out;
+
+ retval = -ESRCH;
+ pid = m->private;
```

```
+ tsk = get_pid_task(pid, PIDTYPE_PID);
+ if (!tsk)
+  goto out_free;
+
+ retval = -EINVAL;
+ 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');
+out_unlock:
+ mutex_unlock(&container_mutex);
+ put_task_struct(tsk);
+out_free:
+ kfree(buf);
+out:
+ return retval;
+}
+
+static int container_open(struct inode *inode, struct file *file)
+{
+ struct pid *pid = PROC_I(inode)->pid;
+ return single_open(file, proc_container_show, pid);
+}
+
+struct file_operations proc_container_operations = {
+ .open  = container_open,
+ .read  = seq_read,
+ .llseek  = seq_lseek,
+ .release = single_release,
+};
Index: container-2.6.20-new/kernel/exit.c
 ===================================================================== =======
--- container-2.6.20-new.orig/kernel/exit.c
+++ container-2.6.20-new/kernel/exit.c
@@ -31,6 +31,7 @@
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
 #include <linux/posix-timers.h>
@@ -927,6 +928,7 @@ fastcall NORET_TYPE void do_exit(long co
  __exit_fs(tsk);
  exit_thread();
```

```
  cpuset_exit(tsk);
+ container_exit(tsk);
  exit_keys(tsk);

  if (group_dead && tsk->signal->leader)
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
@@ -31,6 +31,7 @@
 #include <linux/capability.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/security.h>
 #include <linux/swap.h>
 #include <linux/syscalls.h>
@@ -1058,6 +1059,7 @@ static struct task_struct *copy_process(
  p->io_context = NULL;
  p->io_wait = NULL;
  p->audit_context = NULL;
+ container_fork(p);
  cpuset_fork(p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
@@ -1291,6 +1293,7 @@ bad_fork_cleanup_policy:
 bad_fork_cleanup_cpuset:
 #endif
  cpuset_exit(p);
+ container_exit(p);
 bad_fork_cleanup_delays_binfmt:
  delayacct_tsk_free(p);
  if (p->binfmt)
Index: container-2.6.20-new/kernel/Makefile
 ===================================================================
--- container-2.6.20-new.orig/kernel/Makefile
+++ container-2.6.20-new/kernel/Makefile
@@ -35,6 +35,7 @@ obj-$(CONFIG_PM) += power/
 obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
+obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
Index: container-2.6.20-new/Documentation/containers.txt
 ===================================================================
--- /dev/null
```

+++ container-2.6.20-new/Documentation/containers.txt
@@ -0,0 +1,229 @@
+    CONTAINERS
+    -------
+
+Written by Paul Menage <menage@google.com> based on Documentation/cpusets.txt
+
+Original copyright in cpusets.txt:
+Portions Copyright (C) 2004 BULL SA.
+Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
+Modified by Paul Jackson <pj@sgi.com>
+Modified by Christoph Lameter <clameter@sgi.com>
+
+CONTENTS:
+=========
+
+1. Containers
+  1.1 What are containers ?
+  1.2 Why are containers needed ?
+  1.3 How are containers implemented ?
+  1.4 What does notify_on_release do ?
+  1.5 How do I use containers ?
+2. Usage Examples and Syntax
+  2.1 Basic Usage
+  2.2 Attaching processes
+3. Questions
+4. Contact
+
+1. Containers
+==========
+
+1.1 What are containers ?
+----------------------
+
+Containers provide a mechanism for aggregating sets of tasks, and all
+their children, into hierarchical groups.
+
+Each task has a pointer to a container.  Multiple tasks may reference
+the same container. User level code may create and destroy containers
+by name in the container virtual file system, specify and query to
+which container a task is assigned, and list the task pids assigned to
+a container.
+
+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.

+
+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.
+
+The kernel container patch provides the minimum essential kernel
+mechanisms required to efficiently implement such groups. It has
+minimal impact on the system fast paths, and provides hooks for
+specific subsystems such as cpusets to provide additional behaviour as
+desired.
+
+
+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.
+ - You can list all the tasks (by pid) attached to any container.
+
+The implementation of containers requires a few, simple hooks
+into the rest of the kernel, none in performance critical paths:
+
+ - in init/main.c, to initialize the root container at system boot.
+ - in fork and exit, to attach and detach a task from its container.
+
+In addition a new file system, of type "container" may be mounted,
+typically at /dev/container, to enable browsing and modifying the containers
+presently known to the kernel.  No new system calls are added for
+containers - all support for querying and modifying containers is via
+this container file system.
+
+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.
+
+Each container is represented by a directory in the container file system
+containing the following files describing that container:
+

+ - tasks: list of tasks (by pid) attached to that container
+ - notify_on_release flag: run /sbin/container_release_agent on exit?
+
+Other subsystems such as cpusets may add additional files in each
+container dir
+
+New containers are created using the mkdir system call or shell
+command.  The properties of a container, such as its flags, are
+modified by writing to the appropriate file in that containers
+directory, as listed above.
+
+The named hierarchical structure of nested containers allows partitioning
+a large system into nested, dynamically changeable, "soft-partitions".
+
+The attachment of each task, automatically inherited at fork by any
+children of that task, to a container allows organizing the work load
+on a system into related sets of tasks.  A task may be re-attached to
+any other container, if allowed by the permissions on the necessary
+container file system directories.
+
+The use of a Linux virtual file system (vfs) to represent the
+container hierarchy provides for a familiar permission and name space
+for containers, with a minimum of additional kernel code.
+
+1.4 What does notify_on_release do ?
+----------------------------------
+
+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.
+
+1.5 How do I use containers ?
+---------------------------
+
+To start a new job that is to be contained within a container, the steps are:
+
+ 1) mkdir /dev/container
+ 2) mount -t container container /dev/container
+ 3) Create the new container by doing mkdir's and write's (or echo's) in
+    the /dev/container virtual file system.
+ 4) Start a task that will be the "founding father" of the new job.
+ 5) Attach that task to the new container by writing its pid to the

+    /dev/container tasks file for that container.
+ 6) fork, exec or clone the job tasks from this founding father task.
+
+For example, the following sequence of commands will setup a container
+named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
+and then start a subshell 'sh' in that container:
+
+  mount -t container none /dev/container
+  cd /dev/container
+  mkdir Charlie
+  cd Charlie
+  /bin/echo $$ > tasks
+  sh
+  # The subshell 'sh' is now running in container Charlie
+  # The next line should display '/Charlie'
+  cat /proc/self/container
+
+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
+============================
+
+2.1 Basic Usage
+---------------
+
+Creating, modifying, using the containers can be done through the container
+virtual filesystem.
+
+To mount it, type:
+# mount -t container none /dev/container
+
+Then under /dev/container you can find a tree that corresponds to the
+tree of the containers in the system. For instance, /dev/container
+is the container that holds the whole system.
+
+If you want to create a new container under /dev/container:
+# cd /dev/container
+# mkdir my_container
+
+Now you want to do something with this container.
+# cd my_container
+
+In this directory you can find several files:
+# ls
+notify_on_release  tasks

+
+Now attach your shell to this container:
+# /bin/echo $$ > tasks
+
+You can also create containers inside your container by using mkdir in this
+directory.
+# mkdir my_sub_cs
+
+To remove a container, just use rmdir:
+# rmdir my_sub_cs
+This will fail if the container is in use (has containers inside, or has
+processes attached).
+
+2.2 Attaching processes
+----------------------
+
+# /bin/echo PID > tasks
+
+Note that it is PID, not PIDs. You can only attach ONE task at a time.
+If you have several tasks to attach, you have to do it one after another:
+
+# /bin/echo PID1 > tasks
+# /bin/echo PID2 > tasks
+ ...
+# /bin/echo PIDn > tasks
+
+
+3. Questions
+============
+
+Q: what's up with this '/bin/echo' ?
+A: bash's builtin 'echo' command does not check calls to write() against
+   errors. If you use it in the container file system, you won't be
+   able to tell whether a command succeeded or failed.
+
+Q: When I attach processes, only the first of the line gets really attached !
+A: We can only return one error code per call to write(). So you should also
+   put only ONE pid.
+

--

---

Subject: [PATCH 2/7] Containers (V8): Cpusets hooked into containers
Posted by Paul Menage on Fri, 06 Apr 2007 23:32:23 GMT
View Forum Message <> Reply to Message

This patch removes the process grouping code from the cpusets code,

instead hooking it into the generic container system. This temporarily
adds cpuset-specific code in kernel/container.c, which is removed by
the next patch in the series.

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

---
 Documentation/cpusets.txt |   81 +--
 fs/proc/base.c            |    4
 fs/super.c                |    5
 include/linux/container.h |   10
 include/linux/cpuset.h    |   16
 include/linux/fs.h        |    2
 include/linux/mempolicy.h |    2
 include/linux/sched.h     |    4
 init/Kconfig              |   14
 kernel/container.c        |  117 ++++
 kernel/cpuset.c           | 1100 +++++----------------------------------------
 kernel/exit.c             |    2
 kernel/fork.c             |    7
 13 files changed, 286 insertions(+), 1078 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
@@ -27,9 +27,6 @@ extern struct file_operations proc_conta
 extern void container_lock(void);
 extern void container_unlock(void);

-extern void container_manage_lock(void);
-extern void container_manage_unlock(void);
-
 struct container {
  unsigned long flags;  /* "unsigned long" so bitops work */

@@ -47,6 +44,10 @@ struct container {

  struct container *parent; /* my parent */
  struct dentry *dentry;  /* container fs entry */
+
+#ifdef CONFIG_CPUSETS
+ struct cpuset *cpuset;
+#endif
 };

 /* struct cftype:
@@ -79,6 +80,9 @@ struct cftype {

```
 int container_add_file(struct container *cont, const struct cftype *cft);

 int container_is_removed(const struct container *cont);
+void container_set_release_agent_path(const char *path);
+
+int container_path(const struct container *cont, char *buf, int buflen);

 #else /* !CONFIG_CONTAINERS */

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
@@ -11,16 +11,15 @@
 #include <linux/sched.h>
 #include <linux/cpumask.h>
 #include <linux/nodemask.h>
+#include <linux/container.h>

 #ifdef CONFIG_CPUSETS

-extern int number_of_cpusets; /* How many cpusets are defined in system? */
+extern int number_of_cpusets;  /* How many cpusets are defined in system? */

 extern int cpuset_init_early(void);
 extern int cpuset_init(void);
 extern void cpuset_init_smp(void);
-extern void cpuset_fork(struct task_struct *p);
-extern void cpuset_exit(struct task_struct *p);
 extern cpumask_t cpuset_cpus_allowed(struct task_struct *p);
 extern nodemask_t cpuset_mems_allowed(struct task_struct *p);
 #define cpuset_current_mems_allowed (current->mems_allowed)
@@ -75,13 +74,20 @@ 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);
+
 #else /* !CONFIG_CPUSETS */

 static inline int cpuset_init_early(void) { return 0; }
```

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

 static inline cpumask_t cpuset_cpus_allowed(struct task_struct *p)
 {
Index: container-2.6.20-new/kernel/exit.c
 ===================================================================
--- container-2.6.20-new.orig/kernel/exit.c
+++ container-2.6.20-new/kernel/exit.c
@@ -30,7 +30,6 @@
 #include <linux/mempolicy.h>
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
-#include <linux/cpuset.h>
 #include <linux/container.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
@@ -927,7 +926,6 @@ fastcall NORET_TYPE void do_exit(long co
  __exit_files(tsk);
  __exit_fs(tsk);
  exit_thread();
- cpuset_exit(tsk);
  container_exit(tsk);
  exit_keys(tsk);

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
@@ -30,7 +30,6 @@
 #include <linux/nsproxy.h>
 #include <linux/capability.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
 #include <linux/container.h>
 #include <linux/security.h>
 #include <linux/swap.h>
@@ -1060,13 +1059,12 @@ static struct task_struct *copy_process(
  p->io_wait = NULL;
  p->audit_context = NULL;
  container_fork(p);
- cpuset_fork(p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
   if (IS_ERR(p->mempolicy)) {
    retval = PTR_ERR(p->mempolicy);
```

```
    p->mempolicy = NULL;
-   goto bad_fork_cleanup_cpuset;
+   goto bad_fork_cleanup_container;
   }
  mpol_fix_fork_child_flag(p);
 #endif
@@ -1290,9 +1288,8 @@ bad_fork_cleanup_security:
 bad_fork_cleanup_policy:
 #ifdef CONFIG_NUMA
  mpol_free(p->mempolicy);
-bad_fork_cleanup_cpuset:
+bad_fork_cleanup_container:
 #endif
- cpuset_exit(p);
  container_exit(p);
 bad_fork_cleanup_delays_binfmt:
  delayacct_tsk_free(p);
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
@@ -55,6 +55,7 @@
 #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>
@@ -92,6 +93,18 @@ static struct container top_container =
  .children = LIST_HEAD_INIT(top_container.children),
 };

+/* The path to use for release notifications. No locking between
+ * setting and use - so if userspace updates this while subcontainers
+ * exist, you could miss a notification */
+static char release_agent_path[PATH_MAX] = "/sbin/container_release_agent";
+
+void container_set_release_agent_path(const char *path)
+{
+ container_lock();
+ strcpy(release_agent_path, path);
+ container_unlock();
+}
+
 static struct vfsmount *container_mount;
 static struct super_block *container_sb;
```

```
@@ -303,7 +316,7 @@ static inline struct cftype *__d_cft(str
  * Returns 0 on success, -errno on error.
  */

-static int container_path(const struct container *cont, char *buf, int buflen)
+int container_path(const struct container *cont, char *buf, int buflen)
 {
  char *start;

@@ -367,7 +380,7 @@ static void container_release_agent(cons
   return;

  i = 0;
- argv[i++] = "/sbin/container_release_agent";
+ argv[i++] = release_agent_path;
  argv[i++] = (char *)pathbuf;
  argv[i] = NULL;

@@ -408,6 +421,7 @@ static void check_for_release(struct con
   buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
   if (!buf)
    return;
+
   if (container_path(cont, buf, PAGE_SIZE) < 0)
    kfree(buf);
   else
@@ -454,7 +468,7 @@ static int attach_task(struct container
  pid_t pid;
  struct task_struct *tsk;
  struct container *oldcont;
- int retval;
+ int retval = 0;

  if (sscanf(pidbuf, "%d", &pid) != 1)
   return -EIO;
@@ -481,7 +495,9 @@ static int attach_task(struct container
   get_task_struct(tsk);
  }

- retval = security_task_setscheduler(tsk, 0, NULL);
+#ifdef CONFIG_CPUSETS
+ retval = cpuset_can_attach_task(cont, tsk);
+#endif
  if (retval) {
   put_task_struct(tsk);
   return retval;
@@ -489,15 +505,19 @@ static int attach_task(struct container
```

```
  task_lock(tsk);
  oldcont = tsk->container;
- if (!oldcont) {
-  task_unlock(tsk);
-  put_task_struct(tsk);
-  return -ESRCH;
+  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
+
  put_task_struct(tsk);
  synchronize_rcu();
  if (atomic_dec_and_test(&oldcont->count))
@@ -512,6 +532,7 @@ typedef enum {
  FILE_DIR,
  FILE_NOTIFY_ON_RELEASE,
  FILE_TASKLIST,
+ FILE_RELEASE_AGENT,
 } container_filetype_t;

 static ssize_t container_common_file_write(struct container *cont,
@@ -525,8 +546,7 @@ static ssize_t container_common_file_wri
  char *pathbuf = NULL;
  int retval = 0;

- /* Crude upper limit on largest legitimate cpulist user might write. */
- if (nbytes > 100 + 6 * NR_CPUS)
+ if (nbytes >= PATH_MAX)
   return -E2BIG;

  /* +1 for nul-terminator */
@@ -553,6 +573,20 @@ static ssize_t container_common_file_wri
  case FILE_TASKLIST:
   retval = attach_task(cont, buffer, &pathbuf);
   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;
@@ -606,6 +640,17 @@ static ssize_t container_common_file_rea
  case FILE_NOTIFY_ON_RELEASE:
   *s++ = notify_on_release(cont) ? '1' : '0';
   break;
+ case FILE_RELEASE_AGENT:
+ {
+  size_t n;
+  container_lock();
+  n = strnlen(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;
@@ -941,6 +986,13 @@ static struct cftype cft_notify_on_relea
  .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;
@@ -949,6 +1001,13 @@ static int container_populate_dir(struct
   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
  return 0;
 }

@@ -980,6 +1039,12 @@ static long container_create(struct cont

  cont->parent = parent;

+#ifdef CONFIG_CPUSETS
+ err = cpuset_create(cont);
+ if (err)
+  goto err_unlock_free;
+#endif
+
  list_add(&cont->sibling, &cont->parent->children);
  number_of_containers++;

@@ -999,9 +1064,12 @@ static long container_create(struct cont
  return 0;

  err_remove:
+#ifdef CONFIG_CPUSETS
+ cpuset_destroy(cont);
+#endif
  list_del(&cont->sibling);
  number_of_containers--;
-
+ err_unlock_free:
  mutex_unlock(&container_mutex);
  kfree(cont);
  return err;
@@ -1043,6 +1111,9 @@ static int container_rmdir(struct inode
  container_d_remove_dir(d);
  dput(d);
  number_of_containers--;
+#ifdef CONFIG_CPUSETS
+ cpuset_destroy(cont);
+#endif

  if (list_empty(&parent->children))
   check_for_release(parent, &pathbuf);
```

```
@@ -1142,10 +1213,6 @@ void container_fork(struct task_struct *
  * to any other task to nuke the container immediately, via
  * container_rmdir().
  *
- * We don't need to task_lock() this reference to tsk->container,
- * because tsk is already marked PF_EXITING, so attach_task() won't
- * mess with it, or task is a failed fork, never visible to attach_task.
- *
  * the_top_container_hack:
  *
  *    Set the exiting tasks container to the root container (top_container).
@@ -1184,8 +1251,10 @@ void container_exit(struct task_struct *
 {
  struct container *cont;

+ 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;
@@ -1200,6 +1269,24 @@ void container_exit(struct task_struct *
  }
 }

+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);
+}
+
+
+
 /*
  * proc_container_show()
  *  - Print tasks container path into seq_file.
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
@@ -54,8 +54,6 @@
 #include <asm/atomic.h>
 #include <linux/mutex.h>

-#define CPUSET_SUPER_MAGIC  0x27e0eb
-
 /*
  * Tracks how many cpusets are currently defined in system.
  * When there is only one cpuset (the root cpuset) we can
@@ -77,20 +75,8 @@ struct cpuset {
  cpumask_t cpus_allowed;  /* CPUs allowed to tasks in cpuset */
  nodemask_t mems_allowed; /* Memory Nodes allowed to tasks */

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

  /*
   * Copy of global cpuset_mems_generation as of the most
@@ -106,8 +92,6 @@ typedef enum {
  CS_CPU_EXCLUSIVE,
  CS_MEM_EXCLUSIVE,
  CS_MEMORY_MIGRATE,
- CS_REMOVED,
- CS_NOTIFY_ON_RELEASE,
  CS_SPREAD_PAGE,
  CS_SPREAD_SLAB,
 } cpuset_flagbits_t;
@@ -123,16 +107,6 @@ static inline int is_mem_exclusive(const
  return test_bit(CS_MEM_EXCLUSIVE, &cs->flags);
 }

-static inline int is_removed(const struct cpuset *cs)
```

```
-{
- return test_bit(CS_REMOVED, &cs->flags);
-}
-
-static inline int notify_on_release(const struct cpuset *cs)
-{
- return test_bit(CS_NOTIFY_ON_RELEASE, &cs->flags);
-}
-
 static inline int is_memory_migrate(const struct cpuset *cs)
 {
  return test_bit(CS_MEMORY_MIGRATE, &cs->flags);
@@ -173,14 +147,8 @@ static struct cpuset top_cpuset = {
  .flags = ((1 << CS_CPU_EXCLUSIVE) | (1 << CS_MEM_EXCLUSIVE)),
  .cpus_allowed = CPU_MASK_ALL,
  .mems_allowed = NODE_MASK_ALL,
- .count = ATOMIC_INIT(0),
- .sibling = LIST_HEAD_INIT(top_cpuset.sibling),
- .children = LIST_HEAD_INIT(top_cpuset.children),
 };

-static struct vfsmount *cpuset_mount;
-static struct super_block *cpuset_sb;
-
 /*
  * We have two global cpuset mutexes below.  They can nest.
  * It is ok to first take manage_mutex, then nest callback_mutex.  We also
@@ -264,297 +232,32 @@ static struct super_block *cpuset_sb;
  * the routine cpuset_update_task_memory_state().
  */

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

-/*
- * A couple of forward declarations required, due to cyclic reference loop:
- *  cpuset_mkdir -> cpuset_create -> cpuset_populate_dir -> cpuset_add_file
- *  -> cpuset_create_file -> cpuset_dir_inode_operations -> cpuset_mkdir.
- */
-
-static int cpuset_mkdir(struct inode *dir, struct dentry *dentry, int mode);
-static int cpuset_rmdir(struct inode *unused_dir, struct dentry *dentry);
-
-static struct backing_dev_info cpuset_backing_dev_info = {
- .ra_pages = 0,  /* No readahead */
- .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
-};
-
```

```
-static struct inode *cpuset_new_inode(mode_t mode)
-{
- struct inode *inode = new_inode(cpuset_sb);
-
- if (inode) {
-  inode->i_mode = mode;
-  inode->i_uid = current->fsuid;
-  inode->i_gid = current->fsgid;
-  inode->i_blocks = 0;
-  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
-  inode->i_mapping->backing_dev_info = &cpuset_backing_dev_info;
- }
- return inode;
-}
-
-static void cpuset_diput(struct dentry *dentry, struct inode *inode)
-{
- /* is dentry a directory ? if so, kfree() associated cpuset */
- if (S_ISDIR(inode->i_mode)) {
-  struct cpuset *cs = dentry->d_fsdata;
-  BUG_ON(!(is_removed(cs)));
-  kfree(cs);
- }
- iput(inode);
-}
-
-static struct dentry_operations cpuset_dops = {
- .d_iput = cpuset_diput,
-};
-
-static struct dentry *cpuset_get_dentry(struct dentry *parent, const char *name)
-{
- struct dentry *d = lookup_one_len(name, parent, strlen(name));
- if (!IS_ERR(d))
-  d->d_op = &cpuset_dops;
- return d;
-}
-
-static void remove_dir(struct dentry *d)
-{
- struct dentry *parent = dget(d->d_parent);
-
- d_delete(d);
- simple_rmdir(parent->d_inode, d);
- dput(parent);
-}
-
-/*
```

```
- * NOTE : the dentry must have been dget()'ed
- */
-static void cpuset_d_remove_dir(struct dentry *dentry)
-{
- struct list_head *node;
-
- spin_lock(&dcache_lock);
- node = dentry->d_subdirs.next;
- while (node != &dentry->d_subdirs) {
-  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
-  list_del_init(node);
-  if (d->d_inode) {
-   d = dget_locked(d);
-   spin_unlock(&dcache_lock);
-   d_delete(d);
-   simple_unlink(dentry->d_inode, d);
-   dput(d);
-   spin_lock(&dcache_lock);
-  }
-  node = dentry->d_subdirs.next;
- }
- list_del_init(&dentry->d_u.d_child);
- spin_unlock(&dcache_lock);
- remove_dir(dentry);
-}
-
-static struct super_operations cpuset_ops = {
- .statfs = simple_statfs,
- .drop_inode = generic_delete_inode,
-};
-
-static int cpuset_fill_super(struct super_block *sb, void *unused_data,
-       int unused_silent)
-{
- struct inode *inode;
- struct dentry *root;
-
- sb->s_blocksize = PAGE_CACHE_SIZE;
- sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
- sb->s_magic = CPUSET_SUPER_MAGIC;
- sb->s_op = &cpuset_ops;
- cpuset_sb = sb;
-
- inode = cpuset_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR);
- if (inode) {
-  inode->i_op = &simple_dir_inode_operations;
-  inode->i_fop = &simple_dir_operations;
-  /* directories start off with i_nlink == 2 (for "." entry) */
```

```
-  inc_nlink(inode);
- } else {
-  return -ENOMEM;
- }
-
- root = d_alloc_root(inode);
- if (!root) {
-  iput(inode);
-  return -ENOMEM;
- }
- sb->s_root = root;
- return 0;
-}
-
+/* This is ugly, but preserves the userspace API for existing cpuset
+ * users. If someone tries to mount the "cpuset" filesystem, we
+ * silently switch it to mount "container" instead */
 static int cpuset_get_sb(struct file_system_type *fs_type,
     int flags, const char *unused_dev_name,
     void *data, struct vfsmount *mnt)
 {
- return get_sb_single(fs_type, flags, data, cpuset_fill_super, mnt);
+ struct file_system_type *container_fs = get_fs_type("container");
+ int ret = -ENODEV;
+ 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);
+  put_filesystem(container_fs);
+ }
+ return ret;
 }

 static struct file_system_type cpuset_fs_type = {
  .name = "cpuset",
  .get_sb = cpuset_get_sb,
- .kill_sb = kill_litter_super,
 };

-/* struct cftype:
- *
- * The files in the cpuset filesystem mostly have a very simple read/write
- * handling, some common function will take care of it. Nevertheless some cases
- * (read tasks) are special and therefore I define this structure for every
- * kind of file.
- *
- *
```

```
- * When reading/writing to a file:
- * - the cpuset to use in file->f_path.dentry->d_parent->d_fsdata
- * - the 'cftype' of the file is file->f_path.dentry->d_fsdata
- */
-
-struct cftype {
- char *name;
- int private;
- int (*open) (struct inode *inode, struct file *file);
- ssize_t (*read) (struct file *file, char __user *buf, size_t nbytes,
-       loff_t *ppos);
- int (*write) (struct file *file, const char __user *buf, size_t nbytes,
-       loff_t *ppos);
- int (*release) (struct inode *inode, struct file *file);
-};
-
-static inline struct cpuset *__d_cs(struct dentry *dentry)
-{
- return dentry->d_fsdata;
-}
-
-static inline struct cftype *__d_cft(struct dentry *dentry)
-{
- return dentry->d_fsdata;
-}
-
-/*
- * Call with manage_mutex held.  Writes path of cpuset into buf.
- * Returns 0 on success, -errno on error.
- */
-
-static int cpuset_path(const struct cpuset *cs, char *buf, int buflen)
-{
- char *start;
-
- start = buf + buflen;
-
- *--start = '\0';
- for (;;) {
-  int len = cs->dentry->d_name.len;
-  if ((start -= len) < buf)
-   return -ENAMETOOLONG;
-  memcpy(start, cs->dentry->d_name.name, len);
-  cs = cs->parent;
-  if (!cs)
-   break;
-  if (!cs->parent)
-   continue;
```

```
-  if (--start < buf)
-   return -ENAMETOOLONG;
-  *start = '/';
- }
- memmove(buf, start, buf + buflen - start);
- return 0;
-}
-
-/*
- * Notify userspace when a cpuset is released, by running
- * /sbin/cpuset_release_agent with the name of the cpuset (path
- * relative to the root of cpuset file system) as the argument.
- *
- * Most likely, this user command will try to rmdir this cpuset.
- *
- * This races with the possibility that some other task will be
- * attached to this cpuset before it is removed, or that some other
- * user task will 'mkdir' a child cpuset of this cpuset.  That's ok.
- * The presumed 'rmdir' will fail quietly if this cpuset is no longer
- * unused, and this cpuset will be reprieved from its death sentence,
- * to continue to serve a useful existence.  Next time it's released,
- * we will get notified again, if it still has 'notify_on_release' set.
- *
- * The final arg to call_usermodehelper() is 0, which means don't
- * wait.  The separate /sbin/cpuset_release_agent task is forked by
- * call_usermodehelper(), then control in this thread returns here,
- * without waiting for the release agent task.  We don't bother to
- * wait because the caller of this routine has no use for the exit
- * status of the /sbin/cpuset_release_agent task, so no sense holding
- * our caller up for that.
- *
- * When we had only one cpuset mutex, we had to call this
- * without holding it, to avoid deadlock when call_usermodehelper()
- * allocated memory.  With two locks, we could now call this while
- * holding manage_mutex, but we still don't, so as to minimize
- * the time manage_mutex is held.
- */
-
-static void cpuset_release_agent(const char *pathbuf)
-{
- char *argv[3], *envp[3];
- int i;
-
- if (!pathbuf)
-  return;
-
- i = 0;
- argv[i++] = "/sbin/cpuset_release_agent";
```

```
- argv[i++] = (char *)pathbuf;
- argv[i] = NULL;
-
- i = 0;
- /* minimal command environment */
- envp[i++] = "HOME=/";
- envp[i++] = "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
- envp[i] = NULL;
-
- call_usermodehelper(argv[0], argv, envp, 0);
- kfree(pathbuf);
-}
-
-/*
- * Either cs->count of using tasks transitioned to zero, or the
- * cs->children list of child cpusets just became empty.  If this
- * cs is notify_on_release() and now both the user count is zero and
- * the list of children is empty, prepare cpuset path in a kmalloc'd
- * buffer, to be returned via ppathbuf, so that the caller can invoke
- * cpuset_release_agent() with it later on, once manage_mutex is dropped.
- * Call here with manage_mutex held.
- *
- * This check_for_release() routine is responsible for kmalloc'ing
- * pathbuf.  The above cpuset_release_agent() is responsible for
- * kfree'ing pathbuf.  The caller of these routines is responsible
- * for providing a pathbuf pointer, initialized to NULL, then
- * calling check_for_release() with manage_mutex held and the address
- * of the pathbuf pointer, then dropping manage_mutex, then calling
- * cpuset_release_agent() with pathbuf, as set by check_for_release().
- */
-
-static void check_for_release(struct cpuset *cs, char **ppathbuf)
-{
- if (notify_on_release(cs) && atomic_read(&cs->count) == 0 &&
-     list_empty(&cs->children)) {
-  char *buf;
-
-  buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
-  if (!buf)
-   return;
-  if (cpuset_path(cs, buf, PAGE_SIZE) < 0)
-   kfree(buf);
-  else
-   *ppathbuf = buf;
- }
-}
-
 /*
```

```
  * Return in *pmask the portion of a cpusets's cpus_allowed that
  * are online.  If none are online, walk up the cpuset hierarchy
@@ -652,12 +355,12 @@ void cpuset_update_task_memory_state(voi
  struct task_struct *tsk = current;
  struct cpuset *cs;

- if (tsk->cpuset == &top_cpuset) {
+ if (tsk->container->cpuset == &top_cpuset) {
   /* Don't need rcu for top_cpuset.  It's never freed. */
   my_cpusets_mem_gen = top_cpuset.mems_generation;
  } else {
   rcu_read_lock();
-  cs = rcu_dereference(tsk->cpuset);
+  cs = rcu_dereference(tsk->container->cpuset);
   my_cpusets_mem_gen = cs->mems_generation;
   rcu_read_unlock();
  }
@@ -665,7 +368,7 @@ void cpuset_update_task_memory_state(voi
  if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
   mutex_lock(&callback_mutex);
   task_lock(tsk);
-  cs = tsk->cpuset; /* Maybe changed when task not locked */
+  cs = tsk->container->cpuset; /* 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))
@@ -720,10 +423,12 @@ static int is_cpuset_subset(const struct

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

  /* Each of our child cpusets must be a subset of us */
- list_for_each_entry(c, &cur->children, sibling) {
+ list_for_each_entry(cont, &cur->container->children, sibling) {
+  c = cont->cpuset;
   if (!is_cpuset_subset(c, trial))
    return -EBUSY;
  }
@@ -739,7 +444,8 @@ static int validate_change(const struct
   return -EACCES;

  /* If either I or some sibling (!= me) is exclusive, we can't overlap */
- list_for_each_entry(c, &par->children, sibling) {
+ list_for_each_entry(cont, &par->container->children, sibling) {
+  c = cont->cpuset;
   if ((is_cpu_exclusive(trial) || is_cpu_exclusive(c)) &&
```

```
       c != cur &&
       cpus_intersects(trial->cpus_allowed, c->cpus_allowed))
@@ -769,6 +475,7 @@ static int validate_change(const struct

 static void update_cpu_domains(struct cpuset *cur)
 {
+ struct container *cont;
  struct cpuset *c, *par = cur->parent;
  cpumask_t pspan, cspan;

@@ -780,7 +487,8 @@ static void update_cpu_domains(struct cp
   * children
   */
  pspan = par->cpus_allowed;
- list_for_each_entry(c, &par->children, sibling) {
+ list_for_each_entry(cont, &par->container->children, sibling) {
+  c = cont->cpuset;
   if (is_cpu_exclusive(c))
    cpus_andnot(pspan, pspan, c->cpus_allowed);
  }
@@ -797,7 +505,8 @@ 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(c, &cur->children, sibling) {
+  list_for_each_entry(cont, &cur->container->children, sibling) {
+   c = cont->cpuset;
    if (is_cpu_exclusive(c))
     cpus_andnot(cspan, cspan, c->cpus_allowed);
   }
@@ -885,7 +594,7 @@ static void cpuset_migrate_mm(struct mm_
  do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);

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

@@ -913,12 +622,14 @@ static int update_nodemask(struct cpuset
  int migrate;
  int fudge;
  int retval;
+ struct container *cont;

  /* top_cpuset.mems_allowed tracks node_online_map; it's read-only */
  if (cs == &top_cpuset)
   return -EACCES;
```

```
  trialcs = *cs;
+ cont = cs->container;
  retval = nodelist_parse(buf, trialcs.mems_allowed);
  if (retval < 0)
   goto done;
@@ -955,13 +666,13 @@ static int update_nodemask(struct cpuset
   * enough mmarray[] w/o using GFP_ATOMIC.
   */
  while (1) {
-  ntasks = atomic_read(&cs->count); /* guess */
+  ntasks = atomic_read(&cs->container->count);  /* guess */
   ntasks += fudge;
   mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
   if (!mmarray)
    goto done;
   write_lock_irq(&tasklist_lock);  /* block fork */
-  if (atomic_read(&cs->count) <= ntasks)
+  if (atomic_read(&cs->container->count) <= ntasks)
    break;    /* got enough */
   write_unlock_irq(&tasklist_lock); /* try again */
   kfree(mmarray);
@@ -978,7 +689,7 @@ static int update_nodemask(struct cpuset
     "Cpuset mempolicy rebind incomplete.\n");
    continue;
   }
-  if (p->cpuset != cs)
+  if (p->container != cont)
    continue;
   mm = get_task_mm(p);
   if (!mm)
@@ -1168,85 +879,32 @@ static int fmeter_getrate(struct fmeter
  return val;
 }

-/*
- * Attack task specified by pid in 'pidbuf' to cpuset 'cs', possibly
- * writing the path of the old cpuset in 'ppathbuf' if it needs to be
- * notified on release.
- *
- * Call holding manage_mutex.  May take callback_mutex and task_lock of
- * the task 'pid' during call.
- */
-
-static int attach_task(struct cpuset *cs, char *pidbuf, char **ppathbuf)
+int cpuset_can_attach_task(struct container *cont, struct task_struct *tsk)
 {
- pid_t pid;
```

```
- struct task_struct *tsk;
- struct cpuset *oldcs;
- cpumask_t cpus;
- nodemask_t from, to;
- struct mm_struct *mm;
- int retval;
+ struct cpuset *cs = cont->cpuset;

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

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

- retval = security_task_setscheduler(tsk, 0, NULL);
- if (retval) {
-  put_task_struct(tsk);
-  return retval;
- }
+void cpuset_attach_task(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;

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

  from = oldcs->mems_allowed;
  to = cs->mems_allowed;
-
- mutex_unlock(&callback_mutex);
-
  mm = get_task_mm(tsk);
  if (mm) {
   mpol_rebind_mm(mm, &to);
@@ -1255,40 +913,31 @@ static int attach_task(struct cpuset *cs
   mmput(mm);
  }

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

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

 typedef enum {
```

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

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

  /* Crude upper limit on largest legitimate cpulist user might write. */
@@ -1305,9 +954,9 @@ static ssize_t cpuset_common_file_write(
  }
  buffer[nbytes] = 0; /* nul-terminate */

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

- if (is_removed(cs)) {
+ if (container_is_removed(cont)) {
   retval = -ENODEV;
   goto out2;
  }
@@ -1325,9 +974,6 @@ static ssize_t cpuset_common_file_write(
  case FILE_MEM_EXCLUSIVE:
   retval = update_flag(CS_MEM_EXCLUSIVE, cs, buffer);
   break;
- case FILE_NOTIFY_ON_RELEASE:
```

```
-  retval = update_flag(CS_NOTIFY_ON_RELEASE, cs, buffer);
-  break;
   case FILE_MEMORY_MIGRATE:
    retval = update_flag(CS_MEMORY_MIGRATE, cs, buffer);
    break;
@@ -1345,9 +991,6 @@ static ssize_t cpuset_common_file_write(
    retval = update_flag(CS_SPREAD_SLAB, cs, buffer);
    cs->mems_generation = cpuset_mems_generation++;
    break;
- case FILE_TASKLIST:
-  retval = attach_task(cs, buffer, &pathbuf);
-  break;
   default:
    retval = -EINVAL;
    goto out2;
@@ -1356,30 +999,12 @@ static ssize_t cpuset_common_file_write(
   if (retval == 0)
    retval = nbytes;
  out2:
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
+ container_unlock();
  out1:
   kfree(buffer);
   return retval;
  }

-static ssize_t cpuset_file_write(struct file *file, const char __user *buf,
-      size_t nbytes, loff_t *ppos)
-{
- ssize_t retval = 0;
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (!cft)
-  return -ENODEV;
-
- /* special function ? */
- if (cft->write)
-  retval = cft->write(file, buf, nbytes, ppos);
- else
-  retval = cpuset_common_file_write(file, buf, nbytes, ppos);
-
- return retval;
-}
-
  /*
   * These ascii lists should be read in a single call, by using a user
   * buffer large enough to hold the entire map.  If read in smaller
@@ -1414,11 +1039,13 @@ static int cpuset_sprintf_memlist(char *
```

```
  return nodelist_scnprintf(page, PAGE_SIZE, mask);
 }

-static ssize_t cpuset_common_file_read(struct file *file, char __user *buf,
-    size_t nbytes, loff_t *ppos)
+static ssize_t cpuset_common_file_read(struct container *cont,
+           struct cftype *cft,
+           struct file *file,
+           char __user *buf,
+           size_t nbytes, loff_t *ppos)
 {
- struct cftype *cft = __d_cft(file->f_path.dentry);
- struct cpuset *cs = __d_cs(file->f_path.dentry->d_parent);
+ struct cpuset *cs = cont->cpuset;
  cpuset_filetype_t type = cft->private;
  char *page;
  ssize_t retval = 0;
@@ -1442,9 +1069,6 @@ static ssize_t cpuset_common_file_read(s
  case FILE_MEM_EXCLUSIVE:
   *s++ = is_mem_exclusive(cs) ? '1' : '0';
   break;
- case FILE_NOTIFY_ON_RELEASE:
-  *s++ = notify_on_release(cs) ? '1' : '0';
-  break;
  case FILE_MEMORY_MIGRATE:
   *s++ = is_memory_migrate(cs) ? '1' : '0';
   break;
@@ -1472,391 +1096,96 @@ out:
  return retval;
 }

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

```
-static int cpuset_file_open(struct inode *inode, struct file *file)
-{
- int err;
- struct cftype *cft;
-
- err = generic_file_open(inode, file);
- if (err)
-  return err;
-
- cft = __d_cft(file->f_path.dentry);
- if (!cft)
-  return -ENODEV;
- if (cft->open)
-  err = cft->open(inode, file);
- else
-  err = 0;
-
- return err;
-}
-
-static int cpuset_file_release(struct inode *inode, struct file *file)
-{
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (cft->release)
-  return cft->release(inode, file);
- return 0;
-}
-
-/*
- * cpuset_rename - Only allow simple rename of directories in place.
- */
-static int cpuset_rename(struct inode *old_dir, struct dentry *old_dentry,
-                         struct inode *new_dir, struct dentry *new_dentry)
-{
- if (!S_ISDIR(old_dentry->d_inode->i_mode))
-  return -ENOTDIR;
- if (new_dentry->d_inode)
-  return -EEXIST;
- if (old_dir != new_dir)
-  return -EIO;
- return simple_rename(old_dir, old_dentry, new_dir, new_dentry);
-}
-
-static const struct file_operations cpuset_file_operations = {
- .read = cpuset_file_read,
- .write = cpuset_file_write,
- .llseek = generic_file_llseek,
- .open = cpuset_file_open,
```

```
- .release = cpuset_file_release,
-};
-
-static struct inode_operations cpuset_dir_inode_operations = {
- .lookup = simple_lookup,
- .mkdir = cpuset_mkdir,
- .rmdir = cpuset_rmdir,
- .rename = cpuset_rename,
-};
-
-static int cpuset_create_file(struct dentry *dentry, int mode)
-{
- struct inode *inode;
-
- if (!dentry)
-  return -ENOENT;
- if (dentry->d_inode)
-  return -EEXIST;
-
- inode = cpuset_new_inode(mode);
- if (!inode)
-  return -ENOMEM;
-
- if (S_ISDIR(mode)) {
-  inode->i_op = &cpuset_dir_inode_operations;
-  inode->i_fop = &simple_dir_operations;
-
-  /* start off with i_nlink == 2 (for "." entry) */
-  inc_nlink(inode);
- } else if (S_ISREG(mode)) {
-  inode->i_size = 0;
-  inode->i_fop = &cpuset_file_operations;
- }
-
- d_instantiate(dentry, inode);
- dget(dentry); /* Extra count - pin the dentry in core */
- return 0;
-}
-
-/*
- * cpuset_create_dir - create a directory for an object.
- * cs: the cpuset we create the directory for.
- *  It must have a valid ->parent field
- *  And we are going to fill its ->dentry field.
- * name: The name to give to the cpuset directory. Will be copied.
- * mode: mode to set on new directory.
- */
-
```

```
-static int cpuset_create_dir(struct cpuset *cs, const char *name, int mode)
-{
- struct dentry *dentry = NULL;
- struct dentry *parent;
- int error = 0;
-
- parent = cs->parent->dentry;
- dentry = cpuset_get_dentry(parent, name);
- if (IS_ERR(dentry))
-  return PTR_ERR(dentry);
- error = cpuset_create_file(dentry, S_IFDIR | mode);
- if (!error) {
-  dentry->d_fsdata = cs;
-  inc_nlink(parent->d_inode);
-  cs->dentry = dentry;
- }
- dput(dentry);
-
- return error;
-}
-
-static int cpuset_add_file(struct dentry *dir, const struct cftype *cft)
-{
- struct dentry *dentry;
- int error;
-
- mutex_lock(&dir->d_inode->i_mutex);
- dentry = cpuset_get_dentry(dir, cft->name);
- if (!IS_ERR(dentry)) {
-  error = cpuset_create_file(dentry, 0644 | S_IFREG);
-  if (!error)
-   dentry->d_fsdata = (void *)cft;
-  dput(dentry);
- } else
-  error = PTR_ERR(dentry);
- mutex_unlock(&dir->d_inode->i_mutex);
- return error;
-}
-
-/*
- * Stuff for reading the 'tasks' file.
- *
- * Reading this file can return large amounts of data if a cpuset has
- * *lots* of attached tasks. So it may need several calls to read(),
- * but we cannot guarantee that the information we produce is correct
- * unless we produce it entirely atomically.
- *
- * Upon tasks file open(), a struct ctr_struct is allocated, that
```

```
- * will have a pointer to an array (also allocated here).  The struct
- * ctr_struct * is stored in file->private_data.  Its resources will
- * be freed by release() when the file is closed.  The array is used
- * to sprintf the PIDs and then used by read().
- */
-
-/* cpusets_tasks_read array */
-
-struct ctr_struct {
- char *buf;
- int bufsz;
-};
-
-/*
- * Load into 'pidarray' up to 'npids' of the tasks using cpuset 'cs'.
- * Return actual number of pids loaded.  No need to task_lock(p)
- * when reading out p->cpuset, as we don't really care if it changes
- * on the next cycle, and we are not going to try to dereference it.
- */
-static int pid_array_load(pid_t *pidarray, int npids, struct cpuset *cs)
-{
- int n = 0;
- struct task_struct *g, *p;
-
- read_lock(&tasklist_lock);
-
- do_each_thread(g, p) {
-  if (p->cpuset == cs) {
-   pidarray[n++] = p->pid;
-   if (unlikely(n == npids))
-    goto array_full;
-  }
- } while_each_thread(g, p);
-
-array_full:
- read_unlock(&tasklist_lock);
- return n;
-}
-
-static int cmppid(const void *a, const void *b)
-{
- return *(pid_t *)a - *(pid_t *)b;
-}
-
-/*
- * Convert array 'a' of 'npids' pid_t's to a string of newline separated
- * decimal pids in 'buf'.  Don't write more than 'sz' chars, but return
- * count 'cnt' of how many chars would be written if buf were large enough.
```

```
- */
-static int pid_array_to_buf(char *buf, int sz, pid_t *a, int npids)
-{
- int cnt = 0;
- int i;
-
- for (i = 0; i < npids; i++)
-  cnt += snprintf(buf + cnt, max(sz - cnt, 0), "%d\n", a[i]);
- return cnt;
-}
-
-/*
- * Handle an open on 'tasks' file.  Prepare a buffer listing the
- * process id's of tasks currently attached to the cpuset being opened.
- *
- * Does not require any specific cpuset mutexes, and does not take any.
- */
-static int cpuset_tasks_open(struct inode *unused, struct file *file)
-{
- struct cpuset *cs = __d_cs(file->f_path.dentry->d_parent);
- struct ctr_struct *ctr;
- pid_t *pidarray;
- int npids;
- char c;
-
- if (!(file->f_mode & FMODE_READ))
-  return 0;
-
- ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
- if (!ctr)
-  goto err0;
-
- /*
-  * If cpuset gets more users after we read count, we won't have
-  * enough space - tough.  This race is indistinguishable to the
-  * caller from the case that the additional cpuset users didn't
-  * show up until sometime later on.
-  */
- npids = atomic_read(&cs->count);
- pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
- if (!pidarray)
-  goto err1;
-
- npids = pid_array_load(pidarray, npids, cs);
- sort(pidarray, npids, sizeof(pid_t), cmppid, NULL);
-
- /* Call pid_array_to_buf() twice, first just to get bufsz */
- ctr->bufsz = pid_array_to_buf(&c, sizeof(c), pidarray, npids) + 1;
```

```
- ctr->buf = kmalloc(ctr->bufsz, GFP_KERNEL);
- if (!ctr->buf)
-  goto err2;
- ctr->bufsz = pid_array_to_buf(ctr->buf, ctr->bufsz, pidarray, npids);
-
- kfree(pidarray);
- file->private_data = ctr;
- return 0;
-
-err2:
- kfree(pidarray);
-err1:
- kfree(ctr);
-err0:
- return -ENOMEM;
-}
-
-static ssize_t cpuset_tasks_read(struct file *file, char __user *buf,
-      size_t nbytes, loff_t *ppos)
-{
- struct ctr_struct *ctr = file->private_data;
-
- if (*ppos + nbytes > ctr->bufsz)
-  nbytes = ctr->bufsz - *ppos;
- if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
-  return -EFAULT;
- *ppos += nbytes;
- return nbytes;
-}
-
-static int cpuset_tasks_release(struct inode *unused_inode, struct file *file)
-{
- struct ctr_struct *ctr;
-
- if (file->f_mode & FMODE_READ) {
-  ctr = file->private_data;
-  kfree(ctr->buf);
-  kfree(ctr);
- }
- return 0;
-}
-
 /*
  * for the common functions, 'private' gives the type of file
  */

-static struct cftype cft_tasks = {
- .name = "tasks",
```

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

 static struct cftype cft_mems = {
  .name = "mems",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_MEMLIST,
 };

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

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

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

 static struct cftype cft_memory_pressure_enabled = {
  .name = "memory_pressure_enabled",
```

```
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_MEMORY_PRESSURE_ENABLED,
 };

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

 static struct cftype cft_spread_page = {
  .name = "memory_spread_page",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_SPREAD_PAGE,
 };

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

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

- if ((err = cpuset_add_file(cs_dentry, &cft_cpus)) < 0)
-  return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_mems)) < 0)
+ if ((err = container_add_file(cont, &cft_cpus)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_cpu_exclusive)) < 0)
+ if ((err = container_add_file(cont, &cft_mems)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_mem_exclusive)) < 0)
+ if ((err = container_add_file(cont, &cft_cpu_exclusive)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_notify_on_release)) < 0)
+ if ((err = container_add_file(cont, &cft_mem_exclusive)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_memory_migrate)) < 0)
+ if ((err = container_add_file(cont, &cft_memory_migrate)) < 0)
   return err;
```

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

@@ -1869,66 +1198,31 @@ static int cpuset_populate_dir(struct de
  * Must be called with the mutex on the parent inode held
  */

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

  cs = kmalloc(sizeof(*cs), GFP_KERNEL);
  if (!cs)
   return -ENOMEM;

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

```
  cs->parent = parent;
-
- mutex_lock(&callback_mutex);
- list_add(&cs->sibling, &cs->parent->children);
+ cont->cpuset = cs;
+ cs->container = cont;
  number_of_cpusets++;
- mutex_unlock(&callback_mutex);
-
- err = cpuset_create_dir(cs, name, mode);
- if (err < 0)
-  goto err;
-
- /*
-  * Release manage_mutex before cpuset_populate_dir() because it
-  * will down() this new directory's i_mutex and if we race with
-  * another mkdir, we might deadlock.
-  */
- mutex_unlock(&manage_mutex);
-
- err = cpuset_populate_dir(cs->dentry);
- /* If err < 0, we have a half-filled directory - oh well ;) */
  return 0;
-err:
- list_del(&cs->sibling);
- mutex_unlock(&manage_mutex);
- kfree(cs);
- return err;
-}
-
-static int cpuset_mkdir(struct inode *dir, struct dentry *dentry, int mode)
-{
- struct cpuset *c_parent = dentry->d_parent->d_fsdata;
-
- /* the vfs holds inode->i_mutex already */
- return cpuset_create(c_parent, dentry->d_name.name, mode | S_IFDIR);
 }

 /*
@@ -1942,49 +1236,16 @@ static int cpuset_mkdir(struct inode *di
  * nesting would risk an ABBA deadlock.
  */

-static int cpuset_rmdir(struct inode *unused_dir, struct dentry *dentry)
+void cpuset_destroy(struct container *cont)
 {
- struct cpuset *cs = dentry->d_fsdata;
```

```
- struct dentry *d;
- struct cpuset *parent;
- char *pathbuf = NULL;
-
- /* the vfs holds both inode->i_mutex already */
+ struct cpuset *cs = cont->cpuset;

- mutex_lock(&manage_mutex);
  cpuset_update_task_memory_state();
- if (atomic_read(&cs->count) > 0) {
-  mutex_unlock(&manage_mutex);
-  return -EBUSY;
- }
- if (!list_empty(&cs->children)) {
-  mutex_unlock(&manage_mutex);
-  return -EBUSY;
- }
  if (is_cpu_exclusive(cs)) {
   int retval = update_flag(CS_CPU_EXCLUSIVE, cs, "0");
-  if (retval < 0) {
-   mutex_unlock(&manage_mutex);
-   return retval;
-  }
+  BUG_ON(retval);
  }
- parent = cs->parent;
- mutex_lock(&callback_mutex);
- set_bit(CS_REMOVED, &cs->flags);
- list_del(&cs->sibling); /* delete my sibling from parent->children */
- spin_lock(&cs->dentry->d_lock);
- d = dget(cs->dentry);
- cs->dentry = NULL;
- spin_unlock(&d->d_lock);
- cpuset_d_remove_dir(d);
- dput(d);
  number_of_cpusets--;
- mutex_unlock(&callback_mutex);
- if (list_empty(&parent->children))
-  check_for_release(parent, &pathbuf);
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
- return 0;
 }

 /*
@@ -1995,10 +1256,10 @@ static int cpuset_rmdir(struct inode *un

 int __init cpuset_init_early(void)
```

```
 {
- struct task_struct *tsk = current;
-
- tsk->cpuset = &top_cpuset;
- tsk->cpuset->mems_generation = cpuset_mems_generation++;
+ struct container *cont = current->container;
+ cont->cpuset = &top_cpuset;
+ top_cpuset.container = cont;
+ cont->cpuset->mems_generation = cpuset_mems_generation++;
  return 0;
 }

@@ -2010,39 +1271,19 @@ int __init cpuset_init_early(void)

 int __init cpuset_init(void)
 {
- struct dentry *root;
- int err;
-
+ int err = 0;
  top_cpuset.cpus_allowed = CPU_MASK_ALL;
  top_cpuset.mems_allowed = NODE_MASK_ALL;

  fmeter_init(&top_cpuset.fmeter);
  top_cpuset.mems_generation = cpuset_mems_generation++;

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

```
- if (err == 0)
-  err = cpuset_add_file(root, &cft_memory_pressure_enabled);
-out:
- return err;
+ return 0;
 }

 /*
@@ -2068,10 +1309,12 @@ out:

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

  /* Each of our child cpusets mems must be online */
- list_for_each_entry(c, &cur->children, sibling) {
+ list_for_each_entry(cont, &cur->container->children, sibling) {
+  c = cont->cpuset;
   guarantee_online_cpus_mems_in_subtree(c);
   if (!cpus_empty(c->cpus_allowed))
    guarantee_online_cpus(c, &c->cpus_allowed);
@@ -2098,7 +1341,7 @@ static void guarantee_online_cpus_mems_i

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

  guarantee_online_cpus_mems_in_subtree(&top_cpuset);
@@ -2106,7 +1349,7 @@ static void common_cpu_mem_hotplug_unplu
  top_cpuset.mems_allowed = node_online_map;

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

 /*
@@ -2154,111 +1397,6 @@ void __init cpuset_init_smp(void)
 }

 /**
- * cpuset_fork - attach newly forked task to its parents cpuset.
- * @tsk: pointer to task_struct of forking parent process.
- *
- * Description: A task inherits its parent's cpuset at fork().
```

```
- *
- * A pointer to the shared cpuset was automatically copied in fork.c
- * by dup_task_struct().  However, we ignore that copy, since it was
- * not made under the protection of task_lock(), so might no longer be
- * a valid cpuset pointer.  attach_task() might have already changed
- * current->cpuset, allowing the previously referenced cpuset to
- * be removed and freed.  Instead, we task_lock(current) and copy
- * its present value of current->cpuset for our freshly forked child.
- *
- * At the point that cpuset_fork() is called, 'current' is the parent
- * task, and the passed argument 'child' points to the child task.
- **/
-
-void cpuset_fork(struct task_struct *child)
-{
- task_lock(current);
- child->cpuset = current->cpuset;
- atomic_inc(&child->cpuset->count);
- task_unlock(current);
-}
-
-/**
- * cpuset_exit - detach cpuset from exiting task
- * @tsk: pointer to task_struct of exiting process
- *
- * Description: Detach cpuset from @tsk and release it.
- *
- * Note that cpusets marked notify_on_release force every task in
- * them to take the global manage_mutex mutex when exiting.
- * This could impact scaling on very large systems.  Be reluctant to
- * use notify_on_release cpusets where very high task exit scaling
- * is required on large systems.
- *
- * Don't even think about derefencing 'cs' after the cpuset use count
- * goes to zero, except inside a critical section guarded by manage_mutex
- * or callback_mutex.   Otherwise a zero cpuset use count is a license to
- * any other task to nuke the cpuset immediately, via cpuset_rmdir().
- *
- * This routine has to take manage_mutex, not callback_mutex, because
- * it is holding that mutex while calling check_for_release(),
- * which calls kmalloc(), so can't be called holding callback_mutex().
- *
- * We don't need to task_lock() this reference to tsk->cpuset,
- * because tsk is already marked PF_EXITING, so attach_task() won't
- * mess with it, or task is a failed fork, never visible to attach_task.
- *
- * the_top_cpuset_hack:
- *
```

```
- *    Set the exiting tasks cpuset to the root cpuset (top_cpuset).
- *
- *    Don't leave a task unable to allocate memory, as that is an
- *    accident waiting to happen should someone add a callout in
- *    do_exit() after the cpuset_exit() call that might allocate.
- *    If a task tries to allocate memory with an invalid cpuset,
- *    it will oops in cpuset_update_task_memory_state().
- *
- *    We call cpuset_exit() while the task is still competent to
- *    handle notify_on_release(), then leave the task attached to
- *    the root cpuset (top_cpuset) for the remainder of its exit.
- *
- *    To do this properly, we would increment the reference count on
- *    top_cpuset, and near the very end of the kernel/exit.c do_exit()
- *    code we would add a second cpuset function call, to drop that
- *    reference.  This would just create an unnecessary hot spot on
- *    the top_cpuset reference count, to no avail.
- *
- *    Normally, holding a reference to a cpuset without bumping its
- *    count is unsafe.   The cpuset could go away, or someone could
- *    attach us to a different cpuset, decrementing the count on
- *    the first cpuset that we never incremented.  But in this case,
- *    top_cpuset isn't going away, and either task has PF_EXITING set,
- *    which wards off any attach_task() attempts, or task is a failed
- *    fork, never visible to attach_task.
- *
- *    Another way to do this would be to set the cpuset pointer
- *    to NULL here, and check in cpuset_update_task_memory_state()
- *    for a NULL pointer.  This hack avoids that NULL check, for no
- *    cost (other than this way too long comment ;).
- **/
-
-void cpuset_exit(struct task_struct *tsk)
-{
- struct cpuset *cs;
-
- cs = tsk->cpuset;
- tsk->cpuset = &top_cpuset; /* the_top_cpuset_hack - see above */
-
- if (notify_on_release(cs)) {
-  char *pathbuf = NULL;
-
-  mutex_lock(&manage_mutex);
-  if (atomic_dec_and_test(&cs->count))
-   check_for_release(cs, &pathbuf);
-  mutex_unlock(&manage_mutex);
-  cpuset_release_agent(pathbuf);
- } else {
```

```
-  atomic_dec(&cs->count);
- }
-}
-
-/**
  * cpuset_cpus_allowed - return cpus_allowed mask from a tasks cpuset.
  * @tsk: pointer to task_struct from which to obtain cpuset->cpus_allowed.
  *
@@ -2274,7 +1412,7 @@ cpumask_t cpuset_cpus_allowed(struct tas

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

@@ -2302,7 +1440,7 @@ nodemask_t cpuset_mems_allowed(struct ta

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

@@ -2423,7 +1561,7 @@ int __cpuset_zone_allowed_softwall(struc
  mutex_lock(&callback_mutex);

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

  allowed = node_isset(node, cs->mems_allowed);
@@ -2552,7 +1690,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock(current);
   goto done;
  }
- cs1 = nearest_exclusive_ancestor(current->cpuset);
+ cs1 = nearest_exclusive_ancestor(current->container->cpuset);
  task_unlock(current);

  task_lock((struct task_struct *)p);
@@ -2560,7 +1698,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock((struct task_struct *)p);
   goto done;
  }
```

```
- cs2 = nearest_exclusive_ancestor(p->cpuset);
+ cs2 = nearest_exclusive_ancestor(p->container->cpuset);
  task_unlock((struct task_struct *)p);

  overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
@@ -2599,11 +1737,12 @@ void __cpuset_memory_pressure_bump(void)
  struct cpuset *cs;

  task_lock(current);
- cs = current->cpuset;
+ cs = current->container->cpuset;
  fmeter_markevent(&cs->fmeter);
  task_unlock(current);
 }

+#ifdef CONFIG_PROC_PID_CPUSET
 /*
  * proc_cpuset_show()
  *  - Print tasks cpuset path into seq_file.
@@ -2634,15 +1773,15 @@ static int proc_cpuset_show(struct seq_f
   goto out_free;

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

- retval = cpuset_path(tsk->cpuset, buf, PAGE_SIZE);
+ retval = container_path(tsk->container, buf, PAGE_SIZE);
  if (retval < 0)
   goto out_unlock;
  seq_puts(m, buf);
  seq_putc(m, '\n');
 out_unlock:
- mutex_unlock(&manage_mutex);
+ container_unlock();
  put_task_struct(tsk);
 out_free:
  kfree(buf);
@@ -2662,6 +1801,7 @@ struct file_operations proc_cpuset_opera
  .llseek  = seq_lseek,
  .release = single_release,
 };
+#endif /* CONFIG_PROC_PID_CPUSET */

 /* Display task cpus_allowed, mems_allowed in /proc/<pid>/status file. */
 char *cpuset_task_status_allowed(struct task_struct *task, char *buffer)
Index: container-2.6.20-new/init/Kconfig
 ===================================================================
```

```
--- container-2.6.20-new.orig/init/Kconfig
+++ container-2.6.20-new/init/Kconfig
@@ -239,17 +239,12 @@ config IKCONFIG_PROC
    through /proc/config.gz.

 config CONTAINERS
- bool "Container support"
- help
-   This option will let you create and manage process containers,
-   which can be used to aggregate multiple processes, e.g. for
-   the purposes of resource tracking.
-
-   Say N if unsure
+ bool

 config CPUSETS
  bool "Cpuset support"
  depends on SMP
+ select CONTAINERS
  help
    This option will let you create and manage CPUSETs which
    allow dynamically partitioning a system into sets of CPUs and
@@ -278,6 +273,11 @@ config SYSFS_DEPRECATED
    If you are using a distro that was released in 2006 or later,
    it should be safe to say N here.

+config PROC_PID_CPUSET
+ bool "Include legacy /proc/<pid>/cpuset file"
+ depends on CPUSETS
+ default y
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
Index: container-2.6.20-new/include/linux/sched.h
 ===================================================================
--- container-2.6.20-new.orig/include/linux/sched.h
+++ container-2.6.20-new/include/linux/sched.h
@@ -744,7 +744,6 @@ extern unsigned int max_cache_size;

 struct io_context;   /* See blkdev.h */
 struct container;
-struct cpuset;
 #define NGROUPS_SMALL  32
 #define NGROUPS_PER_BLOCK ((int)(PAGE_SIZE / sizeof(gid_t)))
 struct group_info {
@@ -1026,7 +1025,6 @@ struct task_struct {
  short il_next;
```

```
 #endif
 #ifdef CONFIG_CPUSETS
- struct cpuset *cpuset;
  nodemask_t mems_allowed;
  int cpuset_mems_generation;
  int cpuset_mem_spread_rotor;
@@ -1471,7 +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 ->cpuset.
+ * 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/Documentation/cpusets.txt
 ===================================================================== =======
--- container-2.6.20-new.orig/Documentation/cpusets.txt
+++ container-2.6.20-new/Documentation/cpusets.txt
@@ -7,6 +7,7 @@ Written by Simon.Derr@bull.net
 Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
 Modified by Paul Jackson <pj@sgi.com>
 Modified by Christoph Lameter <clameter@sgi.com>
+Modified by Paul Menage <menage@google.com>

 CONTENTS:
 =========
@@ -16,10 +17,9 @@ CONTENTS:
   1.2 Why are cpusets needed ?
   1.3 How are cpusets implemented ?
   1.4 What are exclusive cpusets ?
-  1.5 What does notify_on_release do ?
-  1.6 What is memory_pressure ?
-  1.7 What is memory spread ?
-  1.8 How do I use cpusets ?
+  1.5 What is memory_pressure ?
+  1.6 What is memory spread ?
+  1.7 How do I use cpusets ?
 2. Usage Examples and Syntax
   2.1 Basic Usage
   2.2 Adding/removing cpus
@@ -43,18 +43,19 @@ hierarchy visible in a virtual file syst
 hooks, beyond what is already present, required to manage dynamic
 job placement on large systems.

-Each task has a pointer to a cpuset.  Multiple tasks may reference
-the same cpuset.  Requests by a task, using the sched_setaffinity(2)
-system call to include CPUs in its CPU affinity mask, and using the
```

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

-User level code may create and destroy cpusets by name in the cpuset
+Requests by a task, using the sched_setaffinity(2) system call to
+include CPUs in its CPU affinity mask, and using the mbind(2) and
+set_mempolicy(2) system calls to include Memory Nodes in its memory
+policy, are both filtered through that tasks cpuset, filtering out any
+CPUs or Memory Nodes not in that cpuset.  The scheduler will not
+schedule a task on a CPU that is not allowed in its cpus_allowed
+vector, and the kernel page allocator will not allocate a page on a
+node that is not allowed in the requesting tasks mems_allowed vector.
+
+User level code may create and destroy cpusets by name in the container
 virtual file system, manage the attributes and permissions of these
 cpusets and which CPUs and Memory Nodes are assigned to each cpuset,
 specify and query to which cpuset a task is assigned, and list the
@@ -117,7 +118,7 @@ Cpusets extends these two mechanisms as
  - Cpusets are sets of allowed CPUs and Memory Nodes, known to the
    kernel.
  - Each task in the system is attached to a cpuset, via a pointer
-   in the task structure to a reference counted cpuset structure.
+   in the task structure to a reference counted container structure.
  - Calls to sched_setaffinity are filtered to just those CPUs
    allowed in that tasks cpuset.
  - Calls to mbind and set_mempolicy are filtered to just
@@ -152,15 +153,10 @@ into the rest of the kernel, none in per
  - in page_alloc.c, to restrict memory to allowed nodes.
  - in vmscan.c, to restrict page recovery to the current cpuset.

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

+new system calls are added for cpusets - all support for querying and
+modifying cpusets is via this cpuset file system.

 The /proc/<pid>/status file for each task has two added lines,
 displaying the tasks cpus_allowed (on which CPUs it may be scheduled)
@@ -170,16 +166,15 @@ in the format seen in the following exam
   Cpus_allowed:   ffffffff,ffffffff,ffffffff,ffffffff
   Mems_allowed:   ffffffff,ffffffff

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

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

 In addition, the root cpuset only has the following file:
@@ -253,21 +248,7 @@ such as requests from interrupt handlers
 outside even a mem_exclusive cpuset.


-1.5 What does notify_on_release do ?
----------
-
-If the notify_on_release flag is enabled (1) in a cpuset, then whenever
-the last task in the cpuset leaves (exits or attaches to some other
-cpuset) and the last child cpuset of that cpuset is removed, then
-the kernel runs the command /sbin/cpuset_release_agent, supplying the
-pathname (relative to the mount point of the cpuset file system) of the
-abandoned cpuset.  This enables automatic removal of abandoned cpusets.
-The default value of notify_on_release in the root cpuset at system
-boot is disabled (0).  The default value of other cpusets at creation
-is the current value of their parents notify_on_release setting.
-
-
-1.6 What is memory_pressure ?
+1.5 What is memory_pressure ?
 ---------
 The memory_pressure of a cpuset provides a simple per-cpuset metric
 of the rate that the tasks in a cpuset are attempting to free up in

@@ -324,7 +305,7 @@ the tasks in the cpuset, in units of rec
 times 1000.


-1.7 What is memory spread ?
+1.6 What is memory spread ?
 ---------------------------
 There are two boolean flag files per cpuset that control where the
 kernel allocates pages for the file system buffers and related in
@@ -395,7 +376,7 @@ data set, the memory allocation across t
 can become very uneven.


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

 In order to minimize the impact of cpusets on critical kernel
@@ -485,7 +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 cpuset none /dev/cpuset
+ 2) mount -t container none /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.
@@ -497,7 +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 cpuset none /dev/cpuset
+  mount -t container none /dev/cpuset
    cd /dev/cpuset
    mkdir Charlie
    cd Charlie
@@ -507,7 +488,7 @@ and then start a subshell 'sh' in that c
    # The subshell 'sh' is now running in cpuset Charlie
    # The next line should display '/Charlie'
-  cat /proc/self/cpuset
+  cat /proc/self/container

 In the future, a C library interface to cpusets will likely be
 available.  For now, the only way to query or modify cpusets is
@@ -529,7 +510,7 @@ Creating, modifying, using the cpusets c
 virtual filesystem.

```
 To mount it, type:
-# mount -t cpuset none /dev/cpuset
+# mount -t container none /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
Index: container-2.6.20-new/fs/super.c
 ===================================================== =======
--- container-2.6.20-new.orig/fs/super.c
+++ container-2.6.20-new/fs/super.c
@@ -39,11 +39,6 @@
 #include <linux/mutex.h>
 #include <asm/uaccess.h>

-
-void get_filesystem(struct file_system_type *fs);
-void put_filesystem(struct file_system_type *fs);
-struct file_system_type *get_fs_type(const char *name);
-
 LIST_HEAD(super_blocks);
 DEFINE_SPINLOCK(sb_lock);

Index: container-2.6.20-new/include/linux/fs.h
 ===================================================== =======
--- container-2.6.20-new.orig/include/linux/fs.h
+++ container-2.6.20-new/include/linux/fs.h
@@ -1841,6 +1841,8 @@ extern int vfs_fstat(unsigned int, struc

 extern int vfs_ioctl(struct file *, unsigned int, unsigned int, unsigned long);

+extern void get_filesystem(struct file_system_type *fs);
+extern void put_filesystem(struct file_system_type *fs);
 extern struct file_system_type *get_fs_type(const char *name);
 extern struct super_block *get_super(struct block_device *);
 extern struct super_block *user_get_super(dev_t);
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
@@ -152,7 +152,7 @@ extern void mpol_fix_fork_child_flag(str

 #ifdef CONFIG_CPUSETS
 #define current_cpuset_is_being_rebound() \
-    (cpuset_being_rebound == current->cpuset)
+ (cpuset_being_rebound == current->container->cpuset)
 #else
 #define current_cpuset_is_being_rebound() 0
 #endif
```

```
Index: container-2.6.20-new/fs/proc/base.c
 ===================================================================
--- container-2.6.20-new.orig/fs/proc/base.c
+++ container-2.6.20-new/fs/proc/base.c
@@ -1868,7 +1868,7 @@ static struct pid_entry tgid_base_stuff[
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat",  S_IRUGO, pid_schedstat),
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",     S_IRUGO, cpuset),
 #endif
 #ifdef CONFIG_CONTAINERS
@@ -2152,7 +2152,7 @@ static struct pid_entry tid_base_stuff[]
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat", S_IRUGO, pid_schedstat),
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",    S_IRUGO, cpuset),
 #endif
 #ifdef CONFIG_CONTAINERS

--
```

---

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

+static int rebind_subsystems(struct containerfs_root *root,
+         unsigned long final_bits)
+{
+ unsigned long added_bits, removed_bits;
+ struct container *cont = &root->top_container;
+ int i;
+
+ removed_bits = root->subsys_bits & ~final_bits;
+ added_bits = final_bits & ~root->subsys_bits;
+ /* Check that any added subsystems are currently free */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  unsigned long long bit = 1ull << i;
+  struct container_subsys *ss = subsys[i];
+  if (!(bit & added_bits))
+   continue;
+  if (ss->root != &rootnode) {
+   /* Subsystem isn't free */
+   return -EBUSY;
+  }
+ }
+
```

```
+ /* Currently we don't handle adding/removing subsystems when
+  * any subcontainers exist. This is theoretically supportable
+  * but involves complex erro r handling, so it's being left until
+  * later */
+ if (!list_empty(&cont->children)) {
+  return -EBUSY;
+ }
+
+ /* Process each subsystem */
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  struct container_subsys *ss = subsys[i];
+  unsigned long bit = 1UL << i;
+  if (bit & added_bits) {
+   /* We're binding this subsystem to this hierarchy */
+   BUG_ON(cont->subsys[i]);
+   BUG_ON(!dummytop->subsys[i]);
+   BUG_ON(dummytop->subsys[i]->container != dummytop);
+   cont->subsys[i] = dummytop->subsys[i];
+   cont->subsys[i]->container = cont;
+   list_add(&ss->sibling, &root->subsys_list);
+   rcu_assign_pointer(ss->root, root);
+   if (ss->bind)
+    ss->bind(ss, cont);
+
+  } else if (bit & removed_bits) {
+   /* We're removing this subsystem */
+   BUG_ON(cont->subsys[i] != dummytop->subsys[i]);
+   BUG_ON(cont->subsys[i]->container != cont);
+   if (ss->bind)
+    ss->bind(ss, dummytop);
+   dummytop->subsys[i]->container = dummytop;
+   cont->subsys[i] = NULL;
+   rcu_assign_pointer(subsys[i]->root, &rootnode);
+   list_del(&ss->sibling);
+  } else if (bit & final_bits) {
+   /* Subsystem state should already exist */
+   BUG_ON(!cont->subsys[i]);
+  } else {
+   /* Subsystem state shouldn't exist */
+   BUG_ON(cont->subsys[i]);
+  }
+ }
+ root->subsys_bits = final_bits;
+ synchronize_rcu();
+
+ return 0;
+}
+
```

```
+/*
+ * Release the last use of a hierarchy.  Will never be called when
+ * there are active subcontainers since each subcontainer bumps the
+ * value of sb->s_active.
+ */
+
+static void container_put_super(struct super_block *sb) {
+
+ struct containerfs_root *root = sb->s_fs_info;
+ struct container *cont = &root->top_container;
+ int ret;
+
+ root->sb = NULL;
+ sb->s_fs_info = NULL;
+
+ mutex_lock(&container_mutex);
+
+ BUG_ON(root->number_of_containers != 1);
+ BUG_ON(!list_empty(&cont->children));
+ BUG_ON(!list_empty(&cont->sibling));
+ BUG_ON(!root->subsys_bits);
+
+ /* Rebind all subsystems back to the default hierarchy */
+ ret = rebind_subsystems(root, 0);
+ BUG_ON(ret);
+
+ kfree(root);
+ mutex_unlock(&container_mutex);
+}
+
+static int container_show_options(struct seq_file *seq, struct vfsmount *vfs)
+{
+ struct containerfs_root *root = vfs->mnt_sb->s_fs_info;
+ struct container_subsys *ss;
+ for_each_subsys(root, ss) {
+  seq_printf(seq, ",%s", ss->name);
+ }
+ return 0;
+}
+
+/* Convert a hierarchy specifier into a bitmask. LL=container_mutex */
+static int parse_containerfs_options(char *opts, unsigned long *bits)
+{
+ char *token, *o = opts ?: "all";
+
+ *bits = 0;
+
+ while ((token = strsep(&o, ",")) != NULL) {
```

```
+  if (!*token)
+   return -EINVAL;
+  if (!strcmp(token, "all")) {
+   *bits = (1 << CONTAINER_SUBSYS_COUNT) - 1;
+  } else {
+   struct container_subsys *ss;
+   int i;
+   for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+    ss = subsys[i];
+    if (!strcmp(token, ss->name)) {
+     *bits |= 1 << i;
+     break;
+    }
+   }
+   if (i == CONTAINER_SUBSYS_COUNT)
+    return -ENOENT;
+  }
+ }
+
+ /* We can't have an empty hierarchy */
+ if (!*bits)
+  return -EINVAL;
+
+ return 0;
+}
+
+static int container_remount(struct super_block *sb, int *flags, char *data)
+{
+ int ret = 0;
+ unsigned long subsys_bits;
+ struct containerfs_root *root = sb->s_fs_info;
+ struct container *cont = &root->top_container;
+
+ mutex_lock(&cont->dentry->d_inode->i_mutex);
+ mutex_lock(&container_mutex);
+
+ /* See what subsystems are wanted */
+ ret = parse_containerfs_options(data, &subsys_bits);
+ if (ret)
+  goto out_unlock;
+
+ ret = rebind_subsystems(root, subsys_bits);
+
+ /* (re)populate subsystem files */
+ if (!ret)
+  container_populate_dir(cont);
+
+ out_unlock:
```

```
+ mutex_unlock(&container_mutex);
+ mutex_unlock(&cont->dentry->d_inode->i_mutex);
+ return ret;
+}
+
 static struct super_operations container_ops = {
  .statfs = simple_statfs,
  .drop_inode = generic_delete_inode,
+ .put_super = container_put_super,
+ .show_options = container_show_options,
+ .remount_fs = container_remount,
 };

-static int container_fill_super(struct super_block *sb, void *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(oldcg);
  return 0;
 }

-
 /*
- * Attack 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
  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 = strnlen(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
  .llseek  = seq_lseek,
  .release = single_release,
 };
+
+/* Display information about each subsystem and each hierarchy */
+static int proc_containerstats_show(struct seq_file *m, void *v)
+{
+ int i;
+ struct containerfs_root *root;
+ mutex_lock(&container_mutex);
+ seq_puts(m, "Hierarchies:\n");
+ for_each_root(root) {
+  struct container_subsys *ss;
+  int first = 1;
+  seq_printf(m, "%p: bits=%lx containers=%d (", root,
+      root->subsys_bits, root->number_of_containers);
+  for_each_subsys(root, ss) {
+   seq_printf(m, "%s%s", first ? "" : ", ", ss->name);
+   first = false;
+  }
+  seq_putc(m, ')');
+  if (root->sb) {
+   seq_printf(m, " s_active=%d",
+       atomic_read(&root->sb->s_active));
+  }
+  seq_putc(m, '\n');
+ }
+ seq_puts(m, "Subsystems:\n");
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  struct container_subsys *ss = subsys[i];
+  seq_printf(m, "%d: name=%s hierarchy=%p\n",
+      i, ss->name, ss->root);
+ }
+ 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_cpusets_mem_gen = top_cpuset.mems_generation;
  } else {
   rcu_read_lock();
-  cs = rcu_dereference(tsk->container->cpuset);
-  my_cpusets_mem_gen = cs->mems_generation;
+  my_cpusets_mem_gen = task_cs(current)->mems_generation;
   rcu_read_unlock();
  }

  if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
   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 hierachies 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
+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.
+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:
+
+       CPU :          Top cpuset
+                      /       \
+              CPUSet1         CPUSet2
+                |               |
+              (Profs)         (Students)
+
+              In addition (system tasks) are attached to topcpuset (so
+              that they can run anywhere) with a limit of 20%
+
+       Memory : Professors (50%), students (30%), system (20%)
+
+       Disk : Prof (50%), students (30%), system (20%)
+
+       Network : WWW browsing (20%), Network File System (60%), others (20%)
+                         / \
+                 Prof (15%) students (5%)
+
+Browsers like firefox/lynx go into the WWW network class, while (k)nfsd go
+into NFS network class.
+
+At the same time firefox/lynx will share an appropriate CPU/Memory class

+depending on who launched it (prof/student).
+
+With the ability to classify tasks differently for different resources
+(by putting those resource subsystems in different hierarchies) then
+the admin can easily set up a script which receives exec notifications
+and depending on who is launching the browser he can
+
+       # echo browser_pid > /mnt/<restype>/<userclass>/tasks
+
+With only a single hierarchy, he now would potentially have to create
+a separate container for every browser launched and associate it with
+approp network and other resource class.  This may lead to
+proliferation of such containers.
+
+Also lets say that the administrator would like to give enhanced network
+access temporarily to a student's browser (since it is night and the user
+wants to do online gaming :)  OR give one of the students simulation
+apps enhanced CPU power,
+
+With ability to write pids directly to resource classes, its just a
+matter of :
+
+       # echo pid > /mnt/network/<new_class>/tasks
+       (after some time)
+       # echo pid > /mnt/network/<orig_class>/tasks
+
+Without this ability, he would have to split the container into
+multiple separate ones and then associate the new containers with the
+new resource classes.
+
+

 1.3 How are containers implemented ?
 ---------------------------------

 Containers extends the kernel as follows:

- - Each task in the system 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 if 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 deregestering 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_subsy *ss, struct task_struct *task)
+LL=callback_mutex, maybe read_lock(tasklist_lock)
+
+Called when a task is forked into a container. Also called during
+registration for all existing tasks.
+
+void exit(struct container_subsys *ss, struct task_struct *task)
+LL=callback_mutex
+
+Called during task exit
+
+int populate(struct container_subsys *ss, struct container *cont)
+LL=none
+
+Called after creation of a container to allow a subsystem to populate
+the container directory with file entries.  The subsystem should make
+calls to container_add_file() with objects of type cftype (see
+include/linux/container.h for details).  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. */
  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:
  mpol_free(p->mempolicy);
 bad_fork_cleanup_container:
 #endif
- container_exit(p);
+ container_exit(p, container_callbacks_done);
 bad_fork_cleanup_delays_binfmt:
  delayacct_tsk_free(p);
  if (p->binfmt)
Index: container-2.6.20-new/kernel/exit.c
===================================================================
--- container-2.6.20-new.orig/kernel/exit.c
+++ container-2.6.20-new/kernel/exit.c
@@ -926,7 +926,7 @@ fastcall NORET_TYPE void do_exit(long co
  __exit_files(tsk);
  __exit_fs(tsk);
  exit_thread();
- container_exit(tsk);
+ container_exit(tsk, 1);
  exit_keys(tsk);

  if (group_dead && tsk->signal->leader)
Index: container-2.6.20-new/include/linux/container_subsys.h
```

```
==========================================================================
--- /dev/null
+++ container-2.6.20-new/include/linux/container_subsys.h
@@ -0,0 +1,14 @@
+/* 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
+ */
+
+/* */
+
+#ifdef CONFIG_CPUSETS
+SUBSYS(cpuset)
+#endif
+
+/* */
+
+/* */

--
```

---

Subject: [PATCH 4/7] Containers (V8): Simple CPU accounting container subsystem
Posted by Paul Menage on Fri, 06 Apr 2007 23:32:25 GMT
View Forum Message <> Reply to Message

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

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

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

```
---
 include/linux/container_subsys.h |    6 +
 include/linux/cpu_acct.h         |   14 ++
 init/Kconfig                     |    7 +
 kernel/Makefile                  |    1
 kernel/cpu_acct.c                |  204 ++++++++++++++++++++++++++++++++++++++++
 kernel/sched.c                   |   14 ++
 6 files changed, 243 insertions(+), 3 deletions(-)

Index: container-2.6.20-new/include/linux/cpu_acct.h
 ========================================================== =======
--- /dev/null
```

```
+++ container-2.6.20-new/include/linux/cpu_acct.h
@@ -0,0 +1,14 @@
+
+#ifndef _LINUX_CPU_ACCT_H
+#define _LINUX_CPU_ACCT_H
+
+#include <linux/container.h>
+#include <asm/cputime.h>
+
+#ifdef CONFIG_CONTAINER_CPUACCT
+extern void cpuacct_charge(struct task_struct *, cputime_t cputime);
+#else
+static void inline cpuacct_charge(struct task_struct *p, cputime_t cputime) {}
+#endif
+
+#endif
Index: container-2.6.20-new/init/Kconfig
 ===================================================================
--- container-2.6.20-new.orig/init/Kconfig
+++ container-2.6.20-new/init/Kconfig
@@ -278,6 +278,13 @@ config PROC_PID_CPUSET
  depends on CPUSETS
  default y

+config CONTAINER_CPUACCT
+ bool "Simple CPU accounting container subsystem"
+ select CONTAINERS
+ help
+   Provides a simple Resource Controller for monitoring the
+   total CPU consumed by the tasks in a container
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
Index: container-2.6.20-new/kernel/cpu_acct.c
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/cpu_acct.c
@@ -0,0 +1,204 @@
+/*
+ * kernel/cpu_acct.c - CPU accounting container subsystem
+ *
+ * Copyright (C) Google Inc, 2006
+ *
+ * Developed by Paul Menage (menage@google.com) and Balbir Singh
+ * (balbir@in.ibm.com)
+ *
+ */
```

```
+
+/*
+ * Container subsystem for reporting total CPU usage of tasks in a
+ * container, along with percentage load over a time interval
+ */
+
+#include <linux/module.h>
+#include <linux/container.h>
+#include <linux/fs.h>
+#include <asm/div64.h>
+
+struct cpuacct {
+ struct container_subsys_state css;
+ spinlock_t lock;
+ /* total time used by this class */
+ cputime64_t time;
+
+ /* time when next load calculation occurs */
+ u64 next_interval_check;
+
+ /* time used in current period */
+ cputime64_t current_interval_time;
+
+ /* time used in last period */
+ cputime64_t last_interval_time;
+};
+
+struct container_subsys cpuacct_subsys;
+
+static inline struct cpuacct *container_ca(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, cpuacct_subsys_id),
+       struct cpuacct, css);
+}
+
+static inline struct cpuacct *task_ca(struct task_struct *task)
+{
+ return container_ca(task_container(task, cpuacct_subsys_id));
+}
+
+#define INTERVAL (HZ * 10)
+
+static inline u64 next_interval_boundary(u64 now) {
+ /* calculate the next interval boundary beyond the
+  * current time */
+ do_div(now, INTERVAL);
+ return (now + 1) * INTERVAL;
+}
```

```
+
+static int cpuacct_create(struct container_subsys *ss, struct container *cont)
+{
+ struct cpuacct *ca = kzalloc(sizeof(*ca), GFP_KERNEL);
+ if (!ca)
+  return -ENOMEM;
+ spin_lock_init(&ca->lock);
+ ca->next_interval_check = next_interval_boundary(get_jiffies_64());
+ cont->subsys[cpuacct_subsys.subsys_id] = &ca->css;
+ return 0;
+}
+
+static void cpuacct_destroy(struct container_subsys *ss,
+       struct container *cont)
+{
+ kfree(container_ca(cont));
+}
+
+/* Lazily update the load calculation if necessary. Called with ca locked */
+static void cpuusage_update(struct cpuacct *ca)
+{
+ u64 now = get_jiffies_64();
+ /* If we're not due for an update, return */
+ if (ca->next_interval_check > now)
+  return;
+
+ if (ca->next_interval_check <= (now - INTERVAL)) {
+  /* If it's been more than an interval since the last
+   * check, then catch up - the last interval must have
+   * been zero load */
+  ca->last_interval_time = 0;
+  ca->next_interval_check = next_interval_boundary(now);
+ } else {
+  /* If a steal takes the last interval time negative,
+   * then we just ignore it */
+  if ((s64)ca->current_interval_time > 0) {
+   ca->last_interval_time = ca->current_interval_time;
+  } else {
+   ca->last_interval_time = 0;
+  }
+  ca->next_interval_check += INTERVAL;
+ }
+ ca->current_interval_time = 0;
+}
+
+static ssize_t cpuusage_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
```

```
+        char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct cpuacct *ca = container_ca(cont);
+ u64 time;
+ char usagebuf[64];
+ char *s = usagebuf;
+
+ spin_lock_irq(&ca->lock);
+ cpuusage_update(ca);
+ time = cputime64_to_jiffies64(ca->time);
+ spin_unlock_irq(&ca->lock);
+
+ /* Convert 64-bit jiffies to seconds */
+ time *= 1000;
+ do_div(time, HZ);
+ s += sprintf(s, "%llu", (unsigned long long) time);
+
+ return simple_read_from_buffer(buf, nbytes, ppos, usagebuf, s - usagebuf);
+}
+
+static ssize_t load_read(struct container *cont,
+    struct cftype *cft,
+    struct file *file,
+    char __user *buf,
+    size_t nbytes, loff_t *ppos)
+{
+ struct cpuacct *ca = container_ca(cont);
+ u64 time;
+ char usagebuf[64];
+ char *s = usagebuf;
+
+ /* Find the time used in the previous interval */
+ spin_lock_irq(&ca->lock);
+ cpuusage_update(ca);
+ time = cputime64_to_jiffies64(ca->last_interval_time);
+ spin_unlock_irq(&ca->lock);
+
+ /* Convert time to a percentage, to give the load in the
+  * previous period */
+ time *= 100;
+ do_div(time, INTERVAL);
+
+ s += sprintf(s, "%llu", (unsigned long long) time);
+
+ return simple_read_from_buffer(buf, nbytes, ppos, usagebuf, s - usagebuf);
+}
+static struct cftype cft_usage = {
```

```
+ .name = "cpuacct.usage",
+ .read = cpuusage_read,
+};
+
+static struct cftype cft_load = {
+ .name = "cpuacct.load",
+ .read = load_read,
+};
+
+static int cpuacct_populate(struct container_subsys *ss,
+       struct container *cont)
+{
+ int err;
+
+ if ((err = container_add_file(cont, &cft_usage)))
+  return err;
+ if ((err = container_add_file(cont, &cft_load)))
+  return err;
+
+ return 0;
+}
+
+
+void cpuacct_charge(struct task_struct *task, cputime_t cputime)
+{
+
+ struct cpuacct *ca;
+ unsigned long flags;
+
+ if (!cpuacct_subsys.active)
+  return;
+ rcu_read_lock();
+ ca = task_ca(task);
+ if (ca) {
+  spin_lock_irqsave(&ca->lock, flags);
+  cpuusage_update(ca);
+  ca->time = cputime64_add(ca->time, cputime);
+  ca->current_interval_time =
+   cputime64_add(ca->current_interval_time, cputime);
+  spin_unlock_irqrestore(&ca->lock, flags);
+ }
+ rcu_read_unlock();
+}
+
+struct container_subsys cpuacct_subsys = {
+ .name = "cpuacct",
+ .create = cpuacct_create,
+ .destroy = cpuacct_destroy,
```

```
+ .populate = cpuacct_populate,
+ .subsys_id = cpuacct_subsys_id,
+};
Index: container-2.6.20-new/kernel/Makefile
 ===================================================================
--- container-2.6.20-new.orig/kernel/Makefile
+++ container-2.6.20-new/kernel/Makefile
@@ -37,6 +37,7 @@ obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
+obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
 obj-$(CONFIG_AUDIT) += audit.o auditfilter.o
Index: container-2.6.20-new/kernel/sched.c
 ===================================================================
--- container-2.6.20-new.orig/kernel/sched.c
+++ container-2.6.20-new/kernel/sched.c
@@ -52,6 +52,7 @@
 #include <linux/tsacct_kern.h>
 #include <linux/kprobes.h>
 #include <linux/delayacct.h>
+#include <linux/cpu_acct.h>
 #include <asm/tlb.h>

 #include <asm/unistd.h>
@@ -3066,9 +3067,13 @@ void account_user_time(struct task_struc
 {
  struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
  cputime64_t tmp;
+ struct rq *rq = this_rq();

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

+ if (p != rq->idle)
+  cpuacct_charge(p, cputime);
+
  /* Add user time to cpustat. */
  tmp = cputime_to_cputime64(cputime);
  if (TASK_NICE(p) > 0)
@@ -3098,9 +3103,10 @@ void account_system_time(struct task_str
   cpustat->irq = cputime64_add(cpustat->irq, tmp);
  else if (softirq_count())
   cpustat->softirq = cputime64_add(cpustat->softirq, tmp);
- else if (p != rq->idle)
+ else if (p != rq->idle) {
   cpustat->system = cputime64_add(cpustat->system, tmp);
```

```
- else if (atomic_read(&rq->nr_iowait) > 0)
+  cpuacct_charge(p, cputime);
+ } else if (atomic_read(&rq->nr_iowait) > 0)
   cpustat->iowait = cputime64_add(cpustat->iowait, tmp);
  else
   cpustat->idle = cputime64_add(cpustat->idle, tmp);
@@ -3125,8 +3131,10 @@ void account_steal_time(struct task_stru
    cpustat->iowait = cputime64_add(cpustat->iowait, tmp);
   else
    cpustat->idle = cputime64_add(cpustat->idle, tmp);
- } else
+ } else {
   cpustat->steal = cputime64_add(cpustat->steal, tmp);
+  cpuacct_charge(p, -tmp);
+ }
 }

 static void task_running_tick(struct rq *rq, struct task_struct *p)
Index: container-2.6.20-new/include/linux/container_subsys.h
 ===================================================== =======
--- container-2.6.20-new.orig/include/linux/container_subsys.h
+++ container-2.6.20-new/include/linux/container_subsys.h
@@ -11,4 +11,10 @@ SUBSYS(cpuset)

 /* */

+#ifdef CONFIG_CONTAINER_CPUACCT
+SUBSYS(cpuacct)
+#endif
+
+/* */
+
 /* */

--
```

---

## Subject: [PATCH 5/7] Containers (V8): Resource Groups over generic containers
Posted by Paul Menage on Fri, 06 Apr 2007 23:32:26 GMT
View Forum Message <> Reply to Message

This patch provides the RG core and numtasks controller as container subsystems, intended as an example of how to implement a more complex resource control system over generic process containers. The changes to the core involve primarily removing the group management, task membership and configfs support and adding interface layers to talk to the generic container layer instead.

Each resource controller becomes an independent container subsystem;
the RG core is essentially a library that the resource controllers can
use to provide the RG API to userspace. Rather than a single shares
and stats file in each group, there's a <controller>_shares and
a <controller>_stats file, each linked to the appropriate resource
controller.

```
 include/linux/container_subsys.h |    6
 include/linux/moduleparam.h      |   12 -
 include/linux/numtasks.h         |   28 ++
 include/linux/res_group.h        |   86 +++++++
 include/linux/res_group_rc.h     |  125 ++++++++++
 init/Kconfig                     |   22 +
 kernel/Makefile                  |    1
 kernel/fork.c                    |    7
 kernel/res_group/Makefile        |    2
 kernel/res_group/local.h         |   38 +++
 kernel/res_group/numtasks.c      |  451 ++++++++++++++++++++++++++++++++++++++++
 kernel/res_group/res_group.c     |  135 ++++++++++
 kernel/res_group/rgcs.c          |  302 ++++++++++++++++++++++++++
 kernel/res_group/shares.c        |  228 ++++++++++++++++++++
 14 files changed, 1439 insertions(+), 4 deletions(-)

Index: container-2.6.20-new/include/linux/moduleparam.h
 ===================================================================== =======
--- container-2.6.20-new.orig/include/linux/moduleparam.h
+++ container-2.6.20-new/include/linux/moduleparam.h
@@ -78,11 +78,17 @@ struct kparam_array
 /* Helper functions: type is byte, short, ushort, int, uint, long,
    ulong, charp, bool or invbool, or XXX if you define param_get_XXX,
    param_set_XXX and param_check_XXX. */
-#define module_param_named(name, value, type, perm)      \
- param_check_##type(name, &(value));       \
- module_param_call(name, param_set_##type, param_get_##type, &value, perm); \
+#define module_param_named_call(name, value, type, set, perm)  \
+ param_check_##type(name, &(value));    \
+ module_param_call(name, set, param_get_##type, &(value), perm); \
   __MODULE_PARM_TYPE(name, #type)

+#define module_param_named(name, value, type, perm)      \
+ module_param_named_call(name, value, type, param_set_##type, perm)
+
+#define module_param_set_call(name, type, setfn, perm) \
+ module_param_named_call(name, name, type, setfn, perm)
+
 #define module_param(name, type, perm)    \
  module_param_named(name, name, type, perm)
```

Index: container-2.6.20-new/include/linux/numtasks.h
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/include/linux/numtasks.h
@@ -0,0 +1,28 @@
+/* numtasks.h - No. of tasks resource controller for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *
+ * Provides No. of tasks resource controller for Resource Groups
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+#ifndef _LINUX_NUMTASKS_H
+#define _LINUX_NUMTASKS_H
+
+#ifdef CONFIG_RES_GROUPS_NUMTASKS
+#include <linux/res_group_rc.h>
+
+extern int numtasks_allow_fork(struct task_struct *);
+
+#else /* CONFIG_RES_GROUPS_NUMTASKS */
+
+#define numtasks_allow_fork(task) (0)
+
+#endif /* CONFIG_RES_GROUPS_NUMTASKS */
+#endif /* _LINUX_NUMTASKS_H */
Index: container-2.6.20-new/include/linux/res_group.h
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/include/linux/res_group.h
@@ -0,0 +1,86 @@
+/*
+ *  res_group.h - Header file to be used by Resource Groups
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003, 2004
+ *  (C) Shailabh Nagar,  IBM Corp. 2003, 2004
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *
+ * Provides data structures, macros and kernel APIs
+ *
+ * More details at http://ckrm.sf.net

```
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+#ifndef _LINUX_RES_GROUP_H
+#define _LINUX_RES_GROUP_H
+
+#ifdef CONFIG_RES_GROUPS
+#include <linux/spinlock.h>
+#include <linux/list.h>
+#include <linux/kref.h>
+#include <linux/container.h>
+
+#define SHARE_UNCHANGED (-1) /* implicitly specified by userspace,
+      * never stored in a resource group'
+      * shares struct; never displayed */
+#define SHARE_UNSUPPORTED (-2) /* If the resource controller doesn't
+      * support user changing a shares value
+      * it sets the corresponding share
+      * value to UNSUPPORTED when it returns
+      * the newly allocated shares data
+      * structure */
+#define SHARE_DONT_CARE (-3)
+
+#define SHARE_DEFAULT_DIVISOR  (100)
+
+#define MAX_DEPTH 5 /* max depth of hierarchy supported */
+
+#define NO_RES_GROUP  NULL
+#define NO_SHARE  NULL
+#define NO_RES_ID  -1 /* Invalid ID */
+
+/*
+ * Share quantities are a child's fraction of the parent's resource
+ * specified by a divisor in the parent and a dividend in the child.
+ *
+ * Shares are represented as a relative quantity between parent and child
+ * to simplify locking when propagating modifications to the shares of a
+ * resource group. Only the parent and the children of the modified
+ * resource group need to be locked.
+*/
+struct res_shares {
+ /* shares only set by userspace */
+ int min_shares; /* minimun fraction of parent's resources allowed */
```

```
+ int max_shares; /* maximum fraction of parent's resources allowed */
+ int child_shares_divisor; /* >= 1, may not be DONT_CARE */
+
+ /*
+  * share values invisible to userspace.  adjusted when userspace
+  * sets shares
+  */
+ int unused_min_shares;
+  /* 0 <= unused_min_shares <= (child_shares_divisor -
+   *    Sum of min_shares of children)
+   */
+ int cur_max_shares; /* max(children's max_shares). need better name */
+
+ /* State maintained by container system - only relevant when
+  * this shares struct is the actual shares struct for a
+  * container */
+ struct container_subsys_state css;
+};
+
+/*
+ * Class is the grouping of tasks with shares of each resource that has
+ * registered a resource controller (see include/linux/res_group_rc.h).
+ */
+
+#define resource_group container
+
+#endif /* CONFIG_RES_GROUPS */
+#endif /* _LINUX_RES_GROUP_H */
Index: container-2.6.20-new/include/linux/res_group_rc.h
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/include/linux/res_group_rc.h
@@ -0,0 +1,125 @@
+/*
+ *  res_group_rc.h - Header file to be used by Resource controllers of
+ *        Resource Groups
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003
+ *  (C) Shailabh Nagar,  IBM Corp. 2003
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *  (C) Vivek Kashyap , IBM Corp. 2004
+ *
+ * Provides data structures, macros and kernel API of Resource Groups for
+ * resource controllers.
+ *
+ * More details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
```

```
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+#ifndef _LINUX_RES_GROUP_RC_H
+#define _LINUX_RES_GROUP_RC_H
+
+#include <linux/res_group.h>
+#include <linux/container.h>
+
+struct res_group_cft {
+ struct cftype cft;
+ struct res_controller *ctlr;
+};
+
+struct res_controller {
+ struct container_subsys *subsys;
+ struct res_group_cft shares_cft;
+ struct res_group_cft stats_cft;
+
+ //const char *name;
+ unsigned int ctlr_id;
+
+ /*
+  * Keeps number of references to this controller structure. kref
+  * does not work as we want to be able to allow removal of a
+  * controller even when some resource group are still defined.
+  */
+ atomic_t count;
+
+ /*
+  * Allocate a new shares struct for this resource controller.
+  * Called when registering a resource controller with pre-existing
+  * resource groups and when new resource group is created by the user.
+  */
+ struct res_shares *(*alloc_shares_struct)(struct container *);
+ /* Corresponding free of shares struct for this resource controller */
+ void (*free_shares_struct)(struct res_shares *);
+
+ /* Notifies the controller when the shares are changed */
+ void (*shares_changed)(struct res_shares *);
+
+ /* resource statistics */
+ ssize_t (*show_stats)(struct res_shares *, char *, size_t);
+ int (*reset_stats)(struct res_shares *, const char *);
+
```

```
+ /*
+  * move_task is called when a task moves from one resource group to
+  * another. First parameter is the task that is moving, the second
+  * is the resource specific shares of the resource group the task
+  * was in, and the third is the shares of the resource group the
+  * task has moved to.
+  */
+ void (*move_task)(struct task_struct *, struct res_shares *,
+    struct res_shares *);
+};
+
+#define DECLARE_RG_CONTROLLER(_name)  \
+struct res_controller _name ## _ctlr;  \
+struct container_subsys _name ## _subsys = { \
+ .name = #_name,     \
+ .create = res_group_create,  \
+ .destroy = res_group_destroy,  \
+ .populate = res_group_populate,  \
+ .attach = res_group_attach,  \
+ .fork = res_group_fork,   \
+ .exit = res_group_exit,   \
+      \
+ .subsys_id = _name ## _subsys_id, \
+ .private = &_name ## _ctlr,  \
+};
+
+extern int res_group_create(struct container_subsys *ss,
+       struct container *cont);
+extern void res_group_destroy(struct container_subsys *ss,
+         struct container *cont);
+extern int res_group_populate(struct container_subsys *ss,
+         struct container *cont);
+extern void res_group_attach(struct container_subsys *ss,
+        struct container *cont,
+        struct container *old_cont,
+        struct task_struct *tsk);
+extern void res_group_fork(struct container_subsys *ss,
+      struct task_struct *task);
+extern void res_group_exit(struct container_subsys *ss,
+      struct task_struct *task);
+
+extern struct resource_group default_res_group;
+static inline int is_res_group_root(const struct resource_group *rgroup)
+{
+ return (rgroup->parent == NULL);
+}
+
+#define for_each_child(child, parent) \
```

```
+ list_for_each_entry(child, &parent->children, sibling)
+
+/* Get controller specific shares structure for the given resource group */
+static inline struct res_shares *get_controller_shares(
+ struct container *rgroup, struct res_controller *ctlr)
+{
+ if (rgroup && ctlr)
+  return container_of(rgroup->subsys[ctlr->subsys->subsys_id],
+        struct res_shares, css);
+ else
+  return NO_SHARE;
+}
+
+#endif /* _LINUX_RES_GROUP_RC_H */
Index: container-2.6.20-new/init/Kconfig
 ===================================================================== =======
--- container-2.6.20-new.orig/init/Kconfig
+++ container-2.6.20-new/init/Kconfig
@@ -329,6 +329,28 @@ config TASK_IO_ACCOUNTING

    Say N if unsure.

+menu "Resource Groups"
+
+config RES_GROUPS
+ bool "Resource Groups"
+ depends on EXPERIMENTAL
+ select CONTAINERS
+ help
+   Resource Groups is a framework for controlling and monitoring
+   resource allocation of user-defined groups of tasks. For more
+   information, please visit http://ckrm.sf.net.
+
+config RES_GROUPS_NUMTASKS
+ bool "Number of Tasks Resource Controller"
+ depends on RES_GROUPS
+ default y
+ help
+   Provides a Resource Controller for Resource Groups that allows
+   limiting number of tasks a resource group can have.
+
+   Say N if unsure, Y to use the feature.
+
+endmenu
 config SYSCTL
  bool

Index: container-2.6.20-new/kernel/Makefile
```

```
 ===================================================================
--- container-2.6.20-new.orig/kernel/Makefile
+++ container-2.6.20-new/kernel/Makefile
@@ -52,6 +52,7 @@ obj-$(CONFIG_RELAY) += relay.o
 obj-$(CONFIG_UTS_NS) += utsname.o
 obj-$(CONFIG_TASK_DELAY_ACCT) += delayacct.o
 obj-$(CONFIG_TASKSTATS) += taskstats.o tsacct.o
+obj-$(CONFIG_RES_GROUPS) += res_group/

 ifneq ($(CONFIG_SCHED_NO_NO_OMIT_FRAME_POINTER),y)
 # According to Alan Modra <alan@linuxcare.com.au>, the -fno-omit-frame-pointer is
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
@@ -49,6 +49,7 @@
 #include <linux/delayacct.h>
 #include <linux/taskstats_kern.h>
 #include <linux/random.h>
+#include <linux/numtasks.h>

 #include <asm/pgtable.h>
 #include <asm/pgalloc.h>
@@ -1362,7 +1363,7 @@ long do_fork(unsigned long clone_flags,
       int __user *child_tidptr)
 {
  struct task_struct *p;
- int trace = 0;
+ int trace = 0, rc;
  struct pid *pid = alloc_pid();
  long nr;

@@ -1375,6 +1376,10 @@ long do_fork(unsigned long clone_flags,
    clone_flags |= CLONE_PTRACE;
  }

+ rc = numtasks_allow_fork(current);
+ if (rc)
+  return rc;
+
  p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr, child_tidptr, nr);
  /*
   * Do this prior waking up the new thread - the thread pointer
Index: container-2.6.20-new/kernel/res_group/Makefile
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/res_group/Makefile
@@ -0,0 +1,2 @@
```

```
+obj-y = res_group.o shares.o rgcs.o
+obj-$(CONFIG_RES_GROUPS_NUMTASKS) += numtasks.o
Index: container-2.6.20-new/kernel/res_group/local.h
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/res_group/local.h
@@ -0,0 +1,38 @@
+/*
+ * Contains function definitions that are local to the Resource Groups.
+ * NOT to be included by controllers.
+ */
+
+#include <linux/res_group_rc.h>
+
+extern struct res_controller *get_controller_by_name(const char *);
+extern struct res_controller *get_controller_by_id(unsigned int);
+extern void put_controller(struct res_controller *);
+extern struct resource_group *alloc_res_group(struct resource_group *,
+       const char *);
+extern int free_res_group(struct resource_group *);
+extern void release_res_group(struct kref *);
+extern int set_controller_shares(struct resource_group *,
+   struct res_controller *, const struct res_shares *);
+/* Set shares for the given resource group and resource to default values */
+extern void set_shares_to_default(struct resource_group *,
+      struct res_controller *);
+extern void res_group_teardown(void);
+extern int set_res_group(pid_t, struct resource_group *);
+extern void move_tasks_to_parent(struct resource_group *);
+
+ssize_t res_group_file_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       char __user *buf,
+       size_t nbytes, loff_t *ppos);
+ssize_t res_group_file_write(struct container *cont,
+        struct cftype *cft,
+        struct file *file,
+        const char __user *userbuf,
+        size_t nbytes, loff_t *ppos);
+
+enum {
+ RG_FILE_SHARES,
+ RG_FILE_STATS,
+};
Index: container-2.6.20-new/kernel/res_group/numtasks.c
 ===================================================================
--- /dev/null
```

```
+++ container-2.6.20-new/kernel/res_group/numtasks.c
@@ -0,0 +1,451 @@
+/* numtasks.c - "Number of tasks" resource controller for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman,  IBM Corp. 2003-2006
+ *       (C) Matt Helsley, IBM Corp. 2006
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+/*
+ * Resource controller for tracking number of tasks in a resource group.
+ */
+#include <linux/module.h>
+#include <linux/res_group_rc.h>
+#include <linux/numtasks.h>
+
+static const char res_ctlr_name[] = "numtasks";
+
+#define UNLIMITED INT_MAX
+#define DEF_TOTAL_NUM_TASKS UNLIMITED
+static int total_numtasks __read_mostly = DEF_TOTAL_NUM_TASKS;
+
+static struct resource_group *root_rgroup;
+static int total_cnt_alloc = 0;
+
+#define DEF_FORKRATE UNLIMITED
+#define DEF_FORKRATE_INTERVAL (1)
+static int forkrate __read_mostly = DEF_FORKRATE;
+static int forkrate_interval __read_mostly = DEF_FORKRATE_INTERVAL;
+
+struct numtasks {
+ struct res_shares shares;
+ int cnt_min_shares; /* num_tasks min_shares in local units */
+ int cnt_unused;  /* has to borrow if more than this is needed */
+ int cnt_max_shares; /* no tasks over this limit. */
+    /* Three above cnt_* fields are protected
+     * by resource group's group_lock */
+ atomic_t cnt_cur_alloc; /* current alloc from self */
+ atomic_t cnt_borrowed; /* borrowed from the parent */
+
+ /* stats */
```

```
+ int successes;
+ int failures;
+ int forkrate_failures;
+
+ /* Fork rate fields */
+ int forks_in_period;
+ unsigned long period_start;
+};
+
+struct res_controller numtasks_ctlr;
+
+static struct numtasks *get_shares_numtasks(struct res_shares *shares)
+{
+ if (shares)
+  return container_of(shares, struct numtasks, shares);
+ return NULL;
+}
+
+static struct numtasks *get_numtasks(struct resource_group *rgroup)
+{
+ return get_shares_numtasks(get_controller_shares(rgroup,
+       &numtasks_ctlr));
+}
+
+static struct resource_group *numtasks_rgroup(struct numtasks *nt)
+{
+ return nt->shares.css.container;
+}
+
+static inline int check_forkrate(struct numtasks *res)
+{
+ if (time_after(jiffies, res->period_start + forkrate_interval * HZ)) {
+  res->period_start = jiffies;
+  res->forks_in_period = 0;
+ }
+
+ if (res->forks_in_period >= forkrate) {
+   res->forkrate_failures++;
+  return -ENOSPC;
+ }
+ res->forks_in_period++;
+ return 0;
+}
+
+int numtasks_allow_fork(struct task_struct *task)
+{
+ int rc = 0;
+ struct numtasks *res;
```

```
+
+ /* task->container won't be deleted during an RCU critical section */
+ rcu_read_lock();
+
+ /* controller is not registered; no resource group is given */
+ if (numtasks_ctlr.ctlr_id == NO_RES_ID)
+  goto out;
+ res = get_numtasks(task_container(task, numtasks_subsys_id));
+
+ /* numtasks not available for this resource group */
+ if (!res)
+  goto out;
+
+ /* Check forkrate before checking resource group's usage */
+ rc = check_forkrate(res);
+ if (rc)
+  goto out;
+
+ if (res->cnt_max_shares == SHARE_DONT_CARE)
+  goto out;
+
+ /* Over the limit ? */
+ if (atomic_read(&res->cnt_cur_alloc) >= res->cnt_max_shares) {
+  res->failures++;
+  rc = -ENOSPC;
+  goto out;
+ }
+ out:
+ rcu_read_unlock();
+ return rc;
+}
+
+static void inc_usage_count(struct numtasks *res)
+{
+ struct resource_group *rgroup = numtasks_rgroup(res);
+ atomic_inc(&res->cnt_cur_alloc);
+
+ if (is_res_group_root(rgroup)) {
+  total_cnt_alloc++;
+  res->successes++;
+  return;
+ }
+ /* Do we need to borrow from our parent ? */
+ if ((res->cnt_unused == SHARE_DONT_CARE) ||
+    (atomic_read(&res->cnt_cur_alloc) > res->cnt_unused)) {
+  inc_usage_count(get_numtasks(rgroup->parent));
+  atomic_inc(&res->cnt_borrowed);
+ } else {
```

```
+  total_cnt_alloc++;
+    res->successes++;
+ }
+}
+
+static void dec_usage_count(struct numtasks *res)
+{
+ if (atomic_read(&res->cnt_cur_alloc) == 0)
+  return;
+ atomic_dec(&res->cnt_cur_alloc);
+ if (atomic_read(&res->cnt_borrowed) > 0) {
+  atomic_dec(&res->cnt_borrowed);
+  dec_usage_count(get_numtasks(numtasks_rgroup(res)->parent));
+ } else
+  total_cnt_alloc--;
+
+}
+
+static void numtasks_move_task(struct task_struct *task,
+  struct res_shares *old, struct res_shares *new)
+{
+ struct numtasks *oldres, *newres;
+
+ if (old == new)
+  return;
+
+ /* Decrement usage count of old resource group */
+ oldres = get_shares_numtasks(old);
+ if (oldres)
+  dec_usage_count(oldres);
+
+ /* Increment usage count of new resource group */
+ newres = get_shares_numtasks(new);
+ if (newres)
+  inc_usage_count(newres);
+}
+
+/* Initialize share struct values */
+static void numtasks_res_init_one(struct numtasks *numtasks_res)
+{
+ numtasks_res->shares.min_shares = SHARE_DONT_CARE;
+ numtasks_res->shares.max_shares = SHARE_DONT_CARE;
+ numtasks_res->shares.child_shares_divisor = SHARE_DEFAULT_DIVISOR;
+ numtasks_res->shares.unused_min_shares = SHARE_DEFAULT_DIVISOR;
+
+ numtasks_res->cnt_min_shares = SHARE_DONT_CARE;
+ numtasks_res->cnt_unused = SHARE_DONT_CARE;
+ numtasks_res->cnt_max_shares = SHARE_DONT_CARE;
```

```
+ numtasks_res->period_start = jiffies;
+}
+
+static struct res_shares *numtasks_alloc_shares_struct(
+     struct resource_group *rgroup)
+{
+ struct numtasks *res;
+
+ res = kzalloc(sizeof(struct numtasks), GFP_KERNEL);
+ if (!res)
+  return NULL;
+ numtasks_res_init_one(res);
+ if (is_res_group_root(rgroup))
+  root_rgroup = rgroup; /* store root's resource group. */
+ return &res->shares;
+}
+
+/*
+ * No locking of this resource group object necessary as we are not
+ * supposed to be assigned (or used) when/after this function is called.
+ */
+static void numtasks_free_shares_struct(struct res_shares *my_res)
+{
+ struct numtasks *res, *parres;
+ int i, borrowed;
+ struct resource_group *rgroup;
+
+ res = get_shares_numtasks(my_res);
+ rgroup = numtasks_rgroup(res);
+ if (!is_res_group_root(rgroup)) {
+  parres = get_numtasks(rgroup->parent);
+  borrowed = atomic_read(&res->cnt_borrowed);
+  for (i = 0; i < borrowed; i++)
+   dec_usage_count(parres);
+ }
+ kfree(res);
+}
+
+static int recalc_shares(int self_shares, int parent_shares, int parent_divisor)
+{
+ u64 numerator;
+
+ if ((self_shares == SHARE_DONT_CARE) ||
+    (parent_shares == SHARE_DONT_CARE))
+  return SHARE_DONT_CARE;
+ if (parent_divisor == 0)
+  return 0;
+ numerator = (u64) self_shares * parent_shares;
```

```
+ do_div(numerator, parent_divisor);
+ return numerator;
+}
+
+static int recalc_unused_shares(int self_cnt_min_shares,
+    int self_unused_min_shares, int self_divisor)
+{
+ u64 numerator;
+
+ if (self_cnt_min_shares == SHARE_DONT_CARE)
+  return SHARE_DONT_CARE;
+ if (self_divisor == 0)
+  return 0;
+ numerator = (u64) self_unused_min_shares * self_cnt_min_shares;
+ do_div(numerator, self_divisor);
+ return numerator;
+}
+
+static void recalc_self(struct numtasks *res,
+    struct numtasks *parres)
+{
+ struct res_shares *par = &parres->shares;
+ struct res_shares *self = &res->shares;
+
+ res->cnt_min_shares = recalc_shares(self->min_shares,
+      parres->cnt_min_shares,
+      par->child_shares_divisor);
+ res->cnt_max_shares = recalc_shares(self->max_shares,
+      parres->cnt_max_shares,
+      par->child_shares_divisor);
+
+ /*
+  * Now that we know the new cnt_min/cnt_max boundaries we can update
+  * the unused quantity.
+  */
+ res->cnt_unused = recalc_unused_shares(res->cnt_min_shares,
+      self->unused_min_shares,
+      self->child_shares_divisor);
+}
+
+
+/*
+ * Recalculate the min_shares and max_shares in real units and propagate the
+ * same to children.
+ * Called with container_manage_lock() held.
+ */
+static void recalc_and_propagate(struct numtasks *res,
+    struct numtasks *parres)
```

```
+{
+ struct resource_group *child = NULL;
+ struct numtasks *childres;
+
+ if (parres)
+  recalc_self(res, parres);
+
+ /* propagate to children */
+ for_each_child(child, numtasks_rgroup(res)) {
+  childres = get_numtasks(child);
+  BUG_ON(!childres);
+  recalc_and_propagate(childres, res);
+ }
+}
+
+static void numtasks_shares_changed(struct res_shares *my_res)
+{
+ struct numtasks *parres, *res;
+ struct res_shares *cur, *par;
+ struct resource_group *rgroup;
+
+ res = get_shares_numtasks(my_res);
+ if (!res)
+  return;
+ rgroup = numtasks_rgroup(res);
+ cur = &res->shares;
+
+ if (!is_res_group_root(rgroup)) {
+  parres = get_numtasks(rgroup->parent);
+  par = &parres->shares;
+ } else {
+  parres = NULL;
+  par = NULL;
+ }
+ if (parres)
+  parres->cnt_unused = recalc_unused_shares(
+      parres->cnt_min_shares,
+      par->unused_min_shares,
+      par->child_shares_divisor);
+ recalc_and_propagate(res, parres);
+}
+
+static ssize_t numtasks_show_stats(struct res_shares *my_res,
+     char *buf, size_t buf_size)
+{
+ ssize_t i, j = 0;
+ struct numtasks *res;
+
```

```
+ res = get_shares_numtasks(my_res);
+ if (!res)
+  return -EINVAL;
+
+ i = snprintf(buf, buf_size, "%s: Current usage %d\n",
+     res_ctlr_name,
+     atomic_read(&res->cnt_cur_alloc));
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of successes %d\n",
+     res_ctlr_name, res->successes);
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of failures %d\n",
+     res_ctlr_name, res->failures);
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of forkrate failures %d\n",
+     res_ctlr_name, res->forkrate_failures);
+ j += i;
+ return j;
+}
+
+DECLARE_RG_CONTROLLER(numtasks);
+
+struct res_controller numtasks_ctlr = {
+ .subsys = &numtasks_subsys,
+ .ctlr_id = NO_RES_ID,
+ .alloc_shares_struct = numtasks_alloc_shares_struct,
+ .free_shares_struct = numtasks_free_shares_struct,
+ .move_task = numtasks_move_task,
+ .shares_changed = numtasks_shares_changed,
+ .show_stats = numtasks_show_stats,
+};
+
+/*
+ * Writeable module parameters use these set_<parameter> functions to respond
+ * to changes. Otherwise the values can be read and used any time.
+ */
+static int set_numtasks_config_val(int *var, int old_value, const char *val,
+    struct kernel_param *kp)
+{
+ int rc = param_set_int(val, kp);
+
+ if (rc < 0)
+  return rc;
+ if (*var < 1) {
+  *var = old_value;
+  return -EINVAL;
+ }
+ return 0;
```

```
+}
+
+static int set_total_numtasks(const char *val, struct kernel_param *kp)
+{
+ int prev = total_numtasks;
+ int rc = set_numtasks_config_val(&total_numtasks, prev, val, kp);
+ struct numtasks *res = NULL;
+
+ if (!root_rgroup)
+  return 0;
+ if (rc < 0)
+  return rc;
+ if (total_numtasks <= total_cnt_alloc) {
+  total_numtasks = prev;
+  return -EINVAL;
+ }
+ container_lock();
+ res = get_numtasks(root_rgroup);
+ res->cnt_min_shares = total_numtasks;
+ res->cnt_unused = total_numtasks;
+ res->cnt_max_shares = total_numtasks;
+ recalc_and_propagate(res, NULL);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(total_numtasks, int, set_total_numtasks,
+   S_IRUGO | S_IWUSR);
+
+static void reset_forkrates(struct resource_group *rgroup, unsigned long now)
+{
+ struct numtasks *res;
+ struct resource_group *child = NULL;
+
+ res = get_numtasks(rgroup);
+ if (!res)
+  return;
+ res->forks_in_period = 0;
+ res->period_start = now;
+
+ for_each_child(child, rgroup)
+  reset_forkrates(child, now);
+}
+
+static int set_forkrate(const char *val, struct kernel_param *kp)
+{
+ int prev = forkrate;
+ int rc = set_numtasks_config_val(&forkrate, prev, val, kp);
+ if (rc < 0)
```

```
+  return rc;
+ container_lock();
+ reset_forkrates(root_rgroup, jiffies);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(forkrate, int, set_forkrate, S_IRUGO | S_IWUSR);
+
+static int set_forkrate_interval(const char *val, struct kernel_param *kp)
+{
+ int prev = forkrate_interval;
+ int rc = set_numtasks_config_val(&forkrate_interval, prev, val, kp);
+ if (rc < 0)
+  return rc;
+ container_lock();
+ reset_forkrates(root_rgroup, jiffies);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(forkrate_interval, int, set_forkrate_interval,
+   S_IRUGO | S_IWUSR);
Index: container-2.6.20-new/kernel/res_group/res_group.c
 ===================================================== =======
--- /dev/null
+++ container-2.6.20-new/kernel/res_group/res_group.c
@@ -0,0 +1,135 @@
+/* res_group.c - Resource Groups: Resource management through grouping of
+ *    unrelated tasks.
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003, 2004
+ *  (C) Shailabh Nagar, IBM Corp. 2003, 2004
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *  (C) Vivek Kashyap, IBM Corp. 2004
+ *  (C) Matt Helsley, IBM Corp. 2006
+ *
+ * Provides kernel API of Resource Groups for in-kernel,per-resource
+ * controllers (one each for cpu, memory and io).
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+#include <linux/module.h>
```

```
+#include <asm/uaccess.h>
+#include <linux/fs.h>
+#include "local.h"
+
+/*
+ * Interface for registering a resource controller. Called when the
+ * container system initializes this subsystem.
+ *
+ * Returns the 0 on success, -errno for failure.
+ * Fills ctlr->ctlr_id with a valid controller id on success.
+ */
+static int register_controller(struct res_controller *ctlr)
+{
+ struct container_subsys *ss;
+
+ if (!ctlr)
+  return -EINVAL;
+
+ ss = ctlr->subsys;
+
+ BUG_ON(ss->active);
+
+ /* Make sure there is an alloc and a free */
+ if (!ctlr->alloc_shares_struct || !ctlr->free_shares_struct)
+  return -EINVAL;
+
+ ctlr->shares_cft.ctlr = ctlr;
+ strcpy(ctlr->shares_cft.cft.name, ss->name);
+ strcat(ctlr->shares_cft.cft.name, ".shares");
+ ctlr->shares_cft.cft.private = RG_FILE_SHARES;
+ ctlr->shares_cft.cft.read = res_group_file_read;
+ ctlr->shares_cft.cft.write = res_group_file_write;
+
+ ctlr->stats_cft.ctlr = ctlr;
+ strcpy(ctlr->stats_cft.cft.name, ss->name);
+ strcat(ctlr->stats_cft.cft.name, ".stats");
+ ctlr->stats_cft.cft.private = RG_FILE_STATS;
+ ctlr->stats_cft.cft.read = res_group_file_read;
+ ctlr->stats_cft.cft.write = res_group_file_write;
+
+ ctlr->ctlr_id = ss->subsys_id;
+
+ return 0;
+}
+
+ int res_group_create(struct container_subsys *ss,
+       struct container *cont)
+{
```

```
+ struct res_controller *ctlr = ss->private;
+ struct res_shares *shares;
+ if (!cont->parent) {
+  int retval = register_controller(ctlr);
+  BUG_ON(retval);
+ }
+ shares = ctlr->alloc_shares_struct(cont);
+ cont->subsys[ss->subsys_id] = &shares->css;
+ return 0;
+}
+
+ void res_group_destroy(struct container_subsys *ss,
+        struct container *cont)
+{
+ struct res_controller *ctlr = ss->private;
+ struct res_shares *shares = get_controller_shares(cont, ctlr);
+ ctlr->free_shares_struct(shares);
+}
+
+ int res_group_populate(struct container_subsys *ss,
+         struct container *cont) {
+ int err;
+ struct res_controller *ctlr = ss->private;
+ if ((err = container_add_file(cont, &ctlr->shares_cft.cft)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &ctlr->stats_cft.cft)) < 0)
+  return err;
+
+ return 0;
+}
+
+ void res_group_attach(struct container_subsys *ss,
+        struct container *cont,
+        struct container *old_cont,
+        struct task_struct *tsk) {
+ struct res_controller *ctlr = ss->private;
+ struct res_shares *oldshares = get_controller_shares(old_cont, ctlr);
+ struct res_shares *newshares = get_controller_shares(cont, ctlr);
+
+ if (ctlr->move_task) {
+  ctlr->move_task(tsk, oldshares, newshares);
+ }
+}
+
+ void res_group_fork(struct container_subsys *ss,
+    struct task_struct *task) {
+ struct res_controller *ctlr = ss->private;
+ struct res_shares *shares =
```

```
+  get_controller_shares(task_container(task, ss->subsys_id), ctlr);
+ if (ctlr->move_task) {
+  ctlr->move_task(task, NULL, shares);
+ }
+}
+
+ void res_group_exit(struct container_subsys *ss,
+      struct task_struct *task) {
+ struct res_controller *ctlr = ss->private;
+ struct res_shares *shares =
+  get_controller_shares(task_container(task, ss->subsys_id), ctlr);
+ if (ctlr->move_task) {
+  ctlr->move_task(task, shares, NULL);
+ }
+}
+
+EXPORT_SYMBOL_GPL(set_controller_shares);
Index: container-2.6.20-new/kernel/res_group/rgcs.c
 ===================================================================== =======
--- /dev/null
+++ container-2.6.20-new/kernel/res_group/rgcs.c
@@ -0,0 +1,302 @@
+/*
+ * kernel/res_group/rgcs.c
+ *
+ * Copyright (C) Shailabh Nagar,  IBM Corp. 2005
+ *        Chandra Seetharaman,   IBM Corp. 2005, 2006
+ *
+ * Resource Group Configfs Subsystem (rgcs) provides the user interface
+ * for Resource groups.
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of version 2 of the  GNU General Public License
+ * as published by the Free Software Foundation.
+ *
+ */
+#include <linux/ctype.h>
+#include <linux/module.h>
+#include <linux/configfs.h>
+#include <linux/parser.h>
+#include <linux/fs.h>
+#include <asm/uaccess.h>
+
+#include "local.h"
+
+#define RES_STRING "res"
```

```
+#define MIN_SHARES_STRING "min_shares"
+#define MAX_SHARES_STRING "max_shares"
+#define CHILD_SHARES_DIVISOR_STRING "child_shares_divisor"
+
+static ssize_t show_stats(struct resource_group *rgroup,
+     struct res_controller *ctlr,
+     char *buf)
+{
+ int j = 0, rc = 0;
+ size_t buf_size = PAGE_SIZE-1; /* allow only PAGE_SIZE # of bytes */
+ struct res_shares *shares;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares && ctlr->show_stats)
+  j = ctlr->show_stats(shares, buf, buf_size);
+ rc += j;
+ buf += j;
+ buf_size -= j;
+ return rc;
+}
+
+enum parse_token_t {
+ parse_res_type, parse_err
+};
+
+static match_table_t parse_tokens = {
+ {parse_res_type, RES_STRING"=%s"},
+ {parse_err, NULL}
+};
+
+static int stats_parse(const char *options,
+    char **resname, char **remaining_line)
+{
+ char *p, *str;
+ int rc = -EINVAL;
+
+ if (!options)
+  return -EINVAL;
+
+ while ((p = strsep((char **)&options, ",")) != NULL) {
+  substring_t args[MAX_OPT_ARGS];
+  int token;
+
+  if (!*p)
+   continue;
+  token = match_token(p, parse_tokens, args);
+  if (token == parse_res_type) {
+   *resname = match_strdup(args);
```

```
+   str = p + strlen(p) + 1;
+   *remaining_line = kmalloc(strlen(str) + 1, GFP_KERNEL);
+   if (*remaining_line == NULL) {
+    kfree(*resname);
+    *resname = NULL;
+    rc = -ENOMEM;
+   } else {
+    strcpy(*remaining_line, str);
+    rc = 0;
+   }
+   break;
+  }
+ }
+ return rc;
+}
+
+static int reset_stats(struct resource_group *rgroup, struct res_controller *ctlr, const char *str)
+{
+ int rc;
+ char *resname = NULL, *statstr = NULL;
+ struct res_shares *shares;
+
+ rc = stats_parse(str, &resname, &statstr);
+ if (rc)
+  return rc;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares && ctlr->reset_stats)
+  rc = ctlr->reset_stats(shares, statstr);
+
+ kfree(resname);
+ kfree(statstr);
+ return rc;
+}
+
+
+enum share_token_t {
+ MIN_SHARES_TOKEN,
+ MAX_SHARES_TOKEN,
+ CHILD_SHARES_DIVISOR_TOKEN,
+ RESOURCE_TYPE_TOKEN,
+ ERROR_TOKEN
+};
+
+/* Token matching for parsing input to this magic file */
+static match_table_t shares_tokens = {
+ {RESOURCE_TYPE_TOKEN, RES_STRING"=%s"},
+ {MIN_SHARES_TOKEN, MIN_SHARES_STRING"=%d"},
```

```
+ {MAX_SHARES_TOKEN, MAX_SHARES_STRING"=%d"},
+ {CHILD_SHARES_DIVISOR_TOKEN, CHILD_SHARES_DIVISOR_STRING"=%d"},
+ {ERROR_TOKEN, NULL}
+};
+
+static int shares_parse(const char *options, char **resname,
+     struct res_shares *shares)
+{
+ char *p;
+ int option, rc = -EINVAL;
+
+ *resname = NULL;
+ if (!options)
+  goto done;
+
+ while ((p = strsep((char **)&options, ",")) != NULL) {
+  substring_t args[MAX_OPT_ARGS];
+  int token;
+
+  if (!*p)
+   continue;
+
+  token = match_token(p, shares_tokens, args);
+  switch (token) {
+  case RESOURCE_TYPE_TOKEN:
+   if (*resname)
+    goto done;
+   *resname = match_strdup(args);
+   break;
+  case MIN_SHARES_TOKEN:
+   if (match_int(args, &option))
+    goto done;
+   shares->min_shares = option;
+   break;
+  case MAX_SHARES_TOKEN:
+   if (match_int(args, &option))
+    goto done;
+   shares->max_shares = option;
+   break;
+  case CHILD_SHARES_DIVISOR_TOKEN:
+   if (match_int(args, &option))
+    goto done;
+   shares->child_shares_divisor = option;
+   break;
+  default:
+   goto done;
+  }
+ }
```

```
+ rc = 0;
+done:
+ if (rc) {
+  kfree(*resname);
+  *resname = NULL;
+ }
+ return rc;
+}
+
+static int set_shares(struct resource_group *rgroup,
+        struct res_controller *ctlr,
+        const char *str)
+{
+ char *resname = NULL;
+ int rc;
+ struct res_shares shares = {
+  .min_shares = SHARE_UNCHANGED,
+  .max_shares = SHARE_UNCHANGED,
+  .child_shares_divisor = SHARE_UNCHANGED,
+ };
+
+ rc = shares_parse(str, &resname, &shares);
+ if (!rc) {
+  rc = set_controller_shares(rgroup, ctlr, &shares);
+  kfree(resname);
+ }
+ return rc;
+}
+
+static ssize_t show_shares(struct resource_group *rgroup,
+      struct res_controller *ctlr,
+      char *buf)
+{
+ ssize_t j, rc = 0, bufsize = PAGE_SIZE;
+ struct res_shares *shares;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares) {
+  j = snprintf(buf, bufsize, "%s=%s,%s=%d,%s=%d,%s=%d\n",
+        RES_STRING, ctlr->subsys->name,
+        MIN_SHARES_STRING, shares->min_shares,
+        MAX_SHARES_STRING, shares->max_shares,
+        CHILD_SHARES_DIVISOR_STRING,
+        shares->child_shares_divisor);
+  rc += j; buf += j; bufsize -= j;
+ }
+ return rc;
+}
```

```
+
+ssize_t res_group_file_write(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       const char __user *userbuf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct res_group_cft *rgcft = container_of(cft, struct res_group_cft, cft);
+ struct res_controller *ctlr = rgcft->ctlr;
+
+ char *buf;
+ ssize_t retval;
+ int filetype = cft->private;
+
+ if (nbytes >= PAGE_SIZE)
+  return -E2BIG;
+
+ buf = kmalloc(nbytes + 1, GFP_USER);
+ if (!buf) return -ENOMEM;
+ if (copy_from_user(buf, userbuf, nbytes)) {
+  retval = -EFAULT;
+  goto out1;
+ }
+ buf[nbytes] = 0; /* nul-terminate */
+
+ container_lock();
+
+ if (container_is_removed(cont)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch(filetype) {
+ case RG_FILE_SHARES:
+  retval = set_shares(cont, ctlr, buf);
+  break;
+ case RG_FILE_STATS:
+  retval = reset_stats(cont, ctlr, buf);
+  break;
+ default:
+  retval = -EINVAL;
+ }
+ if (!retval) retval = nbytes;
+
+ out2:
+ container_unlock();
+ out1:
+ kfree(buf);
```

```
+ return retval;
+}
+
+ssize_t res_group_file_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct res_group_cft *rgcft = container_of(cft, struct res_group_cft, cft);
+ struct res_controller *ctlr = rgcft->ctlr;
+
+ char *page = kmalloc(PAGE_SIZE, GFP_USER);
+ ssize_t retval;
+ int filetype = cft->private;
+
+ if (!page) return -ENOMEM;
+
+ switch(filetype) {
+ case RG_FILE_SHARES:
+  retval = show_shares(cont, ctlr, page);
+  break;
+ case RG_FILE_STATS:
+  retval = show_stats(cont, ctlr, page);
+  break;
+ default:
+  retval = -EINVAL;
+ }
+
+ if (retval >= 0) {
+  retval = simple_read_from_buffer(buf, nbytes,
+       ppos, page, retval);
+ }
+ kfree(page);
+ return retval;
+}
Index: container-2.6.20-new/kernel/res_group/shares.c
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/res_group/shares.c
@@ -0,0 +1,228 @@
+/*
+ * shares.c - Share management functions for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman,  IBM Corp. 2003, 2004, 2005, 2006
+ *  (C) Hubertus Franke,  IBM Corp. 2004
+ *  (C) Matt Helsley,  IBM Corp. 2006
+ *
```

```
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License version 2 as
+ * published by the Free Software Foundation.
+ */
+
+#include <linux/errno.h>
+#include <linux/res_group_rc.h>
+#include <linux/container.h>
+
+/*
+ * Share values can be quantitative (quantity of memory for instance) or
+ * symbolic. The symbolic value DONT_CARE allows for any quantity of a resource
+ * to be substituted in its place. The symbolic value UNCHANGED is only used
+ * when setting share values and means that the old value should be used.
+ */
+
+/* Is the share a quantity (as opposed to "symbols" DONT_CARE or UNCHANGED) */
+static inline int is_share_quantitative(int share)
+{
+ return (share >= 0);
+}
+
+static inline int is_share_symbolic(int share)
+{
+ return !is_share_quantitative(share);
+}
+
+static inline int is_share_valid(int share)
+{
+ return ((share == SHARE_DONT_CARE) ||
+   (share == SHARE_UNSUPPORTED) ||
+   is_share_quantitative(share));
+}
+
+static inline int did_share_change(int share)
+{
+ return (share != SHARE_UNCHANGED);
+}
+
+static inline int change_supported(int share)
+{
+ return (share != SHARE_UNSUPPORTED);
+}
+
+/*
+ * Caller is responsible for protecting 'parent'
```

```
+ * Caller is responsible for making sure that the sum of sibling min_shares
+ * doesn't exceed parent's total min_shares.
+ */
+static inline void child_min_shares_changed(struct res_shares *parent,
+       int child_cur_min_shares,
+       int child_new_min_shares)
+{
+ if (is_share_quantitative(child_new_min_shares))
+  parent->unused_min_shares -= child_new_min_shares;
+ if (is_share_quantitative(child_cur_min_shares))
+  parent->unused_min_shares += child_cur_min_shares;
+}
+
+/*
+ * Set parent's cur_max_shares to the largest 'max_shares' of all
+ * of its children.
+ */
+static inline void set_cur_max_shares(struct resource_group *parent,
+         struct res_controller *ctlr)
+{
+ int max_shares = 0;
+ struct resource_group *child = NULL;
+ struct res_shares *child_shares, *parent_shares;
+
+ for_each_child(child, parent) {
+  child_shares = get_controller_shares(child, ctlr);
+  max_shares = max(max_shares, child_shares->max_shares);
+ }
+
+ parent_shares = get_controller_shares(parent, ctlr);
+ parent_shares->cur_max_shares = max_shares;
+}
+
+/*
+ * Return -EINVAL if the child's shares violate self-consistency or
+ * parent-imposed restrictions. Otherwise return 0.
+ *
+ * This involves checking shares between the child and its parent;
+ * the child and itself (userspace can't be trusted).
+ */
+static inline int are_shares_valid(struct res_shares *child,
+       struct res_shares *parent,
+       int current_usage,
+       int min_shares_increase)
+{
+ /*
+  * CHILD <-> PARENT validation
+  * Increases in child's min_shares or max_shares can't exceed
```

```
+  * limitations imposed by the parent resource group.
+  * Only validate this if we have a parent.
+  */
+ if (parent &&
+     ((is_share_quantitative(child->min_shares) &&
+       (min_shares_increase > parent->unused_min_shares)) ||
+      (is_share_quantitative(child->max_shares) &&
+       (child->max_shares > parent->child_shares_divisor))))
+  return -EINVAL;
+
+ /* CHILD validation: is min valid */
+ if (!is_share_valid(child->min_shares))
+  return -EINVAL;
+
+ /* CHILD validation: is max valid */
+ if (!is_share_valid(child->max_shares))
+  return -EINVAL;
+
+ /*
+  * CHILD validation: is divisor quantitative & current_usage
+  * is not more than the new divisor
+  */
+ if (!is_share_quantitative(child->child_shares_divisor) ||
+    (current_usage > child->child_shares_divisor))
+  return -EINVAL;
+
+ /*
+  * CHILD validation: is the new child_shares_divisor large
+  * enough to accomodate largest max_shares of any of my child
+  */
+ if (child->child_shares_divisor < child->cur_max_shares)
+  return -EINVAL;
+
+ /* CHILD validation: min <= max */
+ if (is_share_quantitative(child->min_shares) &&
+    is_share_quantitative(child->max_shares) &&
+    (child->min_shares > child->max_shares))
+  return -EINVAL;
+
+ return 0;
+}
+
+/*
+ * Set the resource shares of a child resource group given the new shares
+ * specified by userspace, the child's current shares, and the parent
+ * resource group's shares.
+ *
+ * Caller is responsible for holding group_lock of child and parent
```

```
+ * resource groups to protect the shares structures passed to this function.
+ */
+static int set_shares(const struct res_shares *new,
+      struct res_shares *child_shares,
+          struct res_shares *parent_shares)
+{
+ int rc, current_usage, min_shares_increase;
+ struct res_shares final_shares;
+
+ BUG_ON(!new || !child_shares);
+
+ final_shares = *child_shares;
+ if (did_share_change(new->child_shares_divisor) &&
+   change_supported(child_shares->child_shares_divisor))
+  final_shares.child_shares_divisor = new->child_shares_divisor;
+ if (did_share_change(new->min_shares) &&
+   change_supported(child_shares->min_shares))
+  final_shares.min_shares = new->min_shares;
+ if (did_share_change(new->max_shares) &&
+   change_supported(child_shares->max_shares))
+  final_shares.max_shares = new->max_shares;
+
+ current_usage = child_shares->child_shares_divisor -
+        child_shares->unused_min_shares;
+ min_shares_increase = final_shares.min_shares;
+ if (is_share_quantitative(child_shares->min_shares))
+  min_shares_increase -= child_shares->min_shares;
+
+ rc = are_shares_valid(&final_shares, parent_shares, current_usage,
+            min_shares_increase);
+ if (rc)
+  return rc; /* new shares would violate restrictions */
+
+ if (did_share_change(new->child_shares_divisor))
+  final_shares.unused_min_shares =
+   (final_shares.child_shares_divisor - current_usage);
+ *child_shares = final_shares;
+ return 0;
+}
+
+int set_controller_shares(struct resource_group *rgroup,
+     struct res_controller *ctlr,
+     const struct res_shares *new_shares)
+{
+ struct res_shares *shares, *parent_shares;
+ int prev_min, prev_max, rc;
+
+ if (!ctlr->shares_changed)
```

```
+  return -EINVAL;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (!shares)
+  return -EINVAL;
+
+ prev_min = shares->min_shares;
+ prev_max = shares->max_shares;
+
+ container_lock(); /* XXX */
+ //spin_lock(&rgroup->group_lock);
+ parent_shares = get_controller_shares(rgroup->parent, ctlr);
+ rc = set_shares(new_shares, shares, parent_shares);
+
+ if (rc || is_res_group_root(rgroup))
+  goto done;
+
+ /* Notify parent about changes in my shares */
+ child_min_shares_changed(parent_shares, prev_min,
+          shares->min_shares);
+ if (prev_max != shares->max_shares)
+  set_cur_max_shares(rgroup->parent, ctlr);
+
+done:
+ container_unlock(); /* XXX */
+ if (!rc)
+  ctlr->shares_changed(shares);
+ return rc;
+}
Index: container-2.6.20-new/include/linux/container_subsys.h
 ========================================================== =======
--- container-2.6.20-new.orig/include/linux/container_subsys.h
+++ container-2.6.20-new/include/linux/container_subsys.h
@@ -17,4 +17,10 @@ SUBSYS(cpuacct)

 /* */

+#ifdef CONFIG_RES_GROUPS_NUMTASKS
+SUBSYS(numtasks)
+#endif
+
+/* */
+
 /* */

--
```

Subject: [PATCH 6/7] Containers (V8): BeanCounters over generic process containers
Posted by Paul Menage on Fri, 06 Apr 2007 23:32:27 GMT
View Forum Message <> Reply to Message

This patch implements the BeanCounter resource control abstraction
over generic process containers. It contains the beancounter core
code, plus the numfiles resource counter. It doesn't currently contain
any of the memory tracking code or the code for switching beancounter
context in interrupts.

Currently all the beancounters resource counters are lumped into a
single hierarchy; ideally it would be possible for each resource
counter to be a separate container subsystem, allowing them to be
connected to different hierarchies.

---
 fs/file_table.c | 11 +
 include/bc/beancounter.h | 194 ++++++++++++++++++++
 include/bc/misc.h | 27 ++
 include/bc/task.h | 64 ++++++
 include/linux/container_subsys.h | 6
 include/linux/fs.h | 3
 include/linux/sched.h | 3
 init/Kconfig | 4
 init/main.c | 3
 kernel/Makefile | 1
 kernel/bc/Kconfig | 17 +
 kernel/bc/Makefile | 7
 kernel/bc/beancounter.c | 368 ++++++++++++++++++++++++++++++++++++++++++
 kernel/bc/misc.c | 57 ++++++
 kernel/softirq.c | 7
 15 files changed, 771 insertions(+), 1 deletion(-)

Index: container-2.6.20-new/init/Kconfig
 ===================================================== =======
--- container-2.6.20-new.orig/init/Kconfig
+++ container-2.6.20-new/init/Kconfig
@@ -607,6 +607,10 @@ config STOP_MACHINE
   Need stop_machine() primitive.
 endmenu

+menu "Beancounters"
+source "kernel/bc/Kconfig"
+endmenu
+
 menu "Block layer"
 source "block/Kconfig"
 endmenu

```
Index: container-2.6.20-new/kernel/Makefile
 ===================================================================
--- container-2.6.20-new.orig/kernel/Makefile
+++ container-2.6.20-new/kernel/Makefile
@@ -12,6 +12,7 @@ obj-y     = sched.o fork.o exec_domain.o

 obj-$(CONFIG_STACKTRACE) += stacktrace.o
 obj-y += time/
+obj-$(CONFIG_BEANCOUNTERS) += bc/
 obj-$(CONFIG_DEBUG_MUTEXES) += mutex-debug.o
 obj-$(CONFIG_LOCKDEP) += lockdep.o
 ifeq ($(CONFIG_PROC_FS),y)
Index: container-2.6.20-new/kernel/bc/Kconfig
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/bc/Kconfig
@@ -0,0 +1,17 @@
+config BEANCOUNTERS
+ bool "Enable resource accounting/control"
+ default n
+ select CONTAINERS
+ help
+   When Y this option provides accounting and allows configuring
+   limits for user's consumption of exhaustible system resources.
+   The most important resource controlled by this patch is unswappable
+   memory (either mlock'ed or used by internal kernel structures and
+   buffers). The main goal of this patch is to protect processes
+   from running short of important resources because of accidental
+   misbehavior of processes or malicious activity aiming to ``kill''
+   the system. It's worth mentioning that resource limits configured
+   by setrlimit(2) do not give an acceptable level of protection
+   because they cover only a small fraction of resources and work on a
+   per-process basis.  Per-process accounting doesn't prevent malicious
+   users from spawning a lot of resource-consuming processes.
Index: container-2.6.20-new/kernel/bc/Makefile
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/bc/Makefile
@@ -0,0 +1,7 @@
+#
+# kernel/bc/Makefile
+#
+# Copyright (C) 2006 OpenVZ SWsoft Inc.
+#
+
+obj-y = beancounter.o misc.o
Index: container-2.6.20-new/include/bc/beancounter.h
 ===================================================================
```

```
--- /dev/null
+++ container-2.6.20-new/include/bc/beancounter.h
@@ -0,0 +1,194 @@
+/*
+ * include/bc/beancounter.h
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#ifndef __BEANCOUNTER_H__
+#define __BEANCOUNTER_H__
+
+#include <linux/container.h>
+
+enum {
+ BC_KMEMSIZE,
+ BC_PRIVVMPAGES,
+ BC_PHYSPAGES,
+ BC_NUMTASKS,
+ BC_NUMFILES,
+
+ BC_RESOURCES
+};
+
+struct bc_resource_parm {
+ unsigned long barrier;
+ unsigned long limit;
+ unsigned long held;
+ unsigned long minheld;
+ unsigned long maxheld;
+ unsigned long failcnt;
+
+};
+
+#ifdef __KERNEL__
+
+#include <linux/list.h>
+#include <linux/spinlock.h>
+#include <linux/init.h>
+#include <linux/configfs.h>
+#include <asm/atomic.h>
+
+#define BC_MAXVALUE ((unsigned long)LONG_MAX)
+
+enum bc_severity {
+ BC_BARRIER,
+ BC_LIMIT,
```

```
+ BC_FORCE,
+};
+
+struct beancounter;
+
+#ifdef CONFIG_BEANCOUNTERS
+
+enum bc_attr_index {
+ BC_RES_HELD,
+ BC_RES_MAXHELD,
+ BC_RES_MINHELD,
+ BC_RES_BARRIER,
+ BC_RES_LIMIT,
+ BC_RES_FAILCNT,
+
+ BC_ATTRS
+};
+
+struct bc_resource {
+ char *bcr_name;
+ int      res_id;
+
+ int (*bcr_init)(struct beancounter *bc, int res);
+ int (*bcr_change)(struct beancounter *bc,
+   unsigned long new_bar, unsigned long new_lim);
+ void (*bcr_barrier_hit)(struct beancounter *bc);
+ int (*bcr_limit_hit)(struct beancounter *bc, unsigned long val,
+   unsigned long flags);
+ void (*bcr_fini)(struct beancounter *bc);
+
+ /* container file handlers */
+ struct cftype cft_attrs[BC_ATTRS];
+};
+
+extern struct bc_resource *bc_resources[];
+extern struct container_subsys bc_subsys;
+
+struct beancounter {
+ struct container_subsys_state css;
+ spinlock_t  bc_lock;
+
+ struct bc_resource_parm bc_parms[BC_RESOURCES];
+};
+
+/* Update the beancounter for a container */
+static inline void set_container_bc(struct container *cont,
+        struct beancounter *bc)
+{
```

```
+ cont->subsys[bc_subsys.subsys_id] = &bc->css;
+}
+
+/* Retrieve the beancounter for a container */
+static inline struct beancounter *container_bc(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, bc_subsys_id),
+       struct beancounter, css);
+}
+
+/* Retrieve the beancounter for a task */
+static inline struct beancounter *task_bc(struct task_struct *task)
+{
+ return container_of(task_subsys_state(task, bc_subsys_id),
+       struct beancounter, css);
+}
+
+static inline void bc_adjust_maxheld(struct bc_resource_parm *parm)
+{
+ if (parm->maxheld < parm->held)
+  parm->maxheld = parm->held;
+}
+
+static inline void bc_adjust_minheld(struct bc_resource_parm *parm)
+{
+ if (parm->minheld > parm->held)
+  parm->minheld = parm->held;
+}
+
+static inline void bc_init_resource(struct bc_resource_parm *parm,
+  unsigned long bar, unsigned long lim)
+{
+ parm->barrier = bar;
+ parm->limit = lim;
+ parm->held = 0;
+ parm->minheld = 0;
+ parm->maxheld = 0;
+ parm->failcnt = 0;
+}
+
+int bc_change_param(struct beancounter *bc, int res,
+  unsigned long bar, unsigned long lim);
+
+int __must_check bc_charge_locked(struct beancounter *bc, int res_id,
+  unsigned long val, int strict, unsigned long flags);
+static inline int __must_check bc_charge(struct beancounter *bc, int res_id,
+  unsigned long val, int strict)
+{
```

```
+ int ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&bc->bc_lock, flags);
+ ret = bc_charge_locked(bc, res_id, val, strict, flags);
+ spin_unlock_irqrestore(&bc->bc_lock, flags);
+ return ret;
+}
+
+void bc_uncharge_locked(struct beancounter *bc, int res_id,
+         unsigned long val);
+
+static inline void bc_uncharge(struct beancounter *bc, int res_id,
+  unsigned long val)
+{
+ unsigned long flags;
+
+ spin_lock_irqsave(&bc->bc_lock, flags);
+ bc_uncharge_locked(bc, res_id, val);
+ spin_unlock_irqrestore(&bc->bc_lock, flags);
+}
+
+void __init bc_register_resource(int res_id, struct bc_resource *br);
+void __init bc_init_early(void);
+#else /* CONFIG_BEANCOUNTERS */
+static inline int __must_check bc_charge_locked(struct beancounter *bc, int res,
+  unsigned long val, int strict, unsigned long flags)
+{
+ return 0;
+}
+
+static inline int __must_check bc_charge(struct beancounter *bc, int res,
+  unsigned long val, int strict)
+{
+ return 0;
+}
+
+static inline void bc_uncharge_locked(struct beancounter *bc, int res,
+  unsigned long val)
+{
+}
+
+static inline void bc_uncharge(struct beancounter *bc, int res,
+  unsigned long val)
+{
+}
+
+static inline void bc_init_early(void)
```

```
+{
+}
+#endif /* CONFIG_BEANCOUNTERS */
+#endif /* __KERNEL__ */
+#endif
Index: container-2.6.20-new/init/main.c
 ===================================================================
--- container-2.6.20-new.orig/init/main.c
+++ container-2.6.20-new/init/main.c
@@ -54,6 +54,8 @@
 #include <linux/pid_namespace.h>
 #include <linux/device.h>

+#include <bc/beancounter.h>
+
 #include <asm/io.h>
 #include <asm/bugs.h>
 #include <asm/setup.h>
@@ -487,6 +489,7 @@ asmlinkage void __init start_kernel(void
  extern struct kernel_param __start___param[], __stop___param[];

  container_init_early();
+ bc_init_early();
  smp_setup_processor_id();

  /*
Index: container-2.6.20-new/kernel/bc/beancounter.c
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/kernel/bc/beancounter.c
@@ -0,0 +1,368 @@
+/*
+ * kernel/bc/beancounter.c
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#include <linux/sched.h>
+#include <linux/list.h>
+#include <linux/hash.h>
+#include <linux/gfp.h>
+#include <linux/slab.h>
+#include <linux/module.h>
+#include <linux/fs.h>
+#include <linux/uaccess.h>
+
+#include <bc/beancounter.h>
```

```
+
+#define BC_HASH_BITS (8)
+#define BC_HASH_SIZE (1 << BC_HASH_BITS)
+
+static int bc_dummy_init(struct beancounter *bc, int i)
+{
+ bc_init_resource(&bc->bc_parms[i], BC_MAXVALUE, BC_MAXVALUE);
+ return 0;
+}
+
+static struct bc_resource bc_dummy_res = {
+ .bcr_name = "dummy",
+ .bcr_init = bc_dummy_init,
+};
+
+struct bc_resource *bc_resources[BC_RESOURCES] = {
+ [0 ... BC_RESOURCES - 1] = &bc_dummy_res,
+};
+
+struct beancounter init_bc;
+static struct kmem_cache *bc_cache;
+
+static int bc_create(struct container_subsys *ss,
+       struct container *cont)
+{
+ int i;
+ struct beancounter *new_bc;
+
+ if (!cont->parent) {
+  /* Early initialization for top container */
+  set_container_bc(cont, &init_bc);
+  init_bc.css.container = cont;
+  return 0;
+ }
+
+ new_bc = kmem_cache_alloc(bc_cache, GFP_KERNEL);
+ if (!new_bc)
+  return -ENOMEM;
+
+ spin_lock_init(&new_bc->bc_lock);
+
+ for (i = 0; i < BC_RESOURCES; i++) {
+  int err;
+  if ((err = bc_resources[i]->bcr_init(new_bc, i))) {
+   for (i--; i >= 0; i--)
+    if (bc_resources[i]->bcr_fini)
+     bc_resources[i]->bcr_fini(new_bc);
+   kmem_cache_free(bc_cache, new_bc);
```

```
+   return err;
+  }
+ }
+
+ new_bc->css.container = cont;
+ set_container_bc(cont, new_bc);
+ return 0;
+}
+
+static void bc_destroy(struct container_subsys *ss,
+         struct container *cont)
+{
+ struct beancounter *bc = container_bc(cont);
+ kmem_cache_free(bc_cache, bc);
+}
+
+int bc_charge_locked(struct beancounter *bc, int res, unsigned long val,
+  int strict, unsigned long flags)
+{
+ struct bc_resource_parm *parm;
+ unsigned long new_held;
+
+ BUG_ON(val > BC_MAXVALUE);
+
+ parm = &bc->bc_parms[res];
+ new_held = parm->held + val;
+
+ switch (strict) {
+ case BC_LIMIT:
+  if (new_held > parm->limit)
+   break;
+  /* fallthrough */
+ case BC_BARRIER:
+  if (new_held > parm->barrier) {
+   if (strict == BC_BARRIER)
+    break;
+   if (parm->held < parm->barrier &&
+     bc_resources[res]->bcr_barrier_hit)
+    bc_resources[res]->bcr_barrier_hit(bc);
+  }
+  /* fallthrough */
+ case BC_FORCE:
+  parm->held = new_held;
+  bc_adjust_maxheld(parm);
+  return 0;
+ default:
+  BUG();
+ }
```

```
+
+ if (bc_resources[res]->bcr_limit_hit)
+  return bc_resources[res]->bcr_limit_hit(bc, val, flags);
+
+ parm->failcnt++;
+ return -ENOMEM;
+}
+
+void bc_uncharge_locked(struct beancounter *bc, int res, unsigned long val)
+{
+ struct bc_resource_parm *parm;
+
+ BUG_ON(val > BC_MAXVALUE);
+
+ parm = &bc->bc_parms[res];
+ if (unlikely(val > parm->held)) {
+  printk(KERN_ERR "BC: Uncharging too much of %s: %lu vs %lu\n",
+    bc_resources[res]->bcr_name,
+    val, parm->held);
+  val = parm->held;
+ }
+
+ parm->held -= val;
+ bc_adjust_minheld(parm);
+}
+
+int bc_change_param(struct beancounter *bc, int res,
+      unsigned long bar, unsigned long lim)
+{
+ int ret;
+
+ ret = -EINVAL;
+ if (bar > lim)
+  goto out;
+ if (bar > BC_MAXVALUE || lim > BC_MAXVALUE)
+  goto out;
+
+ ret = 0;
+ spin_lock_irq(&bc->bc_lock);
+ if (bc_resources[res]->bcr_change) {
+  ret = bc_resources[res]->bcr_change(bc, bar, lim);
+  if (ret < 0)
+   goto out_unlock;
+ }
+
+ bc->bc_parms[res].barrier = bar;
+ bc->bc_parms[res].limit = lim;
+
```

```
+out_unlock:
+ spin_unlock_irq(&bc->bc_lock);
+out:
+ return ret;
+}
+
+static ssize_t bc_resource_show(struct container *cont, struct cftype *cft,
+    struct file *file, char __user *userbuf,
+    size_t nbytes, loff_t *ppos)
+{
+ struct beancounter *bc = container_bc(cont);
+ int idx = cft->private;
+ struct bc_resource *res = container_of(cft, struct bc_resource,
+            cft_attrs[idx]);
+
+ char *page;
+ ssize_t retval = 0;
+ char *s;
+
+ if (!(page = (char *)__get_free_page(GFP_KERNEL)))
+  return -ENOMEM;
+
+ s = page;
+
+ switch(idx) {
+ case BC_RES_HELD:
+  s += sprintf(page, "%lu\n", bc->bc_parms[res->res_id].held);
+  break;
+ case BC_RES_BARRIER:
+  s += sprintf(page, "%lu\n", bc->bc_parms[res->res_id].barrier);
+  break;
+ case BC_RES_LIMIT:
+  s += sprintf(page, "%lu\n", bc->bc_parms[res->res_id].limit);
+  break;
+ case BC_RES_MAXHELD:
+  s += sprintf(page, "%lu\n", bc->bc_parms[res->res_id].maxheld);
+  break;
+ case BC_RES_MINHELD:
+  s += sprintf(page, "%lu\n", bc->bc_parms[res->res_id].minheld);
+  break;
+ case BC_RES_FAILCNT:
+  s += sprintf(page, "%lu\n", bc->bc_parms[res->res_id].failcnt);
+  break;
+ default:
+  retval = -EINVAL;
+  goto out;
+  break;
+ }
```

```
+
+ retval = simple_read_from_buffer(userbuf, nbytes, ppos, page, s-page);
+ out:
+ free_page((unsigned long) page);
+ return retval;
+}
+
+static ssize_t bc_resource_store(struct container *cont, struct cftype *cft,
+     struct file *file,
+     const char __user *userbuf,
+     size_t nbytes, loff_t *ppos)
+{
+ struct beancounter *bc = container_bc(cont);
+ int idx = cft->private;
+ struct bc_resource *res = container_of(cft, struct bc_resource,
+           cft_attrs[idx]);
+
+ char buffer[64];
+ unsigned long val;
+ char *end;
+ int ret = 0;
+
+ if (nbytes >= sizeof(buffer))
+  return -E2BIG;
+
+ if (copy_from_user(buffer, userbuf, nbytes))
+  return -EFAULT;
+
+ buffer[nbytes] = 0;
+
+ container_lock();
+
+ if (container_is_removed(cont)) {
+  ret = -ENODEV;
+  goto out_unlock;
+ }
+
+ ret = -EINVAL;
+ val = simple_strtoul(buffer, &end, 10);
+ if (*end != '\0')
+  goto out_unlock;
+
+ switch (idx) {
+ case BC_RES_BARRIER:
+  ret = bc_change_param(bc, res->res_id,
+          val, bc->bc_parms[res->res_id].limit);
+  break;
+ case BC_RES_LIMIT:
```

```
+  ret = bc_change_param(bc, res->res_id,
+          bc->bc_parms[res->res_id].barrier, val);
+  break;
+ }
+
+ if (ret == 0)
+  ret = nbytes;
+
+ out_unlock:
+ container_unlock();
+
+ return ret;
+}
+
+
+
+void __init bc_register_resource(int res_id, struct bc_resource *br)
+{
+ int attr;
+ BUG_ON(bc_resources[res_id] != &bc_dummy_res);
+ BUG_ON(res_id >= BC_RESOURCES);
+
+ bc_resources[res_id] = br;
+ br->res_id = res_id;
+
+ for (attr = 0; attr < BC_ATTRS; attr++) {
+
+  /* Construct a file handler for each attribute of this
+   * resource; the name is of the form
+   * "bc.<resname>.<attrname>" */
+
+  struct cftype *cft = &br->cft_attrs[attr];
+  const char *attrname;
+  cft->private = attr;
+  strcpy(cft->name, "bc.");
+  strcat(cft->name, br->bcr_name);
+  strcat(cft->name, ".");
+  switch (attr) {
+  case BC_RES_HELD:
+   attrname = "held"; break;
+  case BC_RES_BARRIER:
+   attrname = "barrier"; break;
+  case BC_RES_LIMIT:
+   attrname = "limit"; break;
+  case BC_RES_MAXHELD:
+   attrname = "maxheld"; break;
+  case BC_RES_MINHELD:
+   attrname = "minheld"; break;
```

```
+  case BC_RES_FAILCNT:
+   attrname = "failcnt"; break;
+  default:
+   BUG();
+  }
+  strcat(cft->name, attrname);
+  cft->read = bc_resource_show;
+  cft->write = bc_resource_store;
+ }
+}
+
+void __init bc_init_early(void)
+{
+ int i;
+
+ spin_lock_init(&init_bc.bc_lock);
+
+ for (i = 0; i < BC_RESOURCES; i++) {
+  init_bc.bc_parms[i].barrier = BC_MAXVALUE;
+  init_bc.bc_parms[i].limit = BC_MAXVALUE;
+
+ }
+}
+
+int __init bc_init_late(void)
+{
+ bc_cache = kmem_cache_create("beancounters",
+   sizeof(struct beancounter), 0,
+   SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL, NULL);
+ return 0;
+}
+
+__initcall(bc_init_late);
+
+static int bc_populate(struct container_subsys *ss, struct container *cont)
+{
+ int err;
+ int attr, res;
+ for (res = 0; res < BC_RESOURCES; res++) {
+  struct bc_resource *bcr = bc_resources[res];
+
+  for (attr = 0; attr < BC_ATTRS; attr++) {
+   struct cftype *cft = &bcr->cft_attrs[attr];
+   if (!cft->name[0]) continue;
+   err = container_add_file(cont, cft);
+   if (err < 0) return err;
+  }
+ }
```

```
+ return 0;
+}
+
+struct container_subsys bc_subsys = {
+ .name = "bc",
+ .create = bc_create,
+ .destroy = bc_destroy,
+ .populate = bc_populate,
+ .subsys_id = bc_subsys_id,
+ .early_init = 1,
+};
+
+EXPORT_SYMBOL(bc_resources);
+EXPORT_SYMBOL(init_bc);
+EXPORT_SYMBOL(bc_change_param);
Index: container-2.6.20-new/include/bc/misc.h
 ===================================================================== =======
--- /dev/null
+++ container-2.6.20-new/include/bc/misc.h
@@ -0,0 +1,27 @@
+/*
+ * include/bc/misc.h
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#ifndef __BC_MISC_H__
+#define __BC_MISC_H__
+
+struct file;
+
+#ifdef CONFIG_BEANCOUNTERS
+int __must_check bc_file_charge(struct file *);
+void bc_file_uncharge(struct file *);
+#else
+static inline int __must_check bc_file_charge(struct file *f)
+{
+ return 0;
+}
+
+static inline void bc_file_uncharge(struct file *f)
+{
+}
+#endif
+
+#endif
Index: container-2.6.20-new/kernel/bc/misc.c
```

```
 ========================================================== =======
--- /dev/null
+++ container-2.6.20-new/kernel/bc/misc.c
@@ -0,0 +1,57 @@
+
+#include <linux/fs.h>
+#include <bc/beancounter.h>
+#include <bc/task.h>
+
+int bc_file_charge(struct file *file)
+{
+ int sev;
+ struct beancounter *bc;
+
+ rcu_read_lock();
+ bc = get_exec_bc();
+ css_get(&bc->css);
+ rcu_read_unlock();
+
+ sev = (capable(CAP_SYS_ADMIN) ? BC_LIMIT : BC_BARRIER);
+
+ if (bc_charge(bc, BC_NUMFILES, 1, sev)) {
+  css_put(&bc->css);
+  return -EMFILE;
+ }
+
+ file->f_bc = bc;
+ return 0;
+}
+
+void bc_file_uncharge(struct file *file)
+{
+ struct beancounter *bc;
+
+ bc = file->f_bc;
+ bc_uncharge(bc, BC_NUMFILES, 1);
+ css_put(&bc->css);
+}
+
+#define BC_NUMFILES_BARRIER 256
+#define BC_NUMFILES_LIMIT 512
+
+static int bc_files_init(struct beancounter *bc, int i)
+{
+ bc_init_resource(&bc->bc_parms[BC_NUMFILES],
+   BC_NUMFILES_BARRIER, BC_NUMFILES_LIMIT);
+ return 0;
+}
```

```
+
+static struct bc_resource bc_files_resource = {
+ .bcr_name = "numfiles",
+ .bcr_init = bc_files_init,
+};
+
+static int __init bc_misc_init_resource(void)
+{
+ bc_register_resource(BC_NUMFILES, &bc_files_resource);
+  return 0;
+}
+
+__initcall(bc_misc_init_resource);
Index: container-2.6.20-new/fs/file_table.c
 ===================================================================== =======
--- container-2.6.20-new.orig/fs/file_table.c
+++ container-2.6.20-new/fs/file_table.c
@@ -22,6 +22,8 @@
 #include <linux/sysctl.h>
 #include <linux/percpu_counter.h>

+#include <bc/misc.h>
+
 #include <asm/atomic.h>

 /* sysctl tunables... */
@@ -43,6 +45,7 @@ static inline void file_free_rcu(struct
 static inline void file_free(struct file *f)
 {
  percpu_counter_dec(&nr_files);
+ bc_file_uncharge(f);
  call_rcu(&f->f_u.fu_rcuhead, file_free_rcu);
 }

@@ -107,8 +110,10 @@ struct file *get_empty_filp(void)
  if (f == NULL)
   goto fail;

- percpu_counter_inc(&nr_files);
  memset(f, 0, sizeof(*f));
+ if (bc_file_charge(f))
+  goto fail_charge;
+ percpu_counter_inc(&nr_files);
  if (security_file_alloc(f))
   goto fail_sec;

@@ -135,6 +140,10 @@ fail_sec:
  file_free(f);
```

```
 fail:
  return NULL;
+
+ fail_charge:
+ kmem_cache_free(filp_cachep, f);
+ return NULL;
 }

 EXPORT_SYMBOL(get_empty_filp);
Index: container-2.6.20-new/include/linux/fs.h
 ===================================================================
--- container-2.6.20-new.orig/include/linux/fs.h
+++ container-2.6.20-new/include/linux/fs.h
@@ -739,6 +739,9 @@ struct file {
  spinlock_t  f_ep_lock;
 #endif /* #ifdef CONFIG_EPOLL */
  struct address_space *f_mapping;
+#ifdef CONFIG_BEANCOUNTERS
+ struct beancounter      *f_bc;
+#endif
 };
 extern spinlock_t files_lock;
 #define file_list_lock() spin_lock(&files_lock);
Index: container-2.6.20-new/include/bc/task.h
 ===================================================================
--- /dev/null
+++ container-2.6.20-new/include/bc/task.h
@@ -0,0 +1,64 @@
+/*
+ * include/bc/task.h
+ *
+ * Copyright (C) 2007 OpenVZ SWsoft Inc
+ * Adapted by Paul Menage <menage@google.com> for generic containers
+ *
+ */
+
+#ifndef __BC_TASK_H__
+#define __BC_TASK_H__
+
+#include <linux/container.h>
+#include <linux/hardirq.h>
+#include <bc/beancounter.h>
+
+struct beancounter;
+struct task_struct;
+
+#ifdef CONFIG_BEANCOUNTERS
+extern struct beancounter init_bc;
```

```
+
+extern struct container_subsys bc_subsys;
+
+
+/*
+ * Caller must be in rcu_read safe section or hold task_lock(current)
+ */
+
+static inline struct beancounter *get_exec_bc(void)
+{
+	struct task_struct *tsk;
+
+	if (in_irq())
+		return &init_bc;
+
+	tsk = current;
+	if (tsk->tmp_exec_bc != NULL)
+		return tsk->tmp_exec_bc;
+
+	return task_bc(tsk);
+}
+
+#define set_exec_bc(bc)		({				\
+		struct task_struct *t;			\
+		struct beancounter *old;		\
+		t = current;				\
+		old = t->tmp_exec_bc;			\
+		t->tmp_exec_bc = bc;			\
+		old;					\
+	})
+
+#define reset_exec_bc(old, expected)	do {		\
+		struct task_struct *t;			\
+		t = current;				\
+		BUG_ON(t->tmp_exec_bc != expected);	\
+		t->tmp_exec_bc = old;			\
+	} while (0)
+
+#else
+
+#define set_exec_bc(bc)		(NULL)
+#define reset_exec_bc(bc, exp) do { } while (0)
+#endif
+#endif
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
```

```
@@ -1052,6 +1052,9 @@ struct task_struct {
 #ifdef CONFIG_FAULT_INJECTION
  int make_it_fail;
 #endif
+#ifdef CONFIG_BEANCOUNTERS
+ struct beancounter *tmp_exec_bc;
+#endif
 };

 static inline pid_t process_group(struct task_struct *tsk)
Index: container-2.6.20-new/kernel/softirq.c
 ===================================================================== =======
--- container-2.6.20-new.orig/kernel/softirq.c
+++ container-2.6.20-new/kernel/softirq.c
@@ -18,6 +18,8 @@
 #include <linux/rcupdate.h>
 #include <linux/smp.h>

+#include <bc/task.h>
+
 #include <asm/irq.h>
 /*
    - No shared variables, all the data are CPU local.
@@ -209,6 +211,7 @@ asmlinkage void __do_softirq(void)
  __u32 pending;
  int max_restart = MAX_SOFTIRQ_RESTART;
  int cpu;
+ struct beancounter *bc;

  pending = local_softirq_pending();
  account_system_vtime(current);
@@ -225,6 +228,8 @@ restart:

  h = softirq_vec;

+ bc = set_exec_bc(&init_bc);
+
  do {
   if (pending & 1) {
    h->action(h);
@@ -234,6 +239,8 @@ restart:
   pending >>= 1;
  } while (pending);

+ reset_exec_bc(bc, &init_bc);
+
  local_irq_disable();
```

```
  pending = local_softirq_pending();
Index: container-2.6.20-new/include/linux/container_subsys.h
 ===================================================== =======
--- container-2.6.20-new.orig/include/linux/container_subsys.h
+++ container-2.6.20-new/include/linux/container_subsys.h
@@ -23,4 +23,10 @@ SUBSYS(numtasks)

 /* */

+#ifdef CONFIG_BEANCOUNTERS
+SUBSYS(bc)
+#endif
+
+/* */
+
 /* */

--
```

---

Subject: [PATCH 7/7] Containers (V8): Container interface to nsproxy subsystem
Posted by Paul Menage on Fri, 06 Apr 2007 23:32:28 GMT
View Forum Message <> Reply to Message

This is intended as a simple illustration of how a virtual server system could be integrated with generic containers, and hence take advantage of other resource-control efforts. A real implementation would probably allow parameters such as configuring what kinds of namespace creations triggered new containers, etc.

When a task enters a new namespace via a clone() or unshare(), a new container is created and the task moves into it. Developed by Serge Hallyn <serue@us.ibm.com>, adapted by Paul Menage <menage@google.com>

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

```
---
 include/linux/container_subsys.h |    6 ++
 include/linux/nsproxy.h          |    6 ++
 init/Kconfig                     |    9 +++
 kernel/Makefile                  |    1
 kernel/fork.c                    |    4 +
 kernel/ns_container.c            |   99 +++++++++++++++++++++++++++++++++++++++
 kernel/nsproxy.c                 |    6 ++
 7 files changed, 131 insertions(+)

Index: container-2.6.20-new/include/linux/nsproxy.h
 ===================================================== =======
```

```
--- container-2.6.20-new.orig/include/linux/nsproxy.h
+++ container-2.6.20-new/include/linux/nsproxy.h
@@ -53,4 +53,10 @@ static inline void exit_task_namespaces(
   put_nsproxy(ns);
  }
 }
+#ifdef CONFIG_CONTAINER_NS
+int ns_container_clone(struct task_struct *tsk);
+#else
+static inline int ns_container_clone(struct task_struct *tsk) { return 0; }
+#endif
+
 #endif
Index: container-2.6.20-new/kernel/Makefile
 ===================================================================
--- container-2.6.20-new.orig/kernel/Makefile
+++ container-2.6.20-new/kernel/Makefile
@@ -39,6 +39,7 @@ obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
+obj-$(CONFIG_CONTAINER_NS) += ns_container.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
 obj-$(CONFIG_AUDIT) += audit.o auditfilter.o
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
@@ -1668,6 +1668,9 @@ asmlinkage long sys_unshare(unsigned lon
    err = -ENOMEM;
    goto bad_unshare_cleanup_ipc;
   }
+  err = ns_container_clone(current);
+  if (err)
+   goto bad_unshare_cleanup_dupns;
  }

  if (new_fs || new_ns || new_mm || new_fd || new_ulist ||
@@ -1722,6 +1725,7 @@ asmlinkage long sys_unshare(unsigned lon
   task_unlock(current);
  }

+bad_unshare_cleanup_dupns:
  if (new_nsproxy)
   put_nsproxy(new_nsproxy);

Index: container-2.6.20-new/kernel/ns_container.c
```

```
 ============================================================ =======
--- /dev/null
+++ container-2.6.20-new/kernel/ns_container.c
@@ -0,0 +1,99 @@
+/*
+ * ns_container.c - namespace container subsystem
+ *
+ * Copyright IBM, 2006
+ */
+
+#include <linux/module.h>
+#include <linux/container.h>
+#include <linux/fs.h>
+
+struct nscont {
+ struct container_subsys_state css;
+ spinlock_t lock;
+};
+
+struct container_subsys ns_subsys;
+
+static inline struct nscont *container_nscont(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, ns_subsys_id),
+       struct nscont, css);
+}
+
+int ns_container_clone(struct task_struct *tsk)
+{
+ return container_clone(tsk, &ns_subsys);
+}
+
+/*
+ * Rules:
+ *   1. you can only enter a container which is a child of your current
+ *      container
+ *   2. you can only place another process into a container if
+ *      a. you have CAP_SYS_ADMIN
+ *      b. your container is an ancestor of tsk's destination container
+ *        (hence either you are in the same container as tsk, or in an
+ *         ancestor container thereof)
+ */
+int ns_can_attach(struct container_subsys *ss,
+     struct container *cont, struct task_struct *tsk)
+{
+ struct container *c;
+
+ if (current != tsk) {
```

```
+  if (!capable(CAP_SYS_ADMIN))
+   return -EPERM;
+
+  if (!container_is_descendant(cont))
+   return -EPERM;
+ }
+
+ if (container_task_count(cont) != 0)
+  return -EPERM;
+
+ c = task_container(tsk, ns_subsys_id);
+ if (c && c != cont->parent)
+  return -EPERM;
+
+ return 0;
+}
+
+/*
+ * Rules: you can only create a container if
+ *     1. you are capable(CAP_SYS_ADMIN)
+ *     2. the target container is a descendant of your own container
+ */
+static int ns_create(struct container_subsys *ss, struct container *cont)
+{
+ struct nscont *ns;
+
+ if (!capable(CAP_SYS_ADMIN))
+  return -EPERM;
+ if (cont->parent && !container_is_descendant(cont))
+  return -EPERM;
+
+ ns = kzalloc(sizeof(*ns), GFP_KERNEL);
+ if (!ns) return -ENOMEM;
+ spin_lock_init(&ns->lock);
+ cont->subsys[ns_subsys.subsys_id] = &ns->css;
+ return 0;
+}
+
+static void ns_destroy(struct container_subsys *ss,
+         struct container *cont)
+{
+ struct nscont *ns = container_nscont(cont);
+ kfree(ns);
+}
+
+struct container_subsys ns_subsys = {
+ .name = "ns",
+ .create = ns_create,
```

```
+ .destroy  = ns_destroy,
+ .can_attach = ns_can_attach,
+ //.attach = ns_attach,
+ //.post_attach = ns_post_attach,
+ //.populate = ns_populate,
+ .subsys_id = ns_subsys_id,
+};
Index: container-2.6.20-new/kernel/nsproxy.c
 ===================================================================
--- container-2.6.20-new.orig/kernel/nsproxy.c
+++ container-2.6.20-new/kernel/nsproxy.c
@@ -116,10 +116,16 @@ int copy_namespaces(int flags, struct ta
  if (err)
   goto out_pid;

+ err = ns_container_clone(tsk);
+ if (err)
+  goto out_container;
 out:
  put_nsproxy(old_ns);
  return err;

+ out_container:
+ if (new_ns->pid_ns)
+  put_pid_ns(new_ns->pid_ns);
 out_pid:
  if (new_ns->ipc_ns)
   put_ipc_ns(new_ns->ipc_ns);
Index: container-2.6.20-new/include/linux/container_subsys.h
 ===================================================================
--- container-2.6.20-new.orig/include/linux/container_subsys.h
+++ container-2.6.20-new/include/linux/container_subsys.h
@@ -29,4 +29,10 @@ SUBSYS(bc)

 /* */

+#ifdef CONFIG_CONTAINER_NS
+SUBSYS(ns)
+#endif
+
+/* */
+
 /* */
Index: container-2.6.20-new/init/Kconfig
 ===================================================================
--- container-2.6.20-new.orig/init/Kconfig
+++ container-2.6.20-new/init/Kconfig
@@ -285,6 +285,15 @@ config CONTAINER_CPUACCT
```

```
   Provides a simple Resource Controller for monitoring the
   total CPU consumed by the tasks in a container

+config CONTAINER_NS
+       bool "Namespace container subsystem"
+       select CONTAINERS
+       help
+         Provides a simple namespace container subsystem to
+         provide hierarchical naming of sets of namespaces,
+         for instance virtual servers and checkpoint/restart
+         jobs.
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help

--
```

---

Subject: Re: [PATCH 6/7] Containers (V8): BeanCounters over generic process containers
Posted by xemul on Mon, 09 Apr 2007 07:43:07 GMT
View Forum Message <> Reply to Message

menage@google.com wrote:
> This patch implements the BeanCounter resource control abstraction
> over generic process containers. It contains the beancounter core
> code, plus the numfiles resource counter. It doesn't currently contain
> any of the memory tracking code or the code for switching beancounter
> context in interrupts.
>
> Currently all the beancounters resource counters are lumped into a
> single hierarchy; ideally it would be possible for each resource
> counter to be a separate container subsystem, allowing them to be
> connected to different hierarchies.

Thanks for your attention, but since we have decided to build a
"new" resource controllers above your containers infrastructure,
I think there is no need in further BC code porting.

Thanks,
Pavel.

---

Subject: Re: [PATCH 6/7] Containers (V8): BeanCounters over generic process containers
Posted by William Lee Irwin III on Mon, 09 Apr 2007 16:44:34 GMT

View Forum Message <> Reply to Message

On Fri, Apr 06, 2007 at 04:32:27PM -0700, menage@google.com wrote:
> This patch implements the BeanCounter resource control abstraction
> over generic process containers. It contains the beancounter core
> code, plus the numfiles resource counter. It doesn't currently contain
> any of the memory tracking code or the code for switching beancounter
> context in interrupts.
> Currently all the beancounters resource counters are lumped into a
> single hierarchy; ideally it would be possible for each resource
> counter to be a separate container subsystem, allowing them to be

I'm very happy to see the BeanCounter work revived. The utility of
this work should extend far beyond feature work like containers and
into stability and reliability areas as well.

-- wli

---

Subject: Re: [PATCH 4/7] Containers (V8): Simple CPU accounting container subsystem
Posted by Srivatsa Vaddagiri on Tue, 10 Apr 2007 13:16:42 GMT
View Forum Message <> Reply to Message

On Fri, Apr 06, 2007 at 04:32:25PM -0700, menage@google.com wrote:
> +struct container_subsys cpuacct_subsys = {
> + .name = "cpuacct",
> + .create = cpuacct_create,
> + .destroy = cpuacct_destroy,
> + .populate = cpuacct_populate,
> + .subsys_id = cpuacct_subsys_id,
> +};

container_register_subsys(&cpuacct_subsys) is missing ..

--
Regards,
vatsa

---

Subject: Re: [PATCH 2/7] Containers (V8): Cpusets hooked into containers
Posted by Vaidyanathan Srinivas on Mon, 23 Apr 2007 10:52:49 GMT
View Forum Message <> Reply to Message

menage@google.com wrote:
> This patch removes the process grouping code from the cpusets code,
> instead hooking it into the generic container system. This temporarily

> adds cpuset-specific code in kernel/container.c, which is removed by
> the next patch in the series.
>
> Signed-off-by: Paul Menage <menage@google.com>
>

[snip]

> Index: container-2.6.20-new/init/Kconfig
> ===================================================== =======
> --- container-2.6.20-new.orig/init/Kconfig
> +++ container-2.6.20-new/init/Kconfig
> @@ -239,17 +239,12 @@ config IKCONFIG_PROC
>  through /proc/config.gz.
>
>  config CONTAINERS
> - bool "Container support"
> - help
> -  This option will let you create and manage process containers,
> -  which can be used to aggregate multiple processes, e.g. for
> -  the purposes of resource tracking.
> -
> -  Say N if unsure
> + bool

Hi Paul,

This looks like some patch generation error.  Description for
containers should not be removed after applying this patch.

--Vaidy

>
>  config CPUSETS
>  bool "Cpuset support"
>  depends on SMP
> + select CONTAINERS
>  help
>  This option will let you create and manage CPUSETs which
>  allow dynamically partitioning a system into sets of CPUs and
> @@ -278,6 +273,11 @@ config SYSFS_DEPRECATED
>  If you are using a distro that was released in 2006 or later,
>  it should be safe to say N here.
>
> +config PROC_PID_CPUSET
> + bool "Include legacy /proc/<pid>/cpuset file"
> + depends on CPUSETS
> + default y

> +
> config RELAY
> bool "Kernel->user space relay support (formerly relayfs)"
> help

[snip]

---

Subject: Re: [PATCH 0/7] Containers (V8): Generic Process Containers
Posted by Vaidyanathan Srinivas on Mon, 23 Apr 2007 11:07:33 GMT
View Forum Message <> Reply to Message

Hi Paul,

In [patch 3/7] Containers (V8): Add generic multi-subsystem API to containers, you have forcefully enabled interrupt in container_init_subsys() with spin_unlock_irq() which breaks on PPC64.

> +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);

Interrupt gets enabled here and on PPC64, the kernel takes a pending decrementer and crashes because it is too early to handle them.

Use of irqsave and restore routines would fix the problem.
I have included the fix along with minor Kconfig correction.

Also your 3/7 patch did not showup on LKML.

--Vaidy

> +
> + need_forkexit_callback |= ss->fork || ss->exit;

---

Subject: Re: [PATCH 2/7] Containers (V8): Cpusets hooked into containers
Posted by Paul Menage on Wed, 25 Apr 2007 04:59:40 GMT
View Forum Message <> Reply to Message

On 4/23/07, Vaidyanathan Srinivasan <svaidy@linux.vnet.ibm.com> wrote:
> >
> > config CONTAINERS
> > - bool "Container support"
> > - help
> > - This option will let you create and manage process containers,
> > - which can be used to aggregate multiple processes, e.g. for
> > - the purposes of resource tracking.
> > -
> > - Say N if unsure
> > + bool
> >
> Hi Paul,
>
> This looks like some patch generation error. Description for
> containers should not be removed after applying this patch.

No, this is intentional - in the first patch in the series,
CONFIG_CONTAINER was a user-selectable option so it had a description;
in the second it becomes an option that's only selected if other
selected systems (e.g. cpusets) depend on it. So it no longer needs
help text.

Cheers,

Paul

---

Subject: Re: [ckrm-tech] [PATCH 0/7] Containers (V8): Generic Process Containers
Posted by Paul Menage on Wed, 25 Apr 2007 05:04:53 GMT
View Forum Message <> Reply to Message

On 4/23/07, Vaidyanathan Srinivasan <svaidy@linux.vnet.ibm.com> wrote:
> Hi Paul,
>
> In [patch 3/7] Containers (V8): Add generic multi-subsystem API to
> containers, you have forcefully enabled interrupt in
> container_init_subsys() with spin_unlock_irq() which breaks on PPC64.
>
>
> > +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);
>
> Interrupt gets enabled here and on PPC64, the kernel takes a pending
> decrementer and crashes because it is too early to handle them.
>
> Use of irqsave and restore routines would fix the problem.

OK, thanks. I'll add that change.

Paul