
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);
+
+extern 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_mutex, however there are
+ * several performance critical places that need to reference
+ * task->container without the expense of grabbing a system global
+ * mutex. Therefore except as noted below, when dereferencing or, as
+ * in attach_task(), modifying a task's container pointer we use
+ * task_lock(), which acts on a spinlock (task->alloc_lock) already in
+ * the task_struct routinely used for such matters.
+ *
+ * P.S. One more locking exception. RCU is used to guard the

```



```

+ * update of a tasks container pointer by attach_task()
+ */
+
+static DEFINE_MUTEX(container_mutex);
+
+/*
+ * 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 kcalloc'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 kcalloc'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 = kcalloc(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;
+}
+
+/*
+ * Attach task specified by pid in 'pidbuf' to container 'cont', possibly
+ * writing the path of the old container in 'ppathbuf' if it needs to be
+ * notified on release.
+ *
+ * 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 cmpgid(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), cmp_pid, 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 dereferencing '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.

+

--