Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Thu, 08 Mar 2007 18:13:53 GMT
View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 11:00:31PM +0530, Srivatsa Vaddagiri wrote:
> > I'd like to see that. I suspect it will be a bit more fiddly than the
> > simple cpu_acct subsystem.
>
> I am almost done with the conversion. And yes cpuset is a beast to
> convert over! Will test and send the patches out tomorrow.

Ok ..I am not in a state yet where I can post the patches to lkml in the usual conventions (breaking down neatly/good documentation etc). But I do have something which seems to work! I could mount cpuset as:

```
mount -t rcfs -ocpuset none cpuset
cd cpuset
mkdir a
cd a
cat tasks # shows nothing
echo 7 > cpus
echo 0 > mems
echo 1 > cpu_exclusive
echo some_pid > tasks
cat tasks # shows some_pid
```

top now shows some_pid running on CPU7, as expected :)

Instead of the usual convention of inlining patches and sending them in separate mails, I am sending all of them as attachments (beware, bugs around!). But this gives you an idea on which direction this is proceeding ..

Todo:

- Introduce refcounting of resource objects (get/put_res_ns)
- rmdir needs to check resource object refcount rather than nsproxy's
- Trace couple of other lockdep warnings I have hit

Patches attached.

--
Regards,
vatsa

```
 linux-2.6.20-vatsa/include/linux/init_task.h |   11
 linux-2.6.20-vatsa/include/linux/nsproxy.h   |   11
 linux-2.6.20-vatsa/init/Kconfig              |   22
 linux-2.6.20-vatsa/init/main.c               |    3
 linux-2.6.20-vatsa/kernel/Makefile           |    1
```

```
 linux-2.6.20.1-vatsa/include/linux/init_task.h |   11
 linux-2.6.20.1-vatsa/include/linux/nsproxy.h   |   11
 linux-2.6.20.1-vatsa/include/linux/rcfs.h      |   76 +
 linux-2.6.20.1-vatsa/init/Kconfig              |   22
 linux-2.6.20.1-vatsa/init/main.c               |    3
 linux-2.6.20.1-vatsa/kernel/Makefile           |    1
 linux-2.6.20.1-vatsa/kernel/nsproxy.c          |   65 +
 linux-2.6.20.1-vatsa/kernel/rcfs.c             | 1202 +++++++++++++++++++++++++
 8 files changed, 1391 insertions(+)
```

```
diff -puN include/linux/init_task.h~rcfs include/linux/init_task.h
--- linux-2.6.20.1/include/linux/init_task.h~rcfs 2007-03-08 21:21:33.000000000 +0530
+++ linux-2.6.20.1-vatsa/include/linux/init_task.h 2007-03-08 21:21:34.000000000 +0530
@@ -71,6 +71,16 @@
 }

 extern struct nsproxy init_nsproxy;
+
+#ifdef CONFIG_RCFS
+#define INIT_RCFS(nsproxy)       \
+ .list   = LIST_HEAD_INIT(nsproxy.list),   \
+ .ctlr_data  = {[ 0 ... CONFIG_MAX_RC_SUBSYS-1 ] = NULL },
+#else
+#define INIT_RCFS(nsproxy)
+#endif
+
+
 #define INIT_NSPROXY(nsproxy) {      \
  .pid_ns  = &init_pid_ns,     \
  .count  = ATOMIC_INIT(1),    \
@@ -78,6 +88,7 @@ extern struct nsproxy init_nsproxy;
  .uts_ns  = &init_uts_ns,     \
  .mnt_ns  = NULL,      \
  INIT_IPC_NS(ipc_ns)      \
+ INIT_RCFS(nsproxy)      \
 }
```

```
 #define INIT_SIGHAND(sighand) {      \
diff -puN include/linux/nsproxy.h~rcfs include/linux/nsproxy.h
--- linux-2.6.20.1/include/linux/nsproxy.h~rcfs 2007-03-08 21:21:33.000000000 +0530
+++ linux-2.6.20.1-vatsa/include/linux/nsproxy.h 2007-03-08 21:21:34.000000000 +0530
@@ -28,6 +28,10 @@ struct nsproxy {
  struct ipc_namespace *ipc_ns;
  struct mnt_namespace *mnt_ns;
  struct pid_namespace *pid_ns;
+#ifdef CONFIG_RCFS
+ struct list_head list;
+ void *ctlr_data[CONFIG_MAX_RC_SUBSYS];
+#endif
 };
 extern struct nsproxy init_nsproxy;

@@ -35,6 +39,13 @@ struct nsproxy *dup_namespaces(struct ns
 int copy_namespaces(int flags, struct task_struct *tsk);
 void get_task_namespaces(struct task_struct *tsk);
 void free_nsproxy(struct nsproxy *ns);
+#ifdef CONFIG_RCFS
+struct nsproxy *find_nsproxy(struct nsproxy *ns);
+int namespaces_init(void);
+int nsproxy_task_count(void *data, int idx);
+#else
+static inline int namespaces_init(void) { return 0;}
+#endif

 static inline void put_nsproxy(struct nsproxy *ns)
 {
diff -puN /dev/null include/linux/rcfs.h
--- /dev/null 2007-03-08 22:46:54.325490448 +0530
+++ linux-2.6.20.1-vatsa/include/linux/rcfs.h 2007-03-08 21:21:34.000000000 +0530
@@ -0,0 +1,76 @@
+#ifndef _LINUX_RCFS_H
+#define _LINUX_RCFS_H
+
+#ifdef CONFIG_RCFS
+
+/* 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;
+#define MAX_CFTYPE_NAME 64
+struct cftype {
+ /* 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 nsproxy *ns, struct cftype *cft,
+    struct file *file,
+    char __user *buf, size_t nbytes, loff_t *ppos);
+ ssize_t (*write) (struct nsproxy *ns, struct cftype *cft,
+     struct file *file,
+     const char __user *buf, size_t nbytes, loff_t *ppos);
+ int (*release) (struct inode *inode, struct file *file);
+};
+
+/* resource control subsystem type. See Documentation/rcfs.txt for details */
+
+struct rc_subsys {
+ int (*create)(struct rc_subsys *ss, struct nsproxy *ns,
+    struct nsproxy *parent);
+ void (*destroy)(struct rc_subsys *ss, struct nsproxy *ns);
+ int (*can_attach)(struct rc_subsys *ss, struct nsproxy *ns,
+     struct task_struct *tsk);
+ void (*attach)(struct rc_subsys *ss, struct nsproxy *new,
+     struct nsproxy *old, struct task_struct *tsk);
+ int (*populate)(struct rc_subsys *ss, struct dentry *d);
+ int subsys_id;
+ int active;
+
+#define MAX_CONTAINER_TYPE_NAMELEN 32
+ const char *name;
+
+ /* Protected by RCU */
+ int hierarchy;
+
+ struct list_head sibling;
+};
+
+int rc_register_subsys(struct rc_subsys *subsys);
+/* Add a new file to the given container directory. Should only be
+ * called by subsystems from within a populate() method */
+int rcfs_add_file(struct dentry *d, const struct cftype *cft);
+extern int rcfs_init(void);
```

```
+extern void rcfs_manage_lock(void);
+extern void rcfs_manage_unlock(void);
+extern int rcfs_dir_removed(struct dentry *d);
+extern int rcfs_path(struct dentry *d, char *buf, int len);
+
+#else
+
+static inline int rcfs_init(void) { return 0; }
+
+#endif
+
+
+#endif
diff -puN init/Kconfig~rcfs init/Kconfig
--- linux-2.6.20.1/init/Kconfig~rcfs 2007-03-08 21:21:33.000000000 +0530
+++ linux-2.6.20.1-vatsa/init/Kconfig 2007-03-08 22:47:50.000000000 +0530
@@ -238,6 +238,28 @@ config IKCONFIG_PROC
    This option enables access to the kernel configuration file
    through /proc/config.gz.

+config RCFS
+ bool "Resource control file system support"
+ default n
+ help
+   This option will let you create and manage resource containers,
+   which can be used to aggregate multiple processes, e.g. for
+   the purposes of resource tracking.
+
+   Say N if unsure
+
+config MAX_RC_SUBSYS
+       int "Number of resource control subsystems to support"
+       depends on RCFS
+       range 1 255
+       default 8
+
+config MAX_RC_HIERARCHIES
+       int "Number of rcfs hierarchies to support"
+       depends on RCFS
+       range 2 255
+       default 4
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
diff -puN init/main.c~rcfs init/main.c
--- linux-2.6.20.1/init/main.c~rcfs 2007-03-08 21:21:33.000000000 +0530
+++ linux-2.6.20.1-vatsa/init/main.c 2007-03-08 21:21:34.000000000 +0530
```

```
@@ -52,6 +52,7 @@
 #include <linux/lockdep.h>
 #include <linux/pid_namespace.h>
 #include <linux/device.h>
+#include <linux/rcfs.h>

 #include <asm/io.h>
 #include <asm/bugs.h>
@@ -512,6 +513,7 @@ asmlinkage void __init start_kernel(void
  setup_per_cpu_areas();
  smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */

+ namespaces_init();
  /*
   * Set up the scheduler prior starting any interrupts (such as the
   * timer interrupt). Full topology setup happens at smp_init()
@@ -578,6 +580,7 @@ asmlinkage void __init start_kernel(void
  }
 #endif
  vfs_caches_init_early();
+ rcfs_init();
  cpuset_init_early();
  mem_init();
  kmem_cache_init();
diff -puN kernel/Makefile~rcfs kernel/Makefile
--- linux-2.6.20.1/kernel/Makefile~rcfs 2007-03-08 21:21:34.000000000 +0530
+++ linux-2.6.20.1-vatsa/kernel/Makefile 2007-03-08 22:47:50.000000000 +0530
@@ -50,6 +50,7 @@ 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_RCFS) += rcfs.o

 ifneq ($(CONFIG_SCHED_NO_NO_OMIT_FRAME_POINTER),y)
 # According to Alan Modra <alan@linuxcare.com.au>, the -fno-omit-frame-pointer is
diff -puN kernel/nsproxy.c~rcfs kernel/nsproxy.c
--- linux-2.6.20.1/kernel/nsproxy.c~rcfs 2007-03-08 21:21:34.000000000 +0530
+++ linux-2.6.20.1-vatsa/kernel/nsproxy.c 2007-03-08 22:54:04.000000000 +0530
@@ -23,6 +23,11 @@

 struct nsproxy init_nsproxy = INIT_NSPROXY(init_nsproxy);

+#ifdef CONFIG_RCFS
+static LIST_HEAD(nslisthead);
+static DEFINE_SPINLOCK(nslistlock);
+#endif
+
 static inline void get_nsproxy(struct nsproxy *ns)
```

```
 {
  atomic_inc(&ns->count);
@@ -71,6 +76,12 @@ struct nsproxy *dup_namespaces(struct ns
    get_pid_ns(ns->pid_ns);
  }

+#ifdef CONFIG_RCFS
+ spin_lock_irq(&nslistlock);
+ list_add(&ns->list, &nslisthead);
+ spin_unlock_irq(&nslistlock);
+#endif
+
  return ns;
 }

@@ -145,5 +156,59 @@ void free_nsproxy(struct nsproxy *ns)
   put_ipc_ns(ns->ipc_ns);
  if (ns->pid_ns)
   put_pid_ns(ns->pid_ns);
+#ifdef CONFIG_RCFS
+ spin_lock_irq(&nslistlock);
+ list_del(&ns->list);
+ spin_unlock_irq(&nslistlock);
+#endif
  kfree(ns);
 }
+
+#ifdef CONFIG_RCFS
+struct nsproxy *find_nsproxy(struct nsproxy *target)
+{
+ struct nsproxy *ns;
+ int i = 0;
+
+ spin_lock_irq(&nslistlock);
+ list_for_each_entry(ns, &nslisthead, list) {
+  for (i= 0; i < CONFIG_MAX_RC_SUBSYS; ++i)
+   if (ns->ctlr_data[i] != target->ctlr_data[i])
+    break;
+
+  if (i == CONFIG_MAX_RC_SUBSYS) {
+   /* Found a hit */
+   get_nsproxy(ns);
+   spin_unlock(&nslistlock);
+   return ns;
+  }
+ }
+
+ spin_unlock_irq(&nslistlock);
```

```
+
+ ns = dup_namespaces(target);
+ return ns;
+}
+
+int __init namespaces_init(void)
+{
+ list_add(&init_nsproxy.list, &nslisthead);
+
+ return 0;
+}
+
+int nsproxy_task_count(void *data, int idx)
+{
+ int count = 0;
+ struct nsproxy *ns;
+ unsigned long flags;
+
+ spin_lock_irqsave(&nslistlock, flags);
+ list_for_each_entry(ns, &nslisthead, list)
+  if (ns->ctlr_data[idx] == data)
+   count += atomic_read(&ns->count);
+ spin_unlock_irqrestore(&nslistlock, flags);
+
+ return count;
+}
+#endif
diff -puN /dev/null kernel/rcfs.c
--- /dev/null 2007-03-08 22:46:54.325490448 +0530
+++ linux-2.6.20.1-vatsa/kernel/rcfs.c 2007-03-08 22:35:23.000000000 +0530
@@ -0,0 +1,1202 @@
+/*
+ *  kernel/rcfs.c
+ *
+ *  Generic resource container 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/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 <linux/nsproxy.h>
+#include <linux/rcfs.h>
+
+#include <asm/uaccess.h>
+#include <asm/atomic.h>
+#include <linux/mutex.h>
+
+#define RCFS_SUPER_MAGIC          0x27e0eb
+
```

```
+/* A rcfs_root represents the root of a resource control hierarchy,
+ * and may be associated with a superblock to form an active
+ * hierarchy */
+struct rcfs_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;
+};
+
+static DEFINE_MUTEX(manage_mutex);
+
+/* The set of hierarchies in use */
+static struct rcfs_root rootnode[CONFIG_MAX_RC_HIERARCHIES];
+
+static struct rc_subsys *subsys[CONFIG_MAX_RC_SUBSYS];
+static int subsys_count = 0;
+
+/* for_each_subsys() allows you to act on each subsystem attached to
+ * an active hierarchy */
+#define for_each_subsys(root, _ss) \
+list_for_each_entry(_ss, &root->subsys_list, sibling)
+
+/* Does a container directory have sub-directories under it ? */
+static int dir_empty(struct dentry *dentry)
+{
+ struct dentry *d;
+ int rc = 1;
+
+ spin_lock(&dcache_lock);
+ list_for_each_entry(d, &dentry->d_subdirs, d_u.d_child) {
+  if (S_ISDIR(d->d_inode->i_mode)) {
+   rc = 0;
+   break;
+  }
+ }
+ spin_unlock(&dcache_lock);
+
+ return rc;
+}
+
+static int rebind_subsystems(struct rcfs_root *root, unsigned long final_bits)
+{
+ unsigned long added_bits, removed_bits;
+ int i, hierarchy;
```

```
+
+ 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 < subsys_count; i++) {
+  unsigned long long bit = 1ull << i;
+  struct rc_subsys *ss = subsys[i];
+
+  if (!(bit & added_bits))
+   continue;
+  if (ss->hierarchy != 0) {
+   /* Subsystem isn't free */
+   return -EBUSY;
+  }
+ }
+
+ /* Currently we don't handle adding/removing subsystems when
+  * any subdirectories exist. This is theoretically supportable
+  * but involves complex erro r handling, so it's being left until
+  * later */
+ /*
+ if (!dir_empty(root->sb->s_root))
+  return -EBUSY;
+ */
+
+ hierarchy = rootnode - root;
+
+ /* Process each subsystem */
+ for (i = 0; i < subsys_count; i++) {
+  struct rc_subsys *ss = subsys[i];
+  unsigned long bit = 1UL << i;
+  if (bit & added_bits) {
+   /* We're binding this subsystem to this hierarchy */
+   list_add(&ss->sibling, &root->subsys_list);
+   rcu_assign_pointer(ss->hierarchy, hierarchy);
+  } else if (bit & removed_bits) {
+   /* We're removing this subsystem */
+   rcu_assign_pointer(subsys[i]->hierarchy, 0);
+   list_del(&ss->sibling);
+  }
+ }
+ root->subsys_bits = final_bits;
+ synchronize_rcu(); /* needed ? */
+
+ 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 rcfs_put_super(struct super_block *sb) {
+
+ struct rcfs_root *root = sb->s_fs_info;
+ int ret;
+
+ mutex_lock(&manage_mutex);
+
+ BUG_ON(!root->subsys_bits);
+
+ /* Rebind all subsystems back to the default hierarchy */
+ ret = rebind_subsystems(root, 0);
+ root->sb = NULL;
+ sb->s_fs_info = NULL;
+
+ mutex_unlock(&manage_mutex);
+}
+
+static int rcfs_show_options(struct seq_file *seq, struct vfsmount *vfs)
+{
+ struct rcfs_root *root = vfs->mnt_sb->s_fs_info;
+ struct rc_subsys *ss;
+
+ for_each_subsys(root, ss)
+  seq_printf(seq, ",%s", ss->name);
+
+ return 0;
+}
+
+/* Convert a hierarchy specifier into a bitmask. LL=manage_mutex */
+static int parse_rcfs_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 << subsys_count) - 1;
+  } else {
+   struct rc_subsys *ss;
+   int i;
+   for (i = 0; i < subsys_count; i++) {
```

```
+    ss = subsys[i];
+    if (!strcmp(token, ss->name)) {
+     *bits |= 1 << i;
+     break;
+    }
+   }
+   if (i == subsys_count)
+    return -ENOENT;
+  }
+ }
+
+ /* We can't have an empty hierarchy */
+ if (!*bits)
+  return -EINVAL;
+
+ return 0;
+}
+
+static struct backing_dev_info rcfs_backing_dev_info = {
+ .ra_pages = 0,  /* No readahead */
+ .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
+};
+
+static struct inode *rcfs_new_inode(mode_t mode, struct super_block *sb)
+{
+ struct inode *inode = new_inode(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 = &rcfs_backing_dev_info;
+ }
+ return inode;
+}
+
+static struct super_operations rcfs_sb_ops = {
+ .statfs = simple_statfs,
+ .drop_inode = generic_delete_inode,
+ .put_super = rcfs_put_super,
+ .show_options = rcfs_show_options,
+ //.remount_fs = rcfs_remount,
+};
+
+static struct inode_operations rcfs_dir_inode_operations;
+static int rcfs_create_dir(struct nsproxy *ns, struct dentry *dentry,
```

```
+    int mode);
+static int rcfs_populate_dir(struct dentry *d);
+static void rcfs_d_remove_dir(struct dentry *dentry);
+
+static int rcfs_fill_super(struct super_block *sb, void *options,
+    int unused_silent)
+{
+ struct inode *inode;
+ struct dentry *root;
+ struct rcfs_root *hroot = options;
+
+ sb->s_blocksize = PAGE_CACHE_SIZE;
+ sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
+ sb->s_magic = RCFS_SUPER_MAGIC;
+ sb->s_op = &rcfs_sb_ops;
+
+ inode = rcfs_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 = &rcfs_dir_inode_operations;
+ /* directories start off with i_nlink == 2 (for "." entry) */
+ inc_nlink(inode);
+
+ root = d_alloc_root(inode);
+ if (!root) {
+  iput(inode);
+  return -ENOMEM;
+ }
+ sb->s_root = root;
+ get_task_namespaces(&init_task);
+ root->d_fsdata = init_task.nsproxy;
+ sb->s_fs_info = hroot;
+ hroot->sb = sb;
+
+ return 0;
+}
+
+static inline struct nsproxy *__d_ns(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+
+static inline struct cftype *__d_cft(struct dentry *dentry)
+{
```

```
+ return dentry->d_fsdata;
+}
+
+/* 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 rcfs_task_count(struct dentry *d)
+{
+ struct nsproxy *ns = __d_ns(d);
+ struct rc_subsys *ss;
+ struct rcfs_root *root = d->d_sb->s_fs_info;
+ int count;
+
+ ss = list_entry(root->subsys_list.next, struct rc_subsys, sibling);
+ count = nsproxy_task_count(ns->ctlr_data[ss->subsys_id], ss->subsys_id);
+
+ return count;
+}
+
+/*
+ * 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, 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 dentry *d)
+{
+ int n = 0, idx;
+ struct task_struct *g, *p;
+ struct nsproxy *ns = __d_ns(d);
+ struct rc_subsys *ss;
+ struct rcfs_root *root = d->d_sb->s_fs_info;
+
+ rcu_read_lock();
+ read_lock(&tasklist_lock);
+
+ ss = list_entry(root->subsys_list.next, struct rc_subsys, sibling);
+ idx = ss->subsys_id;
+
+ do_each_thread(g, p) {
+  if (p->nsproxy->ctlr_data[idx] == ns->ctlr_data[idx]) {
+   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);
+ rcu_read_unlock();
+ 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 rcfs_tasks_open(struct inode *unused, struct file *file)
+{
+ 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 = rcfs_task_count(file->f_dentry->d_parent);
+ pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
+ if (!pidarray)
+  goto err1;
+
+ npids = pid_array_load(pidarray, npids, file->f_dentry->d_parent);
+ 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 rcfs_tasks_read(struct nsproxy *ns,
+        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 rcfs_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;
+}
+/*
+ * Attach task 'tsk' to container 'cont'
+ *
+ * Call holding manage_mutex.  May take callback_mutex and task_lock of
+ * the task 'pid' during call.
+ */
+
+static int attach_task(struct dentry *d, struct task_struct *tsk)
+{
+ int retval = 0;
+ struct rc_subsys *ss;
+ struct rcfs_root *root = d->d_sb->s_fs_info;
+ struct nsproxy *ns = __d_ns(d->d_parent);
+ struct nsproxy *oldns, *newns;
+ struct nsproxy dupns;
```

```
+
+ printk ("attaching task %d to %p \n", tsk->pid, ns);
+
+ /* Nothing to do if the task is already in that container */
+ if (tsk->nsproxy == ns)
+  return 0;
+
+ for_each_subsys(root, ss) {
+  if (ss->can_attach) {
+   retval = ss->can_attach(ss, ns, tsk);
+   if (retval) {
+    put_task_struct(tsk);
+    return retval;
+   }
+  }
+ }
+
+ /* Locate or allocate a new container_group for this task,
+  * based on its final set of containers */
+  get_task_namespaces(tsk);
+ oldns = tsk->nsproxy;
+ memcpy(&dupns, oldns, sizeof(dupns));
+ for_each_subsys(root, ss)
+  dupns.ctlr_data[ss->subsys_id] = ns->ctlr_data[ss->subsys_id];
+ newns = find_nsproxy(&dupns);
+ printk ("find_nsproxy returned %p \n", newns);
+ if (!newns) {
+  put_nsproxy(tsk->nsproxy);
+  put_task_struct(tsk);
+  return -ENOMEM;
+ }
+
+ task_lock(tsk); /* Needed ? */
+ rcu_assign_pointer(tsk->nsproxy, newns);
+ task_unlock(tsk);
+
+ for_each_subsys(root, ss) {
+  if (ss->attach)
+   ss->attach(ss, newns, oldns, tsk);
+ }
+
+ synchronize_rcu();
+ put_nsproxy(oldns);
+ return 0;
+}
+
+
+/*
```

```
+ * Attach task with pid 'pid' to container 'cont'. Call with
+ * manage_mutex, may take callback_mutex and task_lock of task
+ *
+ */
+
+static int attach_task_by_pid(struct dentry *d, char *pidbuf)
+{
+ pid_t pid;
+ struct task_struct *tsk;
+ int ret;
+
+ 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);
+ }
+
+ ret = attach_task(d, tsk);
+ put_task_struct(tsk);
+ return ret;
+}
+
+/* The various types of files and directories in a container file system */
+
+typedef enum {
+ FILE_ROOT,
+ FILE_DIR,
+ FILE_TASKLIST,
+} rcfs_filetype_t;
```

```
+
+static ssize_t rcfs_common_file_write(struct nsproxy *ns, struct cftype *cft,
+        struct file *file,
+        const char __user *userbuf,
+        size_t nbytes, loff_t *unused_ppos)
+{
+ rcfs_filetype_t type = cft->private;
+ char *buffer;
+ int retval = 0;
+
+ if (nbytes >= PATH_MAX)
+  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(&manage_mutex);
+
+ ns = __d_ns(file->f_dentry);
+ if (!atomic_read(&ns->count)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch (type) {
+ case FILE_TASKLIST:
+  retval = attach_task_by_pid(file->f_dentry, buffer);
+  break;
+ default:
+  retval = -EINVAL;
+  goto out2;
+ }
+
+ if (retval == 0)
+  retval = nbytes;
+out2:
+ mutex_unlock(&manage_mutex);
+out1:
+ kfree(buffer);
+ return retval;
+}
```

```
+
+static struct cftype cft_tasks = {
+ .name = "tasks",
+ .open = rcfs_tasks_open,
+ .read = rcfs_tasks_read,
+ .write = rcfs_common_file_write,
+ .release = rcfs_tasks_release,
+ .private = FILE_TASKLIST,
+};
+
+static ssize_t rcfs_file_write(struct file *file, const char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct nsproxy *ns = __d_ns(file->f_dentry->d_parent);
+ if (!cft)
+  return -ENODEV;
+ if (!cft->write)
+  return -EINVAL;
+
+ return cft->write(ns, cft, file, buf, nbytes, ppos);
+}
+
+static ssize_t rcfs_file_read(struct file *file, char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct nsproxy *ns = __d_ns(file->f_dentry->d_parent);
+ if (!cft)
+  return -ENODEV;
+ if (!cft->read)
+  return -EINVAL;
+
+ return cft->read(ns, cft, file, buf, nbytes, ppos);
+}
+
+static int rcfs_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 rcfs_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;
+}
+
+/*
+ * rcfs_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 rcfs_create(struct nsproxy *parent, struct dentry *dentry,
+        int mode)
+{
+ struct rcfs_root *root = dentry->d_sb->s_fs_info;
+ int err = 0;
+ struct rc_subsys *ss;
+ struct super_block *sb = dentry->d_sb;
+ struct nsproxy *ns;
+
+ ns = dup_namespaces(parent);
+ if (!ns)
+  return -ENOMEM;
+
+ printk ("rcfs_create: ns = %p \n", ns);
+
+ /* Grab a reference on the superblock so the hierarchy doesn't
+  * get deleted on unmount if there are child containers.  This
+  * can be done outside manage_mutex, since the sb can't
+  * disappear while someone has an open control file on the
+  * fs */
+ atomic_inc(&sb->s_active);
+
+ mutex_lock(&manage_mutex);
```

```
+
+ for_each_subsys(root, ss) {
+  err = ss->create(ss, ns, parent);
+  if (err) {
+   printk ("%s create failed \n", ss->name);
+   goto err_destroy;
+  }
+ }
+
+ err = rcfs_create_dir(ns, dentry, mode);
+ if (err < 0)
+  goto err_destroy;
+
+ /* The container directory was pre-locked for us */
+ BUG_ON(!mutex_is_locked(&dentry->d_inode->i_mutex));
+
+ err = rcfs_populate_dir(dentry);
+ /* If err < 0, we have a half-filled directory - oh well ;) */
+
+ mutex_unlock(&manage_mutex);
+ mutex_unlock(&dentry->d_inode->i_mutex);
+
+ return 0;
+
+err_destroy:
+
+ for_each_subsys(root, ss)
+  ss->destroy(ss, ns);
+
+ mutex_unlock(&manage_mutex);
+
+ /* Release the reference count that we took on the superblock */
+ deactivate_super(sb);
+
+ free_nsproxy(ns);
+ return err;
+}
+
+static int rcfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
+{
+ struct nsproxy *ns_parent = dentry->d_parent->d_fsdata;
+
+ printk ("rcfs_mkdir : parent_nsproxy = %p (%p) \n", ns_parent, dentry->d_fsdata);
+
+ /* the vfs holds inode->i_mutex already */
+ return rcfs_create(ns_parent, dentry, mode | S_IFDIR);
+}
+
```

```
+static int rcfs_rmdir(struct inode *unused_dir, struct dentry *dentry)
+{
+ struct nsproxy *ns = dentry->d_fsdata;
+ struct dentry *d;
+ struct rc_subsys *ss;
+ struct super_block *sb = dentry->d_sb;
+ struct rcfs_root *root = dentry->d_sb->s_fs_info;
+
+ /* the vfs holds both inode->i_mutex already */
+
+ mutex_lock(&manage_mutex);
+
+ if (atomic_read(&ns->count) > 1) {
+  mutex_unlock(&manage_mutex);
+  return -EBUSY;
+ }
+
+ if (!dir_empty(dentry)) {
+  mutex_unlock(&manage_mutex);
+  return -EBUSY;
+ }
+
+ atomic_set(&ns->count, 0);
+
+ for_each_subsys(root, ss)
+  ss->destroy(ss, ns);
+
+ spin_lock(&dentry->d_lock);
+ d = dget(dentry);
+ spin_unlock(&d->d_lock);
+
+ rcfs_d_remove_dir(d);
+ dput(d);
+
+ mutex_unlock(&manage_mutex);
+ /* Drop the active superblock reference that we took when we
+  * created the container */
+ deactivate_super(sb);
+ return 0;
+}
+
+static struct file_operations rcfs_file_operations = {
+ .read = rcfs_file_read,
+ .write = rcfs_file_write,
+ .llseek = generic_file_llseek,
+ .open = rcfs_file_open,
+ .release = rcfs_file_release,
+};
```

```
+
+static struct inode_operations rcfs_dir_inode_operations = {
+ .lookup = simple_lookup,
+ .mkdir = rcfs_mkdir,
+ .rmdir = rcfs_rmdir,
+ //.rename = rcfs_rename,
+};
+
+static int rcfs_create_file(struct dentry *dentry, int mode,
+     struct super_block *sb)
+{
+ struct inode *inode;
+
+ if (!dentry)
+  return -ENOENT;
+ if (dentry->d_inode)
+  return -EEXIST;
+
+ inode = rcfs_new_inode(mode, sb);
+ if (!inode)
+  return -ENOMEM;
+
+ if (S_ISDIR(mode)) {
+  inode->i_op = &rcfs_dir_inode_operations;
+  inode->i_fop = &simple_dir_operations;
+
+  /* start off with i_nlink == 2 (for "." entry) */
+  inc_nlink(inode);
+
+  /* start with the directory inode held, so that we can
+   * populate it without racing with another mkdir */
+  mutex_lock(&inode->i_mutex);
+ } else if (S_ISREG(mode)) {
+  inode->i_size = 0;
+  inode->i_fop = &rcfs_file_operations;
+ }
+
+ d_instantiate(dentry, inode);
+ dget(dentry); /* Extra count - pin the dentry in core */
+ return 0;
+}
+
+/*
+ * rcfs_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 rcfs_create_dir(struct nsproxy *ns, struct dentry *dentry,
+    int mode)
+{
+ struct dentry *parent;
+ int error = 0;
+
+ parent = dentry->d_parent;
+ if (IS_ERR(dentry))
+  return PTR_ERR(dentry);
+ error = rcfs_create_file(dentry, S_IFDIR | mode, dentry->d_sb);
+ if (!error) {
+  dentry->d_fsdata = ns;
+  inc_nlink(parent->d_inode);
+ }
+ dput(dentry);
+
+ return error;
+}
+
+static void rcfs_diput(struct dentry *dentry, struct inode *inode)
+{
+ /* is dentry a directory ? if so, kfree() associated container */
+ if (S_ISDIR(inode->i_mode)) {
+  struct nsproxy *ns = dentry->d_fsdata;
+
+  free_nsproxy(ns);
+  dentry->d_fsdata = NULL;
+ }
+ iput(inode);
+}
+
+static struct dentry_operations rcfs_dops = {
+ .d_iput = rcfs_diput,
+};
+
+static struct dentry *rcfs_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 = &rcfs_dops;
+ return d;
+}
+
+int rcfs_add_file(struct dentry *dir, const struct cftype *cft)
```

```
+{
+ struct dentry *dentry;
+ int error;
+
+ BUG_ON(!mutex_is_locked(&dir->d_inode->i_mutex));
+ dentry = rcfs_get_dentry(dir, cft->name);
+ if (!IS_ERR(dentry)) {
+  error = rcfs_create_file(dentry, 0644 | S_IFREG, dir->d_sb);
+  if (!error)
+   dentry->d_fsdata = (void *)cft;
+  dput(dentry);
+ } else
+  error = PTR_ERR(dentry);
+ return error;
+}
+
+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);
+}
+
+static void rcfs_clear_directory(struct dentry *dentry)
+{
+ struct list_head *node;
+
+ BUG_ON(!mutex_is_locked(&dentry->d_inode->i_mutex));
+ spin_lock(&dcache_lock);
+ node = dentry->d_subdirs.next;
+ while (node != &dentry->d_subdirs) {
+  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
+  list_del_init(node);
+  if (d->d_inode) {
+   /* This should never be called on a container
+    * directory with child containers */
+   BUG_ON(d->d_inode->i_mode & S_IFDIR);
+   d = dget_locked(d);
+   spin_unlock(&dcache_lock);
+   d_delete(d);
+   simple_unlink(dentry->d_inode, d);
+   dput(d);
+   spin_lock(&dcache_lock);
+  }
+  node = dentry->d_subdirs.next;
+ }
```

```
+ spin_unlock(&dcache_lock);
+}
+
+/*
+ * NOTE : the dentry must have been dget()'ed
+ */
+static void rcfs_d_remove_dir(struct dentry *dentry)
+{
+ rcfs_clear_directory(dentry);
+
+ spin_lock(&dcache_lock);
+ list_del_init(&dentry->d_u.d_child);
+ spin_unlock(&dcache_lock);
+ remove_dir(dentry);
+}
+
+static int rcfs_populate_dir(struct dentry *d)
+{
+ int err;
+ struct rc_subsys *ss;
+ struct rcfs_root *root = d->d_sb->s_fs_info;
+
+ /* First clear out any existing files */
+ rcfs_clear_directory(d);
+
+ if ((err = rcfs_add_file(d, &cft_tasks)) < 0)
+  return err;
+
+ for_each_subsys(root, ss)
+  if (ss->populate && (err = ss->populate(ss, d)) < 0)
+   return err;
+
+ return 0;
+}
+
+static int rcfs_get_sb(struct file_system_type *fs_type,
+                       int flags, const char *unused_dev_name,
+                       void *data, struct vfsmount *mnt)
+{
+ int i;
+ unsigned long subsys_bits = 0;
+ int ret = 0;
+ struct rcfs_root *root = NULL;
+
+ mutex_lock(&manage_mutex);
+
+ /* First find the desired set of resource controllers */
+ ret = parse_rcfs_options(data, &subsys_bits);
```

```
+ if (ret)
+  goto out_unlock;
+
+ /* See if we already have a hierarchy containing this set */
+
+ for (i = 0; i < CONFIG_MAX_RC_HIERARCHIES; i++) {
+  root = &rootnode[i];
+  /* We match - use this hieracrchy */
+  if (root->subsys_bits == subsys_bits) break;
+  /* We clash - fail */
+  if (root->subsys_bits & subsys_bits) {
+   ret = -EBUSY;
+   goto out_unlock;
+  }
+ }
+
+ if (i == CONFIG_MAX_RC_HIERARCHIES) {
+  /* No existing hierarchy matched this set - but we
+   * know that all the subsystems are free */
+  for (i = 0; i < CONFIG_MAX_RC_HIERARCHIES; i++) {
+   root = &rootnode[i];
+   if (!root->sb && !root->subsys_bits) break;
+  }
+ }
+
+ if (i == CONFIG_MAX_RC_HIERARCHIES) {
+  ret = -ENOSPC;
+  goto out_unlock;
+ }
+
+ if (!root->sb) {
+  BUG_ON(root->subsys_bits);
+  ret = get_sb_nodev(fs_type, flags, root,
+        rcfs_fill_super, mnt);
+  if (ret)
+   goto out_unlock;
+
+  ret = rebind_subsystems(root, subsys_bits);
+  BUG_ON(ret);
+
+  /* It's safe to nest i_mutex inside manage_mutex in
+   * this case, since no-one else can be accessing this
+   * directory yet */
+  mutex_lock(&root->sb->s_root->d_inode->i_mutex);
+  rcfs_populate_dir(root->sb->s_root);
+  mutex_unlock(&root->sb->s_root->d_inode->i_mutex);
+
+ } else {
```

```
+  /* Reuse the existing superblock */
+  ret = simple_set_mnt(mnt, root->sb);
+  if (!ret)
+   atomic_inc(&root->sb->s_active);
+ }
+
+out_unlock:
+ mutex_unlock(&manage_mutex);
+ return ret;
+}
+
+static struct file_system_type rcfs_type = {
+ .name = "rcfs",
+ .get_sb = rcfs_get_sb,
+ .kill_sb = kill_litter_super,
+};
+
+int __init rcfs_init(void)
+{
+ int i, err;
+
+ for (i=0; i < CONFIG_MAX_RC_HIERARCHIES; ++i)
+  INIT_LIST_HEAD(&rootnode[i].subsys_list);
+
+ err = register_filesystem(&rcfs_type);
+
+ return err;
+}
+
+int rc_register_subsys(struct rc_subsys *new_subsys)
+{
+ int retval = 0;
+ int i;
+ int ss_id;
+
+ BUG_ON(new_subsys->hierarchy);
+ BUG_ON(new_subsys->active);
+
+ mutex_lock(&manage_mutex);
+
+ if (subsys_count == CONFIG_MAX_RC_SUBSYS) {
+  retval = -ENOSPC;
+  goto out;
+ }
+
+ /* Sanity check the subsystem */
+ if (!new_subsys->name ||
+     (strlen(new_subsys->name) > MAX_CONTAINER_TYPE_NAMELEN) ||
```

```
+     !new_subsys->create || !new_subsys->destroy) {
+  retval = -EINVAL;
+  goto out;
+ }
+
+ /* Check this isn't a duplicate */
+ for (i = 0; i < subsys_count; i++) {
+  if (!strcmp(subsys[i]->name, new_subsys->name)) {
+   retval = -EEXIST;
+   goto out;
+  }
+ }
+
+ /* Create the top container state for this subsystem */
+ ss_id = new_subsys->subsys_id = subsys_count;
+ retval = new_subsys->create(new_subsys, &init_nsproxy, NULL);
+ if (retval) {
+  new_subsys->subsys_id = -1;
+  goto out;
+ }
+
+ subsys[subsys_count++] = new_subsys;
+ new_subsys->active = 1;
+out:
+  mutex_unlock(&manage_mutex);
+  return retval;
+}
+
+void rcfs_manage_lock(void)
+{
+ mutex_lock(&manage_mutex);
+}
+
+/**
+ * container_manage_unlock - release lock on container changes
+ *
+ * Undo the lock taken in a previous container_manage_lock() call.
+ */
+void rcfs_manage_unlock(void)
+{
+ mutex_unlock(&manage_mutex);
+}
+
+int rcfs_dir_removed(struct dentry *d)
+{
+ struct nsproxy *ns = __d_ns(d);
+
+ if (!atomic_read(&ns->count))
```

```
+  return 1;
+
+ return 0;
+}
+
+/*
+ * Call with manage_mutex held.  Writes path of container into buf.
+ * Returns 0 on success, -errno on error.
+ */
+
+int rcfs_path(struct dentry *dentry, char *buf, int buflen)
+{
+ char *start;
+
+ start = buf + buflen;
+
+ *--start = '\0';
+ for (;;) {
+  int len = dentry->d_name.len;
+  if ((start -= len) < buf)
+   return -ENAMETOOLONG;
+  memcpy(start, dentry->d_name.name, len);
+  dentry = dentry->d_parent;
+  if (!dentry)
+   break;
+  if (--start < buf)
+   return -ENAMETOOLONG;
+  *start = '/';
+ }
+ memmove(buf, start, buf + buflen - start);
+ return 0;
+}
+
_
```

This demonstrates how to use the generic container subsystem for a simple resource tracker that counts the total CPU time used by all processes in a container, during the time that they're members of the container.

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

---
 kernel/Makefile | 1

Index: container-2.6.20/include/linux/cpu_acct.h
===

---

 linux-2.6.20-vatsa/init/Kconfig | 7
 linux-2.6.20-vatsa/kernel/Makefile | 1

---

 linux-2.6.20.1-vatsa/include/linux/cpu_acct.h | 14 +
 linux-2.6.20.1-vatsa/init/Kconfig | 7
 linux-2.6.20.1-vatsa/kernel/Makefile | 1
 linux-2.6.20.1-vatsa/kernel/cpu_acct.c | 221 ++++++++++++++++++++++++++
 linux-2.6.20.1-vatsa/kernel/sched.c | 14 +
 5 files changed, 254 insertions(+), 3 deletions(-)

diff -puN /dev/null include/linux/cpu_acct.h
--- /dev/null 2007-03-08 22:15:35.669495160 +0530
+++ linux-2.6.20.1-vatsa/include/linux/cpu_acct.h 2007-03-08 22:35:32.000000000 +0530
@@ -0,0 +1,14 @@
+
+#ifndef _LINUX_CPU_ACCT_H
+#define _LINUX_CPU_ACCT_H
+
+#include <linux/rcfs.h>
+#include <asm/cputime.h>
+
+#ifdef CONFIG_RC_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
diff -puN init/Kconfig~cpu_acct init/Kconfig
--- linux-2.6.20.1/init/Kconfig~cpu_acct 2007-03-08 22:35:32.000000000 +0530
+++ linux-2.6.20.1-vatsa/init/Kconfig 2007-03-08 22:35:32.000000000 +0530
@@ -291,6 +291,13 @@ 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 RC_CPUACCT
+ bool "Simple CPU accounting container subsystem"
+ select RCFS
+ 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
diff -puN /dev/null kernel/cpu_acct.c
--- /dev/null 2007-03-08 22:15:35.669495160 +0530
+++ linux-2.6.20.1-vatsa/kernel/cpu_acct.c 2007-03-08 22:35:32.000000000 +0530
@@ -0,0 +1,221 @@
+/*
+ * 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/nsproxy.h>
+#include <linux/rcfs.h>
+#include <linux/fs.h>
+#include <asm/div64.h>
+
+struct cpuacct {
+ 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;
+};
+
+static struct rc_subsys cpuacct_subsys;
+
+static inline struct cpuacct *nsproxy_ca(struct nsproxy *ns)
+{
+ if (!ns)
+  return NULL;
```

```
+
+ return ns->ctlr_data[cpuacct_subsys.subsys_id];
+}
+
+static inline struct cpuacct *task_ca(struct task_struct *task)
+{
+ return nsproxy_ca(task->nsproxy);
+}
+
+#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 rc_subsys *ss, struct nsproxy *ns,
+     struct nsproxy *parent)
+{
+ struct cpuacct *ca;
+
+ if (parent && (parent != &init_nsproxy))
+  return -EINVAL;
+
+ 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());
+ ns->ctlr_data[cpuacct_subsys.subsys_id] = ca;
+ return 0;
+}
+
+static void cpuacct_destroy(struct rc_subsys *ss, struct nsproxy *ns)
+{
+ kfree(nsproxy_ca(ns));
+}
+
+/* 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 nsproxy *ns,
+        struct cftype *cft,
+        struct file *file,
+        char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct cpuacct *ca = nsproxy_ca(ns);
+ 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 nsproxy *ns,
+    struct cftype *cft,
+    struct file *file,
+    char __user *buf,
+    size_t nbytes, loff_t *ppos)
```

```
+{
+ struct cpuacct *ca = nsproxy_ca(ns);
+ 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 rc_subsys *ss,
+       struct dentry *d)
+{
+ int err;
+
+ if ((err = rcfs_add_file(d, &cft_usage)))
+  return err;
+ if ((err = rcfs_add_file(d, &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();
+}
+
+static struct rc_subsys cpuacct_subsys = {
+ .name = "cpuacct",
+ .create = cpuacct_create,
+ .destroy = cpuacct_destroy,
+ .populate = cpuacct_populate,
+ .subsys_id = -1,
+};
+
+
+int __init init_cpuacct(void)
+{
+ int id = rc_register_subsys(&cpuacct_subsys);
+ return id < 0 ? id : 0;
+}
+
+module_init(init_cpuacct)
diff -puN kernel/Makefile~cpu_acct kernel/Makefile
--- linux-2.6.20.1/kernel/Makefile~cpu_acct 2007-03-08 22:35:32.000000000 +0530
+++ linux-2.6.20.1-vatsa/kernel/Makefile 2007-03-08 22:35:32.000000000 +0530
@@ -36,6 +36,7 @@ obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
+obj-$(CONFIG_RC_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
 obj-$(CONFIG_AUDIT) += audit.o auditfilter.o
diff -puN kernel/sched.c~cpu_acct kernel/sched.c
--- linux-2.6.20.1/kernel/sched.c~cpu_acct 2007-03-08 22:35:32.000000000 +0530
```

```
+++ linux-2.6.20.1-vatsa/kernel/sched.c 2007-03-08 22:35:32.000000000 +0530
@@ -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)
```

_

---

 linux-2.6.20-vatsa/fs/proc/base.c | 4
 linux-2.6.20-vatsa/fs/super.c | 5
 linux-2.6.20-vatsa/include/linux/cpuset.h | 11
 linux-2.6.20-vatsa/include/linux/fs.h | 2
 linux-2.6.20-vatsa/include/linux/mempolicy.h | 12
 linux-2.6.20-vatsa/include/linux/sched.h | 2
 linux-2.6.20-vatsa/init/Kconfig | 5
 linux-2.6.20-vatsa/kernel/exit.c | 2
 linux-2.6.20-vatsa/kernel/fork.c | 6

---

 linux-2.6.20.1-vatsa/fs/proc/base.c | 4
 linux-2.6.20.1-vatsa/fs/super.c | 5
 linux-2.6.20.1-vatsa/include/linux/cpuset.h | 11
 linux-2.6.20.1-vatsa/include/linux/fs.h | 2
 linux-2.6.20.1-vatsa/include/linux/mempolicy.h | 12
 linux-2.6.20.1-vatsa/include/linux/sched.h | 2
 linux-2.6.20.1-vatsa/init/Kconfig | 5
 linux-2.6.20.1-vatsa/kernel/cpuset.c | 1190 +++----------------------
 linux-2.6.20.1-vatsa/kernel/exit.c | 2
 linux-2.6.20.1-vatsa/kernel/fork.c | 6
 10 files changed, 180 insertions(+), 1059 deletions(-)

diff -puN fs/proc/base.c~cpuset_uses_rcfs fs/proc/base.c
--- linux-2.6.20.1/fs/proc/base.c~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/fs/proc/base.c 2007-03-08 22:35:35.000000000 +0530
@@ -1867,7 +1867,7 @@ 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
  INF("oom_score", S_IRUGO, oom_score),
@@ -2148,7 +2148,7 @@ 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
  INF("oom_score", S_IRUGO, oom_score),
diff -puN fs/super.c~cpuset_uses_rcfs fs/super.c
--- linux-2.6.20.1/fs/super.c~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/fs/super.c 2007-03-08 22:35:35.000000000 +0530
@@ -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);

diff -puN include/linux/cpuset.h~cpuset_uses_rcfs include/linux/cpuset.h
--- linux-2.6.20.1/include/linux/cpuset.h~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/include/linux/cpuset.h 2007-03-08 22:35:35.000000000 +0530
@@ -11,6 +11,7 @@
 #include <linux/sched.h>
 #include <linux/cpumask.h>
 #include <linux/nodemask.h>
+#include <linux/rcfs.h>

 #ifdef CONFIG_CPUSETS

@@ -19,8 +20,6 @@ extern int number_of_cpusets; /* How man
 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)
@@ -74,14 +73,13 @@ static inline int cpuset_do_slab_mem_spr
 }

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

 #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)
 {
@@ -146,6 +144,11 @@ 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 */
diff -puN include/linux/fs.h~cpuset_uses_rcfs include/linux/fs.h
--- linux-2.6.20.1/include/linux/fs.h~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/include/linux/fs.h 2007-03-08 22:35:35.000000000 +0530
@@ -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);
diff -puN include/linux/mempolicy.h~cpuset_uses_rcfs include/linux/mempolicy.h
--- linux-2.6.20.1/include/linux/mempolicy.h~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000
+0530
+++ linux-2.6.20.1-vatsa/include/linux/mempolicy.h 2007-03-08 22:35:35.000000000 +0530
@@ -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->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)
 {
diff -puN include/linux/sched.h~cpuset_uses_rcfs include/linux/sched.h
--- linux-2.6.20.1/include/linux/sched.h~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/include/linux/sched.h 2007-03-08 22:35:35.000000000 +0530
@@ -743,7 +743,6 @@ extern unsigned int max_cache_size;


 struct io_context;   /* See blkdev.h */
-struct cpuset;

 #define NGROUPS_SMALL  32
 #define NGROUPS_PER_BLOCK ((int)(PAGE_SIZE / sizeof(gid_t)))
@@ -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;
diff -puN init/Kconfig~cpuset_uses_rcfs init/Kconfig
--- linux-2.6.20.1/init/Kconfig~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/init/Kconfig 2007-03-08 22:35:35.000000000 +0530
@@ -298,6 +298,11 @@ config RC_CPUACCT
    Provides a simple Resource Controller for monitoring the
    total CPU consumed by the tasks in a container

+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
diff -puN kernel/cpuset.c~cpuset_uses_rcfs kernel/cpuset.c
--- linux-2.6.20.1/kernel/cpuset.c~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/kernel/cpuset.c 2007-03-08 22:35:35.000000000 +0530
@@ -49,13 +49,13 @@
 #include <linux/time.h>
 #include <linux/backing-dev.h>
 #include <linux/sort.h>
+#include <linux/rcfs.h>
+#include <linux/nsproxy.h>

 #include <asm/uaccess.h>
 #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
@@ -63,6 +63,10 @@
  */
 int number_of_cpusets __read_mostly;

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

 struct fmeter {
@@ -90,7 +94,7 @@ struct cpuset {
  struct list_head children; /* my children */

  struct cpuset *parent;  /* my parent */
- struct dentry *dentry;  /* cpuset fs entry */
+ struct dentry *dentry;          /* cpuset fs entry */

  /*
   * Copy of global cpuset_mems_generation as of the most
@@ -106,8 +110,6 @@ 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 +125,6 @@ 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);
@@ -178,383 +170,53 @@ static struct cpuset top_cpuset = {
  .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
- * require taking task_lock() when dereferencing a tasks cpuset pointer.
- * See "The task_lock() exception", at the end of this comment.
- *
- * A task must hold both mutexes to modify cpusets.  If a task
- * holds manage_mutex, then it blocks others wanting that mutex,
- * ensuring that it is the only task able to also acquire callback_mutex
- * and be able to modify cpusets.  It can perform various checks on
- * the cpuset structure first, knowing nothing will change.  It can
- * also allocate memory while just holding manage_mutex.  While it is
- * performing these checks, various callback routines can briefly
- * acquire callback_mutex to query cpusets.  Once it is ready to make
- * the changes, it takes callback_mutex, blocking everyone else.
- *
- * Calls to the kernel memory allocator can not be made while holding
- * callback_mutex, as that would risk double tripping on callback_mutex
- * from one of the callbacks into the cpuset code from within
```

- * __alloc_pages().
- *
- * If a task is only holding callback_mutex, then it has read-only
- * access to cpusets.
- *
- * The task_struct fields mems_allowed and mems_generation may only
- * be accessed in the context of that task, so require no locks.
- *
- * Any task can increment and decrement the count field without lock.
- * So in general, code holding manage_mutex or callback_mutex can't rely
- * on the count field not changing. However, if the count goes to
- * zero, then only attach_task(), which holds both mutexes, 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 cpuset can fork (the other way to increment the count).
- * So code holding manage_mutex or callback_mutex can safely assume that
- * if the count is zero, it will stay zero. Similarly, if a task
- * holds manage_mutex or callback_mutex on a cpuset with zero count, it
- * knows that the cpuset won't be removed, as cpuset_rmdir() needs
- * both of those mutexes.
- *
- * The cpuset_common_file_write handler for operations that modify
- * the cpuset hierarchy holds manage_mutex across the entire operation,
- * single threading all such cpuset modifications across the system.
- *
- * The cpuset_common_file_read() handlers only hold callback_mutex across
- * small pieces of code, such as when reading out possibly multi-word
- * cpumasks and nodemasks.
- *
- * The fork and exit callbacks cpuset_fork() and cpuset_exit(), don't
- * (usually) take either mutex. These are the two most performance
- * critical pieces of code here. The exception occurs on cpuset_exit(),
- * when a task in a notify_on_release cpuset exits. Then manage_mutex
- * is taken, and if the cpuset count is zero, a usermode call made
- * to /sbin/cpuset_release_agent with the name of the cpuset (path
- * relative to the root of cpuset file system) as the argument.
- *
- * A cpuset can only be deleted if both its 'count' of using tasks
- * is zero, and its list of 'children' cpusets is empty. Since all
- * tasks in the system use _some_ cpuset, and since there is always at
- * least one task in the system (init), therefore, top_cpuset
- * always has either children cpusets and/or using tasks. So we don't
- * need a special hack to ensure that top_cpuset cannot be deleted.
- *
- * The above "Tale of Two Semaphores" would be complete, but for:
- *
- * The task_lock() exception
- *

```
- * The need for this exception arises from the action of attach_task(),
- * which overwrites one tasks cpuset pointer with another.  It does
- * so using both mutexes, however there are several performance
- * critical places that need to reference task->cpuset without the
- * expense of grabbing a system global mutex.  Therefore except as
- * noted below, when dereferencing or, as in attach_task(), modifying
- * a tasks cpuset 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 cpuset pointer by attach_task() and the
- * access of task->cpuset->mems_generation via that pointer in
- * 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)
+/* Update the cpuset for a container */
+static inline void set_cs(struct nsproxy *ns, struct cpuset *cs)
 {
- struct dentry *parent = dget(d->d_parent);
-
- d_delete(d);
- simple_rmdir(parent->d_inode, d);
- dput(parent);
+       ns->ctlr_data[cpuset_subsys.subsys_id] = cs;
 }

-/*
- * NOTE : the dentry must have been dget()'ed
- */
-static void cpuset_d_remove_dir(struct dentry *dentry)
+static inline struct cpuset *ns_cs(struct nsproxy *ns)
 {
- 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);
+ return ns->ctlr_data[cpuset_subsys.subsys_id];
 }

-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)
+static inline struct cpuset *task_cs(struct task_struct *tsk)
 {
- 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;
+  if (!tsk->nsproxy) {
+  printk ("nsproxy NULL \n");
+  return &top_cpuset;
   }

- root = d_alloc_root(inode);
```

```
- if (!root) {
-  iput(inode);
-  return -ENOMEM;
- }
- sb->s_root = root;
- return 0;
+ return ns_cs(tsk->nsproxy);
 }

+/* 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 "rcfs" 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 *rcfs = get_fs_type("rcfs");
+ int ret = -ENODEV;
+
+ if (rcfs) {
+  ret = rcfs->get_sb(rcfs, flags, unused_dev_name, "cpuset", mnt);
+  put_filesystem(rcfs);
+ }
+
+ 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
@@ -652,20 +314,19 @@ void cpuset_update_task_memory_state(voi
  struct task_struct *tsk = current;
  struct cpuset *cs;
```

```
- if (tsk->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->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->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))
@@ -885,7 +546,7 @@ 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(task_cs(tsk), &tsk->mems_allowed);
  mutex_unlock(&callback_mutex);
 }

@@ -903,6 +564,8 @@ 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;
@@ -941,7 +604,7 @@ 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 */
```

```
@@ -955,13 +618,14 @@ static int update_nodemask(struct cpuset
   * enough mmarray[] w/o using GFP_ATOMIC.
   */
  while (1) {
-  ntasks = atomic_read(&cs->count); /* guess */
+  /* guess */
+  ntasks = nsproxy_task_count(cs, cpuset_subsys.subsys_id);
   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 (nsproxy_task_count(cs, cpuset_subsys.subsys_id) <= ntasks)
    break;    /* got enough */
   write_unlock_irq(&tasklist_lock); /* try again */
   kfree(mmarray);
@@ -978,7 +642,7 @@ static int update_nodemask(struct cpuset
     "Cpuset mempolicy rebind incomplete.\n");
    continue;
   }
-  if (p->cpuset != cs)
+  if (task_cs(p) != cs)
    continue;
   mm = get_task_mm(p);
   if (!mm)
@@ -1012,12 +676,18 @@ 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.
  */
@@ -1168,85 +838,32 @@ 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(struct rc_subsys *ss, struct nsproxy *ns,
+       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 = ns_cs(ns);

- 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);
- }
-
- retval = security_task_setscheduler(tsk, 0, NULL);
- if (retval) {
-  put_task_struct(tsk);
-  return retval;
- }
-
- mutex_lock(&callback_mutex);
+ return security_task_setscheduler(tsk, 0, NULL);
+}

- 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);
+void cpuset_attach(struct rc_subsys *ss, struct nsproxy *ns,
+     struct nsproxy *old_ns, struct task_struct *tsk)
+{
+ struct cpuset *oldcs = ns_cs(old_ns), *cs = ns_cs(ns);
+ cpumask_t cpus;
+ nodemask_t from, to;
+ struct mm_struct *mm;

+ /* container_lock not strictly needed - we already hold manage_mutex */
  guarantee_online_cpus(cs, &cpus);
  set_cpus_allowed(tsk, cpus);

  from = oldcs->mems_allowed;
  to = cs->mems_allowed;

- mutex_unlock(&callback_mutex);
-
  mm = get_task_mm(tsk);
  if (mm) {
```

```
   mpol_rebind_mm(mm, &to);
@@ -1254,41 +871,31 @@ static int attach_task(struct cpuset *cs
    cpuset_migrate_mm(mm, &from, &to);
   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 nsproxy *ns,
+     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 = ns_cs(ns);
  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 +912,9 @@ static ssize_t cpuset_common_file_write(
```

```
  }
  buffer[nbytes] = 0; /* nul-terminate */

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

- if (is_removed(cs)) {
+ if (rcfs_dir_removed(file->f_dentry->d_parent)) {
   retval = -ENODEV;
   goto out2;
  }
@@ -1325,9 +932,6 @@ 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 +949,6 @@ 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 +957,12 @@ static ssize_t cpuset_common_file_write(
  if (retval == 0)
   retval = nbytes;
 out2:
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
+ rcfs_manage_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 +997,13 @@ 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 nsproxy *ns,
+           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 = ns_cs(ns);
  cpuset_filetype_t type = cft->private;
  char *page;
  ssize_t retval = 0;
@@ -1442,9 +1027,6 @@ 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 +1054,101 @@ 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(struct rc_subsys *ss, struct dentry *cs_dentry)
 {
   int err;
+ struct nsproxy *ns = cs_dentry->d_fsdata;
+ struct cpuset *cs = ns_cs(ns);
```

```
- if ((err = cpuset_add_file(cs_dentry, &cft_cpus)) < 0)
-  return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_mems)) < 0)
-  return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_cpu_exclusive)) < 0)
+ cs->dentry = cs_dentry; /* do we need to d_get? */
+
+ if ((err = rcfs_add_file(cs_dentry, &cft_cpus)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_mem_exclusive)) < 0)
+ if ((err = rcfs_add_file(cs_dentry, &cft_mems)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_notify_on_release)) < 0)
+ if ((err = rcfs_add_file(cs_dentry, &cft_cpu_exclusive)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_memory_migrate)) < 0)
+ if ((err = rcfs_add_file(cs_dentry, &cft_mem_exclusive)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_memory_pressure)) < 0)
+ if ((err = rcfs_add_file(cs_dentry, &cft_memory_migrate)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_spread_page)) < 0)
+ if ((err = rcfs_add_file(cs_dentry, &cft_memory_pressure)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_spread_slab)) < 0)
+ if ((err = rcfs_add_file(cs_dentry, &cft_spread_page)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_tasks)) < 0)
+ if ((err = rcfs_add_file(cs_dentry, &cft_spread_slab)) < 0)
   return err;
+ /* memory_pressure_enabled is in root cpuset only */
+ if (err == 0 && !cs->parent)
+  err = rcfs_add_file(cs_dentry, &cft_memory_pressure_enabled);
+
  return 0;
 }

@@ -1869,23 +1161,28 @@ 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 rc_subsys *ss, struct nsproxy *ns,
+     struct nsproxy *parent)
 {
- struct cpuset *cs;
- int err;
```

```
+ struct cpuset *cs, *parent_cs;
+
+ if (!parent) {
+  /* This is early initialization for the top container */
+  set_cs(ns, &top_cpuset);
+  top_cpuset.mems_generation = cpuset_mems_generation++;
+  return 0;
+ }

  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))
+ parent_cs = ns_cs(parent);
+ if (is_spread_page(parent_cs))
   set_bit(CS_SPREAD_PAGE, &cs->flags);
- if (is_spread_slab(parent))
+ if (is_spread_slab(parent_cs))
   set_bit(CS_SPREAD_SLAB, &cs->flags);
  cs->cpus_allowed = CPU_MASK_NONE;
  cs->mems_allowed = NODE_MASK_NONE;
@@ -1895,40 +1192,16 @@ static long cpuset_create(struct cpuset
  cs->mems_generation = cpuset_mems_generation++;
  fmeter_init(&cs->fmeter);

- cs->parent = parent;
+ cs->parent = parent_cs;
+
+ set_cs(ns, cs);

  mutex_lock(&callback_mutex);
  list_add(&cs->sibling, &cs->parent->children);
  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);
 }

 /*
@@ -1942,51 +1215,39 @@ 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 rc_subsys *ss, struct nsproxy *ns)
 {
- struct cpuset *cs = dentry->d_fsdata;
- struct dentry *d;
+ struct cpuset *cs = ns_cs(ns);
  struct cpuset *parent;
- char *pathbuf = NULL;
-
- /* the vfs holds both inode->i_mutex already */

- 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 (atomic_read(&cs->count) > 0 || !list_empty(&cs->children))
+  BUG();
+
  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;
+ kfree(cs); /* Should it be moved to put_cs ? */
 }

+static struct rc_subsys cpuset_subsys = {
+       .name = "cpuset",
+       .create = cpuset_create,
+       .destroy  = cpuset_destroy,
+       .can_attach = cpuset_can_attach,
+       .attach = cpuset_attach,
+       .populate = cpuset_populate,
+       .subsys_id = -1,
+};
+
+
 /*
  * cpuset_init_early - just enough so that the calls to
  * cpuset_update_task_memory_state() in early init code
@@ -1995,10 +1256,10 @@ static int cpuset_rmdir(struct inode *un
```

```
 int __init cpuset_init_early(void)
 {
- struct task_struct *tsk = current;
+ if (rc_register_subsys(&cpuset_subsys) < 0)
+  panic("Couldn't register cpuset subsystem");
+ top_cpuset.mems_generation = cpuset_mems_generation++;

- tsk->cpuset = &top_cpuset;
- tsk->cpuset->mems_generation = cpuset_mems_generation++;
  return 0;
 }

@@ -2010,7 +1271,6 @@ int __init cpuset_init_early(void)

 int __init cpuset_init(void)
 {
- struct dentry *root;
  int err;

  top_cpuset.cpus_allowed = CPU_MASK_ALL;
@@ -2019,30 +1279,11 @@ int __init cpuset_init(void)
  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;
 }

 /*
@@ -2098,7 +1339,7 @@ static void guarantee_online_cpus_mems_i

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

  guarantee_online_cpus_mems_in_subtree(&top_cpuset);
@@ -2106,7 +1347,7 @@ static void common_cpu_mem_hotplug_unplu
  top_cpuset.mems_allowed = node_online_map;

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

 /*
@@ -2154,111 +1395,6 @@ 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 +1410,7 @@ cpumask_t cpuset_cpus_allowed(struct tas

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

```
@@ -2302,7 +1438,7 @@ nodemask_t cpuset_mems_allowed(struct ta

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

@@ -2423,7 +1559,7 @@ int __cpuset_zone_allowed_softwall(struc
  mutex_lock(&callback_mutex);

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

  allowed = node_isset(node, cs->mems_allowed);
@@ -2552,7 +1688,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock(current);
   goto done;
  }
- cs1 = nearest_exclusive_ancestor(current->cpuset);
+ cs1 = nearest_exclusive_ancestor(task_cs(current));
  task_unlock(current);

  task_lock((struct task_struct *)p);
@@ -2560,7 +1696,7 @@ 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(task_cs(p));
  task_unlock((struct task_struct *)p);

  overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
@@ -2596,14 +1732,13 @@ int cpuset_memory_pressure_enabled __rea

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

```
+#ifdef CONFIG_PROC_PID_CPUSET
+
 /*
  * proc_cpuset_show()
  *  - Print tasks cpuset path into seq_file.
@@ -2634,15 +1769,15 @@ static int proc_cpuset_show(struct seq_f
   goto out_free;

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

- retval = cpuset_path(tsk->cpuset, buf, PAGE_SIZE);
+ retval = rcfs_path(task_cs(tsk)->dentry, buf, PAGE_SIZE);
  if (retval < 0)
   goto out_unlock;
  seq_puts(m, buf);
  seq_putc(m, '\n');
 out_unlock:
- mutex_unlock(&manage_mutex);
+ rcfs_manage_unlock();
  put_task_struct(tsk);
 out_free:
  kfree(buf);
@@ -2662,6 +1797,7 @@ 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)
diff -puN kernel/exit.c~cpuset_uses_rcfs kernel/exit.c
--- linux-2.6.20.1/kernel/exit.c~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/kernel/exit.c 2007-03-08 22:35:35.000000000 +0530
@@ -30,7 +30,6 @@
 #include <linux/mempolicy.h>
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
-#include <linux/cpuset.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
 #include <linux/posix-timers.h>
@@ -926,7 +925,6 @@ fastcall NORET_TYPE void do_exit(long co
  __exit_files(tsk);
  __exit_fs(tsk);
  exit_thread();
```

```
- cpuset_exit(tsk);
  exit_keys(tsk);

  if (group_dead && tsk->signal->leader)
diff -puN kernel/fork.c~cpuset_uses_rcfs kernel/fork.c
--- linux-2.6.20.1/kernel/fork.c~cpuset_uses_rcfs 2007-03-08 22:35:35.000000000 +0530
+++ linux-2.6.20.1-vatsa/kernel/fork.c 2007-03-08 22:35:35.000000000 +0530
@@ -30,7 +30,6 @@
 #include <linux/nsproxy.h>
 #include <linux/capability.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
 #include <linux/security.h>
 #include <linux/swap.h>
 #include <linux/syscalls.h>
@@ -1058,13 +1057,12 @@ static struct task_struct *copy_process(
  p->io_context = NULL;
  p->io_wait = NULL;
  p->audit_context = NULL;
- 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_delays_binfmt;
   }
  mpol_fix_fork_child_flag(p);
 #endif
@@ -1288,9 +1286,7 @@ bad_fork_cleanup_security:
 bad_fork_cleanup_policy:
 #ifdef CONFIG_NUMA
  mpol_free(p->mempolicy);
-bad_fork_cleanup_cpuset:
 #endif
- cpuset_exit(p);
 bad_fork_cleanup_delays_binfmt:
  delayacct_tsk_free(p);
  if (p->binfmt)
_
```

Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

File Attachments

1) rcfs.patch, downloaded 317 times
2) cpu_acct.patch, downloaded 329 times
3) cpuset_uses_rcfs.patch, downloaded 335 times