Subject: Re: [PATCH 1/2] rcfs core patch
Posted by serue on Thu, 01 Mar 2007 16:31:26 GMT
View Forum Message <> Reply to Message

Quoting Srivatsa Vaddagiri (vatsa@in.ibm.com):
> Heavily based on Paul Menage's (inturn cpuset) work. The big difference
> is that the patch uses task->nsproxy to group tasks for resource control
> purpose (instead of task->containers).

Hi,

we've already had some trouble with nsproxy holding things with
different lifetimes. As it happens the solution this time was to put
the pid namespace where it belongs - not in nsproxy - so maybe moving
this info into nsproxies will be fine, it just rings a warning bell.

This patch would be all the more reason to do
http://lkml.org/lkml/2007/2/21/217
first though, as 'exit_task_namespaces' becomes even more of a lie.

-serge

> The patch retains the same user interface as Paul Menage's patches. In
> particular, you can have multiple hierarchies, each hierarchy giving a
> different composition/view of task-groups.
>
> (Ideally this patch should have been split into 2 or 3 sub-patches, but
> will do that on a subsequent version post)
>
> Signed-off-by : Srivatsa Vaddagiri <vatsa@in.ibm.com>
> Signed-off-by : Paul Menage <menage@google.com>
>
>
> ---
>
> linux-2.6.20-vatsa/include/linux/init_task.h | 4
> linux-2.6.20-vatsa/include/linux/nsproxy.h | 5
> linux-2.6.20-vatsa/init/Kconfig | 22
> linux-2.6.20-vatsa/init/main.c | 1
> linux-2.6.20-vatsa/kernel/Makefile | 1
>
>
> ---
>
>
> diff -puN include/linux/init_task.h~rcfs include/linux/init_task.h
> --- linux-2.6.20/include/linux/init_task.h~rcfs 2007-03-01 14:20:47.000000000 +0530
> +++ linux-2.6.20-vatsa/include/linux/init_task.h 2007-03-01 14:20:47.000000000 +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/include/linux/nsproxy.h~rcfs 2007-03-01 14:20:47.000000000 +0530
> +++ linux-2.6.20-vatsa/include/linux/nsproxy.h 2007-03-01 14:20:47.000000000 +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 +39,12 @@ 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);
> +#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 2006-02-25 03:06:56.000000000 +0530
> +++ linux-2.6.20-vatsa/include/linux/rcfs.h 2007-03-01 14:20:47.000000000 +0530
> @@ -0,0 +1,72 @@
> +#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, void *new, void *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);
> +
> +#else
> +
> +static inline int rcfs_init(void) { return 0; }
> +
> +#endif
> +
> +
> +#endif
> diff -puN init/Kconfig~rcfs init/Kconfig
> --- linux-2.6.20/init/Kconfig~rcfs 2007-03-01 14:20:47.000000000 +0530
> +++ linux-2.6.20-vatsa/init/Kconfig 2007-03-01 16:52:50.000000000 +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/init/main.c~rcfs 2007-03-01 14:20:47.000000000 +0530
> +++ linux-2.6.20-vatsa/init/main.c 2007-03-01 14:20:47.000000000 +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()
> @@ -608,6 +610,7 @@ asmlinkage void __init start_kernel(void
> #ifdef CONFIG_PROC_FS
> 	proc_root_init();
> #endif
> +	rcfs_init();
> 	cpuset_init();
> 	taskstats_init_early();
> 	delayacct_init();
> diff -puN kernel/Makefile~rcfs kernel/Makefile
> --- linux-2.6.20/kernel/Makefile~rcfs 2007-03-01 14:20:47.000000000 +0530
> +++ linux-2.6.20-vatsa/kernel/Makefile 2007-03-01 16:52:50.000000000 +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/kernel/nsproxy.c~rcfs 2007-03-01 14:20:47.000000000 +0530
> +++ linux-2.6.20-vatsa/kernel/nsproxy.c 2007-03-01 14:20:47.000000000 +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(&nslistlock);
> + list_add(&ns->list, &nslisthead);
> + spin_unlock(&nslistlock);
> +#endif
> +
> return ns;
> }
>
> @@ -145,5 +156,44 @@ 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(&nslistlock);
> + list_del(&ns->list);
> + spin_unlock(&nslistlock);
> +#endif
> kfree(ns);
> }
> +
> +#ifdef CONFIG_RCFS
> +struct nsproxy *find_nsproxy(struct nsproxy *target)

```
> +{
> + struct nsproxy *ns;
> + int i = 0;
> +
> + spin_lock(&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(&nslistlock);
> +
> + ns = dup_namespaces(target);
> + return ns;
> +}
> +
> +int __init namespaces_init(void)
> +{
> + list_add(&init_nsproxy.list, &nslisthead);
> +
> + return 0;
> +}
> +#endif
> diff -puN /dev/null kernel/rcfs.c
> --- /dev/null 2006-02-25 03:06:56.000000000 +0530
> +++ linux-2.6.20-vatsa/kernel/rcfs.c 2007-03-01 16:53:24.000000000 +0530
> @@ -0,0 +1,1138 @@
> +/*
> + *  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;
> +}
> +
> +/* 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(const struct nsproxy *ns)
> +{
> + int count = 0;
> +
> + count = atomic_read(&ns->count);
> +
> + 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 nsproxy *ns)
> +{
> + int n = 0;
> + struct task_struct *g, *p;
> +
> + rcu_read_lock();
> + read_lock(&tasklist_lock);
> +
> + do_each_thread(g, p) {
```

```
> +  if (p->nsproxy == ns) {
> +   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;
> +}
> +
> +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;
> +}
> +
> +/*
> + * 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 nsproxy *ns = __d_ns(file->f_dentry->d_parent);
> + struct ctr_struct *ctr;
> + pid_t *pidarray;
> + int npids;
> + char c;
> +
> + if (!(file->f_mode & FMODE_READ))
> +  return 0;
> +
> + ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
> + if (!ctr)
> +  goto err0;
> +
> + /*
> +  * If container gets more users after we read count, we won't have
> +  * enough space - tough.  This race is indistinguishable to the
> +  * caller from the case that the additional container users didn't
> +  * show up until sometime later on.
> +  */
> + npids = rcfs_task_count(ns);
> + pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
> + if (!pidarray)
> +  goto err1;
> +
> + npids = pid_array_load(pidarray, npids, ns);
> + 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 (!ns) {
> +  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;
> + }
> +
> + 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;
> +}
> +
> _
>
> --
> Regards,
> vatsa
```

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