Subject: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Srivatsa Vaddagiri on Thu, 01 Mar 2007 13:35:43 GMT

View Forum Message <> Reply to Message

Paul,

 Based on some of the feedback to container patches, I have
respun them to avoid the "container" structure abstraction and instead use
nsproxy structure in the kernel. User interface (which I felt was neat
in your patches) has been retained to be same.

What follows is the core (big) patch and the cpu_acct subsystem to serve
as an example of how to use it. I suspect we can make cpusets also work
on top of this very easily.

Oh and since most of the code is serving the purpose of being a
filesystem, I have renamed the patch to be a resource control file
system - rcfs!

--
Regards,
vatsa

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

Subject: [PATCH 1/2] rcfs core patch

Posted by Srivatsa Vaddagiri on Thu, 01 Mar 2007 13:45:28 GMT

View Forum Message <> Reply to Message

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).

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

Subject: [PATCH 2/2] cpu_accounting controller
Posted by Srivatsa Vaddagiri on Thu, 01 Mar 2007 13:50:08 GMT
View Forum Message <> Reply to Message

This patch demonstrates how a resource controller can work with rcfs.

The controller counts the total CPU time used by all processes in a
resource container, during the time that they're members of the
container.

Written by Paul Menage. Adapted to work with rcfs by Srivatsa.

Signed-off-by : Paul Menage <menage@google.com>
Signed-off-by : Srivatsa Vaddagiri <vatsa@in.ibm.com>

```
diff -puN /dev/null include/linux/cpu_acct.h
--- /dev/null 2006-02-25 03:06:56.000000000 +0530
+++ linux-2.6.20-vatsa/include/linux/cpu_acct.h 2007-03-01 16:53:39.000000000 +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/init/Kconfig~cpu_acct 2007-03-01 16:53:39.000000000 +0530
+++ linux-2.6.20-vatsa/init/Kconfig 2007-03-01 16:53:39.000000000 +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 2006-02-25 03:06:56.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/cpu_acct.c 2007-03-01 16:53:39.000000000 +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/kernel/Makefile~cpu_acct 2007-03-01 16:53:39.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/Makefile 2007-03-01 16:53:39.000000000 +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/kernel/sched.c~cpu_acct 2007-03-01 16:53:39.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-03-01 16:53:39.000000000 +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)
_
```

--
Regards,
vatsa
____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [ckrm-tech] [PATCH 1/2] rcfs core patch
Posted by Balbir Singh on Fri, 02 Mar 2007 05:06:49 GMT

View Forum Message <> Reply to Message

Srivatsa Vaddagiri wrote:

> 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).
>
> 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)

With this don't we end up with a lot of duplicate between cpusets and rcfs.

```
> 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
>
>
> ---
```

The diffstat does not look quite right.

--
 Warm Regards,
 Balbir Singh
 Linux Technology Center
 IBM, ISTL

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by akpm on Fri, 02 Mar 2007 16:52:24 GMT

View Forum Message <> Reply to Message

On Fri, 02 Mar 2007 18:45:06 +0300 Kirill Korotaev <dev@openvz.org> wrote:

> > I'm wagering you'll break either the semantics, and/or the
> > performance, of cpusets doing this.
> I like Paul's containers patch. It looks good and pretty well.
> After some of the context issues are resolved it's fine.
> Maybe it is even the best way of doing things.

Have you thought about the relationship between it and UBC?

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by dev on Fri, 02 Mar 2007 17:25:36 GMT

View Forum Message <> Reply to Message

Andrew,

>>>I'm wagering you'll break either the semantics, and/or the
>>>performance, of cpusets doing this.
>>
>>I like Paul's containers patch. It looks good and pretty well.
>>After some of the context issues are resolved it's fine.

>>Maybe it is even the best way of doing things.
>
>
> Have you thought about the relationship between it and UBC?

Sure.

Mostly containers patch does 2 things:
1. user space interfaces (if people don't like system calls used in UBC
  we are fine with filesystems approach. why not?)
2. context handling

So (1) is ok with us.
(2) requires some more work to be done before we are fine.
Actually all we want is lockless context handling all over the code
and looks like it is clear how to do it. Good.

UBC on the other hand can provide containers subsystems, i.e.
memory and other resources accounting and limiting etc.
This is what we are working on right now.

Thanks,
Kirill

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

Subject: Re: [ckrm-tech] [PATCH 1/2] rcfs core patch
Posted by Srivatsa Vaddagiri on Sat, 03 Mar 2007 09:38:24 GMT

View Forum Message <> Reply to Message

On Fri, Mar 02, 2007 at 10:36:49AM +0530, Balbir Singh wrote:
> With this don't we end up with a lot of duplicate between cpusets and rcfs.

Unless we remove the duplication in cpusets and make it work with
rcfs/containers!

I wonder if we can avoid so much of filesystem code and use something
like configfs to configure the resource objects. In particular I dont
know if, using configfs, it is possible to provide a multi-hierarchy feature
(each hierarchy bound to separate set of controllers)

```
> > 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
> >
> >
> >---
>
> The diffstat does not look quite right.

Hmm that was generated using refpatch. Will find out what went wrong.

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Jackson on Sat, 03 Mar 2007 10:21:00 GMT
View Forum Message <> Reply to Message

> Regarding semantics, can you be more specific?

Unfortunately not - sorry.

I've been off in other areas, and not found the time
to read through this current PATCH or think about it
carefully enough to be really useful.

Your reply seemed reasonable enough.

> It should have the same perf overhead as the original container patches
> (basically a double dereference - task->containers/nsproxy->cpuset -
> required to get to the cpuset from a task).

There is just one spot that this might matter to cpusets.

Except for one hook, cpusets uses the mems_allowed and cpus_allowed
masks in the task struct to avoid having to look at the cpuset on hot
code paths.

There is one RCU guarded reference per memory allocation to
 current->cpuset->mems_generation
in the call to cpuset_update_task_memory_state(), for tasks that are in
some cpuset -other- than the default top cpuset, on systems that have
explicitly created additional (other than the top cpuset) cpusets after

boot.

If that RCU guarded reference turned into taking a global lock, or
pulling in a cache line that was frequently off dirty in some other
node, that would be unfortunate.

But that's the key hook so far as cpuset performance impact is
concerned.

Perhaps you could summarize what becomes of this hook, in this
brave new world of rcfs ...

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Jackson on Sat, 03 Mar 2007 21:22:44 GMT

View Forum Message <> Reply to Message

Herbert wrote:
> I agree here, there is not much difference for the
> following aspects:

Whether two somewhat similar needs should be met by one shared
mechanism, or two distinct mechanisms, cannot really be decided by
listing the similarities.

One has to determine if there are any significant differences in
needs that are too difficult for a shared mechanism to provide.

A couple of things you wrote in your second message might
touch on such possible significant differences:
- resources must be hierarchically suballocated, and
- key resource management code hooks can't cause hot cache lines.

In a later message, Herbert wrote:
> well, the thing is, as nsproxy is working now, you
> will get a new one (with a changed subset of entries)
> every time a task does a clone() with one of the
> space flags set, which means, that you will end up

> with quite a lot of them, but resource limits have
> to address a group of them, not a single nsproxy
> (or act in a deeply hierarchical way which is not
> there atm, and probably will never be, as it simply
> adds too much overhead)

I still can't claim to have my head around this, but what you write here, Herbert, writes here touches on what I suspect is a key difference between namespaces and resources that would make it impractical to accomplish both with a shared mechanism for aggregating tasks.

It is a natural and desirable capability when managing resources, which are relatively scarce (that's why they're worth all this trouble) commodities of which we have some limited amount, to subdivide allowances of them. Some group of tasks gets the right to use certain memory pages or cpu time slices, and in turn suballocates that allotment to some subgroup of itself. This naturally leads to a hierarchy of allocated resources.

There is no such necessary, hierarchy for name spaces. One name space might be derived from another at setup, by some arbitrary conventions, but once initialized, this way or that, they are separate name spaces, or at least naturally can (must?) be separate.

The cpuset hierarchy is an important part of the API that cpusets presents to user space, where that hierarchy reflects the suballocation of resources. If B is a child of A in the cpuset hierarchy, then the CPUs and Memory Nodes allowed to B -must- be a subset of those allowed to A. That is the key semantic of the cpuset hierarchy. This includes forcing the removal of a resource from B if for some reason it must be removed from A, in order to preserve the hierarchical suballocation, which requirement is causing a fair bit of hard work for the cpu hot unplug folks.

I am quite willing to believe that name spaces has no need for such a hierarchy, and further that it probably never will have such ... "too much overhead" as you say.

> > It should have the same perf overhead as the original
> > container patches (basically a double dereference -
> > task->containers/nsproxy->cpuset - required to get to the
> > cpuset from a task).
>
> on every limit accounting or check? I think that
> is quite a lot of overhead ...

Do either of these dereferences require locks?

The two critical resources that cpusets manages, memory pages and
time slices on a cpu, cannot afford such dereferences or locking
in the key code paths (allocating a page or scheduling a cpu.) The
existing cpuset code is down to one RCU guarded dereference of
current->cpuset in the page allocation code path (and then only on
systems actively using cpusets), and no such dereferences at all in the
scheduler path.

It took a fair bit of hard work (for someone of my modest abilities)
to get that far; I doubt we can accept much regression on this point.

Most likely the other folks doing resource management will have similar
concerns in many cases - memory pages and cpu slices are not the only
resources we're trying to manage on critical code paths.

In short - the issues seem to be:
 - resources need to be hierarchical, name spaces don't (can't?), and
 - no hot cache lines allowed by the resource hooks in key code paths.

--
                  I won't rest till it's the best ...
                  Programmer, Linux Scalability
                  Paul Jackson <pj@sgi.com> 1.925.600.0401

____

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Mon, 05 Mar 2007 17:02:44 GMT
View Forum Message <> Reply to Message

On Sat, Mar 03, 2007 at 02:21:00AM -0800, Paul Jackson wrote:
> Perhaps you could summarize what becomes of this hook, in this
> brave new world of rcfs ...

attach_task() still uses a synchronize_rcu before doing a put_nsproxy in
the rcfs patches. This means cpuset_update_task_memory_state() can read a
task's cpuset->mems_generation under just a rcu_read_lock() (as it is
doing currently).

```
void cpuset_update_task_memory_state(void)
{
 tsk = current;
 ...

	} else {
		rcu_read_lock();
		ns = rcu_dereference(tsk->nsproxy);
		my_cpusets_mem_gen =
  ns->ctlr_data[cpuset_ctlr.subsys_id]->mems_generation;
		rcu_read_unlock();
	}

 ...
}
```

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Mon, 05 Mar 2007 17:47:50 GMT

View Forum Message <> Reply to Message

On Sat, Mar 03, 2007 at 01:22:44PM -0800, Paul Jackson wrote:
> I still can't claim to have my head around this, but what you write
> here, Herbert, writes here touches on what I suspect is a key
> difference between namespaces and resources that would make it
> impractical to accomplish both with a shared mechanism for aggregating
> tasks.

The way nsproxy is structured, its all pointers to actual namespace (or
in case of rcfs patch) resource objects. This lets namespaces objects be in a
flat hierarchy while resource objects are in tree-like hierarchy.
nsproxy itself doesnt decide any hierarchy. Its those objects pointed to
by nsproxy which can form different hierarchies. In fact the rcfs
patches allows such a combination afaics.

> > on every limit accounting or check? I think that
> > is quite a lot of overhead ...
>
> Do either of these dereferences require locks?

A rcu_read_lock() should be required, which is not that expensive.

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Paul Menage on Wed, 07 Mar 2007 02:32:07 GMT

View Forum Message <> Reply to Message

Hi Vatsa,

Sorry for the delayed reply - the last week has been very busy ...

On 3/1/07, Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:
> Paul,
>   Based on some of the feedback to container patches, I have
> respun them to avoid the "container" structure abstraction and instead use
> nsproxy structure in the kernel. User interface (which I felt was neat
> in your patches) has been retained to be same.

I'm not really sure that I see the value of having this be part of nsproxy rather than the previous independent container (and container_group) structure. As far as I can see, you're putting the container subsystem state pointers and the various task namespace pointers into the same structure (nsproxy) but then they're remaining pretty much independent in terms of code.

The impression that I'm getting (correct me if I'm wrong) is:

- when you do a mkdir within an rcfs directory, the nsproxy associated with the parent is duplicated, and then each rcfs subsystem gets to set a subsystem-state pointer in that nsproxy

- when you move a task into an rcfs container, you create a new nsproxy consisting of the task's old namespaces and its new subsystem pointers. Then you look through the current list of nsproxy objects to see if you find one that matches. If you do, you reuse it, else you create a new nsproxy and link it into the list

- when you do sys_unshare() or a clone that creates new namespaces, then the task (or its child) will get a new nsproxy that has the rcfs subsystem state associated with the old nsproxy, and one or more

namespace pointers cloned to point to new namespaces. So this means
that the nsproxy for the task is no longer the nsproxy associated with
any directory in rcfs. (So the task will disappear from any "tasks"
file in rcfs?)

You seem to have lost some features, including fork/exit subsystem callbacks

>
> What follows is the core (big) patch and the cpu_acct subsystem to serve
> as an example of how to use it. I suspect we can make cpusets also work
> on top of this very easily.

I'd like to see that. I suspect it will be a bit more fiddly than the
simple cpu_acct subsystem.

Paul

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Menage on Wed, 07 Mar 2007 17:29:12 GMT

View Forum Message <> Reply to Message

On 3/7/07, Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:
>
> > - when you do sys_unshare() or a clone that creates new namespaces,
> > then the task (or its child) will get a new nsproxy that has the rcfs
> > subsystem state associated with the old nsproxy, and one or more
> > namespace pointers cloned to point to new namespaces. So this means
> > that the nsproxy for the task is no longer the nsproxy associated with
> > any directory in rcfs. (So the task will disappear from any "tasks"
> > file in rcfs?)
>
> it "should" disappear yes, although I haven't carefully studied the
> unshare requirements yet.

That seems bad. With the current way you're doing it, if I mount
hierarchies A and B on /mnt/A and /mnt/B, then initially all tasks are
in /mnt/A/tasks and /mnt/B/tasks. If I then create /mnt/A/foo and move
a process into it, that process disappears from /mnt/B/tasks, since
its nsproxy no longer matches the nsproxy of B's root container. Or am
I missing something?

Paul

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Wed, 07 Mar 2007 17:30:31 GMT

View Forum Message <> Reply to Message

On Tue, Mar 06, 2007 at 06:32:07PM -0800, Paul Menage wrote:

> I'm not really sure that I see the value of having this be part of
> nsproxy rather than the previous independent container (and
> container_group) structure.

shrug

I wrote the patch mainly to see whether the stuff container folks (Sam Vilain et al) were complaining abt (that container structure abstraction inside the kernel is redundant/unnecessary) made sense or not.

The rcfs patches demonstrate that it is possible to implement resource control on top of just nsproxy -and- give the same interface that you now have. In essense, I would say that the rcfs patches are about 70% same as your original V7 container patches.

However as I am converting over cpusets to work on top of nsproxy, I have learnt few things:

container structure in your patches provides for these things:

a. A way to group tasks
b. A way to maintain several hierarchies of such groups

If you consider just a. then I agree that container abstraction is redundant, esp for vserver resource control (nsproxy can already be used to group tasks).

What nsproxy doesn't provide is b - a way to represent hierarchies of groups.

So we got several choices here.

1. Introduce the container abstraction as is in your patches
2. Extend nsproxy somehow to represent hierarchies
3. Let individual resource controllers that -actually- support

hierarchical resource management maintain hierarchy in their code.

In the last option, nsproxy still is unaware of any hierarchy. Some of the resource objects it points to (for ex: cpuset) may maintain a hierarchy. For ex: nsproxy->ctlr_data[cpuset_subsys.subsys_id] points to a 'struct cpuset' structure which could maintains the hierarchical relationship among cpuset objects.

If we consider that most resource controllers may not implement hierarchical resource management, then 3 may not be a bad compromise. OTOH if we expect *most* resource controllers to support hierarchical resource management, then we could be better of with option 1.

Anyway, summarizing on "why nsproxy", the main point (I think) is about using existing abstraction in the kernel.

> As far as I can see, you're putting the
> container subsystem state pointers and the various task namespace
> pointers into the same structure (nsproxy) but then they're remaining
> pretty much independent in terms of code.
>
> The impression that I'm getting (correct me if I'm wrong) is:
>
> - when you do a mkdir within an rcfs directory, the nsproxy associated
> with the parent is duplicated, and then each rcfs subsystem gets to
> set a subsystem-state pointer in that nsproxy

yes.

> - when you move a task into an rcfs container, you create a new
> nsproxy consisting of the task's old namespaces and its new subsystem
> pointers. Then you look through the current list of nsproxy objects to
> see if you find one that matches. If you do, you reuse it, else you
> create a new nsproxy and link it into the list

yes

> - when you do sys_unshare() or a clone that creates new namespaces,
> then the task (or its child) will get a new nsproxy that has the rcfs
> subsystem state associated with the old nsproxy, and one or more
> namespace pointers cloned to point to new namespaces. So this means
> that the nsproxy for the task is no longer the nsproxy associated with
> any directory in rcfs. (So the task will disappear from any "tasks"
> file in rcfs?)

it "should" disappear yes, although I haven't carefully studied the unshare requirements yet.

> You seem to have lost some features, including fork/exit subsystem callbacks

That was mainly to keep it simple for a proof-of-concept patch! We can add it
back later.

> >What follows is the core (big) patch and the cpu_acct subsystem to serve
> >as an example of how to use it. I suspect we can make cpusets also work
> >on top of this very easily.
>
> 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.

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Wed, 07 Mar 2007 17:32:26 GMT

View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 11:00:31PM +0530, Srivatsa Vaddagiri wrote:
> So we got several choices here.
>
> 1. Introduce the container abstraction as is in your patches
> 2. Extend nsproxy somehow to represent hierarchies
> 3. Let individual resource controllers that -actually- support
>   hierarchical resource management maintain hierarchy in their code.

3 is what I am following for the cpuset conversion (currently under
test).

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by serue on Wed, 07 Mar 2007 17:43:46 GMT
View Forum Message <> Reply to Message

Quoting Srivatsa Vaddagiri (vatsa@in.ibm.com):
> On Tue, Mar 06, 2007 at 06:32:07PM -0800, Paul Menage wrote:
> > I'm not really sure that I see the value of having this be part of
> > nsproxy rather than the previous independent container (and
> > container_group) structure.
>
> *shrug*
>
> I wrote the patch mainly to see whether the stuff container folks (Sam Vilain
> et al) were complaining abt (that container structure abstraction
> inside the kernel is redundant/unnecessary) made sense or not.

I still think the complaint was about terminology, not implementation.
They just didn't want you calling them containers.

> The rcfs patches demonstrate that it is possible to implement resource control
> on top of just nsproxy -and- give the same interface that you now
> have. In essense, I would say that the rcfs patches are about 70% same as your
> original V7 container patches.
>
> However as I am converting over cpusets to work on top of nsproxy, I
> have learnt few things:
>
> container structure in your patches provides for these things:
>
> a. A way to group tasks
> b. A way to maintain several hierarchies of such groups
>
> If you consider just a. then I agree that container abstraction is
> redundant, esp for vserver resource control (nsproxy can already be used
> to group tasks).
>
> What nsproxy doesn't provide is b - a way to represent hierarchies of
> groups.
>
> So we got several choices here.
>
> 1. Introduce the container abstraction as is in your patches
> 2. Extend nsproxy somehow to represent hierarchies
> 3. Let individual resource controllers that -actually- support
>   hierarchical resource management maintain hierarchy in their code.
>
> In the last option, nsproxy still is unaware of any hierarchy. Some of
> the resource objects it points to (for ex: cpuset) may maintain a

> hierarchy. For ex: nsproxy->ctlr_data[cpuset_subsys.subsys_id] points to
> a 'struct cpuset' structure which could maintains the hierarchical
> relationship among cpuset objects.
>
> If we consider that most resource controllers may not implement hierarchical
> resource management, then 3 may not be a bad compromise. OTOH if we
> expect *most* resource controllers to support hierarchical resource
> management, then we could be better of with option 1.
>
> Anyway, summarizing on "why nsproxy", the main point (I think) is about
> using existing abstraction in the kernel.

But nsproxy is not an abstraction, it's an implementation
detail/optimization. I'm mostly being quiet because i don't
particularly care if it gets expanded upon, but it's nothing more than
that right now.

> > As far as I can see, you're putting the
> > container subsystem state pointers and the various task namespace
> > pointers into the same structure (nsproxy) but then they're remaining
> > pretty much independent in terms of code.
> >
> > The impression that I'm getting (correct me if I'm wrong) is:
> >
> > - when you do a mkdir within an rcfs directory, the nsproxy associated
> > with the parent is duplicated, and then each rcfs subsystem gets to
> > set a subsystem-state pointer in that nsproxy
>
> yes.
>
> > - when you move a task into an rcfs container, you create a new
> > nsproxy consisting of the task's old namespaces and its new subsystem
> > pointers. Then you look through the current list of nsproxy objects to
> > see if you find one that matches. If you do, you reuse it, else you
> > create a new nsproxy and link it into the list
>
> yes
>
> > - when you do sys_unshare() or a clone that creates new namespaces,
> > then the task (or its child) will get a new nsproxy that has the rcfs
> > subsystem state associated with the old nsproxy, and one or more
> > namespace pointers cloned to point to new namespaces. So this means
> > that the nsproxy for the task is no longer the nsproxy associated with
> > any directory in rcfs. (So the task will disappear from any "tasks"
> > file in rcfs?)
>
> it "should" disappear yes, although I haven't carefully studied the
> unshare requirements yet.

>
> > You seem to have lost some features, including fork/exit subsystem callbacks
>
> That was mainly to keep it simple for a proof-of-concept patch! We can add it
> back later.
>
> > >What follows is the core (big) patch and the cpu_acct subsystem to serve
> > >as an example of how to use it. I suspect we can make cpusets also work
> > >on top of this very easily.
> >
> > 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.
>
> --
> Regards,
> vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Menage on Wed, 07 Mar 2007 17:46:35 GMT

View Forum Message <> Reply to Message

On 3/7/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
> Quoting Srivatsa Vaddagiri (vatsa@in.ibm.com):
> > On Tue, Mar 06, 2007 at 06:32:07PM -0800, Paul Menage wrote:
> > > I'm not really sure that I see the value of having this be part of
> > > nsproxy rather than the previous independent container (and
> > > container_group) structure.
> >
> > *shrug*
> >
> > I wrote the patch mainly to see whether the stuff container folks (Sam Vilain
> > et al) were complaining abt (that container structure abstraction
> > inside the kernel is redundant/unnecessary) made sense or not.
>
> I still think the complaint was about terminology, not implementation.

No, Sam was saying that nsproxy should be the object that all resource
controllers hook off.

Paul

Subject: Re: [ckrm-tech] [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Srivatsa Vaddagiri on Wed, 07 Mar 2007 17:52:57 GMT

View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 09:29:12AM -0800, Paul Menage wrote:
> That seems bad. With the current way you're doing it, if I mount
> hierarchies A and B on /mnt/A and /mnt/B, then initially all tasks are
> in /mnt/A/tasks and /mnt/B/tasks. If I then create /mnt/A/foo and move
> a process into it, that process disappears from /mnt/B/tasks, since
> its nsproxy no longer matches the nsproxy of B's root container. Or am
> I missing something?

I realized that bug as I was doing cpuset conversion.

Basically, we can't use just tsk->nsproxy to find what tasks are in
a directory (/mnt/B for ex). Here's what I was think we should be doing instead:

```
struct nsproxy *ns;
void *data;

ns = dentry_of(/mnt/B/tasks)->d_parent->d_fsdata;
data = ns->ctlr_data[some subsystem id which is bound in /mnt/B hierarchy]
```

we now scan tasklist and find a match if:

```
 tsk->nsproxy->ctlr_data[the above id] == data
```

(maybe we need to match on all data from all subsystems bound to B)

There is a similar bug in rcfs_rmdir also. We can't just use the nsproxy
pointed to by dentry to know whether the resource objects are free or
not. I am thinking (if at all resource control has to be provided on top
of nsproxy) that we should have a get_res_ns, similar to get_mnt_ns or
get_uts_ns, which will track number of nsproxies pointing to the same
resource object. If we do that, then rmdir() needs to go and check
those resource object's refcounts to see if a dir is in use or not.

--
Regards,

vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Srivatsa Vaddagiri on Wed, 07 Mar 2007 18:00:55 GMT

View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 11:43:46AM -0600, Serge E. Hallyn wrote:
> I still think the complaint was about terminology, not implementation.

I don't think that is what http://lkml.org/lkml/2007/2/12/426 conveyed!

> They just didn't want you calling them containers.

Yes that too.

> > Anyway, summarizing on "why nsproxy", the main point (I think) is about
> > using existing abstraction in the kernel.

s/abstraction/"implementation detail" then :)

> But nsproxy is not an abstraction, it's an implementation
> detail/optimization.

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by serue on Wed, 07 Mar 2007 20:58:46 GMT

View Forum Message <> Reply to Message

Quoting Srivatsa Vaddagiri (vatsa@in.ibm.com):
> On Wed, Mar 07, 2007 at 11:43:46AM -0600, Serge E. Hallyn wrote:
> > I still think the complaint was about terminology, not implementation.

>
> I don't think that is what http://lkml.org/lkml/2007/2/12/426 conveyed!

I don't have that in my inbox anymore so won't reply to it itself
unfortunately, but what it conveyed is also not that nsproxy should be
the 'resource control' object. If anything it seems to argue that all of
Paul's patchset should be done in userspace.

Sam writes

> That's a great idea for a set of
> tightly integrated userland utilities to simplify the presentation to
> the admin, but I don't see why you need to enshrine this in the kernel.
> Certainly not for any of the other patches in your set as far as I can
> see.

I disagree.

Sam, there are two very separate concepts here. Actually three.

What you had originally presented in a patchset was resource
virtualization: so when process A asks for some resource X, rather than
get resource_table[X] he gets resource_table[hash(x)]. The concensus
you mention at the start, and which you say Eric was arguing for, was to
not do such translation, but just get rid of the global
'resource_table'. By allowing processes to have and manipulate their
own private resource_table, you implement namespaces.

And as I've said, the nsproxy is just an implementation detail to keep
namespace pointers out of the task struct. Using nsproxy or not has
nothing to do with the namespace approach versus global resource tables
with id translation at all userspace->kernel boundaries.

So virtualization via explicit translation through global resource
tables is one concept, and private namespaces could be said to be a
second. The third is resource controls, which Paul's container patchset
implements. It has nothing to do with the previous two, and it is what
Paul's patches are addressing.

Sam asks:

> Ask yourself this - what do you need the container structure for so
> badly, that virtualising the individual resources does not provide for?

To keep track of a process' place in any of several hierarchies, where
each node in a tree inherit default values from the parent and can
customize in some way.

> You don't need it to copy the namespaces of another process ("enter")
> and you don't need it for checkpoint/migration.

Because these have nothing to do with resource controls.

> What does it mean to make a new container?

It means to create a new set of limits, probably inheriting defaults from a parent set of limits, customizable but probably within some limits imposed by the parent. For instance, if there is a cpuset container which limits its tasks to cpus 7 and 8, then when a new container is created as a child of that one, it can be further restricted, but can never have more than cpus 7 and 8.

> That's a great idea for a set of
> tightly integrated userland utilities to simplify the presentation to
> the admin, but I don't see why you need to enshrine this in the kernel.

If you want to argue that resource controls should be done in userspace i *suspect* you'll find that approach insufficient but am interested to see attempts to do so.

But just moving the container structure into nsproxy is just that - moving the container structure. Since Sam argues vehemently that there should be no such thing, I don't see how he can be seen as wanting to move it.

All that being said, if it were going to save space without overly complicating things I'm actually not opposed to using nsproxy, but it looks to me like it does complicate things. One reason for this is that through the nsproxy subsystem we are mixing pointers to data (the namespaces) and pointers to pointers to the same data (nsproxy subsystem containers pointing to nsproxies) in the same structure. Yuck.

-serge

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Paul Menage on Wed, 07 Mar 2007 21:20:18 GMT

View Forum Message <> Reply to Message

On 3/7/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
>

> All that being said, if it were going to save space without overly
> complicating things I'm actually not opposed to using nsproxy, but it

If space-saving is the main issue, then the latest version of my
containers patches uses just a single pointer in the task_struct, and
all tasks in the same set of containers (across all hierarchies) will
share a single container_group object, which holds the actual pointers
to container state.

Effectively, container_group is to container as nsproxy is to namespace.

Paul

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by serue on Wed, 07 Mar 2007 21:59:19 GMT
View Forum Message <> Reply to Message

Quoting Paul Menage (menage@google.com):
> On 3/7/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
> >
> >All that being said, if it were going to save space without overly
> >complicating things I'm actually not opposed to using nsproxy, but it
>
> If space-saving is the main issue, then the latest version of my

Space saving was the only reason for nsproxy to exist.

Now of course it also provides the teensiest reduction in # instructions
since every clone results in just one reference count inc for the
nsproxy rather than one for each namespace.

> containers patches uses just a single pointer in the task_struct, and
> all tasks in the same set of containers (across all hierarchies) will
> share a single container_group object, which holds the actual pointers
> to container state.

Yes, that's why this consolidation doesn't make sense to me.

Especially considering again that we will now have nsproxies pointing to
containers pointing to... nsproxies.

> Effectively, container_group is to container as nsproxy is to namespace.

>
> Paul

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Dave Hansen on Wed, 07 Mar 2007 22:13:46 GMT
View Forum Message <> Reply to Message

On Wed, 2007-03-07 at 15:59 -0600, Serge E. Hallyn wrote:
> Space saving was the only reason for nsproxy to exist.
>
> Now of course it also provides the teensiest reduction in # instructions
> since every clone results in just one reference count inc for the
> nsproxy rather than one for each namespace.

If we have 7 or 8 namespaces, then it can save us a significant number of atomic instructions on _each_ of the refcounts, plus touching all of the cachelines, etc...

-- Dave

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by ebiederm on Wed, 07 Mar 2007 22:32:48 GMT
View Forum Message <> Reply to Message

"Paul Menage" <menage@google.com> writes:

> On 3/7/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
>>
>> All that being said, if it were going to save space without overly
>> complicating things I'm actually not opposed to using nsproxy, but it
>
> If space-saving is the main issue, then the latest version of my
> containers patches uses just a single pointer in the task_struct, and
> all tasks in the same set of containers (across all hierarchies) will

> share a single container_group object, which holds the actual pointers
> to container state.

Yes.

However:
> Effectively, container_group is to container as nsproxy is to namespace.

The statement above nicely summarizes the confusion in terminology.

In the namespace world when we say container we mean roughly at the level
of nsproxy and container_group. Although it is expected to be a user space
concept like an application, not a concept implemented directly in the kernel.
i.e. User space is expected to combine separate resource controls and namespaces
and run processes inside that combination.

You are calling something that is on par with a namespace a container. Which
seriously muddies the waters. About as much as calling as referring to your
shoe as your whole outfit.

Without fixing the terminology it is going to be very hard to
successfully communicate.

Eric

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by ebiederm on Wed, 07 Mar 2007 23:13:15 GMT

View Forum Message <> Reply to Message

Dave Hansen <hansendc@us.ibm.com> writes:

> On Wed, 2007-03-07 at 15:59 -0600, Serge E. Hallyn wrote:
>> Space saving was the only reason for nsproxy to exist.
>>
>> Now of course it also provides the teensiest reduction in # instructions
>> since every clone results in just one reference count inc for the
>> nsproxy rather than one for each namespace.
>
> If we have 7 or 8 namespaces, then it can save us a significant number
> of atomic instructions on _each_ of the refcounts, plus touching all of
> the cachelines, etc...

Well we still have a global lock on the fork path so there is only so much we can do to improve things. The global process list, and there are some interesting posix signal handling rules that limit how much we can relax that restriction.

However with namespaces we have a natural limit on how many we will have. There aren't that many spaces for global names.

I don't know the situation well enough for resource controllers but I suspect we might not have any kind of natural limit (except what a single person can comprehend) to the kind of resource we will ultimately want to control which tends to imply we will have more of those.

Eric

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by ebiederm on Wed, 07 Mar 2007 23:16:00 GMT

View Forum Message <> Reply to Message

"Paul Menage" <menage@google.com> writes:

> No, Sam was saying that nsproxy should be the object that all resource
> controllers hook off.

I think implementation wise this tends to make sense.
However it should have nothing to do with semantics.

If we have a lot of independent resource controllers. Placing the pointer to their data structures directly in nsproxy instead of in task_struct sounds like a reasonable idea but it should not be user visible.

Eric

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of

nsproxy!
Posted by Paul Menage on Wed, 07 Mar 2007 23:18:25 GMT
View Forum Message <> Reply to Message

On 3/7/07, Eric W. Biederman <ebiederm@xmission.com> wrote:
> > Effectively, container_group is to container as nsproxy is to namespace.
>
> The statement above nicely summarizes the confusion in terminology.
>
> In the namespace world when we say container we mean roughly at the level
> of nsproxy and container_group.

So you're saying that a task can only be in a single system-wide container.

My patch provides multiple potentially-independent ways of dividing up
the tasks on the system - if the "container" is the set of all
divisions that the process is in, what's an appropriate term for the
sub-units?

>
> You are calling something that is on par with a namespace a container.

Yes.

> Which
> seriously muddies the waters. About as much as calling as referring to your
> shoe as your whole outfit.
> Without fixing the terminology it is going to be very hard to
> successfully communicate.

That assumes the viewpoint that your terminology is "correct" and
other people's needs "fixing". :-)

But as I've said I'm not particularly wedded to the term "container"
if that really turned out to be what's blocking acceptance from people
like Andrew or Linus. Do you have a suggestion for a better name? To
me, "process container" seems like the ideal name, since it's an
abstraction that "contains" processes and associates them with some
(subsystem-provided) state.

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of

nsproxy!
Posted by Sam Vilain on Thu, 08 Mar 2007 00:35:46 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
>> In the namespace world when we say container we mean roughly at the level
>> of nsproxy and container_group.
>>
> So you're saying that a task can only be in a single system-wide container.
>

Nope, we didn't make the mistake of nailing down what a "container" was
too far before it is implemented. We talked before about
containers-within-containers because, inevitably if you provide a
feature you'll end up having to deal with virtualising systems that in
turn use that feature.

> My patch provides multiple potentially-independent ways of dividing up
> the tasks on the system - if the "container" is the set of all
> divisions that the process is in, what's an appropriate term for the
> sub-units?
>

namespace, since 2.4.x

> That assumes the viewpoint that your terminology is "correct" and
> other people's needs "fixing". :-)
>

Absolutely. Please respect the semantics established so far; changing
them adds nothing at the cost of much confusion.

> But as I've said I'm not particularly wedded to the term "container"
> if that really turned out to be what's blocking acceptance from people
> like Andrew or Linus. Do you have a suggestion for a better name? To
> me, "process container" seems like the ideal name, since it's an
> abstraction that "contains" processes and associates them with some
> (subsystem-provided) state.
>

It's not even really the term, it's the semantics.

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Menage on Thu, 08 Mar 2007 00:42:05 GMT
View Forum Message <> Reply to Message

On 3/7/07, Sam Vilain <sam@vilain.net> wrote:
> Paul Menage wrote:
> >> In the namespace world when we say container we mean roughly at the level
> >> of nsproxy and container_group.
> >>
> > So you're saying that a task can only be in a single system-wide container.
> >
>
> Nope, we didn't make the mistake of nailing down what a "container" was
> too far before it is implemented. We talked before about
> containers-within-containers because, inevitably if you provide a
> feature you'll end up having to deal with virtualising systems that in
> turn use that feature.

Sure, my aproach allows containers hierarchically as children of other
containers too.

>
> > My patch provides multiple potentially-independent ways of dividing up
> > the tasks on the system - if the "container" is the set of all
> > divisions that the process is in, what's an appropriate term for the
> > sub-units?
> >
>
> namespace, since 2.4.x
>
> > That assumes the viewpoint that your terminology is "correct" and
> > other people's needs "fixing". :-)
> >
>
> Absolutely. Please respect the semantics established so far; changing
> them adds nothing at the cost of much confusion.

But "namespace" has well-established historical semantics too - a way
of changing the mappings of local names to global objects. This
doesn't describe things liek resource controllers, cpusets, resource
monitoring, etc.

Trying to extend the well-known term namespace to refer to things that
aren't namespaces isn't a useful approach, IMO.

Paul

____
Containers mailing list

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Sam Vilain on Thu, 08 Mar 2007 00:50:01 GMT
View Forum Message <> Reply to Message

Srivatsa Vaddagiri wrote:

> container structure in your patches provides for these things:
>
> a. A way to group tasks
> b. A way to maintain several hierarchies of such groups
>
> If you consider just a. then I agree that container abstraction is
> redundant, esp for vserver resource control (nsproxy can already be used
> to group tasks).
>
> What nsproxy doesn't provide is b - a way to represent hierarchies of
> groups.
>

Well, that's like saying you can't put hierarchical data in a relational database.

The hierarchy question is an interesting one, though. However I believe it first needs to be broken down into subsystems and considered on a subsystem-by-subsystem basis again, and if general patterns are observed, then a common solution should stand out.

Let's go back to the namespaces we know about and discuss how hierarchies apply to them. Please those able to brainstorm, do so - I call green hat time.

1. UTS namespaces

Can a UTS namespace set any value it likes?

Can you inspect or set the UTS namespace values of a subservient UTS namespace?

2. IPC namespaces

Can a process in an IPC namespace send a signal to those in a subservient namespace?

3. PID namespaces

Can a process in a PID namespace see the processes in a subservient namespace?

Do the processes in a subservient namespace appear in a higher level namespace mapped to a different set of PIDs?

4. Filesystem namespaces

Can we see all of the mounts in a subservient namespace?

Does our namespace receive updates when their namespace mounts change? (perhaps under a sub-directory)

5. L2 network namespaces

Can we see or alter the subservient network namespace's interfaces/iptables/routing?

Are any of the subservient network namespace's interfaces visible in our namespace, and by which mapping?

6. L3 network namespaces

Can we bind to a subservient network namespace's addresses?

Can we give or remove addresses to and from the subservient network namespace's namespace?

Can we allow the namespace access to modify particular IP tables?

7. resource namespaces

Is the subservient namespace's resource usage counting against ours too?

Can we dynamically alter the subservient namespace's resource allocations?

8. anyone else?

So, we can see some general trends here - but it's never quite the same question, and I think the best answers will come from a tailored approach for each subsystem.

Each one *does* have some common questions - for instance, "is the namespace allowed to create more namespaces of this type". That's probably a capability bit for each, though.

So let's bring this back to your patches. If they are providing

visibility of ns_proxy, then it should be called namesfs or some such.
It doesn't really matter if processes disappear from namespace
aggregates, because that's what's really happening anyway. The only
problem is that if you try to freeze a namespace that has visibility of
things at this level, you might not be able to reconstruct the
filesystem in the same way. This may or may not be considered a problem,
but open filehandles and directory handles etc surviving a freeze/thaw
is part of what we're trying to achieve. Then again, perhaps some
visibility is better than none for the time being.

If they are restricted entirely to resource control, then don't use the
nsproxy directly - use the structure or structures which hang off the
nsproxy (or even task_struct) related to resource control.

Sam.

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Sam Vilain on Thu, 08 Mar 2007 00:53:05 GMT

View Forum Message <> Reply to Message

Paul Menage wrote:
> But "namespace" has well-established historical semantics too - a way
> of changing the mappings of local names to global objects. This
> doesn't describe things liek resource controllers, cpusets, resource
> monitoring, etc.
>
> Trying to extend the well-known term namespace to refer to things that
> aren't namespaces isn't a useful approach, IMO.
>
> Paul
>

But "namespace" has well-established historical semantics too - a way
of changing the mappings of local * to global objects. This
accurately describes things liek resource controllers, cpusets, resource
monitoring, etc.

Trying to extend the well-known term namespace to refer to things that
are semantically equivalent namespaces is a useful approach, IMHO.

Sam.

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by ebiederm on Thu, 08 Mar 2007 03:12:00 GMT

View Forum Message <> Reply to Message

Srivatsa Vaddagiri <vatsa@in.ibm.com> writes:

> 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).
>
> 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)

After looking at the discussion that happened immediately after this was
posted this feels like the right general direction to get the different
parties talking to each other. I'm not convinced about the whole idea
yet but this looks like a step in a useful direction.

I have a big request.

Please next time this kind of patch is posted add a description of
what is happening and why. I have yet to see people explain why
this is a good idea. Why the current semantics were chosen.

The review is still largely happening at the why level but no
one is addressing that yet. So please can we have a why.

I have a question? What does rcfs look like if we start with
the code that is in the kernel? That is start with namespaces
and nsproxy and just build a filesystem to display/manipulate them?
With the code built so it will support adding resource controllers
when they are ready?

> 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/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 {
```

We probably want to rename this struct task_proxy....
And then we can rename most of the users things like:
dup_task_proxy, clone_task_proxy, get_task_proxy, free_task_proxy,
put_task_proxy, exit_task_proxy, init_task_proxy....

```
>   struct ipc_namespace *ipc_ns;
>   struct mnt_namespace *mnt_ns;
>   struct pid_namespace *pid_ns;
> +#ifdef CONFIG_RCFS
> + struct list_head list;
```

This extra list of nsproxy's is unneeded and a performance problem the way it is used. In general we want to talk about the individual resource controllers not the nsproxy.

```
> + void *ctlr_data[CONFIG_MAX_RC_SUBSYS];
```

I still don't understand why these pointers are so abstract,
and why we need an array lookup into them?

```
> +#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.
```

I'm still inclined to think this should be part of /proc, instead of a purely
separate fs. But I might be missing something.

Eric

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Paul Menage on Thu, 08 Mar 2007 09:10:24 GMT

View Forum Message <> Reply to Message

On 3/7/07, Eric W. Biederman <ebiederm@xmission.com> wrote:
>
> Please next time this kind of patch is posted add a description of
> what is happening and why. I have yet to see people explain why
> this is a good idea. Why the current semantics were chosen.

OK. I thought that the descriptions in my last patch 0/7 and
Documentation/containers.txt gave a reasonable amount of "why", but I
can look at adding more details.

>

> I have a question? What does rcfs look like if we start with
> the code that is in the kernel? That is start with namespaces
> and nsproxy and just build a filesystem to display/manipulate them?
> With the code built so it will support adding resource controllers
> when they are ready?

There's at least one resource controller that's already in the kernel - cpusets.

> We probably want to rename this struct task_proxy....
> And then we can rename most of the users things like:
> dup_task_proxy, clone_task_proxy, get_task_proxy, free_task_proxy,
> put_task_proxy, exit_task_proxy, init_task_proxy....

That could be a good start.

>
> This extra list of nsproxy's is unneeded and a performance problem the
> way it is used. In general we want to talk about the individual resource
> controllers not the nsproxy.

There's one important reason why it's needed, and highlights one of
the ways that "resource controllers" are different from the way that
"namespaces" have currently been used.

Currently with a namespace, you can only unshare, either by
sys_unshare() or clone() - you can't "reshare" a namespace with some
other task. But resource controllers tend to have the concept a lot
more of being able to move between resource classes. If you're going
to have an ns_proxy/container_group object that gathers together a
group of pointers to namespaces/subsystem-states, then either:

1) you only allow a task to reshare *all* namespaces/subsystems with
another task, i.e. you can update current->task_proxy to point to
other->task_proxy. But that restricts flexibility of movement. It
would be impossible to have a process that could enter, say, an
existing process' network namespace without also entering its
pid/ipc/uts namespaces and all of its resource limits.

2) you allow a task to selectively reshare namespaces/subsystems with
another task, i.e. you can update current->task_proxy to point to a
proxy that matches your existing task_proxy in some ways and the
task_proxy of your destination in others. In that case a trivial
implementation would be to allocate a new task_proxy and copy some
pointers from the old task_proxy and some from the new. But then
whenever a task moves between different groupings it acquires a new
unique task_proxy. So moving a bunch of tasks between two groupings,
they'd all end up with unique task_proxy objects with identical
contents.

So it would be much more space efficient to be able to locate an existing task_proxy with an identical set of namespace/subsystem pointers in that event. The linked list approach that I put in my last containers patch was a simple way to do that, and Vatsa's reused it for his patches. My intention is to replace it with a more efficient lookup (maybe using a hash of the desired pointers?) in a future patch.

>
> > +   void *ctlr_data[CONFIG_MAX_RC_SUBSYS];
>
> I still don't understand why these pointers are so abstract,
> and why we need an array lookup into them?
>

For the same reason that we have:

- generic notifier chains rather than having a big pile of #ifdef'd calls to the various notification sites

- linker sections to define initcalls and per-cpu variables, rather than hard-coding all init calls into init/main.c and having a big per-cpu structure (both of which would again be full of #ifdefs)

It makes the code much more readable, and makes patches much simpler and less likely to stomp on one another.

OK, so my current approaches have involved an approach like notifier chains, i.e. have a generic list/array, and do something to all the objects on that array.

How about a radically different approach based around the initcall/percpu way (linker sections)? Something like:

- each namespace or subsystem defines itself in its own code, via a macro such as:

```
struct task_subsys {
  const char *name;
  ...
};

#define DECLARE_TASKGROUP_SUBSYSTEM(ss) \
    __attribute__((__section__(".data.tasksubsys"))) struct
task_subsys *ss##_ptr = &ss
```

It would be used like:

```
struct taskgroup_subsys uts_ns = {
  .name = "uts",
  .unshare = uts_unshare,
};

DECLARE_TASKGROUP_SUBSYSTEM(uts_ns);

...

struct taskgroup_subsys cpuset_ss {
  .name = "cpuset",
  .create = cpuset_create,
  .attach = cpuset_attach,
};

DECLARE_TASKGROUP_SUBSYSTEM(cpuset_ss);
```

At boot time, the task_proxy init code would figure out from the size of the task_subsys section how many pointers had to be in the task_proxy object (maybe add a few spares for dynamically-loaded modules?). The offset of the subsystem pointer within the task_subsys data section would also be the offset of that subsystem's per-task-group state within the task_proxy object, which should allow accesses to be pretty efficient (with macros providing user-friendly access to the appropriate locations in the task_proxy)

The loops in container.c in my patch that iterate over the subsys array to perform callbacks, and the code in nsproxy.c that performs the same action for each namespace type, would be replaced with iterations over the task_subsys data section; possibly some pre-processing of the various linked-in subsystems could be done to remove unnecessary iterations. The generic code would handle things like reference counting.

The existing unshare()/clone() interface would be a way to create a child "container" (for want of a better term) that shared some subsystem pointers with its parent and had cloned versions of others (perhaps only for the namespace-like subsystems?); the filesystem interface would allow you to create new "containers" that weren't explicitly associated with processes, and to move processes between "containers". Also, the filesystem interface would allow you to bind multiple subsystems together to allow easier manipulation from userspace, in a similar way to my current containers patch.

So in summary, it takes the concepts that resource controllers and namespaces share (that of grouping tasks) and unifies them, while not

forcing them to behave exactly the same way. I can envisage some other
per-task pointers that are generally inherited by children being
possibly moved into this in the same way, e.g. task->user and
task->mempolicy, if we could come up with a solution that handles
groupings with sufficiently different lifetimes.

Thoughts?

Paul

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Srivatsa Vaddagiri on Thu, 08 Mar 2007 10:13:47 GMT

View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 08:12:00PM -0700, Eric W. Biederman wrote:
> The review is still largely happening at the why level but no
> one is addressing that yet. So please can we have a why.

Here's a brief summary of what's happening and why. If its not clear, pls get back to us with specific questions.

There have been various projects attempting to provide resource management support in Linux, including CKRM/Resource Groups and UBC. Each had its own task-grouping mechanism.

Paul Menage observed [1] that cpusets in the kernel already has a grouping mechanism which was working well for cpusets. He went ahead and generalized the grouping code in cpusets so that it could be used for overall resource management purpose. With his patches, it is possible to even create multiple hierarchies of groups (see [2] on why multiple hierarchies) as follows:

```
mount -t container -o cpuset none /dev/cpuset <- cpuset hierarchy
mount -t container -o mem,cpu none /dev/mem <- memory/cpu hierarchy
mount -t container -o disk none /dev/disk <- disk hierarchy
```

In each hierarchy, you can create task groups and manipulate the resource parameters of each group. You can also move tasks between groups at run-time (see [3] on why this is required). Each hierarchy is also manipulated independent of the other.

Paul's patches also introduced a 'struct container' in the kernel, which serves these key purposes:

- Task-grouping
 'struct container' represents a task-group created in each hierarchy.
 So every directory created under /dev/cpuset or /dev/mem above will
 have a corresponding 'struct container' inside the kernel.
 All tasks pointing to the same 'struct container' are considered
 to be part of a group

 The 'struct container' in turn has pointers to resource objects
 which store actual resource parameters for that group. In above
 example, 'struct container' created under /dev/cpuset will have a
 pointer to 'struct cpuset' while 'struct container' created
 under /dev/disk will have pointer to 'struct disk_quota_or_whatever'.

- Maintain hierarchical information
 The 'struct container' also keeps track of hierarchical relationship
 between groups.

The filesystem interface in the patches essentially serves these
purposes:

 - Provide an interface to manipulate task-groups. This includes
   creating/deleting groups, listing tasks present in a group and
   moving tasks across groups

 - Provdes an interface to manipulate the resource objects
   (limits etc) pointed to by 'struct container'.

As you know, the introduction of 'struct container' was objected to and
was felt redundant as a means to group tasks. Thats where I took a shot
at converting over Paul Menage's patch to avoid 'struct container'
abstraction and insead work with 'struct nsproxy'. In the rcfs patch,
each directory (in /dev/cpuset or /dev/disk) is associated with a
'struct nsproxy' instead. The most important need of the filesystem
interface is not to manipulate the nsproxy objects directly, but to manipulate
the resource objects (nsproxy->ctlr_data[] in the patches) which store
information like limit etc.

> I have a question? What does rcfs look like if we start with
> the code that is in the kernel? That is start with namespaces
> and nsproxy and just build a filesystem to display/manipulate them?
> With the code built so it will support adding resource controllers
> when they are ready?

If I am not mistaken, Serge did attempt something in that direction,
only that it was based on Paul's container patches. rcfs can no doubt
support the same feature.

```
> >   struct ipc_namespace *ipc_ns;
> >   struct mnt_namespace *mnt_ns;
> >   struct pid_namespace *pid_ns;
> > +#ifdef CONFIG_RCFS
> > + struct list_head list;
>
```

> This extra list of nsproxy's is unneeded and a performance problem the
> way it is used. In general we want to talk about the individual resource
> controllers not the nsproxy.

I think if you consider the multiple hierarchy picture, the need becomes obvious.

Lets say that you had these hierarchies : /dev/cpuset, /dev/mem, /dev/disk and the various resource classes (task-groups) under them as below:

 /dev/cpuset/C1, /dev/cpuset/C1/C11, /dev/cpuset/C2
 /dev/mem/M1, /dev/mem/M2, /dev/mem/M3
 /dev/disk/D1, /dev/disk/D2, /dev/disk/D3

The nsproxy structure basically has pointers to a resource objects in each of these hierarchies.

 nsproxy { ..., C1, M1, D1} could be one nsproxy
 nsproxy { ..., C1, M2, D3} could be another nsproxy and so on

So you see, because of multi-hierachies, we can have different combinations of resource classes.

When we support task movement across resource classes, we need to find a nsproxy which has the right combination of resource classes that the task's nsproxy can be hooked to.

That's where we need the nsproxy list. Hope this makes it clear.

```
> > + void *ctlr_data[CONFIG_MAX_RC_SUBSYS];
>
```

> I still don't understand why these pointers are so abstract,
> and why we need an array lookup into them?

we can avoid these abstract pointers and instead have a set of pointers like this:

```
 struct nsproxy {
  ...
  struct cpu_limit *cpu; /* cpu control namespace */
  struct rss_limit *rss; /* rss control namespace */
  struct cpuset *cs; /* cpuset namespace */
```

}

But that will make some code (like searching for a right nsproxy when a task moves across classes/groups) very awkward.

> I'm still inclined to think this should be part of /proc, instead of a purely
> separate fs. But I might be missing something.

A separate filesystem would give us more flexibility like the implementing multi-hierarchy support described above.

--
Regards,
vatsa

References:

1. http://lkml.org/lkml/2006/09/20/200
2. http://lkml.org/lkml/2006/11/6/95
3. http://lkml.org/lkml/2006/09/5/178

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Thu, 08 Mar 2007 11:30:54 GMT
View Forum Message <> Reply to Message

On Thu, Mar 08, 2007 at 01:50:01PM +1300, Sam Vilain wrote:
> 7. resource namespaces

It should be. Imagine giving 20% bandwidth to a user X. X wants to divide this bandwidth further between multi-media (10%), kernel compilation (5%) and rest (5%). So,

> Is the subservient namespace's resource usage counting against ours too?

Yes, the resource usage of children should be accounted when capping parent resource usage.

> Can we dynamically alter the subservient namespace's resource allocations?

Should be possible yes. That lets user X completely manage his
allocation among whatever sub-groups he creates.

> So let's bring this back to your patches. If they are providing
> visibility of ns_proxy, then it should be called namesfs or some such.

The patches should give visibility to both nsproxy objects (by showing
what tasks share the same nsproxy objects and letting tasks move across
nsproxy objects if allowed) and the resource control objects pointed to
by nsproxy (struct cpuset, struct cpu_limit, struct rss_limit etc).

> It doesn't really matter if processes disappear from namespace
> aggregates, because that's what's really happening anyway. The only
> problem is that if you try to freeze a namespace that has visibility of
> things at this level, you might not be able to reconstruct the
> filesystem in the same way. This may or may not be considered a problem,
> but open filehandles and directory handles etc surviving a freeze/thaw
> is part of what we're trying to achieve. Then again, perhaps some
> visibility is better than none for the time being.
>
> If they are restricted entirely to resource control, then don't use the
> nsproxy directly - use the structure or structures which hang off the
> nsproxy (or even task_struct) related to resource control.

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Thu, 08 Mar 2007 11:39:41 GMT

View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 04:16:00PM -0700, Eric W. Biederman wrote:
> I think implementation wise this tends to make sense.
> However it should have nothing to do with semantics.
>
> If we have a lot of independent resource controllers. Placing the
> pointer to their data structures directly in nsproxy instead of in
> task_struct sounds like a reasonable idea

Thats what the rcfs patches do.

> but it should not be user visible.

What do you mean by this? We do want the user to be able to manipulate
the resource parameters (which are normally present in the data
structures/resource objects pointed to by nsproxy -
nsproxy->ctlr_data[])

--
Regards,
vatsa

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

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Fri, 09 Mar 2007 00:38:19 GMT
View Forum Message <> Reply to Message

On Thu, Mar 08, 2007 at 01:10:24AM -0800, Paul Menage wrote:
> On 3/7/07, Eric W. Biederman <ebiederm@xmission.com> wrote:
> >
> > Please next time this kind of patch is posted add a description of
> > what is happening and why. I have yet to see people explain why
> > this is a good idea. Why the current semantics were chosen.
>
> OK. I thought that the descriptions in my last patch 0/7 and
> Documentation/containers.txt gave a reasonable amount of "why", but I
> can look at adding more details.
>
> >
> > I have a question? What does rcfs look like if we start with
> > the code that is in the kernel? That is start with namespaces
> > and nsproxy and just build a filesystem to display/manipulate them?
> > With the code built so it will support adding resource controllers
> > when they are ready?
>
> There's at least one resource controller that's already in the kernel - cpusets.
>
> > We probably want to rename this struct task_proxy....
> > And then we can rename most of the users things like:
> > dup_task_proxy, clone_task_proxy, get_task_proxy, free_task_proxy,

> > put_task_proxy, exit_task_proxy, init_task_proxy....
>
> That could be a good start.
>
> >
> > This extra list of nsproxy's is unneeded and a performance problem the
> > way it is used. In general we want to talk about the individual resource
> > controllers not the nsproxy.
>
> There's one important reason why it's needed, and highlights one of
> the ways that "resource controllers" are different from the way that
> "namespaces" have currently been used.
>
> Currently with a namespace, you can only unshare, either by
> sys_unshare() or clone() - you can't "reshare" a namespace with some
> other task. But resource controllers tend to have the concept a lot
> more of being able to move between resource classes. If you're going
> to have an ns_proxy/container_group object that gathers together a
> group of pointers to namespaces/subsystem-states, then either:
>
> 1) you only allow a task to reshare *all* namespaces/subsystems with
>   another task, i.e. you can update current->task_proxy to point to
>   other->task_proxy. But that restricts flexibility of movement.
>   It would be impossible to have a process that could enter, say,
>   an existing process' network namespace without also entering its
>   pid/ipc/uts namespaces and all of its resource limits.
>
> 2) you allow a task to selectively reshare namespaces/subsystems with
>   another task, i.e. you can update current->task_proxy to point to
>   a proxy that matches your existing task_proxy in some ways and the
>   task_proxy of your destination in others. In that case a trivial
>   implementation would be to allocate a new task_proxy and copy some
>   pointers from the old task_proxy and some from the new. But then
>   whenever a task moves between different groupings it acquires a
>   new unique task_proxy. So moving a bunch of tasks between two
>   groupings, they'd all end up with unique task_proxy objects with
>   identical contents.

this is exactly what Linux-VServer does right now, and I'm
still not convinced that the nsproxy really buys us anything
compared to a number of different pointers to various spaces
(located in the task struct)

> So it would be much more space efficient to be able to locate an
> existing task_proxy with an identical set of namespace/subsystem
> pointers in that event. The linked list approach that I put in my last
> containers patch was a simple way to do that, and Vatsa's reused it
> for his patches. My intention is to replace it with a more efficient

> lookup (maybe using a hash of the desired pointers?) in a future
> patch.

IMHO that is getting quite complicated and probably very
inefficient, especially if you think hundreds of guests
with a dozent spaces each ... and still we do not know if
the nsproxy is a real benefit either memory or performance
wise ...

> > > +   void *ctlr_data[CONFIG_MAX_RC_SUBSYS];
> >
> > I still don't understand why these pointers are so abstract,
> > and why we need an array lookup into them?
> >
>
> For the same reason that we have:
>
> - generic notifier chains rather than having a big pile of #ifdef'd
>   calls to the various notification sites
>
> - linker sections to define initcalls and per-cpu variables, rather
>   than hard-coding all init calls into init/main.c and having a big
>   per-cpu structure (both of which would again be full of #ifdefs)
>
> It makes the code much more readable, and makes patches much simpler
> and less likely to stomp on one another.
>
> OK, so my current approaches have involved an approach like notifier
> chains, i.e. have a generic list/array, and do something to all the
> objects on that array.

I'd prefer to do accounting (and limits) in a very simple
and especially performant way, and the reason for doing
so is quite simple:

 nobody actually cares about a precise accounting and
 calculating shares or partitions of whatever resource,
 all that matters is that you have a way to prevent a
 potential hostile environment from sucking up all your
 resources (or even a single one) resulting in a DoS

so the main purpose of a resource limit (or accounting)
is to get an idea how much a certain guest uses up, not
more and not less ...

> How about a radically different approach based around the
> initcall/percpu way (linker sections)? Something like:
>

> - each namespace or subsystem defines itself in its own code, via a
> macro such as:
>
> struct task_subsys {
>   const char *name;
>   ...
> };
>
> #define DECLARE_TASKGROUP_SUBSYSTEM(ss) \
>   __attribute__((__section__(".data.tasksubsys"))) struct
> task_subsys *ss##_ptr = &ss
>
>
> It would be used like:
>
> struct taskgroup_subsys uts_ns = {
>   .name = "uts",
>   .unshare = uts_unshare,
> };
>
> DECLARE_TASKGROUP_SUBSYSTEM(uts_ns);
>
> ...
>
> struct taskgroup_subsys cpuset_ss {
>   .name = "cpuset",
>   .create = cpuset_create,
>   .attach = cpuset_attach,
> };
>
> DECLARE_TASKGROUP_SUBSYSTEM(cpuset_ss);
>
> At boot time, the task_proxy init code would figure out from the size
> of the task_subsys section how many pointers had to be in the
> task_proxy object (maybe add a few spares for dynamically-loaded
> modules?). The offset of the subsystem pointer within the task_subsys
> data section would also be the offset of that subsystem's
> per-task-group state within the task_proxy object, which should allow
> accesses to be pretty efficient (with macros providing user-friendly
> access to the appropriate locations in the task_proxy)
>
> The loops in container.c in my patch that iterate over the subsys
> array to perform callbacks, and the code in nsproxy.c that performs
> the same action for each namespace type, would be replaced with
> iterations over the task_subsys data section; possibly some
> pre-processing of the various linked-in subsystems could be done to
> remove unnecessary iterations. The generic code would handle things
> like reference counting.

>
> The existing unshare()/clone() interface would be a way to create a
> child "container" (for want of a better term) that shared some
> subsystem pointers with its parent and had cloned versions of others
> (perhaps only for the namespace-like subsystems?); the filesystem
> interface would allow you to create new "containers" that weren't
> explicitly associated with processes, and to move processes between
> "containers". Also, the filesystem interface would allow you to bind
> multiple subsystems together to allow easier manipulation from
> userspace, in a similar way to my current containers patch.
>
> So in summary, it takes the concepts that resource controllers and
> namespaces share (that of grouping tasks) and unifies them, while
> not forcing them to behave exactly the same way. I can envisage some
> other per-task pointers that are generally inherited by children
> being possibly moved into this in the same way, e.g. task->user and
> task->mempolicy, if we could come up with a solution that handles
> groupings with sufficiently different lifetimes.
>
> Thoughts?

sounds quite complicated and fragile to me ...

but I guess I have to go through that one again
before I can give a final statement ...

best,
Herbert

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

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Fri, 09 Mar 2007 00:48:16 GMT

View Forum Message <> Reply to Message

On Thu, Mar 08, 2007 at 03:43:47PM +0530, Srivatsa Vaddagiri wrote:
> On Wed, Mar 07, 2007 at 08:12:00PM -0700, Eric W. Biederman wrote:
> > The review is still largely happening at the why level but no

> > one is addressing that yet. So please can we have a why.
>
> Here's a brief summary of what's happening and why. If its not clear,
> pls get back to us with specific questions.
>
> There have been various projects attempting to provide resource
> management support in Linux, including CKRM/Resource Groups and UBC.

let me note here, once again, that you forgot Linux-VServer
which does quite non-intrusive resource management ...

> Each had its own task-grouping mechanism.

the basic 'context' (pid space) is the grouping mechanism
we use for resource management too

> Paul Menage observed [1] that cpusets in the kernel already has a
> grouping mechanism which was working well for cpusets. He went ahead
> and generalized the grouping code in cpusets so that it could be used
> for overall resource management purpose.

> With his patches, it is possible to even create multiple hierarchies
> of groups (see [2] on why multiple hierarchies) as follows:

do we need or even want that? IMHO the hierarchical
concept CKRM was designed with, was also the reason
for it being slow, unuseable and complicated

> mount -t container -o cpuset none /dev/cpuset <- cpuset hierarchy
> mount -t container -o mem,cpu none /dev/mem <- memory/cpu hierarchy
> mount -t container -o disk none /dev/disk <- disk hierarchy
>
> In each hierarchy, you can create task groups and manipulate the
> resource parameters of each group. You can also move tasks between
> groups at run-time (see [3] on why this is required).

> Each hierarchy is also manipulated independent of the other.

> Paul's patches also introduced a 'struct container' in the kernel,
> which serves these key purposes:
>
> - Task-grouping
>   'struct container' represents a task-group created in each hierarchy.
>   So every directory created under /dev/cpuset or /dev/mem above will
>   have a corresponding 'struct container' inside the kernel. All tasks
>   pointing to the same 'struct container' are considered to be part of
>   a group
>

>   The 'struct container' in turn has pointers to resource objects which
>   store actual resource parameters for that group. In above example,
>   'struct container' created under /dev/cpuset will have a pointer to
>   'struct cpuset' while 'struct container' created under /dev/disk will
>   have pointer to 'struct disk_quota_or_whatever'.
>
> - Maintain hierarchical information
>   The 'struct container' also keeps track of hierarchical relationship
>   between groups.
>
> The filesystem interface in the patches essentially serves these
> purposes:
>
> - Provide an interface to manipulate task-groups. This includes
>   creating/deleting groups, listing tasks present in a group and
>   moving tasks across groups
>
> - Provdes an interface to manipulate the resource objects
>   (limits etc) pointed to by 'struct container'.
>
> As you know, the introduction of 'struct container' was objected
> to and was felt redundant as a means to group tasks. Thats where I
> took a shot at converting over Paul Menage's patch to avoid 'struct
> container' abstraction and insead work with 'struct nsproxy'.

which IMHO isn't a step in the right direction, as
you will need to handle different nsproxies within
the same 'resource container' (see previous email)

> In the rcfs patch, each directory (in /dev/cpuset or /dev/disk) is
> associated with a 'struct nsproxy' instead. The most important need
> of the filesystem interface is not to manipulate the nsproxy objects
> directly, but to manipulate the resource objects (nsproxy->ctlr_data[]
> in the patches) which store information like limit etc.
>
> > I have a question? What does rcfs look like if we start with
> > the code that is in the kernel? That is start with namespaces
> > and nsproxy and just build a filesystem to display/manipulate them?
> > With the code built so it will support adding resource controllers
> > when they are ready?
>
> If I am not mistaken, Serge did attempt something in that direction,
> only that it was based on Paul's container patches. rcfs can no doubt
> support the same feature.
>
> > >   struct ipc_namespace *ipc_ns;
> > >   struct mnt_namespace *mnt_ns;
> > >   struct pid_namespace *pid_ns;

> > > +#ifdef CONFIG_RCFS
> > > + struct list_head list;
> >
> > This extra list of nsproxy's is unneeded and a performance problem the
> > way it is used. In general we want to talk about the individual resource
> > controllers not the nsproxy.
>
> I think if you consider the multiple hierarchy picture, the need
> becomes obvious.
>
> Lets say that you had these hierarchies : /dev/cpuset, /dev/mem, /dev/disk
> and the various resource classes (task-groups) under them as below:
>
> /dev/cpuset/C1, /dev/cpuset/C1/C11, /dev/cpuset/C2
> /dev/mem/M1, /dev/mem/M2, /dev/mem/M3
> /dev/disk/D1, /dev/disk/D2, /dev/disk/D3
>
> The nsproxy structure basically has pointers to a resource objects in
> each of these hierarchies.
>
> nsproxy { ..., C1, M1, D1} could be one nsproxy
> nsproxy { ..., C1, M2, D3} could be another nsproxy and so on
>
> So you see, because of multi-hierachies, we can have different
> combinations of resource classes.
>
> When we support task movement across resource classes, we need to find a
> nsproxy which has the right combination of resource classes that the
> task's nsproxy can be hooked to.

no, not necessarily, we can simply create a new one
and give it the proper resource or whatever-spaces

> That's where we need the nsproxy list. Hope this makes it clear.
>
> > > + void *ctlr_data[CONFIG_MAX_RC_SUBSYS];
> >
> > I still don't understand why these pointers are so abstract,
> > and why we need an array lookup into them?
>
> we can avoid these abstract pointers and instead have a set of pointers
> like this:
>
> struct nsproxy {
>   ...
>   struct cpu_limit *cpu; /* cpu control namespace */
>   struct rss_limit *rss; /* rss control namespace */
>   struct cpuset *cs; /* cpuset namespace */

>
> }
>
> But that will make some code (like searching for a right nsproxy when a
> task moves across classes/groups) very awkward.
>
> > I'm still inclined to think this should be part of /proc, instead of a purely
> > separate fs. But I might be missing something.
>
> A separate filesystem would give us more flexibility like the
> implementing multi-hierarchy support described above.

why is the filesystem approach so favored for this
kind of manipulations?

IMHO it is one of the worst interfaces I can imagine
(to move tasks between spaces and/or assign resources)
but yes, I'm aware that filesystems are 'in' nowadays

best,
Herbert

> --
> Regards,
> vatsa
>
>
> References:
>
> 1. http://lkml.org/lkml/2006/09/20/200
> 2. http://lkml.org/lkml/2006/11/6/95
> 3. http://lkml.org/lkml/2006/09/5/178
>
> ____
> Containers mailing list
> Containers@lists.osdl.org
> https://lists.osdl.org/mailman/listinfo/containers

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Herbert Poetzl on Fri, 09 Mar 2007 01:16:08 GMT
View Forum Message <> Reply to Message

On Thu, Mar 08, 2007 at 05:00:54PM +0530, Srivatsa Vaddagiri wrote:
> On Thu, Mar 08, 2007 at 01:50:01PM +1300, Sam Vilain wrote:
> > 7. resource namespaces
>
> It should be. Imagine giving 20% bandwidth to a user X. X wants to
> divide this bandwidth further between multi-media (10%), kernel
> compilation (5%) and rest (5%). So,

sounds quite nice, but ...

> > Is the subservient namespace's resource usage counting against ours too?
>
> Yes, the resource usage of children should be accounted when capping
> parent resource usage.

it will require to do accounting many times
(and limit checks of course), which in itself
might be a way to DoS the kernel by creating
more and more resource groups
>
> > Can we dynamically alter the subservient namespace's resource
> > allocations?
>
> Should be possible yes. That lets user X completely manage his
> allocation among whatever sub-groups he creates.

what happens if the parent changes, how is
the resource change (if it was a reduction)
propagated to the children?

e.g. your guest has 1024 file handles, now
you reduce it to 512, but the guest had two
children, both with 256 file handles each ...

> > So let's bring this back to your patches. If they are providing
> > visibility of ns_proxy, then it should be called namesfs or some
> > such.
>
> The patches should give visibility to both nsproxy objects (by showing
> what tasks share the same nsproxy objects and letting tasks move across
> nsproxy objects if allowed) and the resource control objects pointed to
> by nsproxy (struct cpuset, struct cpu_limit, struct rss_limit etc).

the nsproxy is not really relevant, as it
is some kind of strange indirection, which
does not necessarily depict the real relations,
regardless wether you do the re-sharing of
those nsproies or not .. let me know if you

need examples to verify that ...

best,
Herbert

> > It doesn't really matter if processes disappear from namespace
> > aggregates, because that's what's really happening anyway. The only
> > problem is that if you try to freeze a namespace that has visibility
> > of things at this level, you might not be able to reconstruct the
> > filesystem in the same way. This may or may not be considered a
> > problem, but open filehandles and directory handles etc surviving
> > a freeze/thaw is part of what we're trying to achieve. Then again,
> > perhaps some visibility is better than none for the time being.
> >
> > If they are restricted entirely to resource control, then don't use
> > the nsproxy directly - use the structure or structures which hang
> > off the nsproxy (or even task_struct) related to resource control.
>
> --
> Regards,
> vatsa
> ____
> Containers mailing list
> Containers@lists.osdl.org
> https://lists.osdl.org/mailman/listinfo/containers

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Paul Jackson on Fri, 09 Mar 2007 02:35:07 GMT
View Forum Message <> Reply to Message

Herbert wrote:
> why is the filesystem approach so favored for this
> kind of manipulations?

I don't have any clear sense of whether the additional uses of file
systems being considered here are a good idea or not, but the use of a
file system for cpusets has turned out quite well, in my (vain and
biased ;) view.

Cpusets are subsets of the CPUs and memory nodes on a system.

These subsets naturally form a partial ordering, where one cpuset is
below another if its CPUs and nodes are a subset of the other ones.

This forms a natural hierarchical space. It is quite convenient to be able to add names and file system like attributes, so that one can do things like -name- the set of CPUs to which you are attaching a job, as in "this job is to run on the CPUs in cpuset /foo/bar", and to further have file system like permissions on these subsets, to control who can access or modify them.

For such hierarchical data structures, especially ones where names and permissions are useful, file systems are a more natural interface than traditional system call usage patterns.

The key, in my view, is the 'shape' of the data. If the data schema is basically a single table, with uniform rows having a few fields each, where each field is a simple integer or string (not a fancy formatted string encoding some more elaborate shape) then classic system call patterns work well. If the schema is tree shaped, and especially if the usual file system attributes such as a hierarchical name space and permissions are useful, then a file system based API is probably best.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

____

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Jackson on Fri, 09 Mar 2007 04:27:13 GMT
View Forum Message <> Reply to Message

> But "namespace" has well-established historical semantics too - a way
> of changing the mappings of local * to global objects. This
> accurately describes things liek resource controllers, cpusets, resource
> monitoring, etc.

No!

Cpusets don't rename or change the mapping of objects.

I suspect you seriously misunderstand cpusets and are trying to cram them into a 'namespace' remapping role into which they don't fit.

So far as cpusets are concerned, CPU #17 is CPU #17, for all tasks, regardless of what cpuset they are in. They just might not happen to be allowed to execute on CPU #17 at the moment, because that CPU is not allowed by the cpuset they are in.

But they still call it CPU #17.

Similary the namespace of cpusets and of tasks (pid's) are single system-wide namespaces, so far as cpusets are concerned.

Cpusets are not about alternative or multiple or variant name spaces.
They are about (considering just CPUs for the moment):
 1) creating a set of maps M0, M1, ... from the set of CPUs to a Boolean,
 2) creating a mapping Q from the set of tasks to these M0, ... maps, and
 3) imposing constraints on where tasks can run, as follows:
 For any task t, that task is allowed to run on CPU x iff Q(t)(x)
 is True. Here, Q(t) will be one of the maps M0, ... aka a cpuset.

So far as cpusets are concerned, there is only one each of:
 A] a namespace numbering CPUs,
 B] a namespace numbering tasks (the process id),
 C] a namespace naming cpusets (the hierarchical name space normally
    mounted at /dev/cpuset, and corresponding to the Mn maps above) and
 D] a mapping of tasks to cpusets, system wide (just a map, not a namespace.)

All tasks (of sufficient authority) can see each of these, using a single system wide name space for each of [A], [B], and [C].

Unless, that is, you call any mapping a "way of changing mappings".
To do so would be a senseless abuse of the phrase, in my view.

More generally, these resource managers all tend to divide some external limited physical resource into multiple separately allocatable units.

If the resource is amorphous (one atom or cycle of it is interchangeable with another) then we usually do something like divide it into 100 equal units and speak of percentages. If the resource is naturally subdivided into sufficiently small units (sufficient for the granularity of resource management we require) then we take those units as is. Occassionally, as in the 'fake numa node' patch by David Rientjes <rientjes@cs.washington.edu>, who worked at Google over the last summer, if the natural units are not of sufficient granularity, we fake up a somewhat finer division.

Then, in any case, and somewhat separately, we divide the tasks running on the system into subsets. More precisely, we partition the tasks, where a partition of a set is a set of subsets of that set, pairwise disjoint, whose union equals that set.

Then, finally, we map the task subsets (partition element) to the
resource units, and add hooks in the kernel where this particular
resource is allocated or scheduled to constrain the tasks to only using
the units to which their task partition element is mapped.

These hooks are usually the 'interesting' part of a resource management
patch; one needs to minimize impact on both the kernel source code and
on the runtime performance, and for these hooks, that can be a
challenge. In particular, what are naturally system wide resource
management stuctures cannot be allowed to impose system wide locks on
critical resource allocation code paths (and it's usually the most
critical resources, such as memory, cpu and network, that we most need
to manage in the first place.)

==> This has nothing to do with remapping namespaces as I might use that
phrase though I cannot claim to be qualified enough to speak on behalf
of the Generally Established Principles of Computer Science.

I am as qualified as anyone to speak on behalf of cpusets, and I suspect
you are not accurately understanding them if you think of them as remapping
namespaces.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by dev on Fri, 09 Mar 2007 09:07:27 GMT
View Forum Message <> Reply to Message

> nobody actually cares about a precise accounting and
> calculating shares or partitions of whatever resource,
> all that matters is that you have a way to prevent a
> potential hostile environment from sucking up all your
> resources (or even a single one) resulting in a DoS

This is not true. People care. Reasons:
- resource planning
- fairness
- guarantees

What you talk is about security only. Not the above issues.
So good precision is required. If there is no precision at all,
security sucks as well and can be exploited, e.g. for CPU

schedulers doing an accounting based on
jiffies accounting in scheduler_tick() it is easy to build
an application consuming 90% of CPU, but ~0% from scheduler POV.

Kirill

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by dev on Fri, 09 Mar 2007 09:23:55 GMT

View Forum Message <> Reply to Message

>>There have been various projects attempting to provide resource
>>management support in Linux, including CKRM/Resource Groups and UBC.
>
>
> let me note here, once again, that you forgot Linux-VServer
> which does quite non-intrusive resource management ...
Herbert, do you care to send patches except for ask others to do
something that works for you?

Looks like your main argument is non-intrusive...
"working", "secure", "flexible" are not required to people any more? :/

>> Each had its own task-grouping mechanism.
>
>
> the basic 'context' (pid space) is the grouping mechanism
> we use for resource management too
>
>
>>Paul Menage observed [1] that cpusets in the kernel already has a
>>grouping mechanism which was working well for cpusets. He went ahead
>>and generalized the grouping code in cpusets so that it could be used
>>for overall resource management purpose.
>
>
>>With his patches, it is possible to even create multiple hierarchies
>>of groups (see [2] on why multiple hierarchies) as follows:
>
>
> do we need or even want that? IMHO the hierarchical
> concept CKRM was designed with, was also the reason
> for it being slow, unuseable and complicated
1. cpusets are hierarchical already. So hierarchy is required.

2. As it was discussed on the call controllers which are flat
   can just prohibit creation of hierarchy on the filesystem.
   i.e. allow only 1 depth and continue being fast.

>>mount -t container -o cpuset none /dev/cpuset <- cpuset hierarchy
>>mount -t container -o mem,cpu none /dev/mem <- memory/cpu hierarchy
>>mount -t container -o disk none /dev/disk <- disk hierarchy
>>
>>In each hierarchy, you can create task groups and manipulate the
>>resource parameters of each group. You can also move tasks between
>>groups at run-time (see [3] on why this is required).
>
>
>>Each hierarchy is also manipulated independent of the other.
>
>
>>Paul's patches also introduced a 'struct container' in the kernel,
>>which serves these key purposes:
>>
>>- Task-grouping
>> 'struct container' represents a task-group created in each hierarchy.
>> So every directory created under /dev/cpuset or /dev/mem above will
>> have a corresponding 'struct container' inside the kernel. All tasks
>> pointing to the same 'struct container' are considered to be part of
>> a group
>>
>> The 'struct container' in turn has pointers to resource objects which
>> store actual resource parameters for that group. In above example,
>> 'struct container' created under /dev/cpuset will have a pointer to
>> 'struct cpuset' while 'struct container' created under /dev/disk will
>> have pointer to 'struct disk_quota_or_whatever'.
>>
>>- Maintain hierarchical information
>> The 'struct container' also keeps track of hierarchical relationship
>> between groups.
>>
>>The filesystem interface in the patches essentially serves these
>>purposes:
>>
>> - Provide an interface to manipulate task-groups. This includes
>>   creating/deleting groups, listing tasks present in a group and
>>   moving tasks across groups
>>
>> - Provdes an interface to manipulate the resource objects
>>   (limits etc) pointed to by 'struct container'.
>>
>>As you know, the introduction of 'struct container' was objected
>>to and was felt redundant as a means to group tasks. Thats where I

>>took a shot at converting over Paul Menage's patch to avoid 'struct
>>container' abstraction and insead work with 'struct nsproxy'.
>
>
> which IMHO isn't a step in the right direction, as
> you will need to handle different nsproxies within
> the same 'resource container' (see previous email)
tend to agree.
Looks like Paul's original patch was in the right way.

[...]
>>A separate filesystem would give us more flexibility like the
>>implementing multi-hierarchy support described above.
>
>
> why is the filesystem approach so favored for this
> kind of manipulations?
>
> IMHO it is one of the worst interfaces I can imagine
> (to move tasks between spaces and/or assign resources)
> but yes, I'm aware that filesystems are 'in' nowadays
I also hate filesystems approach being used nowdays everywhere.
But, looks like there are reasons still:
1. cpusets already use fs interface.
2. each controller can have a bit of specific information/controls exported easily.

Can you suggest any other extensible/flexible interface for these?

Thanks,
Kirill

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Paul Jackson on Fri, 09 Mar 2007 09:38:17 GMT

View Forum Message <> Reply to Message

Kirill, responding to Herbert:
> > do we need or even want that? IMHO the hierarchical
> > concept CKRM was designed with, was also the reason
> > for it being slow, unuseable and complicated
> 1. cpusets are hierarchical already. So hierarchy is required.

I think that CKRM has a harder time doing a hierarchy than cpusets.

CKRM is trying to account for and control how much of an amorphous
resource is used, whereas cpusets is trying to control whether a
specifically identifiable resource is used, or not used, not how
much of it is used.

A child cpuset gets configured to allow certain CPUs and Nodes, and
then does not need to dynamically pass back any information about
what is actually used - it's a one-way control with no feedback.
That's a relatively easier problem.

CKRM (as I recall it, from long ago ...) has to track the amount
of usage dynamically, across parent and child groups (whatever they
were called.) That's a harder problem.

So, yes, as Kirill observes, we need the hierarchy because cpusets
has it, cpuset users make good use of the hierarchy, and the hierarchy
works fine in that case, even if a hierarchy is more difficult for CKRM.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Fri, 09 Mar 2007 13:21:37 GMT

View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 12:23:55PM +0300, Kirill Korotaev wrote:
>>> There have been various projects attempting to provide
>>> resource management support in Linux, including
>>> CKRM/Resource Groups and UBC.
>>
>> let me note here, once again, that you forgot Linux-VServer
>> which does quite non-intrusive resource management ...

> Herbert, do you care to send patches except for ask
> others to do something that works for you?

sorry, I'm not in the lucky position that I get payed
for sending patches to LKML, so I have to think twice
before I invest time in coding up extra patches ...

i.e. you will have to live with my comments for now

> Looks like your main argument is non-intrusive...
> "working", "secure", "flexible" are not required to
> people any more? :/

well, Linux-VServer is "working", "secure", "flexible"
and non-intrusive ... it is quite natural that less
won't work for me ... and regarding patches, there
will be a 2.2 release soon, with all the patches ...

>>> Each had its own task-grouping mechanism.

>> the basic 'context' (pid space) is the grouping mechanism
>> we use for resource management too

>>> Paul Menage observed [1] that cpusets in the kernel already has a
>>> grouping mechanism which was working well for cpusets. He went ahead
>>> and generalized the grouping code in cpusets so that it could be
>>> used for overall resource management purpose.

>>> With his patches, it is possible to even create multiple hierarchies
>>> of groups (see [2] on why multiple hierarchies) as follows:

>> do we need or even want that? IMHO the hierarchical
>> concept CKRM was designed with, was also the reason
>> for it being slow, unuseable and complicated

> 1. cpusets are hierarchical already. So hierarchy is required.
> 2. As it was discussed on the call controllers which are flat
>   can just prohibit creation of hierarchy on the filesystem.
>   i.e. allow only 1 depth and continue being fast.
>
>>> mount -t container -o cpuset none /dev/cpuset <- cpuset hierarchy
>>> mount -t container -o mem,cpu none /dev/mem <- memory/cpu hierarchy
>>> mount -t container -o disk none /dev/disk <- disk hierarchy
>>>
>>> In each hierarchy, you can create task groups and manipulate the
>>> resource parameters of each group. You can also move tasks between
>>> groups at run-time (see [3] on why this is required).

>>> Each hierarchy is also manipulated independent of the other.

>>> Paul's patches also introduced a 'struct container' in the kernel,
>>> which serves these key purposes:
>>>
>>> - Task-grouping
>>>   'struct container' represents a task-group created in each hierarchy.

>>>   So every directory created under /dev/cpuset or /dev/mem above will
>>>   have a corresponding 'struct container' inside the kernel. All tasks
>>>   pointing to the same 'struct container' are considered to be part of
>>>   a group
>>>
>>>   The 'struct container' in turn has pointers to resource objects which
>>>   store actual resource parameters for that group. In above example,
>>>   'struct container' created under /dev/cpuset will have a pointer to
>>>   'struct cpuset' while 'struct container' created under /dev/disk will
>>>   have pointer to 'struct disk_quota_or_whatever'.
>>>
>>> - Maintain hierarchical information
>>>   The 'struct container' also keeps track of hierarchical relationship
>>>   between groups.
>>>
>>> The filesystem interface in the patches essentially serves these
>>> purposes:
>>>
>>> - Provide an interface to manipulate task-groups. This includes
>>>   creating/deleting groups, listing tasks present in a group and
>>>   moving tasks across groups
>>>
>>> - Provdes an interface to manipulate the resource objects
>>>   (limits etc) pointed to by 'struct container'.
>>>
>>> As you know, the introduction of 'struct container' was objected
>>> to and was felt redundant as a means to group tasks. Thats where I
>>> took a shot at converting over Paul Menage's patch to avoid 'struct
>>> container' abstraction and insead work with 'struct nsproxy'.

>> which IMHO isn't a step in the right direction, as
>> you will need to handle different nsproxies within
>> the same 'resource container' (see previous email)

> tend to agree.
> Looks like Paul's original patch was in the right way.

> [...]

>>> A separate filesystem would give us more flexibility like the
>>> implementing multi-hierarchy support described above.

>> why is the filesystem approach so favored for this
>> kind of manipulations?

>> IMHO it is one of the worst interfaces I can imagine
>> (to move tasks between spaces and/or assign resources)
>> but yes, I'm aware that filesystems are 'in' nowadays

> I also hate filesystems approach being used nowdays everywhere.
> But, looks like there are reasons still:
> 1. cpusets already use fs interface.
> 2. each controller can have a bit of specific
>   information/controls exported easily.

yes, but there are certain drawbacks too, like:

- performance of filesystem interfaces is quite bad
- you need to do a lot to make the fs consistant for
  e.g. find and friends (regarding links and filesize)
- you have a quite hard time to do atomic operations
  (except for the ioctl interface, which nobody likes)
- vfs/mnt namespaces complicate the access to this
  new filesystem once you start moving around (between
  the spaces)

> Can you suggest any other extensible/flexible interface for these?

well, as you know, all current solutions use a syscall
interface to do most of the work, in the OpenVZ/Virtuozzo
case several, unassigned syscalls are used, while
FreeVPS and Linux-VServer use a registered and versioned
(multiplexed) system call, which works quite fine for
all known purposes ...

I'm quite happy with the extensibility and flexibility
the versioned syscall interface has, the only thing I'd
change if I would redesign that interface is, that I
would add another pointer argument to eliminate 32/64bit
issues completely (i.e. use 4 args instead of the 3)

best,
Herbert

> Thanks,
> Kirill
>
> -
> To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
> Please read the FAQ at http://www.tux.org/lkml/

Subject: Re: [PATCH 1/2] rcfs core patch

Posted by Herbert Poetzl on Fri, 09 Mar 2007 13:29:22 GMT

View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 12:07:27PM +0300, Kirill Korotaev wrote:

>> nobody actually cares about a precise accounting and
>> calculating shares or partitions of whatever resource,
>> all that matters is that you have a way to prevent a
>> potential hostile environment from sucking up all your
>> resources (or even a single one) resulting in a DoS

> This is not true. People care. Reasons:
> - resource planning
> - fairness
> - guarantees

let me make that a little more clear ...

nobody cares wether a shared memory page is
accounted as full page or as fraction of a page
(depending on the number of guests sharing it)
as long as the accounted amount is substracted
correctly when the page is disposed

so there _is_ a difference between _false_
accounting (which seems what you are referring
to in the next paragraph) and imprecise, but
consistant accounting (which is what I was
talking about)

best,
Herbert

> What you talk is about security only. Not the above issues.
> So good precision is required. If there is no precision at all,
> security sucks as well and can be exploited, e.g. for CPU
> schedulers doing an accounting based on jiffies accounting in
> scheduler_tick() it is easy to build an application consuming
> 90% of CPU, but ~0% from scheduler POV.

> Kirill

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

Subject: Re: [PATCH 1/2] rcfs core patch

Posted by serue on Fri, 09 Mar 2007 16:16:18 GMT
View Forum Message <> Reply to Message

Quoting Srivatsa Vaddagiri (vatsa@in.ibm.com):
> On Wed, Mar 07, 2007 at 08:12:00PM -0700, Eric W. Biederman wrote:
> > I have a question? What does rcfs look like if we start with
> > the code that is in the kernel? That is start with namespaces
> > and nsproxy and just build a filesystem to display/manipulate them?
> > With the code built so it will support adding resource controllers
> > when they are ready?
>
> If I am not mistaken, Serge did attempt something in that direction,
> only that it was based on Paul's container patches. rcfs can no doubt
> support the same feature.

My first nsproxy control fs was not based on Paul's set, but was very
nsproxy-specific. After that I based it on Paul's set. The nsproxy
subsystem is in Paul's latest patchset.

On top of that set I've posted a set to implement namespace entering,
but that is on hold (or dropped) because the only people so far
interested in namespace entering don't like an fs interface.

If you wanted to take my original non-container fs nsproxy interface and
try to rebuild resource containers on top of them I don't think you'll
get anything better than containers, but then maybe you would...

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Srivatsa Vaddagiri on Fri, 09 Mar 2007 16:34:30 GMT
View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 01:20:18PM -0800, Paul Menage wrote:
> On 3/7/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
> >
> >All that being said, if it were going to save space without overly
> >complicating things I'm actually not opposed to using nsproxy, but it
>
> If space-saving is the main issue, then the latest version of my
> containers patches uses just a single pointer in the task_struct, and
> all tasks in the same set of containers (across all hierarchies) will

> share a single container_group object, which holds the actual pointers
> to container state.

Paul,
Some more thoughts, mostly coming from the point of view of vservers/containers/"whaever is the set of tasks sharing a nsproxy is called".

1. What is the fundamental unit over which resource-management is applied? Individual tasks or individual containers?

/me thinks latter. In which case, it makes sense to stick resource control information in the container somewhere. Just like when controlling a user's resource consumption, 'struct user_struct' may be a natural place to put these resource limits.

2. Regarding space savings, if 100 tasks are in a container (I dont know what is a typical number) -and- lets say that all tasks are to share the same resource allocation (which seems to be natural), then having a 'struct container_group *' pointer in each task_struct seems to be not very efficient (simply because we dont need that task-level granularity of managing resource allocation).

3. This next leads me to think that 'tasks' file in each directory doesnt make sense for containers. In fact it can lend itself to error situations (by administrator/script mistake) when some tasks of a container are in one resource class while others are in a different class.

Instead, from a containers pov, it may be usefull to write a 'container id' (if such a thing exists) into the tasks file which will move all the tasks of the container into the new resource class. This is the same requirement we discussed long back of moving all threads of a process into new resource class.

--
Regards,
vatsa
____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [ckrm-tech] [PATCH 0/2] resource control file system - aka containers

on top of nsproxy!
Posted by Srivatsa Vaddagiri on Fri, 09 Mar 2007 16:41:18 GMT
View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 10:04:30PM +0530, Srivatsa Vaddagiri wrote:
> 2. Regarding space savings, if 100 tasks are in a container (I dont know
>   what is a typical number) -and- lets say that all tasks are to share
>   the same resource allocation (which seems to be natural), then having
>   a 'struct container_group *' pointer in each task_struct seems to be not
>   very efficient (simply because we dont need that task-level granularity of
>   managing resource allocation).

Note that this 'struct container_group *' pointer is in addition to the
'struct nsproxy *' pointer already in task_struct. If the set of tasks
over which resorce control is applied is typically the same set of tasks
which share the same 'struct nsproxy *' pointer, then IMHO 'struct
container_group *' in each task_struct is not very optimal.

--
Regards,
vatsa

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Srivatsa Vaddagiri on Fri, 09 Mar 2007 17:57:07 GMT
View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 01:38:19AM +0100, Herbert Poetzl wrote:
> > 2) you allow a task to selectively reshare namespaces/subsystems with
> >   another task, i.e. you can update current->task_proxy to point to
> >   a proxy that matches your existing task_proxy in some ways and the
> >   task_proxy of your destination in others. In that case a trivial
> >   implementation would be to allocate a new task_proxy and copy some
> >   pointers from the old task_proxy and some from the new. But then
> >   whenever a task moves between different groupings it acquires a
> >   new unique task_proxy. So moving a bunch of tasks between two
> >   groupings, they'd all end up with unique task_proxy objects with
> >   identical contents.
>
> this is exactly what Linux-VServer does right now, and I'm
> still not convinced that the nsproxy really buys us anything
> compared to a number of different pointers to various spaces
> (located in the task struct)

Are you saying that the current scheme of storing pointers to different
spaces (uts_ns, ipc_ns etc) in nsproxy doesn't buy anything?

Or are you referring to storage of pointers to resource (name)spaces
in nsproxy doesn't buy anything?

In either case, doesn't it buy speed and storage space?

> I'd prefer to do accounting (and limits) in a very simple
> and especially performant way, and the reason for doing
> so is quite simple:

Can you elaborate on the relationship between data structures used to store
those limits to the task_struct? Does task_struct store pointers to those
objects directly?

--
Regards,
vatsa

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Srivatsa Vaddagiri on Fri, 09 Mar 2007 18:14:22 GMT
View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 01:48:16AM +0100, Herbert Poetzl wrote:
> > There have been various projects attempting to provide resource
> > management support in Linux, including CKRM/Resource Groups and UBC.
>
> let me note here, once again, that you forgot Linux-VServer
> which does quite non-intrusive resource management ...

Sorry, not intentionally. Maybe it slipped because I haven't seen much res mgmt
related patches from Linux Vserver on lkml recently. Note that I -did- talk
about VServer at one point in past (http://lkml.org/lkml/2006/06/15/112)!

> the basic 'context' (pid space) is the grouping mechanism
> we use for resource management too

so tasks sharing the same nsproxy->pid_ns is the fundamental unit of
resource management (as far as vserver/container goes)?

> > As you know, the introduction of 'struct container' was objected
> > to and was felt redundant as a means to group tasks. Thats where I

> > took a shot at converting over Paul Menage's patch to avoid 'struct
> > container' abstraction and insead work with 'struct nsproxy'.
>
> which IMHO isn't a step in the right direction, as
> you will need to handle different nsproxies within
> the same 'resource container' (see previous email)

Isn't that made simple because of the fact that we have pointers to namespace objects (and not actual objects themselves) in nsproxy?

I mean, all that is required to manage multiple nsproxy's is to have the pointer to the same resource object in all of them.

In system call terms, if someone does a unshare of uts namespace, he will get into a new nsproxy object sure (which has a pointer to the new uts namespace) but the new nsproxy object will still be pointing to the old resource controlling objects.

> > When we support task movement across resource classes, we need to find a
> > nsproxy which has the right combination of resource classes that the
> > task's nsproxy can be hooked to.
>
> no, not necessarily, we can simply create a new one
> and give it the proper resource or whatever-spaces

That would be the simplest, agreeably. But not optimal in terms of storage?

Pls note that task-movement can be not-so-infrequent (in other words, frequent) in context of non-container workload management.

> why is the filesystem approach so favored for this
> kind of manipulations?
>
> IMHO it is one of the worst interfaces I can imagine
> (to move tasks between spaces and/or assign resources)
> but yes, I'm aware that filesystems are 'in' nowadays

Ease of use maybe. Scripts can be more readily used with a fs-based interface.

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Fri, 09 Mar 2007 18:41:05 GMT
View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 02:16:08AM +0100, Herbert Poetzl wrote:
> On Thu, Mar 08, 2007 at 05:00:54PM +0530, Srivatsa Vaddagiri wrote:
> > On Thu, Mar 08, 2007 at 01:50:01PM +1300, Sam Vilain wrote:
> > > 7. resource namespaces
> >
> > It should be. Imagine giving 20% bandwidth to a user X. X wants to
> > divide this bandwidth further between multi-media (10%), kernel
> > compilation (5%) and rest (5%). So,
>
> sounds quite nice, but ...
>
> > > Is the subservient namespace's resource usage counting against ours too?
> >
> > Yes, the resource usage of children should be accounted when capping
> > parent resource usage.
>
> it will require to do accounting many times
> (and limit checks of course), which in itself
> might be a way to DoS the kernel by creating
> more and more resource groups

I was only pointing out the usefullness of the feature and not
necessarily saying it -should- be implemented! Ofcourse I understand it
will make the controller complicated and thats why probably none of the
recontrollers we are seeing posted on lkml don't support hierarchical
res mgmt.

> > > Can we dynamically alter the subservient namespace's resource
> > > allocations?
> >
> > Should be possible yes. That lets user X completely manage his
> > allocation among whatever sub-groups he creates.
>
> what happens if the parent changes, how is
> the resource change (if it was a reduction)
> propagated to the children?
>
> e.g. your guest has 1024 file handles, now
> you reduce it to 512, but the guest had two
> children, both with 256 file handles each ...

I believe CKRM handled this quite neatly (by defining child shares to be
relative to parent shares).

In your example, 256+256 add up to 512 which is within the parent's new limit,
so nothing happens :) You also picked an example of exhaustible/non-reclaimable
resource, which makes it hard to define what should happen if parent's limit
goes below 512. Either nothing happens or perhaps a task is killed,
don't know. In case of memory, I would say that some of child's pages may
get kicked out and in case of cpu, child will start getting fewer
cycles.

> > The patches should give visibility to both nsproxy objects (by showing
> > what tasks share the same nsproxy objects and letting tasks move across
> > nsproxy objects if allowed) and the resource control objects pointed to
> > by nsproxy (struct cpuset, struct cpu_limit, struct rss_limit etc).
>
> the nsproxy is not really relevant, as it
> is some kind of strange indirection, which
> does not necessarily depict the real relations,
> regardless wether you do the re-sharing of
> those nsproies or not ..

So what are you recommending we do instead? My thought was whatever is
the fundamental unit to which resource management needs to be applied,
lets store resource parameters (or pointers to them) there (rather than
duplicating the information in each task_struct).

--
Regards,
vatsa
____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Paul Jackson on Fri, 09 Mar 2007 19:25:47 GMT
View Forum Message <> Reply to Message

> Ease of use maybe. Scripts can be more readily used with a fs-based
> interface.

And, as I might have already stated, file system API's are a natural
fit for hierarchically shaped data, especially if the nodes in the
hierarchy would benefit from file system like permission attributes.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!

Posted by Paul Menage on Fri, 09 Mar 2007 22:09:35 GMT
View Forum Message <> Reply to Message

On 3/9/07, Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:
>
> 1. What is the fundamental unit over which resource-management is
> applied? Individual tasks or individual containers?
>
>   /me thinks latter.

Yes

> In which case, it makes sense to stick
>   resource control information in the container somewhere.

Yes, that's what all my patches have been doing.

> 2. Regarding space savings, if 100 tasks are in a container (I dont know
>   what is a typical number) -and- lets say that all tasks are to share
>   the same resource allocation (which seems to be natural), then having
>   a 'struct container_group *' pointer in each task_struct seems to be not
>   very efficient (simply because we dont need that task-level granularity of
>   managing resource allocation).

I think you should re-read my patches.

Previously, each task had N pointers, one for its container in each
potential hierarchy. The container_group concept means that each task
has 1 pointer, to a set of container pointers (one per hierarchy)
shared by all tasks that have exactly the same set of containers (in
the various different hierarchies).

It doesn't give task-level granularity of resource management (unless
you create a separate container for each task), it just gives a space
saving.

>
> 3. This next leads me to think that 'tasks' file in each directory doesnt make
>   sense for containers. In fact it can lend itself to error situations (by
>   administrator/script mistake) when some tasks of a container are in one

> resource class while others are in a different class.
>
> Instead, from a containers pov, it may be usefull to write
> a 'container id' (if such a thing exists) into the tasks file
> which will move all the tasks of the container into
> the new resource class. This is the same requirement we
> discussed long back of moving all threads of a process into new
> resource class.

I think you need to give a more concrete example and use case of what
you're trying to propose here. I don't really see what advantage
you're getting.

Paul

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Sat, 10 Mar 2007 00:56:43 GMT

View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 11:44:22PM +0530, Srivatsa Vaddagiri wrote:
> On Fri, Mar 09, 2007 at 01:48:16AM +0100, Herbert Poetzl wrote:
> > > There have been various projects attempting to provide resource
> > > management support in Linux, including CKRM/Resource Groups and UBC.

> > let me note here, once again, that you forgot Linux-VServer
> > which does quite non-intrusive resource management ...

> Sorry, not intentionally. Maybe it slipped because I haven't
> seen much res mgmt related patches from Linux Vserver on
> lkml recently.

mainly because I got the impression that we planned
to work on the various spaces first, and handle things
like resource management later .. but it seems that
resource management is now in focus, while the spaces
got somewhat delayed ...

> Note that I -did- talk about VServer at one point in past
> (http://lkml.org/lkml/2006/06/15/112)!

noted and appreciated (although this was about CPU
resources, which IMHO is a special resource like
the networking, as you are mostly interested in

'bandwidth' limitations there, not in resource
limits per se (and of course, it wasn't even cited
correctly, as it is Linux-VServer not vserver ...)

> > the basic 'context' (pid space) is the grouping mechanism
> > we use for resource management too

> so tasks sharing the same nsproxy->pid_ns is the fundamental
> unit of resource management (as far as vserver/container goes)?

we currently have a 'process' context, which holds
the administrative data (capabilities and flags) and
the resource accounting and limits, which basically
contains the pid namespace, so yes and no

it contains a reference to the 'main' nsproxy, which
is used to copy spaces from when you enter the guest
(or some set of spaces), and it defines the unit we
consider a process container

> > > As you know, the introduction of 'struct container' was objected
> > > to and was felt redundant as a means to group tasks. Thats where
> > > I took a shot at converting over Paul Menage's patch to avoid
> > > 'struct container' abstraction and insead work with 'struct
> > > nsproxy'.
> >
> > which IMHO isn't a step in the right direction, as
> > you will need to handle different nsproxies within
> > the same 'resource container' (see previous email)
>
> Isn't that made simple because of the fact that we have pointers to
> namespace objects (and not actual objects themselves) in nsproxy?
>
> I mean, all that is required to manage multiple nsproxy's
> is to have the pointer to the same resource object in all of them.
>
> In system call terms, if someone does a unshare of uts namespace,
> he will get into a new nsproxy object sure (which has a pointer to the
> new uts namespace) but the new nsproxy object will still be pointing
> to the old resource controlling objects.

yes, that is why I agreed, that the container (or
resource limit/accounting/controlling object) can
be seen as space too (and handled like that)

> > > When we support task movement across resource classes, we need to
> > > find a nsproxy which has the right combination of resource classes
> > > that the task's nsproxy can be hooked to.

> >
> > no, not necessarily, we can simply create a new one
> > and give it the proper resource or whatever-spaces
>
> That would be the simplest, agreeably. But not optimal in terms of
> storage?
>
> Pls note that task-movement can be not-so-infrequent
> (in other words, frequent) in context of non-container workload
> management.

not only there, also with solutions like Linux-VServer
(it is quite common to enter guests or subsets of the
space mix assigned)

> > why is the filesystem approach so favored for this
> > kind of manipulations?
> >
> > IMHO it is one of the worst interfaces I can imagine
> > (to move tasks between spaces and/or assign resources)
> > but yes, I'm aware that filesystems are 'in' nowadays
>
> Ease of use maybe. Scripts can be more readily used with a fs-based
> interface.

correct, but what about security and/or atomicity?
i.e. how to assure that some action really was
taken and/or how to wait for completion?

sure, all this _can_ be done, no doubt, but it
is much harder to do with a fs based interface than
with e.g. a syscall interface ...

> --
> Regards,
> vatsa
> ____
> Containers mailing list
> Containers@lists.osdl.org
> https://lists.osdl.org/mailman/listinfo/containers
____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH 1/2] rcfs core patch

Posted by Herbert Poetzl on Sat, 10 Mar 2007 01:00:41 GMT

View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 11:25:47AM -0800, Paul Jackson wrote:

> > Ease of use maybe. Scripts can be more readily used with a fs-based
> > interface.
>
> And, as I might have already stated, file system API's are a natural
> fit for hierarchically shaped data, especially if the nodes in the
> hierarchy would benefit from file system like permission attributes.

personally, I'd prefer to avoid hierarchical
structures wherever possible, because they tend
to make processing and checks a lot more complicated
than necessary, and if we really want hierarchical
structures, it might be more than sufficient to
keep the hierarchy in userspace, and use a flat
representation inside the kernel ...

but hey, I'm all for running a hypervisor under
a hypervisor running inside a hypervisor :)

best,
Herbert

> --
>   I won't rest till it's the best ...
>   Programmer, Linux Scalability
>   Paul Jackson <pj@sgi.com> 1.925.600.0401
> ____
> Containers mailing list
> Containers@lists.osdl.org
> https://lists.osdl.org/mailman/listinfo/containers

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Sat, 10 Mar 2007 01:19:53 GMT

View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 11:27:07PM +0530, Srivatsa Vaddagiri wrote:

> On Fri, Mar 09, 2007 at 01:38:19AM +0100, Herbert Poetzl wrote:
> > > 2) you allow a task to selectively reshare namespaces/subsystems with
> > >   another task, i.e. you can update current->task_proxy to point to
> > >   a proxy that matches your existing task_proxy in some ways and the

> > > task_proxy of your destination in others. In that case a trivial
> > > implementation would be to allocate a new task_proxy and copy some
> > > pointers from the old task_proxy and some from the new. But then
> > > whenever a task moves between different groupings it acquires a
> > > new unique task_proxy. So moving a bunch of tasks between two
> > > groupings, they'd all end up with unique task_proxy objects with
> > > identical contents.

> > this is exactly what Linux-VServer does right now, and I'm
> > still not convinced that the nsproxy really buys us anything
> > compared to a number of different pointers to various spaces
> > (located in the task struct)

> Are you saying that the current scheme of storing pointers to
> different spaces (uts_ns, ipc_ns etc) in nsproxy doesn't buy
> anything?

> Or are you referring to storage of pointers to resource
> (name)spaces in nsproxy doesn't buy anything?

> In either case, doesn't it buy speed and storage space?

let's do a few examples here, just to illustrate the
advantages and disadvantages of nsproxy as separate
structure over nsproxy as part of the task_struct

1) typical setup, 100 guests as shell servers, 5
tasks each when unused, 10 tasks when used 10%
used in average

a) separate nsproxy, we need at least 100
structs to handle that (saves some space)

we might end up with ~500 nsproxies, if
the shell clones a new namespace (so might
not save that much space)

we do a single inc/dec when the nsproxy
is reused, but do the full N inc/dec when
we have to copy an nsproxy (might save
some refcounting)

we need to do the indirection step, from
task to nsproxy to space (and data)

b) we have ~600 tasks with 600 times the
nsproxy data (uses up some more space)

we have to do the full N inc/dev when we create a new task (more refcounting)

we do not need to do the indirection, we access spaces directly from the 'hot' task struct (makes hot pathes quite fast)

so basically we trade a little more space and overhead on task creation for having no indirection to the data accessed quite often throughout the tasks life (hopefully)

2) context migration: for whatever reason, we decide to migrate a task into a subset (space mix) of a context 1000 times

a) separate nsproxy, we need to create a new one consisting of the 'new' mix, which will

- allocate the nsproxy struct
- inc refcounts to all copied spaces
- inc refcount nsproxy and assign to task
- dec refcount existing task nsproxy

after task completion
- dec nsproxy refcount
- dec refcounts for all spaces
- free up nsproxy struct

b) nsproxy data in task struct

- inc/dec refcounts to changed spaces

after task completion
- dec refcounts to spaces

so here we gain nothing with the nsproxy, unless the chosen subset is identical to the one already used, where we end up with a single refcount instead of N

> > I'd prefer to do accounting (and limits) in a very simple
> > and especially performant way, and the reason for doing
> > so is quite simple:

> Can you elaborate on the relationship between data structures
> used to store those limits to the task_struct?

sure it is one to many, i.e. each task points to
exactly one context struct, while a context can
consist of zero, one or many tasks (no back-
pointers there)

> Does task_struct store pointers to those objects directly?

it contains a single pointer to the context struct,
and that contains (as a substruct) the accounting
and limit information

HTC,
Herbert

> --
> Regards,
> vatsa
> ____
> Containers mailing list
> Containers@lists.osdl.org
> https://lists.osdl.org/mailman/listinfo/containers
____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Paul Jackson on Sat, 10 Mar 2007 01:31:15 GMT
View Forum Message <> Reply to Message

Herbert wrote:
> personally, I'd prefer to avoid hierarchical
> structures wherever possible,

Sure - avoid them if you like. But sometimes they work out rather
well. And file system API's are sometimes the best fit for them.

I'm all for choosing the simplest API topology that makes sense.

But encoding higher dimension topologies into lower dimension API's,
just because they seem "simpler" results in obfuscation, convolution
and obscurity, which ends up costing everyone more than getting the
natural fit.

"Make everything as simple as possible, but not simpler."
-- Albert Einstein

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Sat, 10 Mar 2007 02:02:20 GMT
View Forum Message <> Reply to Message

I think maybe I didnt communicate what I mean by a container here
(although I thought I did). I am referring to a container in a vserver
context (set of tasks which share the same namespace).

On Fri, Mar 09, 2007 at 02:09:35PM -0800, Paul Menage wrote:
> >2. Regarding space savings, if 100 tasks are in a container (I dont know
> >   what is a typical number) -and- lets say that all tasks are to share
> >   the same resource allocation (which seems to be natural), then having
> >   a 'struct container_group *' pointer in each task_struct seems to be not
> >   very efficient (simply because we dont need that task-level granularity
> >   of
> >   managing resource allocation).
>
> I think you should re-read my patches.
>
> Previously, each task had N pointers, one for its container in each
> potential hierarchy. The container_group concept means that each task
> has 1 pointer, to a set of container pointers (one per hierarchy)
> shared by all tasks that have exactly the same set of containers (in
> the various different hierarchies).

Ok, let me see if I can convey what I had in mind better:

```
    uts_ns pid_ns ipc_ns
  \    |    /
  ---------------
      | nsproxy   |
       ----------------
             /  |  \    \ <-- 'nsproxy' pointer
  T1  T2  T3 ...T1000
  |   |   |     | <-- 'containers' pointer (4/8 KB for 1000 task)
      -------------------
     | container_group   |
      ------------------
  /
     ----------
    | container |
     ----------
  |
     ----------
    | cpu_limit |
     ----------
```

(T1, T2, T3 ..T1000) are part of a vserver lets say sharing the same uts/pid/ipc_ns. Now where do we store the resource control information for this unit/set-of-tasks in your patches?

```
 (tsk->containers->container[cpu_ctlr.hierarchy] + X)->cpu_limit
```

(The X is to account for the fact that cotainer structure points to a 'struct container_subsys_state' embedded in some other structure. Its usually zero if the structure is embedded at the top)

I understand that container_group also points directly to 'struct container_subsys_state', in which case, the above is optimized to:

```
 (tsk->containers->subsys[cpu_ctlr.subsys_id] + X)->cpu_limit
```

Did I get that correct?

Compare that to:

```
         -----------
     | cpu_limit |
    uts_ns pid_ns ipc_ns   ----------
  \    |    /     |
  ------------------------
      |  nsproxy   |
       ------------------------
              /  |   \  |
  T1  T2  T3 .....T1000
```

We save on 4/8 KB (for 1000 tasks) by avoiding the 'containers' pointer in each task_struct (just to get to the resource limit information).

So my observation was (again note primarily from a vserver context): given that (T1, T2, T3 ..T1000) will all need to be managed as a unit (because they are all sharing the same nsproxy pointer), then having the '->containers' pointer in -each- one of them to tell the unit's limit is not optimal. Instead store the limit in the proper unit structure (in this case nsproxy - but

whatever else is more suitable vserver datastructure (pid_ns?) which
represent the fundamental unit of res mgmt in vservers).

(I will respond to remaining comments later ..too early in the morning now!)

--
Regards,
vatsa

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Herbert Poetzl on Sat, 10 Mar 2007 02:03:48 GMT

View Forum Message <> Reply to Message

On Sat, Mar 10, 2007 at 12:11:05AM +0530, Srivatsa Vaddagiri wrote:
> On Fri, Mar 09, 2007 at 02:16:08AM +0100, Herbert Poetzl wrote:
> > On Thu, Mar 08, 2007 at 05:00:54PM +0530, Srivatsa Vaddagiri wrote:
> > > On Thu, Mar 08, 2007 at 01:50:01PM +1300, Sam Vilain wrote:
> > > > 7. resource namespaces
> > >
> > > It should be. Imagine giving 20% bandwidth to a user X. X wants to
> > > divide this bandwidth further between multi-media (10%), kernel
> > > compilation (5%) and rest (5%). So,
> >
> > sounds quite nice, but ...
> >
> > > > Is the subservient namespace's resource usage counting against
> > > > ours too?
> > >
> > > Yes, the resource usage of children should be accounted when capping
> > > parent resource usage.
> >
> > it will require to do accounting many times
> > (and limit checks of course), which in itself
> > might be a way to DoS the kernel by creating
> > more and more resource groups
>
> I was only pointing out the usefullness of the feature and not
> necessarily saying it -should- be implemented! Ofcourse I understand it
> will make the controller complicated and thats why probably none of the
> recontrollers we are seeing posted on lkml don't support hierarchical
> res mgmt.
>

> > > > Can we dynamically alter the subservient namespace's resource
> > > > allocations?
> > >
> > > Should be possible yes. That lets user X completely manage his
> > > allocation among whatever sub-groups he creates.
> >
> > what happens if the parent changes, how is
> > the resource change (if it was a reduction)
> > propagated to the children?
> >
> > e.g. your guest has 1024 file handles, now
> > you reduce it to 512, but the guest had two
> > children, both with 256 file handles each ...
>
> I believe CKRM handled this quite neatly (by defining child shares to be
> relative to parent shares).
>
> In your example, 256+256 add up to 512 which is within the parent's
> new limit, so nothing happens :)

yes, but that might as well be fatal, because now the
children can easily DoS the parent by using up all the
file handles, where the 'original' setup (2 x 256)
left 512 file handles 'reserved' ...

of course, you could as well have adjusted that to
2 x 128 + 256 for the parent, but that is policy and
IMHO policy does not belong into the kernel, it should
be handled by userspace (maybe invoked by the kernel
in some kind of helper functionality or so)

> You also picked an example of exhaustible/non-reclaimable resource,
> which makes it hard to define what should happen if parent's limit
> goes below 512.

which was quite intentional, and brings us to another
issues when adjusting resource limits (not even in
a hierarchical way)

> Either nothing happens or perhaps a task is killed, don't know.

> In case of memory, I would say that some of child's pages may
> get kicked out and in case of cpu, child will start getting fewer
> cycles.

btw, kicking out pages when rss limit is reached might
be the obvious choice (if we think Virtual Machine here)
but it might not be the best choice from the overall

performance PoV, which might be much better off by
keeping the page in memory (if there is enough memory
available) but penalizing the guest like the page was
actually kicked out (and needs to be fetched later on)

note: this is something we should think about when we
want to address specific limits like RSS, because IMHO
we should not optimize for the single guest case, but
for the big picture ...

> > > The patches should give visibility to both nsproxy objects (by
> > > showing what tasks share the same nsproxy objects and letting
> > > tasks move across nsproxy objects if allowed) and the resource
> > > control objects pointed to by nsproxy (struct cpuset, struct
> > > cpu_limit, struct rss_limit etc).
> >
> > the nsproxy is not really relevant, as it
> > is some kind of strange indirection, which
> > does not necessarily depict the real relations,
> > regardless wether you do the re-sharing of
> > those nsproies or not ..
>
> So what are you recommending we do instead?
> My thought was whatever is the fundamental unit to which resource
> management needs to be applied, lets store resource parameters (or
> pointers to them) there (rather than duplicating the information in
> each task_struct).

we do not want to duplicate any information in the task
struct, but we might want to put some (or maybe all)
of the spaces back (as pointer reference) to the task
struct, just to avoid the nsproxy indirection

note that IMHO not all spaces make sense to be separated
e.g. while it is quite useful to have network and pid
space separated, others might be joined to form larger
consistant structures ...

for example, I could as well live with pid and resource
accounting/limits sharing one common struct/space ...
(doesn't mean that separate spaces are not nice :)

best,
Herbert

> --
> Regards,
> vatsa

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

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

Subject: Re: [ckrm-tech] [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Sat, 10 Mar 2007 03:19:09 GMT
View Forum Message <> Reply to Message

On Sat, Mar 10, 2007 at 07:32:20AM +0530, Srivatsa Vaddagiri wrote:
> Ok, let me see if I can convey what I had in mind better:

```
>
>     uts_ns pid_ns ipc_ns
>   \    |    /
>   ---------------
>        | nsproxy   |
>         ----------------
>                /  |   \    \ <-- 'nsproxy' pointer
>   T1  T2  T3 ...T1000
>   |   |   |      | <-- 'containers' pointer (4/8 KB for 1000 task)
>        -------------------
>       | container_group   |
>        ------------------
>   /
>      ----------
>     | container |
>      ----------
>   |
>      ----------
>     | cpu_limit |
>      ----------
```

[snip]

> We save on 4/8 KB (for 1000 tasks) by avoiding the 'containers' pointer
> in each task_struct (just to get to the resource limit information).

Having the 'containers' pointer in each task-struct is great from a non-container res mgmt perspective. It lets you dynamically decide what is the fundamental unit of res mgmt.

It could be {T1, T5} tasks/threads of a process, or {T1, T3, T8, T10} tasks of

a session (for limiting login time per session), or {T1, T2 ..T10, T18, T27}
tasks of a user etc.

But from a vserver/container pov, this level flexibility (at a -task- level) of
deciding the unit of res mgmt is IMHO not needed. The
vserver/container/namespace (tsk->nsproxy->some_ns) to which a task
belongs automatically defines that unit of res mgmt.

--
Regards,
vatsa

____

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Sam Vilain on Sat, 10 Mar 2007 08:52:35 GMT

View Forum Message <> Reply to Message

Paul Jackson wrote:
>> But "namespace" has well-established historical semantics too - a way
>> of changing the mappings of local * to global objects. This
>> accurately describes things liek resource controllers, cpusets, resource
>> monitoring, etc.
>>
>
> No!
>
> Cpusets don't rename or change the mapping of objects.
>
> I suspect you seriously misunderstand cpusets and are trying to cram them
> into a 'namespace' remapping role into which they don't fit.
>

Look, you're absolutely right, I'm stretching the terms much too far.

namespaces implies some kind of domain, which is the namespace, and
entities within the domain, which are the names, and there is a (task,
domain) mapping. I was thinking that this implies all similar (task,
domain) mappings could be treated in the same way. But when you apply
this to something like cpusets, it gets a little abstract. Like the
entities are (task,cpu) pairs and the domains the set of cpus that a
process can run on.

Sam.

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

Subject: Re: [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Paul Jackson on Sat, 10 Mar 2007 09:11:15 GMT

View Forum Message <> Reply to Message

Sam wrote:
> But when you apply this to something like cpusets, it gets a little abstract.

Just a tad abstract <grin>.

Thanks.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by serue on Sun, 11 Mar 2007 16:36:04 GMT

View Forum Message <> Reply to Message

Quoting Herbert Poetzl (herbert@13thfloor.at):
> On Fri, Mar 09, 2007 at 11:27:07PM +0530, Srivatsa Vaddagiri wrote:
> > On Fri, Mar 09, 2007 at 01:38:19AM +0100, Herbert Poetzl wrote:
> > > > 2) you allow a task to selectively reshare namespaces/subsystems with
> > > >   another task, i.e. you can update current->task_proxy to point to
> > > >   a proxy that matches your existing task_proxy in some ways and the
> > > >   task_proxy of your destination in others. In that case a trivial
> > > >   implementation would be to allocate a new task_proxy and copy some
> > > >   pointers from the old task_proxy and some from the new. But then
> > > >   whenever a task moves between different groupings it acquires a
> > > >   new unique task_proxy. So moving a bunch of tasks between two
> > > >   groupings, they'd all end up with unique task_proxy objects with
> > > >   identical contents.
>

> > > this is exactly what Linux-VServer does right now, and I'm
> > > still not convinced that the nsproxy really buys us anything
> > > compared to a number of different pointers to various spaces
> > > (located in the task struct)
>
> > Are you saying that the current scheme of storing pointers to
> > different spaces (uts_ns, ipc_ns etc) in nsproxy doesn't buy
> > anything?
>
> > Or are you referring to storage of pointers to resource
> > (name)spaces in nsproxy doesn't buy anything?
>
> > In either case, doesn't it buy speed and storage space?
>
> let's do a few examples here, just to illustrate the
> advantages and disadvantages of nsproxy as separate
> structure over nsproxy as part of the task_struct

But you're forgetting the *common* case, which is hundreds or thousands
of tasks with just one nsproxy. That's case for which we have to
optimize.

When that case is no longer the common case, we can yank the nsproxy.
As I keep saying, it *is* just an optimization.

-serge

> 1) typical setup, 100 guests as shell servers, 5
>   tasks each when unused, 10 tasks when used 10%
>   used in average
>
>   a) separate nsproxy, we need at least 100
>   structs to handle that (saves some space)
>
>   we might end up with ~500 nsproxies, if
>   the shell clones a new namespace (so might
>   not save that much space)
>
>   we do a single inc/dec when the nsproxy
>   is reused, but do the full N inc/dec when
>   we have to copy an nsproxy (might save
>   some refcounting)
>
>   we need to do the indirection step, from
>   task to nsproxy to space (and data)
>
>   b) we have ~600 tasks with 600 times the
>   nsproxy data (uses up some more space)

>
>   we have to do the full N inc/dev when
>   we create a new task (more refcounting)
>
>   we do not need to do the indirection, we
>   access spaces directly from the 'hot'
>   task struct (makes hot pathes quite fast)
>
>   so basically we trade a little more space and
>   overhead on task creation for having no
>   indirection to the data accessed quite often
>   throughout the tasks life (hopefully)
>
> 2) context migration: for whatever reason, we decide
>   to migrate a task into a subset (space mix) of a
>   context 1000 times
>
>   a) separate nsproxy, we need to create a new one
>   consisting of the 'new' mix, which will
>
>   - allocate the nsproxy struct
>   - inc refcounts to all copied spaces
>   - inc refcount nsproxy and assign to task
>   - dec refcount existing task nsproxy
>
>   after task completion
>   - dec nsproxy refcount
>   - dec refcounts for all spaces
>   - free up nsproxy struct
>
>   b) nsproxy data in task struct
>
>   - inc/dec refcounts to changed spaces
>
>   after task completion
>   - dec refcounts to spaces
>
>   so here we gain nothing with the nsproxy, unless
>   the chosen subset is identical to the one already
>   used, where we end up with a single refcount
>   instead of N
>
> > > I'd prefer to do accounting (and limits) in a very simple
> > > and especially performant way, and the reason for doing
> > > so is quite simple:
>
> > Can you elaborate on the relationship between data structures
> > used to store those limits to the task_struct?

>
> sure it is one to many, i.e. each task points to
> exactly one context struct, while a context can
> consist of zero, one or many tasks (no back-
> pointers there)
>
> > Does task_struct store pointers to those objects directly?
>
> it contains a single pointer to the context struct,
> and that contains (as a substruct) the accounting
> and limit information
>
> HTC,
> Herbert
>
> > --
> > Regards,
> > vatsa
> > ____
> > Containers mailing list
> > Containers@lists.osdl.org
> > https://lists.osdl.org/mailman/listinfo/containers
> ____
> Containers mailing list
> Containers@lists.osdl.org
> https://lists.osdl.org/mailman/listinfo/containers

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by dev on Sun, 11 Mar 2007 17:09:29 GMT

View Forum Message <> Reply to Message

Herbert,

> sorry, I'm not in the lucky position that I get payed
> for sending patches to LKML, so I have to think twice
> before I invest time in coding up extra patches ...
>
> i.e. you will have to live with my comments for now
looks like you have no better argurments then that...

>>Looks like your main argument is non-intrusive...
>>"working", "secure", "flexible" are not required to
>>people any more? :/

>
>
> well, Linux-VServer is "working", "secure", "flexible"
> _and_ non-intrusive ... it is quite natural that less
> won't work for me ... and regarding patches, there
> will be a 2.2 release soon, with all the patches ...

ok. please check your dcache and slab accounting then
(analyzed according to patch-2.6.20.1-vs2.3.0.11.diff):
Both are full of races and problems. Some of them:
1. Slabs allocated from interrupt context are charged to current context.
   So charged values contain arbitrary mess, since during interrupts
   context can be arbitrary.
2. Due to (1) I guess you do not make any limiting of slabs.
   So there are number of ways how to consume a lot of kernel
   memory from inside container and
   OOM killer will kill arbitrary tasks in case of memory-shortage after that.
   Don't think it is secure... real DoS.
3. Dcache accounting simply doesn't work, since
   charges/uncharges are done on current context (sic!!!), which is arbitrary.
   i.e. lookup can be done in VE context, while dcache shrink can be done
   from another context.
   So the whole problem with dcache DoS is not solved at all, it is just hard to trigger.
4. Dcache accounting is racy, since your checks look like:
```
if (atomic_read(de->d_count))
   charge();
```
   which obviously races with other dput()'s/lookups.
5. Dcache accounting can be hit if someone does `find /` inside container.
   After that it is impossible to open something new,
   since all the dentries for directories in dcache will have d_count > 0
   (due it's children).
   It is a BUG.
6. Counters can be non-zero on container stop due to all of the above.

There are more and more points which arise when such a non-intrusive
accounting is concerned. I'm really suprised, that you don't see them
or try to behave as you don't see them :/
And, please, believe me, I would not suggest so much complicated patches
If everything was so easy and I had no reasons simply to accept vserver code.

> well, as you know, all current solutions use a syscall
> interface to do most of the work, in the OpenVZ/Virtuozzo
> case several, unassigned syscalls are used, while
> FreeVPS and Linux-VServer use a registered and versioned
> (multiplexed) system call, which works quite fine for
> all known purposes ...
>
> I'm quite happy with the extensibility and flexibility
> the versioned syscall interface has, the only thing I'd

> change if I would redesign that interface is, that I
> would add another pointer argument to eliminate 32/64bit
> issues completely (i.e. use 4 args instead of the 3)
Well, I would be happy with syscalls also.
But my guess is that cpuset guys who already use fs approach won't be happy :/
Maybe we can use both?

Thanks,
Kirill

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

Subject: Re: [ckrm-tech] [PATCH 0/2] resource control file system - aka containers on top of nsproxy!
Posted by Srivatsa Vaddagiri on Mon, 12 Mar 2007 14:11:44 GMT
View Forum Message <> Reply to Message

On Wed, Mar 07, 2007 at 03:59:19PM -0600, Serge E. Hallyn wrote:
> > containers patches uses just a single pointer in the task_struct, and
> > all tasks in the same set of containers (across all hierarchies) will
> > share a single container_group object, which holds the actual pointers
> > to container state.
>
> Yes, that's why this consolidation doesn't make sense to me.
>
> Especially considering again that we will now have nsproxies pointing to
> containers pointing to... nsproxies.

nsproxies needn't point to containers. It (or as Herbert pointed -
nsproxy->pid_ns) can have direct pointers to resource objects (whatever
struct container->subsys[] points to).

--
Regards,
vatsa
____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [ckrm-tech] [PATCH 0/2] resource control file system - aka containers

on top of nsproxy!
Posted by Srivatsa Vaddagiri on Mon, 12 Mar 2007 15:07:56 GMT
View Forum Message <> Reply to Message

On Fri, Mar 09, 2007 at 02:09:35PM -0800, Paul Menage wrote:
> > 3. This next leads me to think that 'tasks' file in each directory doesnt make
> >   sense for containers. In fact it can lend itself to error situations (by
> >   administrator/script mistake) when some tasks of a container are in one
> >   resource class while others are in a different class.
> >
> >   Instead, from a containers pov, it may be usefull to write
> >   a 'container id' (if such a thing exists) into the tasks file
> >   which will move all the tasks of the container into
> >   the new resource class. This is the same requirement we
> >   discussed long back of moving all threads of a process into new
> >   resource class.
>
> I think you need to give a more concrete example and use case of what
> you're trying to propose here. I don't really see what advantage
> you're getting.

Ok, this is what I had in mind:

```
 mount -t container -o ns /dev/namespace
 mount -t container -o cpu /dev/cpu
```

Lets we have the namespaces/resource-groups created as under:

```
 /dev/namespace
      |-- prof
      |  |- tasks <- (T1, T2)
      |    |- container_id <- 1 (doesnt exist today perhaps)
      |
      |-- student
      |    |- tasks <- (T3, T4)
      |    |- container_id <- 2 (doesnt exist today perhaps)

 /dev/cpu
        |-- prof
        |    |-- tasks
        |    |-- cpu_limit (40%)
        |
        |-- student
        |    |-- tasks
        |    |-- cpu_limit (20%)
        |
        |
```

Is it possible to create the above structure in container patches?
/me thinks so.

If so, then accidentally someone can do this:

```
 echo T1 > /dev/cpu/prof/tasks
 echo T2 > /dev/cpu/student/tasks
```

with the result that tasks of the same container are now in different resource classes.

Thats why in case of containers I felt we shldnt allow individual tasks to be cat'ed to tasks file.

Or rather, it may be nice to say :

```
 echo "cid 2" > /dev/cpu/prof/tasks
```

and have all tasks belonging to container id 2 move to the new resource group.

--
Regards,
vatsa

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Mon, 12 Mar 2007 23:00:53 GMT

View Forum Message <> Reply to Message

On Sun, Mar 11, 2007 at 08:09:29PM +0300, Kirill Korotaev wrote:
> Herbert,
>
> > sorry, I'm not in the lucky position that I get payed
> > for sending patches to LKML, so I have to think twice
> > before I invest time in coding up extra patches ...
> >
> > i.e. you will have to live with my comments for now

> looks like you have no better argurments then that...

pardon?

if you want to make that personal, please do it
offline ... I'm sick of (lkml) folks wasting
time for (political) hick hack instead of trying
to improve the kernel ...

>>> Looks like your main argument is non-intrusive...
>>> "working", "secure", "flexible" are not required to
>>> people any more? :/

>> well, Linux-VServer is "working", "secure", "flexible"
>> _and_ non-intrusive ... it is quite natural that less
>> won't work for me ... and regarding patches, there
>> will be a 2.2 release soon, with all the patches ...

> ok. please check your dcache and slab accounting then
> (analyzed according to patch-2.6.20.1-vs2.3.0.11.diff):

development branch, good choice for new features
and code which is currently tested ...

> Both are full of races and problems. Some of them:
> 1. Slabs allocated from interrupt context are charged to
>   current context.
>   So charged values contain arbitrary mess, since during
>   interrupts context can be arbitrary.

> 2. Due to (1) I guess you do not make any limiting of slabs.
>   So there are number of ways how to consume a lot of kernel
>   memory from inside container and
>   OOM killer will kill arbitrary tasks in case of
>   memory-shortage after that.
>   Don't think it is secure... real DoS.

> 3. Dcache accounting simply doesn't work, since
>   charges/uncharges are done on current context (sic!!!),
>   which is arbitrary. i.e. lookup can be done in VE context,
>   while dcache shrink can be done from another context.
>   So the whole problem with dcache DoS is not solved at
>   all, it is just hard to trigger.

> 4. Dcache accounting is racy, since your checks look like:
>   if (atomic_read(de->d_count))
>   charge();
>   which obviously races with other dput()'s/lookups.

> 5. Dcache accounting can be hit if someone does `find /`

> inside container.
> After that it is impossible to open something new,
> since all the dentries for directories in dcache will
> have d_count > 0 (due it's children).
> It is a BUG.

> 6. Counters can be non-zero on container stop due to all
> of the above.

looks like for the the first time you are actually
looking at the code, or at least providing feedback
and/or suggestions for improvements (well, not many
of them, but hey, nobody is perfect :)

> There are more and more points which arise when such a
> non-intrusive accounting is concerned.

never claimed that Linux-VServer code is perfect,
(the Linux accounting isn't perfect either in many
ways) and Linux-VServer is constantly improving
(see my other email) ... but IIRC, we are _not_
discussing Linux-VServer code at all, we are talking
about a superior solution, which combines the best
of both worlds ...

> I'm really suprised, that you don't see them
> or try to behave as you don't see them :/

all I'm saying is that there is no point in achieving
perfect accounting and limits (and everything else)
when all you get is Xen performance and resource usage

> And, please, believe me, I would not suggest so much
> complicated patches If everything was so easy and I
> had no reasons simply to accept vserver code.

no, you are suggesting those patches, because that
is what your company came up with after being confronted
with the task (of creating OS-Level virtualization) and
the arising problems ... so it definitely _is_ a
solution to those problems, but not necessarily the
best and definitely not the only one :)

> > well, as you know, all current solutions use a syscall
> > interface to do most of the work, in the OpenVZ/Virtuozzo
> > case several, unassigned syscalls are used, while
> > FreeVPS and Linux-VServer use a registered and versioned
> > (multiplexed) system call, which works quite fine for

> > all known purposes ...
> >
> > I'm quite happy with the extensibility and flexibility
> > the versioned syscall interface has, the only thing I'd
> > change if I would redesign that interface is, that I
> > would add another pointer argument to eliminate 32/64bit
> > issues completely (i.e. use 4 args instead of the 3)

> Well, I would be happy with syscalls also.

> But my guess is that cpuset guys who already use fs
> approach won't be happy :/

> Maybe we can use both?

I'm fine with either here, though my preference is
for syscalls (and we will probably keep the versioned
syscall commands for Linux-VServer anyway)

best,
Herbert

> Thanks,
> Kirill

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Mon, 12 Mar 2007 23:16:18 GMT

View Forum Message <> Reply to Message

On Sun, Mar 11, 2007 at 11:36:04AM -0500, Serge E. Hallyn wrote:
> Quoting Herbert Poetzl (herbert@13thfloor.at):
> > On Fri, Mar 09, 2007 at 11:27:07PM +0530, Srivatsa Vaddagiri wrote:
> > > On Fri, Mar 09, 2007 at 01:38:19AM +0100, Herbert Poetzl wrote:
> > > > > > 2) you allow a task to selectively reshare namespaces/subsystems with
> > > > >   another task, i.e. you can update current->task_proxy to point to
> > > > >   a proxy that matches your existing task_proxy in some ways and the
> > > > >   task_proxy of your destination in others. In that case a trivial
> > > > >   implementation would be to allocate a new task_proxy and copy some
> > > > >   pointers from the old task_proxy and some from the new. But then
> > > > >   whenever a task moves between different groupings it acquires a
> > > > >   new unique task_proxy. So moving a bunch of tasks between two
> > > > >   groupings, they'd all end up with unique task_proxy objects with
> > > > >   identical contents.

> >
> > > > this is exactly what Linux-VServer does right now, and I'm
> > > > still not convinced that the nsproxy really buys us anything
> > > > compared to a number of different pointers to various spaces
> > > > (located in the task struct)
> >
> > > Are you saying that the current scheme of storing pointers to
> > > different spaces (uts_ns, ipc_ns etc) in nsproxy doesn't buy
> > > anything?
> >
> > > Or are you referring to storage of pointers to resource
> > > (name)spaces in nsproxy doesn't buy anything?
> >
> > > In either case, doesn't it buy speed and storage space?
> >
> > let's do a few examples here, just to illustrate the
> > advantages and disadvantages of nsproxy as separate
> > structure over nsproxy as part of the task_struct
>
> But you're forgetting the *common* case, which is hundreds or
> thousands of tasks with just one nsproxy. That's case for
> which we have to optimize.

yes, I agree here, maybe we should do something
I suggested (and submitted a patch for some time
ago) and add some kind of accounting for the various
spaces (and the nsproxy) so that we can get a feeling
how many of them are there and how many create/destroy
cycles really happen ...

those things will definitely be accounted in the
Linux-VServer devel versions, don't know about OVZ

> When that case is no longer the common case, we can yank the
> nsproxy. As I keep saying, it *is* just an optimization.

yes, fine with me, just wanted to paint a picture ...

best,
Herbert

> -serge
>
> > 1) typical setup, 100 guests as shell servers, 5
> >   tasks each when unused, 10 tasks when used 10%
> >   used in average
> >
> >   a) separate nsproxy, we need at least 100

> >   structs to handle that (saves some space)
> >
> >   we might end up with ~500 nsproxies, if
> >   the shell clones a new namespace (so might
> >   not save that much space)
> >
> >   we do a single inc/dec when the nsproxy
> >   is reused, but do the full N inc/dec when
> >   we have to copy an nsproxy (might save
> >   some refcounting)
> >
> >   we need to do the indirection step, from
> >   task to nsproxy to space (and data)
> >
> >   b) we have ~600 tasks with 600 times the
> >   nsproxy data (uses up some more space)
> >
> >   we have to do the full N inc/dev when
> >   we create a new task (more refcounting)
> >
> >   we do not need to do the indirection, we
> >   access spaces directly from the 'hot'
> >   task struct (makes hot pathes quite fast)
> >
> >   so basically we trade a little more space and
> >   overhead on task creation for having no
> >   indirection to the data accessed quite often
> >   throughout the tasks life (hopefully)
> >
> > 2) context migration: for whatever reason, we decide
> >   to migrate a task into a subset (space mix) of a
> >   context 1000 times
> >
> >   a) separate nsproxy, we need to create a new one
> >   consisting of the 'new' mix, which will
> >
> >   - allocate the nsproxy struct
> >   - inc refcounts to all copied spaces
> >   - inc refcount nsproxy and assign to task
> >   - dec refcount existing task nsproxy
> >
> >   after task completion
> >   - dec nsproxy refcount
> >   - dec refcounts for all spaces
> >   - free up nsproxy struct
> >
> >   b) nsproxy data in task struct
> >

> >   - inc/dec refcounts to changed spaces
> >
> >   after task completion
> >   - dec refcounts to spaces
> >
> >   so here we gain nothing with the nsproxy, unless
> >   the chosen subset is identical to the one already
> >   used, where we end up with a single refcount
> >   instead of N
> >
> > > > I'd prefer to do accounting (and limits) in a very simple
> > > > and especially performant way, and the reason for doing
> > > > so is quite simple:
> >
> > > Can you elaborate on the relationship between data structures
> > > used to store those limits to the task_struct?
> >
> > sure it is one to many, i.e. each task points to
> > exactly one context struct, while a context can
> > consist of zero, one or many tasks (no back-
> > pointers there)
> >
> > > Does task_struct store pointers to those objects directly?
> >
> > it contains a single pointer to the context struct,
> > and that contains (as a substruct) the accounting
> > and limit information
> >
> > HTC,
> > Herbert
> >
> > > --
> > > Regards,
> > > vatsa
> > > ____
> > > Containers mailing list
> > > Containers@lists.osdl.org
> > > https://lists.osdl.org/mailman/listinfo/containers
> > ____
> > Containers mailing list
> > Containers@lists.osdl.org
> > https://lists.osdl.org/mailman/listinfo/containers
> ____
> Containers mailing list
> Containers@lists.osdl.org
> https://lists.osdl.org/mailman/listinfo/containers

____
Containers mailing list

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by dev on Tue, 13 Mar 2007 08:28:06 GMT
View Forum Message <> Reply to Message

```
>>>well, Linux-VServer is "working", "secure", "flexible"
>>>_and_ non-intrusive ... it is quite natural that less
>>>won't work for me ... and regarding patches, there
>>>will be a 2.2 release soon, with all the patches ...
>
>
>>ok. please check your dcache and slab accounting then
>>(analyzed according to patch-2.6.20.1-vs2.3.0.11.diff):
>
>
> development branch, good choice for new features
> and code which is currently tested ...
```

you know better than I that stable branch doesn't differ much,
especially in securiy (because it lacks these controls at all).

BTW, killing arbitrary task in case of RSS limit hit
doesn't look acceptable resource management approach, does it?

```
>>Both are full of races and problems. Some of them:
>>1. Slabs allocated from interrupt context are charged to
>>   current context.
>>   So charged values contain arbitrary mess, since during
>>   interrupts context can be arbitrary.
>
>
>>2. Due to (1) I guess you do not make any limiting of slabs.
>>   So there are number of ways how to consume a lot of kernel
>>   memory from inside container and
>>   OOM killer will kill arbitrary tasks in case of
>>   memory-shortage after that.
>>   Don't think it is secure... real DoS.
>
>
>>3. Dcache accounting simply doesn't work, since
>>   charges/uncharges are done on current context (sic!!!),
>>   which is arbitrary. i.e. lookup can be done in VE context,
>>   while dcache shrink can be done from another context.
>>   So the whole problem with dcache DoS is not solved at
>>   all, it is just hard to trigger.
>
```

>
>>4. Dcache accounting is racy, since your checks look like:
>>   if (atomic_read(de->d_count))
>>   charge();
>>   which obviously races with other dput()'s/lookups.
>
>
>>5. Dcache accounting can be hit if someone does `find /`
>>   inside container.
>>   After that it is impossible to open something new,
>>   since all the dentries for directories in dcache will
>>   have d_count > 0 (due it's children).
>>   It is a BUG.
>
>
>>6. Counters can be non-zero on container stop due to all
>>   of the above.
>
>
> looks like for the the first time you are actually
> looking at the code, or at least providing feedback
> and/or suggestions for improvements (well, not many
> of them, but hey, nobody is perfect :)
It's a pity, but it took me only 5 minutes of looking into the code,
so "not perfect" is a wrong word here, sorry.

>>There are more and more points which arise when such a
>>non-intrusive accounting is concerned.
>
>
> never claimed that Linux-VServer code is perfect,
> (the Linux accounting isn't perfect either in many
> ways) and Linux-VServer is constantly improving
> (see my other email) ... but IIRC, we are _not_
> discussing Linux-VServer code at all, we are talking
> about a superior solution, which combines the best
> of both worlds ...
Forget about Vserver and OpenVZ. It is not a war.
We are looking for something working, new and robust.
I'm just trying you to show that non-intrusive and pretty small
accounting/limiting code like in Vserver
simply doesn't work. The problem of resource controls is much more complicated.
So non-intrusiveness is a very weird argument from you (and the only).

>>I'm really suprised, that you don't see them
>>or try to behave as you don't see them :/
>
>

> all I'm saying is that there is no point in achieving
> perfect accounting and limits (and everything else)
> when all you get is Xen performance and resource usage
then please elaborate on what you mean by
perfect and non-perfect accounting and limits?
I would be happy to sent a patch with a "non-perfect"
accounting if it really works correct and good and suits all the people needs.

BTW, Xen overhead comes mostly from different things (not resource management) -
inability to share data effectively, emulation overhead etc.

>>And, please, believe me, I would not suggest so much
>>complicated patches If everything was so easy and I
>>had no reasons simply to accept vserver code.
>
>
> no, you are suggesting those patches, because that
> is what your company came up with after being confronted
> with the task (of creating OS-Level virtualization) and
> the arising problems ... so it definitely _is_ a
> solution to those problems, but not necessarily the
> best and definitely not the only one :)
You judge so because you want to.
Have you had some time to compare UBC patches from OVZ
and those sent to LKML (container + RSS)?
You would notice too litle in common.
Patches in LKML has non-OVZ interfaces, no shared pages accounting,
RSS accounting which is not used in OVZ at all.
So do you see any similarities except for stupid and simple
controls like numtask/numfile?

Thanks,
Kirill

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

Subject: Re: [PATCH 1/2] rcfs core patch
Posted by Herbert Poetzl on Tue, 13 Mar 2007 13:55:05 GMT

View Forum Message <> Reply to Message

On Tue, Mar 13, 2007 at 11:28:06AM +0300, Kirill Korotaev wrote:
> >>>well, Linux-VServer is "working", "secure", "flexible"
> >>>_and_ non-intrusive ... it is quite natural that less
> >>>won't work for me ... and regarding patches, there

> >>>will be a 2.2 release soon, with all the patches ...

first, fix your mail client to get the quoting right,
it is quite unreadable the way it is (not the first
time I tell you that)

> >>ok. please check your dcache and slab accounting then
> >>(analyzed according to patch-2.6.20.1-vs2.3.0.11.diff):
> >
> >
> > development branch, good choice for new features
> > and code which is currently tested ...
> you know better than I that stable branch doesn't differ much,
> especially in securiy (because it lacks these controls at all).
>
> BTW, killing arbitrary task in case of RSS limit hit
> doesn't look acceptable resource management approach, does it?
>
> >>Both are full of races and problems. Some of them:
> >>1. Slabs allocated from interrupt context are charged to
> >>   current context.
> >>   So charged values contain arbitrary mess, since during
> >>   interrupts context can be arbitrary.
> >
> >
> >>2. Due to (1) I guess you do not make any limiting of slabs.
> >>   So there are number of ways how to consume a lot of kernel
> >>   memory from inside container and
> >>   OOM killer will kill arbitrary tasks in case of
> >>   memory-shortage after that.
> >>   Don't think it is secure... real DoS.
> >
> >
> >>3. Dcache accounting simply doesn't work, since
> >>   charges/uncharges are done on current context (sic!!!),
> >>   which is arbitrary. i.e. lookup can be done in VE context,
> >>   while dcache shrink can be done from another context.
> >>   So the whole problem with dcache DoS is not solved at
> >>   all, it is just hard to trigger.
> >
> >
> >>4. Dcache accounting is racy, since your checks look like:
> >>   if (atomic_read(de->d_count))
> >>   charge();
> >>   which obviously races with other dput()'s/lookups.
> >
> >
> >>5. Dcache accounting can be hit if someone does `find /`

> >>   inside container.
> >>   After that it is impossible to open something new,
> >>   since all the dentries for directories in dcache will
> >>   have d_count > 0 (due it's children).
> >>   It is a BUG.
> >
> >
> >>6. Counters can be non-zero on container stop due to all
> >>   of the above.
> >
> >
> > looks like for the the first time you are actually
> > looking at the code, or at least providing feedback
> > and/or suggestions for improvements (well, not many
> > of them, but hey, nobody is perfect :)
> It's a pity, but it took me only 5 minutes of looking into the code,
> so "not perfect" is a wrong word here, sorry.

see how readable and easily understandable the code is?
it takes me several hours to read OpenVZ code, and that's
not just me :)

> >>There are more and more points which arise when such a
> >>non-intrusive accounting is concerned.
> >
> >
> > never claimed that Linux-VServer code is perfect,
> > (the Linux accounting isn't perfect either in many
> > ways) and Linux-VServer is constantly improving
> > (see my other email) ... but IIRC, we are _not_
> > discussing Linux-VServer code at all, we are talking
> > about a superior solution, which combines the best
> > of both worlds ...
> Forget about Vserver and OpenVZ. It is not a war.
> We are looking for something working, new and robust.

you forgot efficient and performant here ...

> I'm just trying you to show that non-intrusive and pretty small
> accounting/limiting code like in Vserver simply doesn't work.

simply doesn't work?
because you didn't try to make it work?
because you didn't succeed in making it work?

> The problem of resource controls is much more complicated.
> So non-intrusiveness is a very weird argument from you
> (and the only).

no comment, read my previous emails ...

> >>I'm really suprised, that you don't see them
> >>or try to behave as you don't see them :/
> >
> >
> > all I'm saying is that there is no point in achieving
> > perfect accounting and limits (and everything else)
> > when all you get is Xen performance and resource usage
> then please elaborate on what you mean by
> perfect and non-perfect accounting and limits?

as we are discussing RSS limits, there are actually three different (existing) approaches we have talked about:

- 'the 'perfect RAM counter'
  each page is accounted exactly once, when used in a guest, regardless of how many times it is shared between different guest tasks

- the 'RSS sum' approach
  each page is accounted for every task mapping it (will account shared pages inside a guest several times and doesn't reflect the actual RAM usage)

- the 'first user owns' approach
  each page, when mapped the first time, gets accounted to the guest who mapped it, regardless of the fact that it might be shared with other guests lateron

the first one is 'perfect' IMHO, while all three are 'consistant' if done properly, although they will show quite different results and require different limit settings ...

> I would be happy to sent a patch with a "non-perfect"
> accounting if it really works correct and good and suits
> all the people needs.

good, but what you currently do is providing 'your' implementation with 'your' design and approach, which _doesn't_ really suit _my_ needs ...

> BTW, Xen overhead comes mostly from different things
> (not resource management) - inability to share data
> effectively, emulation overhead etc.

no comment ...

> >>And, please, believe me, I would not suggest so much
> >>complicated patches If everything was so easy and I
> >>had no reasons simply to accept vserver code.
> >
> >
> > no, you are suggesting those patches, because that
> > is what your company came up with after being confronted
> > with the task (of creating OS-Level virtualization) and
> > the arising problems ... so it definitely _is_ a
> > solution to those problems, but not necessarily the
> > best and definitely not the only one :)
> You judge so because you want to.
> Have you had some time to compare UBC patches from OVZ
> and those sent to LKML (container + RSS)?
> You would notice too litle in common.
> Patches in LKML has non-OVZ interfaces, no shared pages accounting,
> RSS accounting which is not used in OVZ at all.
> So do you see any similarities except for stupid and simple
> controls like numtask/numfile?

yes, tons of locking, complicated indirections and
a lot of (partially hard to understand) code ...

best,
Herbert

> Thanks,
> Kirill

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

Subject: Re: [ckrm-tech] [PATCH 1/2] rcfs core patch
Posted by Srivatsa Vaddagiri on Tue, 13 Mar 2007 14:11:37 GMT

View Forum Message <> Reply to Message

On Tue, Mar 13, 2007 at 02:55:05PM +0100, Herbert Poetzl wrote:
> yes, tons of locking, complicated indirections and
> a lot of (partially hard to understand) code ...

Are you referring to these issues in the general Paul Menage's container code
or in the RSS-control code posted by Pavel?

--
Regards,
vatsa

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