Subject: [PATCH 0/7] Generic Process Containers (+ ResGroups/BeanCounters)
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:48 GMT
View Forum Message <> Reply to Message

This is an update to my multi-hierarchy generic containers patch (against 2.6.19-rc6). Changes include:

- an example patch implementing the BeanCounters core and numfiles counters over generic containers. The addition of the BeanCounters code unifies the three main process grouping abstractions (Cpusets, ResGroups and BeanCounters).

- a patch splitting Cpusets into two independently groupable subsystems, Cpusets and Memsets.

- support for a subsystem to keep a container alive via refcounts (e.g. the BeanCounters numfiles counter has a reference to the beancounter object from each file charged to that beancounter, so needs to be able to keep the beancounter alive until the file is destroyed)

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

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

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

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

The change is implemented in five stages plus two additional example patches:

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

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

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

4) add a simple CPU accounting container subsystem as an example

5) example of implementing ResGroups and its numtasks controller over
   generic containers - not intended to be applied with this patch set

6) split cpusets into two subsystems, cpusets and memsets

7) example of implementing BeanCounters and its numfiles counter over
   generic containers - not intended to be applied with this patch set

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

- the userspace APIs are (somewhat) normalised

- it's easier to test out e.g. the ResGroups CPU controller in
 conjunction with the BeanCounters memory controller

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

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

--

Subject: [PATCH 1/7] Generic container system abstracted from cpusets code
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:49 GMT
View Forum Message <> Reply to Message

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

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

 Documentation/containers.txt | 229 +++++++
 fs/proc/base.c | 11
 include/linux/container.h | 96 +++

```
 include/linux/sched.h        |    5
 init/Kconfig                 |    9
 init/main.c                  |    3
 kernel/Makefile              |    1
 kernel/container.c           | 1343 ++++++++++++++++++++++++++++++++++++++++++++++
 kernel/exit.c                |    2
 kernel/fork.c                |    3
 10 files changed, 1699 insertions(+), 3 deletions(-)

Index: container-2.6.19-rc5/fs/proc/base.c
 ===================================================================
--- container-2.6.19-rc5.orig/fs/proc/base.c
+++ container-2.6.19-rc5/fs/proc/base.c
@@ -68,6 +68,7 @@
 #include <linux/security.h>
 #include <linux/ptrace.h>
 #include <linux/seccomp.h>
+#include <linux/container.h>
 #include <linux/cpuset.h>
 #include <linux/audit.h>
 #include <linux/poll.h>
@@ -1782,7 +1783,10 @@ static struct pid_entry tgid_base_stuff[
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat",  S_IRUGO, pid_schedstat),
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_CONTAINERS
+ REG("container",  S_IRUGO, container),
+#endif
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",     S_IRUGO, cpuset),
 #endif
  INF("oom_score",  S_IRUGO, oom_score),
@@ -2056,7 +2060,10 @@ static struct pid_entry tid_base_stuff[]
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat", S_IRUGO, pid_schedstat),
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_CONTAINERS
+ REG("container",  S_IRUGO, container),
+#endif
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",    S_IRUGO, cpuset),
 #endif
  INF("oom_score", S_IRUGO, oom_score),
Index: container-2.6.19-rc5/include/linux/container.h
 ===================================================================
--- /dev/null
```

```
+++ container-2.6.19-rc5/include/linux/container.h
@@ -0,0 +1,96 @@
+#ifndef _LINUX_CONTAINER_H
+#define _LINUX_CONTAINER_H
+/*
+ *  container interface
+ *
+ *  Copyright (C) 2003 BULL SA
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ */
+
+#include <linux/sched.h>
+#include <linux/cpumask.h>
+#include <linux/nodemask.h>
+
+#ifdef CONFIG_CONTAINERS
+
+extern int number_of_containers; /* How many containers are defined in system? */
+
+extern int container_init_early(void);
+extern int container_init(void);
+extern void container_init_smp(void);
+extern void container_fork(struct task_struct *p);
+extern void container_exit(struct task_struct *p);
+
+extern struct file_operations proc_container_operations;
+
+extern void container_lock(void);
+extern void container_unlock(void);
+
+extern void container_manage_lock(void);
+extern void container_manage_unlock(void);
+
+struct container {
+ unsigned long flags;  /* "unsigned long" so bitops work */
+
+ /*
+  * Count is atomic so can incr (fork) or decr (exit) without a lock.
+  */
+ atomic_t count;   /* count tasks using this container */
+
+ /*
+  * We link our 'sibling' struct into our parent's 'children'.
+  * Our children link their 'sibling' into our 'children'.
+  */
+ struct list_head sibling; /* my parent's children */
+ struct list_head children; /* my children */
```

```
+
+ struct container *parent; /* my parent */
+ struct dentry *dentry;  /* container fs entry */
+};
+
+/* struct cftype:
+ *
+ * The files in the container filesystem mostly have a very simple read/write
+ * handling, some common function will take care of it. Nevertheless some cases
+ * (read tasks) are special and therefore I define this structure for every
+ * kind of file.
+ *
+ *
+ * When reading/writing to a file:
+ * - the container to use in file->f_dentry->d_parent->d_fsdata
+ * - the 'cftype' of the file is file->f_dentry->d_fsdata
+ */
+
+struct inode;
+struct cftype {
+ char *name;
+ int private;
+ int (*open) (struct inode *inode, struct file *file);
+ ssize_t (*read) (struct container *cont, struct cftype *cft,
+    struct file *file,
+    char __user *buf, size_t nbytes, loff_t *ppos);
+ ssize_t (*write) (struct container *cont, struct cftype *cft,
+     struct file *file,
+     const char __user *buf, size_t nbytes, loff_t *ppos);
+ int (*release) (struct inode *inode, struct file *file);
+};
+
+int container_add_file(struct container *cont, const struct cftype *cft);
+
+int container_is_removed(const struct container *cont);
+
+#else /* !CONFIG_CONTAINERS */
+
+static inline int container_init_early(void) { return 0; }
+static inline int container_init(void) { return 0; }
+static inline void container_init_smp(void) {}
+static inline void container_fork(struct task_struct *p) {}
+static inline void container_exit(struct task_struct *p) {}
+
+static inline void container_lock(void) {}
+static inline void container_unlock(void) {}
+
+#endif /* !CONFIG_CONTAINERS */
```

```
+
+#endif /* _LINUX_CONTAINER_H */
Index: container-2.6.19-rc5/include/linux/sched.h
 ===================================================== =======
--- container-2.6.19-rc5.orig/include/linux/sched.h
+++ container-2.6.19-rc5/include/linux/sched.h
@@ -719,8 +719,8 @@ extern unsigned int max_cache_size;


 struct io_context;   /* See blkdev.h */
+struct container;
 struct cpuset;
-
 #define NGROUPS_SMALL  32
 #define NGROUPS_PER_BLOCK ((int)(PAGE_SIZE / sizeof(gid_t)))
 struct group_info {
@@ -1006,6 +1006,9 @@ struct task_struct {
  int cpuset_mems_generation;
  int cpuset_mem_spread_rotor;
 #endif
+#ifdef CONFIG_CONTAINERS
+ struct container *container;
+#endif
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
  struct compat_robust_list_head __user *compat_robust_list;
Index: container-2.6.19-rc5/init/Kconfig
 ===================================================== =======
--- container-2.6.19-rc5.orig/init/Kconfig
+++ container-2.6.19-rc5/init/Kconfig
@@ -238,6 +238,15 @@ config IKCONFIG_PROC
    This option enables access to the kernel configuration file
    through /proc/config.gz.

+config CONTAINERS
+ bool "Container support"
+ help
+   This option will let you create and manage process containers,
+   which can be used to aggregate multiple processes, e.g. for
+   the purposes of resource tracking.
+
+   Say N if unsure
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
Index: container-2.6.19-rc5/init/main.c
 ===================================================== =======
```

```
--- container-2.6.19-rc5.orig/init/main.c
+++ container-2.6.19-rc5/init/main.c
@@ -38,6 +38,7 @@
 #include <linux/writeback.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/efi.h>
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
@@ -568,6 +569,7 @@ asmlinkage void __init start_kernel(void
  }
 #endif
  vfs_caches_init_early();
+ container_init_early();
  cpuset_init_early();
  mem_init();
  kmem_cache_init();
@@ -598,6 +600,7 @@ asmlinkage void __init start_kernel(void
 #ifdef CONFIG_PROC_FS
  proc_root_init();
 #endif
+ container_init();
  cpuset_init();
  taskstats_init_early();
  delayacct_init();
Index: container-2.6.19-rc5/kernel/container.c
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc5/kernel/container.c
@@ -0,0 +1,1343 @@
+/*
+ *  kernel/container.c
+ *
+ *  Generic process-grouping system.
+ *
+ *  Based originally on the cpuset system, extracted by Paul Menage
+ *  Copyright (C) 2006 Google, Inc
+ *
+ *  Copyright notices from the original cpuset code:
+ *  --------------------------------------------------
+ *  Copyright (C) 2003 BULL SA.
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ *  Portions derived from Patrick Mochel's sysfs code.
+ *  sysfs is Copyright (c) 2001-3 Patrick Mochel
+ *
+ *  2003-10-10 Written by Simon Derr.
```

```
+ *  2003-10-22 Updates by Stephen Hemminger.
+ *  2004 May-July Rework by Paul Jackson.
+ *  ---------------------------------------------------
+ *
+ *  This file is subject to the terms and conditions of the GNU General Public
+ *  License.  See the file COPYING in the main directory of the Linux
+ *  distribution for more details.
+ */
+
+#include <linux/cpu.h>
+#include <linux/cpumask.h>
+#include <linux/container.h>
+#include <linux/err.h>
+#include <linux/errno.h>
+#include <linux/file.h>
+#include <linux/fs.h>
+#include <linux/init.h>
+#include <linux/interrupt.h>
+#include <linux/kernel.h>
+#include <linux/kmod.h>
+#include <linux/list.h>
+#include <linux/mempolicy.h>
+#include <linux/mm.h>
+#include <linux/module.h>
+#include <linux/mount.h>
+#include <linux/namei.h>
+#include <linux/pagemap.h>
+#include <linux/proc_fs.h>
+#include <linux/rcupdate.h>
+#include <linux/sched.h>
+#include <linux/seq_file.h>
+#include <linux/security.h>
+#include <linux/slab.h>
+#include <linux/smp_lock.h>
+#include <linux/spinlock.h>
+#include <linux/stat.h>
+#include <linux/string.h>
+#include <linux/time.h>
+#include <linux/backing-dev.h>
+#include <linux/sort.h>
+
+#include <asm/uaccess.h>
+#include <asm/atomic.h>
+#include <linux/mutex.h>
+
+#define CONTAINER_SUPER_MAGIC  0x27e0eb
+
+/*
```

```
+ * Tracks how many containers are currently defined in system.
+ * When there is only one container (the root container) we can
+ * short circuit some hooks.
+ */
+int number_of_containers __read_mostly;
+
+/* bits in struct container flags field */
+typedef enum {
+ CONT_REMOVED,
+ CONT_NOTIFY_ON_RELEASE,
+} container_flagbits_t;
+
+/* convenient tests for these bits */
+inline int container_is_removed(const struct container *cont)
+{
+ return test_bit(CONT_REMOVED, &cont->flags);
+}
+
+static inline int notify_on_release(const struct container *cont)
+{
+ return test_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+}
+
+static struct container top_container = {
+ .count = ATOMIC_INIT(0),
+ .sibling = LIST_HEAD_INIT(top_container.sibling),
+ .children = LIST_HEAD_INIT(top_container.children),
+};
+
+static struct vfsmount *container_mount;
+static struct super_block *container_sb;
+
+/*
+ * We have two global container 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 container pointer.
+ * See "The task_lock() exception", at the end of this comment.
+ *
+ * A task must hold both mutexes to modify containers.  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 containers.  It can perform various checks on
+ * the container 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 containers.  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 container code from within
+ * __alloc_pages().
+ *
+ * If a task is only holding callback_mutex, then it has read-only
+ * access to containers.
+ *
+ * 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 container 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 container with zero count, it
+ * knows that the container won't be removed, as container_rmdir() needs
+ * both of those mutexes.
+ *
+ * The container_common_file_write handler for operations that modify
+ * the container hierarchy holds manage_mutex across the entire operation,
+ * single threading all such container modifications across the system.
+ *
+ * The container_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 container_fork() and container_exit(), don't
+ * (usually) take either mutex.  These are the two most performance
+ * critical pieces of code here.  The exception occurs on container_exit(),
+ * when a task in a notify_on_release container exits.  Then manage_mutex
+ * is taken, and if the container count is zero, a usermode call made
+ * to /sbin/container_release_agent with the name of the container (path
+ * relative to the root of container file system) as the argument.
+ *
+ * A container can only be deleted if both its 'count' of using tasks
+ * is zero, and its list of 'children' containers is empty.  Since all
+ * tasks in the system use _some_ container, and since there is always at
+ * least one task in the system (init, pid == 1), therefore, top_container
+ * always has either children containers and/or using tasks.  So we don't
+ * need a special hack to ensure that top_container cannot be deleted.
+ *
+ * The 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 container pointer with another.  It does
+ * so using both mutexes, however there are several performance
+ * critical places that need to reference task->container without the
+ * expense of grabbing a system global mutex.  Therefore except as
+ * noted below, when dereferencing or, as in attach_task(), modifying
+ * a tasks container pointer we use task_lock(), which acts on a spinlock
+ * (task->alloc_lock) already in the task_struct routinely used for
+ * such matters.
+ *
+ * P.S.  One more locking exception.  RCU is used to guard the
+ * update of a tasks container pointer by attach_task() and the
+ * access of task->container->mems_generation via that pointer in
+ * the routine container_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:
+ *  container_mkdir -> container_create -> container_populate_dir -> container_add_file
+ *  -> container_create_file -> container_dir_inode_operations -> container_mkdir.
+ */
+
+static int container_mkdir(struct inode *dir, struct dentry *dentry, int mode);
+static int container_rmdir(struct inode *unused_dir, struct dentry *dentry);
+
+static struct backing_dev_info container_backing_dev_info = {
+ .ra_pages = 0,  /* No readahead */
+ .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
+};
+
+static struct inode *container_new_inode(mode_t mode)
+{
+ struct inode *inode = new_inode(container_sb);
+
+ if (inode) {
+  inode->i_mode = mode;
+  inode->i_uid = current->fsuid;
+  inode->i_gid = current->fsgid;
+  inode->i_blocks = 0;
+  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
+  inode->i_mapping->backing_dev_info = &container_backing_dev_info;
+ }
```

```
+ return inode;
+}
+
+static void container_diput(struct dentry *dentry, struct inode *inode)
+{
+ /* is dentry a directory ? if so, kfree() associated container */
+ if (S_ISDIR(inode->i_mode)) {
+  struct container *cont = dentry->d_fsdata;
+  BUG_ON(!(container_is_removed(cont)));
+  kfree(cont);
+ }
+ iput(inode);
+}
+
+static struct dentry_operations container_dops = {
+ .d_iput = container_diput,
+};
+
+static struct dentry *container_get_dentry(struct dentry *parent, const char *name)
+{
+ struct dentry *d = lookup_one_len(name, parent, strlen(name));
+ if (!IS_ERR(d))
+  d->d_op = &container_dops;
+ return d;
+}
+
+static void remove_dir(struct dentry *d)
+{
+ struct dentry *parent = dget(d->d_parent);
+
+ d_delete(d);
+ simple_rmdir(parent->d_inode, d);
+ dput(parent);
+}
+
+/*
+ * NOTE : the dentry must have been dget()'ed
+ */
+static void container_d_remove_dir(struct dentry *dentry)
+{
+ struct list_head *node;
+
+ spin_lock(&dcache_lock);
+ node = dentry->d_subdirs.next;
+ while (node != &dentry->d_subdirs) {
+  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
+  list_del_init(node);
+  if (d->d_inode) {
```

```
+   d = dget_locked(d);
+   spin_unlock(&dcache_lock);
+   d_delete(d);
+   simple_unlink(dentry->d_inode, d);
+   dput(d);
+   spin_lock(&dcache_lock);
+  }
+  node = dentry->d_subdirs.next;
+ }
+ list_del_init(&dentry->d_u.d_child);
+ spin_unlock(&dcache_lock);
+ remove_dir(dentry);
+}
+
+static struct super_operations container_ops = {
+ .statfs = simple_statfs,
+ .drop_inode = generic_delete_inode,
+};
+
+static int container_fill_super(struct super_block *sb, void *unused_data,
+       int unused_silent)
+{
+ struct inode *inode;
+ struct dentry *root;
+
+ sb->s_blocksize = PAGE_CACHE_SIZE;
+ sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
+ sb->s_magic = CONTAINER_SUPER_MAGIC;
+ sb->s_op = &container_ops;
+ container_sb = sb;
+
+ inode = container_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR);
+ if (inode) {
+  inode->i_op = &simple_dir_inode_operations;
+  inode->i_fop = &simple_dir_operations;
+  /* directories start off with i_nlink == 2 (for "." entry) */
+  inode->i_nlink++;
+ } else {
+  return -ENOMEM;
+ }
+
+ root = d_alloc_root(inode);
+ if (!root) {
+  iput(inode);
+  return -ENOMEM;
+ }
+ sb->s_root = root;
+ return 0;
```

```
+}
+
+static int container_get_sb(struct file_system_type *fs_type,
+    int flags, const char *unused_dev_name,
+    void *data, struct vfsmount *mnt)
+{
+ return get_sb_single(fs_type, flags, data, container_fill_super, mnt);
+}
+
+static struct file_system_type container_fs_type = {
+ .name = "container",
+ .get_sb = container_get_sb,
+ .kill_sb = kill_litter_super,
+};
+
+static inline struct container *__d_cont(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+static inline struct cftype *__d_cft(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+/*
+ * Call with manage_mutex held.  Writes path of container into buf.
+ * Returns 0 on success, -errno on error.
+ */
+
+static int container_path(const struct container *cont, char *buf, int buflen)
+{
+ char *start;
+
+ start = buf + buflen;
+
+ *--start = '\0';
+ for (;;) {
+  int len = cont->dentry->d_name.len;
+  if ((start -= len) < buf)
+   return -ENAMETOOLONG;
+  memcpy(start, cont->dentry->d_name.name, len);
+  cont = cont->parent;
+  if (!cont)
+   break;
+  if (!cont->parent)
+   continue;
+  if (--start < buf)
```

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

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

```
+ * update_flag - read a 0 or a 1 in a file and update associated flag
+ * bit: the bit to update (CONT_NOTIFY_ON_RELEASE)
+ * cont: the container to update
+ * buf: the buffer where we read the 0 or 1
+ *
+ * Call with manage_mutex held.
+ */
+
+static int update_flag(container_flagbits_t bit, struct container *cont, char *buf)
+{
+ int turning_on;
+
+ turning_on = (simple_strtoul(buf, NULL, 10) != 0);
+
+ mutex_lock(&callback_mutex);
+ if (turning_on)
+  set_bit(bit, &cont->flags);
+ else
+  clear_bit(bit, &cont->flags);
+ mutex_unlock(&callback_mutex);
+
+ return 0;
+}
+
+
+/*
+ * Attack task specified by pid in 'pidbuf' to container 'cont', possibly
+ * writing the path of the old container in 'ppathbuf' if it needs to be
+ * notified on release.
+ *
+ * Call holding manage_mutex.  May take callback_mutex and task_lock of
+ * the task 'pid' during call.
+ */
+
+static int attach_task(struct container *cont, char *pidbuf, char **ppathbuf)
+{
+ pid_t pid;
+ struct task_struct *tsk;
+ struct container *oldcont;
+ int retval;
+
+ if (sscanf(pidbuf, "%d", &pid) != 1)
+  return -EIO;
+
+ if (pid) {
+  read_lock(&tasklist_lock);
+
+  tsk = find_task_by_pid(pid);
```

```
+  if (!tsk || tsk->flags & PF_EXITING) {
+   read_unlock(&tasklist_lock);
+   return -ESRCH;
+  }
+
+  get_task_struct(tsk);
+  read_unlock(&tasklist_lock);
+
+  if ((current->euid) && (current->euid != tsk->uid)
+      && (current->euid != tsk->suid)) {
+   put_task_struct(tsk);
+   return -EACCES;
+  }
+ } else {
+  tsk = current;
+  get_task_struct(tsk);
+ }
+
+ retval = security_task_setscheduler(tsk, 0, NULL);
+ if (retval) {
+  put_task_struct(tsk);
+  return retval;
+ }
+
+ mutex_lock(&callback_mutex);
+
+ task_lock(tsk);
+ oldcont = tsk->container;
+ if (!oldcont) {
+  task_unlock(tsk);
+  mutex_unlock(&callback_mutex);
+  put_task_struct(tsk);
+  return -ESRCH;
+ }
+ atomic_inc(&cont->count);
+ rcu_assign_pointer(tsk->container, cont);
+ task_unlock(tsk);
+
+ mutex_unlock(&callback_mutex);
+
+ put_task_struct(tsk);
+ synchronize_rcu();
+ if (atomic_dec_and_test(&oldcont->count))
+  check_for_release(oldcont, ppathbuf);
+ return 0;
+}
+
+/* The various types of files and directories in a container file system */
```

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

```
+  retval = -EINVAL;
+  goto out2;
+ }
+
+ if (retval == 0)
+  retval = nbytes;
+out2:
+ mutex_unlock(&manage_mutex);
+ container_release_agent(pathbuf);
+out1:
+ kfree(buffer);
+ return retval;
+}
+
+static ssize_t container_file_write(struct file *file, const char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ ssize_t retval = 0;
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct container *cont = __d_cont(file->f_dentry->d_parent);
+ if (!cft)
+  return -ENODEV;
+
+ /* special function ? */
+ if (cft->write)
+  retval = cft->write(cont, cft, file, buf, nbytes, ppos);
+ else
+  retval = -EINVAL;
+
+ return retval;
+}
+
+static ssize_t container_common_file_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ container_filetype_t type = cft->private;
+ char *page;
+ ssize_t retval = 0;
+ char *s;
+
+ if (!(page = (char *)__get_free_page(GFP_KERNEL)))
+  return -ENOMEM;
+
+ s = page;
+
```

```
+ switch (type) {
+ case FILE_NOTIFY_ON_RELEASE:
+  *s++ = notify_on_release(cont) ? '1' : '0';
+  break;
+ default:
+  retval = -EINVAL;
+  goto out;
+ }
+ *s++ = '\n';
+
+ retval = simple_read_from_buffer(buf, nbytes, ppos, page, s - page);
+out:
+ free_page((unsigned long)page);
+ return retval;
+}
+
+static ssize_t container_file_read(struct file *file, char __user *buf, size_t nbytes,
+        loff_t *ppos)
+{
+ ssize_t retval = 0;
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct container *cont = __d_cont(file->f_dentry->d_parent);
+ if (!cft)
+  return -ENODEV;
+
+ /* special function ? */
+ if (cft->read)
+  retval = cft->read(cont, cft, file, buf, nbytes, ppos);
+ else
+  retval = -EINVAL;
+
+ return retval;
+}
+
+static int container_file_open(struct inode *inode, struct file *file)
+{
+ int err;
+ struct cftype *cft;
+
+ err = generic_file_open(inode, file);
+ if (err)
+  return err;
+
+ cft = __d_cft(file->f_dentry);
+ if (!cft)
+  return -ENODEV;
+ if (cft->open)
+  err = cft->open(inode, file);
```

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

```
+ if (!dentry)
+  return -ENOENT;
+ if (dentry->d_inode)
+  return -EEXIST;
+
+ inode = container_new_inode(mode);
+ if (!inode)
+  return -ENOMEM;
+
+ if (S_ISDIR(mode)) {
+  inode->i_op = &container_dir_inode_operations;
+  inode->i_fop = &simple_dir_operations;
+
+  /* start off with i_nlink == 2 (for "." entry) */
+  inode->i_nlink++;
+ } else if (S_ISREG(mode)) {
+  inode->i_size = 0;
+  inode->i_fop = &container_file_operations;
+ }
+
+ d_instantiate(dentry, inode);
+ dget(dentry); /* Extra count - pin the dentry in core */
+ return 0;
+}
+
+/*
+ * container_create_dir - create a directory for an object.
+ * cont: the container we create the directory for.
+ *  It must have a valid ->parent field
+ *  And we are going to fill its ->dentry field.
+ * name: The name to give to the container directory. Will be copied.
+ * mode: mode to set on new directory.
+ */
+
+static int container_create_dir(struct container *cont, const char *name, int mode)
+{
+ struct dentry *dentry = NULL;
+ struct dentry *parent;
+ int error = 0;
+
+ parent = cont->parent->dentry;
+ dentry = container_get_dentry(parent, name);
+ if (IS_ERR(dentry))
+  return PTR_ERR(dentry);
+ error = container_create_file(dentry, S_IFDIR | mode);
+ if (!error) {
+  dentry->d_fsdata = cont;
+  parent->d_inode->i_nlink++;
```

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

```
+/*
+ * Load into 'pidarray' up to 'npids' of the tasks using container 'cont'.
+ * Return actual number of pids loaded.  No need to task_lock(p)
+ * when reading out p->container, as we don't really care if it changes
+ * on the next cycle, and we are not going to try to dereference it.
+ */
+static int pid_array_load(pid_t *pidarray, int npids, struct container *cont)
+{
+ int n = 0;
+ struct task_struct *g, *p;
+
+ read_lock(&tasklist_lock);
+
+ do_each_thread(g, p) {
+  if (p->container == cont) {
+   pidarray[n++] = 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 container being opened.
+ *
+ * Does not require any specific container mutexes, and does not take any.
+ */
+static int container_tasks_open(struct inode *unused, struct file *file)
+{
+ struct container *cont = __d_cont(file->f_dentry->d_parent);
+ struct ctr_struct *ctr;
+ pid_t *pidarray;
+ int npids;
+ char c;
+
+ if (!(file->f_mode & FMODE_READ))
+  return 0;
+
+ ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
+ if (!ctr)
+  goto err0;
+
+ /*
+  * If container gets more users after we read count, we won't have
+  * enough space - tough.  This race is indistinguishable to the
+  * caller from the case that the additional container users didn't
+  * show up until sometime later on.
+  */
+ npids = atomic_read(&cont->count);
+ pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
+ if (!pidarray)
+  goto err1;
+
+ npids = pid_array_load(pidarray, npids, cont);
+ sort(pidarray, npids, sizeof(pid_t), cmppid, NULL);
+
+ /* Call pid_array_to_buf() twice, first just to get bufsz */
+ ctr->bufsz = pid_array_to_buf(&c, sizeof(c), pidarray, npids) + 1;
+ ctr->buf = kmalloc(ctr->bufsz, GFP_KERNEL);
+ if (!ctr->buf)
+  goto err2;
+ ctr->bufsz = pid_array_to_buf(ctr->buf, ctr->bufsz, pidarray, npids);
+
+ kfree(pidarray);
+ file->private_data = ctr;
+ return 0;
+
+err2:
+ kfree(pidarray);
+err1:
+ kfree(ctr);
```

```
+err0:
+ return -ENOMEM;
+}
+
+static ssize_t container_tasks_read(struct container *cont,
+        struct cftype *cft,
+        struct file *file, char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct ctr_struct *ctr = file->private_data;
+
+ if (*ppos + nbytes > ctr->bufsz)
+  nbytes = ctr->bufsz - *ppos;
+ if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
+  return -EFAULT;
+ *ppos += nbytes;
+ return nbytes;
+}
+
+static int container_tasks_release(struct inode *unused_inode, struct file *file)
+{
+ struct ctr_struct *ctr;
+
+ if (file->f_mode & FMODE_READ) {
+  ctr = file->private_data;
+  kfree(ctr->buf);
+  kfree(ctr);
+ }
+ return 0;
+}
+
+/*
+ * for the common functions, 'private' gives the type of file
+ */
+
+static struct cftype cft_tasks = {
+ .name = "tasks",
+ .open = container_tasks_open,
+ .read = container_tasks_read,
+ .write = container_common_file_write,
+ .release = container_tasks_release,
+ .private = FILE_TASKLIST,
+};
+
+static struct cftype cft_notify_on_release = {
+ .name = "notify_on_release",
+ .read = container_common_file_read,
+ .write = container_common_file_write,
```

```
+ .private = FILE_NOTIFY_ON_RELEASE,
+};
+
+static int container_populate_dir(struct container *cont)
+{
+ int err;
+
+ if ((err = container_add_file(cont, &cft_notify_on_release)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_tasks)) < 0)
+  return err;
+ return 0;
+}
+
+/*
+ * container_create - create a container
+ * parent: container that will be parent of the new container.
+ * name:  name of the new container. Will be strcpy'ed.
+ * mode:  mode to set on new inode
+ *
+ * Must be called with the mutex on the parent inode held
+ */
+
+static long container_create(struct container *parent, const char *name, int mode)
+{
+ struct container *cont;
+ int err;
+
+ cont = kmalloc(sizeof(*cont), GFP_KERNEL);
+ if (!cont)
+  return -ENOMEM;
+
+ mutex_lock(&manage_mutex);
+ cont->flags = 0;
+ if (notify_on_release(parent))
+  set_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+ atomic_set(&cont->count, 0);
+ INIT_LIST_HEAD(&cont->sibling);
+ INIT_LIST_HEAD(&cont->children);
+
+ cont->parent = parent;
+
+ mutex_lock(&callback_mutex);
+ list_add(&cont->sibling, &cont->parent->children);
+ number_of_containers++;
+ mutex_unlock(&callback_mutex);
+
+ err = container_create_dir(cont, name, mode);
```

```
+ if (err < 0)
+  goto err_remove;
+
+ /*
+  * Release manage_mutex before container_populate_dir() because it
+  * will down() this new directory's i_mutex and if we race with
+  * another mkdir, we might deadlock.
+  */
+ mutex_unlock(&manage_mutex);
+
+ err = container_populate_dir(cont);
+ /* If err < 0, we have a half-filled directory - oh well ;) */
+ return 0;
+
+ err_remove:
+ mutex_lock(&callback_mutex);
+ list_del(&cont->sibling);
+ number_of_containers--;
+ mutex_unlock(&callback_mutex);
+
+ mutex_unlock(&manage_mutex);
+ kfree(cont);
+ return err;
+}
+
+static int container_mkdir(struct inode *dir, struct dentry *dentry, int mode)
+{
+ struct container *c_parent = dentry->d_parent->d_fsdata;
+
+ /* the vfs holds inode->i_mutex already */
+ return container_create(c_parent, dentry->d_name.name, mode | S_IFDIR);
+}
+
+/*
+ * Locking note on the strange update_flag() call below:
+ *
+ * If the container being removed is marked cpu_exclusive, then simulate
+ * turning cpu_exclusive off, which will call update_cpu_domains().
+ * The lock_cpu_hotplug() call in update_cpu_domains() must not be
+ * made while holding callback_mutex.  Elsewhere the kernel nests
+ * callback_mutex inside lock_cpu_hotplug() calls.  So the reverse
+ * nesting would risk an ABBA deadlock.
+ */
+
+static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)
+{
+ struct container *cont = dentry->d_fsdata;
+ struct dentry *d;
```

```
+ struct container *parent;
+ char *pathbuf = NULL;
+
+ /* the vfs holds both inode->i_mutex already */
+
+ mutex_lock(&manage_mutex);
+ if (atomic_read(&cont->count) > 0) {
+  mutex_unlock(&manage_mutex);
+  return -EBUSY;
+ }
+ if (!list_empty(&cont->children)) {
+  mutex_unlock(&manage_mutex);
+  return -EBUSY;
+ }
+ parent = cont->parent;
+ mutex_lock(&callback_mutex);
+ set_bit(CONT_REMOVED, &cont->flags);
+ list_del(&cont->sibling); /* delete my sibling from parent->children */
+ spin_lock(&cont->dentry->d_lock);
+ d = dget(cont->dentry);
+ cont->dentry = NULL;
+ spin_unlock(&d->d_lock);
+ container_d_remove_dir(d);
+ dput(d);
+ number_of_containers--;
+ mutex_unlock(&callback_mutex);
+ if (list_empty(&parent->children))
+  check_for_release(parent, &pathbuf);
+ mutex_unlock(&manage_mutex);
+ container_release_agent(pathbuf);
+ return 0;
+}
+
+/*
+ * container_init_early - probably not needed yet, but will be needed
+ * once cpusets are hooked into this code
+ */
+
+int __init container_init_early(void)
+{
+ struct task_struct *tsk = current;
+
+ tsk->container = &top_container;
+ return 0;
+}
+
+/**
+ * container_init - initialize containers at system boot
```

```
+ *
+ * Description: Initialize top_container and the container internal file system,
+ **/
+
+int __init container_init(void)
+{
+ struct dentry *root;
+ int err;
+
+ init_task.container = &top_container;
+
+ err = register_filesystem(&container_fs_type);
+ if (err < 0)
+  goto out;
+ container_mount = kern_mount(&container_fs_type);
+ if (IS_ERR(container_mount)) {
+  printk(KERN_ERR "container: could not mount!\n");
+  err = PTR_ERR(container_mount);
+  container_mount = NULL;
+  goto out;
+ }
+ root = container_mount->mnt_sb->s_root;
+ root->d_fsdata = &top_container;
+ root->d_inode->i_nlink++;
+ top_container.dentry = root;
+ root->d_inode->i_op = &container_dir_inode_operations;
+ number_of_containers = 1;
+ err = container_populate_dir(&top_container);
+out:
+ return err;
+}
+
+/**
+ * container_fork - attach newly forked task to its parents container.
+ * @tsk: pointer to task_struct of forking parent process.
+ *
+ * Description: A task inherits its parent's container at fork().
+ *
+ * A pointer to the shared container was automatically copied in fork.c
+ * by dup_task_struct().  However, we ignore that copy, since it was
+ * not made under the protection of task_lock(), so might no longer be
+ * a valid container pointer.  attach_task() might have already changed
+ * current->container, allowing the previously referenced container to
+ * be removed and freed.  Instead, we task_lock(current) and copy
+ * its present value of current->container for our freshly forked child.
+ *
+ * At the point that container_fork() is called, 'current' is the parent
+ * task, and the passed argument 'child' points to the child task.
```

```
+ **/
+
+void container_fork(struct task_struct *child)
+{
+ task_lock(current);
+ child->container = current->container;
+ atomic_inc(&child->container->count);
+ task_unlock(current);
+}
+
+/**
+ * container_exit - detach container from exiting task
+ * @tsk: pointer to task_struct of exiting process
+ *
+ * Description: Detach container from @tsk and release it.
+ *
+ * Note that containers marked notify_on_release force every task in
+ * them to take the global manage_mutex mutex when exiting.
+ * This could impact scaling on very large systems.  Be reluctant to
+ * use notify_on_release containers where very high task exit scaling
+ * is required on large systems.
+ *
+ * Don't even think about derefencing 'cont' after the container use count
+ * goes to zero, except inside a critical section guarded by manage_mutex
+ * or callback_mutex.   Otherwise a zero container use count is a license to
+ * any other task to nuke the container immediately, via container_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->container,
+ * because tsk is already marked PF_EXITING, so attach_task() won't
+ * mess with it, or task is a failed fork, never visible to attach_task.
+ *
+ * the_top_container_hack:
+ *
+ *    Set the exiting tasks container to the root container (top_container).
+ *
+ *    Don't leave a task unable to allocate memory, as that is an
+ *    accident waiting to happen should someone add a callout in
+ *    do_exit() after the container_exit() call that might allocate.
+ *    If a task tries to allocate memory with an invalid container,
+ *    it will oops in container_update_task_memory_state().
+ *
+ *    We call container_exit() while the task is still competent to
+ *    handle notify_on_release(), then leave the task attached to
+ *    the root container (top_container) for the remainder of its exit.
```

```
+ *
+ *    To do this properly, we would increment the reference count on
+ *    top_container, and near the very end of the kernel/exit.c do_exit()
+ *    code we would add a second container function call, to drop that
+ *    reference.  This would just create an unnecessary hot spot on
+ *    the top_container reference count, to no avail.
+ *
+ *    Normally, holding a reference to a container without bumping its
+ *    count is unsafe.   The container could go away, or someone could
+ *    attach us to a different container, decrementing the count on
+ *    the first container that we never incremented.  But in this case,
+ *    top_container isn't going away, and either task has PF_EXITING set,
+ *    which wards off any attach_task() attempts, or task is a failed
+ *    fork, never visible to attach_task.
+ *
+ *    Another way to do this would be to set the container pointer
+ *    to NULL here, and check in container_update_task_memory_state()
+ *    for a NULL pointer.  This hack avoids that NULL check, for no
+ *    cost (other than this way too long comment ;).
+ **/
+
+void container_exit(struct task_struct *tsk)
+{
+ struct container *cont;
+
+ cont = tsk->container;
+ tsk->container = &top_container; /* the_top_container_hack - see above */
+
+ if (notify_on_release(cont)) {
+  char *pathbuf = NULL;
+
+  mutex_lock(&manage_mutex);
+  if (atomic_dec_and_test(&cont->count))
+   check_for_release(cont, &pathbuf);
+  mutex_unlock(&manage_mutex);
+  container_release_agent(pathbuf);
+ } else {
+  atomic_dec(&cont->count);
+ }
+}
+
+/**
+ * container_lock - lock out any changes to container structures
+ *
+ * The out of memory (oom) code needs to mutex_lock containers
+ * from being changed while it scans the tasklist looking for a
+ * task in an overlapping container.  Expose callback_mutex via this
+ * container_lock() routine, so the oom code can lock it, before
```

```
+ * locking the task list.  The tasklist_lock is a spinlock, so
+ * must be taken inside callback_mutex.
+ */
+
+void container_lock(void)
+{
+ mutex_lock(&callback_mutex);
+}
+
+/**
+ * container_unlock - release lock on container changes
+ *
+ * Undo the lock taken in a previous container_lock() call.
+ */
+
+void container_unlock(void)
+{
+ mutex_unlock(&callback_mutex);
+}
+
+/*
+ * proc_container_show()
+ *  - Print tasks container path into seq_file.
+ *  - Used for /proc/<pid>/container.
+ *  - No need to task_lock(tsk) on this tsk->container reference, as it
+ *    doesn't really matter if tsk->container changes after we read it,
+ *    and we take manage_mutex, keeping attach_task() from changing it
+ *    anyway.  No need to check that tsk->container != NULL, thanks to
+ *    the_top_container_hack in container_exit(), which sets an exiting tasks
+ *    container to top_container.
+ */
+static int proc_container_show(struct seq_file *m, void *v)
+{
+ struct pid *pid;
+ struct task_struct *tsk;
+ char *buf;
+ int retval;
+
+ retval = -ENOMEM;
+ buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
+ if (!buf)
+  goto out;
+
+ retval = -ESRCH;
+ pid = m->private;
+ tsk = get_pid_task(pid, PIDTYPE_PID);
+ if (!tsk)
+  goto out_free;
```

```
+
+ retval = -EINVAL;
+ mutex_lock(&manage_mutex);
+
+ retval = container_path(tsk->container, buf, PAGE_SIZE);
+ if (retval < 0)
+  goto out_unlock;
+ seq_puts(m, buf);
+ seq_putc(m, '\n');
+out_unlock:
+ mutex_unlock(&manage_mutex);
+ put_task_struct(tsk);
+out_free:
+ kfree(buf);
+out:
+ return retval;
+}
+
+static int container_open(struct inode *inode, struct file *file)
+{
+ struct pid *pid = PROC_I(inode)->pid;
+ return single_open(file, proc_container_show, pid);
+}
+
+struct file_operations proc_container_operations = {
+ .open  = container_open,
+ .read  = seq_read,
+ .llseek  = seq_lseek,
+ .release = single_release,
+};
Index: container-2.6.19-rc5/kernel/exit.c
 ===================================================================
--- container-2.6.19-rc5.orig/kernel/exit.c
+++ container-2.6.19-rc5/kernel/exit.c
@@ -30,6 +30,7 @@
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
 #include <linux/posix-timers.h>
@@ -923,6 +924,7 @@ fastcall NORET_TYPE void do_exit(long co
  __exit_fs(tsk);
  exit_thread();
  cpuset_exit(tsk);
+ container_exit(tsk);
  exit_keys(tsk);
```

```
  if (group_dead && tsk->signal->leader)
Index: container-2.6.19-rc5/kernel/fork.c
 ===================================================================
--- container-2.6.19-rc5.orig/kernel/fork.c
+++ container-2.6.19-rc5/kernel/fork.c
@@ -31,6 +31,7 @@
 #include <linux/capability.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/security.h>
 #include <linux/swap.h>
 #include <linux/syscalls.h>
@@ -1054,6 +1055,7 @@ static struct task_struct *copy_process(
   p->io_context = NULL;
   p->io_wait = NULL;
   p->audit_context = NULL;
+ container_fork(p);
   cpuset_fork(p);
 #ifdef CONFIG_NUMA
    p->mempolicy = mpol_copy(p->mempolicy);
@@ -1287,6 +1289,7 @@ bad_fork_cleanup_policy:
 bad_fork_cleanup_cpuset:
 #endif
   cpuset_exit(p);
+ container_exit(p);
 bad_fork_cleanup_delays_binfmt:
   delayacct_tsk_free(p);
   if (p->binfmt)
Index: container-2.6.19-rc5/kernel/Makefile
 ===================================================================
--- container-2.6.19-rc5.orig/kernel/Makefile
+++ container-2.6.19-rc5/kernel/Makefile
@@ -36,6 +36,7 @@ obj-$(CONFIG_PM) += power/
 obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
+obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
Index: container-2.6.19-rc5/Documentation/containers.txt
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc5/Documentation/containers.txt
@@ -0,0 +1,229 @@
+    CONTAINERS
```

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

+
+There are multiple efforts to provide process aggregations in the
+Linux kernel, mainly for resource tracking purposes. Such efforts
+include cpusets, CKRM/ResGroups, and UserBeanCounters. These all
+require the basic notion of a grouping of processes, with newly forked
+processes ending in the same group (container) as their parent
+process.
+
+The kernel container patch provides the minimum essential kernel
+mechanisms required to efficiently implement such groups. It has
+minimal impact on the system fast paths, and provides hooks for
+specific subsystems such as cpusets to provide additional behaviour as
+desired.
+
+
+1.3 How are containers implemented ?
+---------------------------------
+
+Containers extends the kernel as follows:
+
+ - Each task in the system is attached to a container, via a pointer
+   in the task structure to a reference counted container structure.
+ - The hierarchy of containers can be mounted at /dev/container (or
+   elsewhere), for browsing and manipulation from user space.
+ - You can list all the tasks (by pid) attached to any container.
+
+The implementation of containers requires a few, simple hooks
+into the rest of the kernel, none in performance critical paths:
+
+ - in init/main.c, to initialize the root container at system boot.
+ - in fork and exit, to attach and detach a task from its container.
+
+In addition a new file system, of type "container" may be mounted,
+typically at /dev/container, to enable browsing and modifying the containers
+presently known to the kernel. No new system calls are added for
+containers - all support for querying and modifying containers is via
+this container file system.
+
+Each task under /proc has an added file named 'container', displaying
+the container name, as the path relative to the root of the container file
+system.
+
+Each container is represented by a directory in the container file system
+containing the following files describing that container:
+
+ - tasks: list of tasks (by pid) attached to that container
+ - notify_on_release flag: run /sbin/container_release_agent on exit?
+

+Other subsystems such as cpusets may add additional files in each
+container dir
+
+New containers are created using the mkdir system call or shell
+command. The properties of a container, such as its flags, are
+modified by writing to the appropriate file in that containers
+directory, as listed above.
+
+The named hierarchical structure of nested containers allows partitioning
+a large system into nested, dynamically changeable, "soft-partitions".
+
+The attachment of each task, automatically inherited at fork by any
+children of that task, to a container allows organizing the work load
+on a system into related sets of tasks. A task may be re-attached to
+any other container, if allowed by the permissions on the necessary
+container file system directories.
+
+The use of a Linux virtual file system (vfs) to represent the
+container hierarchy provides for a familiar permission and name space
+for containers, with a minimum of additional kernel code.
+
+1.4 What does notify_on_release do ?
+------------------------------------
+
+If the notify_on_release flag is enabled (1) in a container, then whenever
+the last task in the container leaves (exits or attaches to some other
+container) and the last child container of that container is removed, then
+the kernel runs the command /sbin/container_release_agent, supplying the
+pathname (relative to the mount point of the container file system) of the
+abandoned container. This enables automatic removal of abandoned containers.
+The default value of notify_on_release in the root container at system
+boot is disabled (0). The default value of other containers at creation
+is the current value of their parents notify_on_release setting.
+
+1.5 How do I use containers ?
+-------------------------
+
+To start a new job that is to be contained within a container, the steps are:
+
+ 1) mkdir /dev/container
+ 2) mount -t container container /dev/container
+ 3) Create the new container by doing mkdir's and write's (or echo's) in
+   the /dev/container virtual file system.
+ 4) Start a task that will be the "founding father" of the new job.
+ 5) Attach that task to the new container by writing its pid to the
+   /dev/container tasks file for that container.
+ 6) fork, exec or clone the job tasks from this founding father task.
+

+For example, the following sequence of commands will setup a container
+named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
+and then start a subshell 'sh' in that container:
+
+ mount -t container none /dev/container
+ cd /dev/container
+ mkdir Charlie
+ cd Charlie
+ /bin/echo $$ > tasks
+ sh
+ # The subshell 'sh' is now running in container Charlie
+ # The next line should display '/Charlie'
+ cat /proc/self/container
+
+In the future, a C library interface to containers will likely be
+available. For now, the only way to query or modify containers is
+via the container file system, using the various cd, mkdir, echo, cat,
+rmdir commands from the shell, or their equivalent from C.
+
+2. Usage Examples and Syntax
+============================
+
+2.1 Basic Usage
+---------------
+
+Creating, modifying, using the containers can be done through the container
+virtual filesystem.
+
+To mount it, type:
+# mount -t container none /dev/container
+
+Then under /dev/container you can find a tree that corresponds to the
+tree of the containers in the system. For instance, /dev/container
+is the container that holds the whole system.
+
+If you want to create a new container under /dev/container:
+# cd /dev/container
+# mkdir my_container
+
+Now you want to do something with this container.
+# cd my_container
+
+In this directory you can find several files:
+# ls
+notify_on_release tasks
+
+Now attach your shell to this container:
+# /bin/echo $$ > tasks

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

--

Subject: [PATCH 2/7] Cpusets hooked into containers
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:50 GMT
View Forum Message <> Reply to Message

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

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

 Documentation/cpusets.txt | 81 +-
 fs/super.c | 5
 include/linux/container.h | 7
 include/linux/cpuset.h | 25
 include/linux/fs.h | 2
 include/linux/mempolicy.h | 2
 include/linux/sched.h | 4
 init/Kconfig | 23
 kernel/container.c | 107 +++
 kernel/cpuset.c | 1273 +++++--
 kernel/exit.c | 2
 kernel/fork.c | 7
 mm/oom_kill.c | 6
 13 files changed, 330 insertions(+), 1214 deletions(-)

Index: container-2.6.19-rc5/include/linux/container.h
 === =======
--- container-2.6.19-rc5.orig/include/linux/container.h
+++ container-2.6.19-rc5/include/linux/container.h
@@ -47,6 +47,10 @@ struct container {

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

 /* struct cftype:
@@ -79,6 +83,9 @@ struct cftype {
 int container_add_file(struct container *cont, const struct cftype *cft);

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

 #else /* !CONFIG_CONTAINERS */

Index: container-2.6.19-rc5/include/linux/cpuset.h
 === =======
--- container-2.6.19-rc5.orig/include/linux/cpuset.h
+++ container-2.6.19-rc5/include/linux/cpuset.h

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

 #ifdef CONFIG_CPUSETS

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

 extern int cpuset_init_early(void);
 extern int cpuset_init(void);
 extern void cpuset_init_smp(void);
-extern void cpuset_fork(struct task_struct *p);
-extern void cpuset_exit(struct task_struct *p);
 extern cpumask_t cpuset_cpus_allowed(struct task_struct *p);
 extern nodemask_t cpuset_mems_allowed(struct task_struct *p);
 void cpuset_init_current_mems_allowed(void);
@@ -47,10 +46,6 @@ extern void __cpuset_memory_pressure_bum

 extern struct file_operations proc_cpuset_operations;
 extern char *cpuset_task_status_allowed(struct task_struct *task, char *buffer);
-
-extern void cpuset_lock(void);
-extern void cpuset_unlock(void);
-
 extern int cpuset_mem_spread_node(void);

 static inline int cpuset_do_page_mem_spread(void)
@@ -65,13 +60,22 @@ static inline int cpuset_do_slab_mem_spr

 extern void cpuset_track_online_nodes(void);

+extern int cpuset_can_attach_task(struct container *cont,
+      struct task_struct *tsk);
+extern void cpuset_attach_task(struct container *cont,
+    struct task_struct *tsk);
+extern void cpuset_post_attach_task(struct container *cont,
+        struct container *oldcont,
+        struct task_struct *tsk);
+extern int cpuset_populate_dir(struct container *cont);
+extern int cpuset_create(struct container *cont);
+extern void cpuset_destroy(struct container *cont);
+
 #else /* !CONFIG_CPUSETS */

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

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

 static inline cpumask_t cpuset_cpus_allowed(struct task_struct *p)
 {
@@ -110,9 +114,6 @@ static inline char *cpuset_task_status_a
  return buffer;
 }

-static inline void cpuset_lock(void) {}
-static inline void cpuset_unlock(void) {}
-
 static inline int cpuset_mem_spread_node(void)
 {
  return 0;
Index: container-2.6.19-rc5/kernel/exit.c
 ===================================================================
--- container-2.6.19-rc5.orig/kernel/exit.c
+++ container-2.6.19-rc5/kernel/exit.c
@@ -29,7 +29,6 @@
 #include <linux/mempolicy.h>
 #include <linux/taskstats_kern.h>
 #include <linux/delayacct.h>
-#include <linux/cpuset.h>
 #include <linux/container.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
@@ -923,7 +922,6 @@ fastcall NORET_TYPE void do_exit(long co
  __exit_files(tsk);
  __exit_fs(tsk);
  exit_thread();
- cpuset_exit(tsk);
  container_exit(tsk);
  exit_keys(tsk);

Index: container-2.6.19-rc5/kernel/fork.c
 ===================================================================
--- container-2.6.19-rc5.orig/kernel/fork.c
+++ container-2.6.19-rc5/kernel/fork.c
@@ -30,7 +30,6 @@
 #include <linux/nsproxy.h>
 #include <linux/capability.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
 #include <linux/container.h>
 #include <linux/security.h>
```

```
 #include <linux/swap.h>
@@ -1056,13 +1055,12 @@ static struct task_struct *copy_process(
  p->io_wait = NULL;
  p->audit_context = NULL;
  container_fork(p);
- cpuset_fork(p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
   if (IS_ERR(p->mempolicy)) {
    retval = PTR_ERR(p->mempolicy);
    p->mempolicy = NULL;
-   goto bad_fork_cleanup_cpuset;
+   goto bad_fork_cleanup_container;
   }
  mpol_fix_fork_child_flag(p);
 #endif
@@ -1286,9 +1284,8 @@ bad_fork_cleanup_security:
 bad_fork_cleanup_policy:
 #ifdef CONFIG_NUMA
  mpol_free(p->mempolicy);
-bad_fork_cleanup_cpuset:
+bad_fork_cleanup_container:
 #endif
- cpuset_exit(p);
  container_exit(p);
 bad_fork_cleanup_delays_binfmt:
  delayacct_tsk_free(p);
Index: container-2.6.19-rc5/kernel/container.c
 ========================================================== =======
--- container-2.6.19-rc5.orig/kernel/container.c
+++ container-2.6.19-rc5/kernel/container.c
@@ -55,6 +55,7 @@
 #include <linux/time.h>
 #include <linux/backing-dev.h>
 #include <linux/sort.h>
+#include <linux/cpuset.h>

 #include <asm/uaccess.h>
 #include <asm/atomic.h>
@@ -92,6 +93,18 @@ static struct container top_container =
  .children = LIST_HEAD_INIT(top_container.children),
 };

+/* The path to use for release notifications. No locking between
+ * setting and use - so if userspace updates this while subcontainers
+ * exist, you could miss a notification */
+static char release_agent_path[PATH_MAX] = "/sbin/container_release_agent";
+
```

```
+void container_set_release_agent_path(const char *path)
+{
+ container_manage_lock();
+ strcpy(release_agent_path, path);
+ container_manage_unlock();
+}
+
 static struct vfsmount *container_mount;
 static struct super_block *container_sb;

@@ -333,7 +346,7 @@ static inline struct cftype *__d_cft(str
  * Returns 0 on success, -errno on error.
  */

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

@@ -397,7 +410,7 @@ static void container_release_agent(cons
   return;

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

@@ -438,6 +451,7 @@ static void check_for_release(struct con
   buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
   if (!buf)
    return;
+
   if (container_path(cont, buf, PAGE_SIZE) < 0)
    kfree(buf);
   else
@@ -486,7 +500,7 @@ static int attach_task(struct container
  pid_t pid;
  struct task_struct *tsk;
  struct container *oldcont;
- int retval;
+ int retval = 0;

  if (sscanf(pidbuf, "%d", &pid) != 1)
   return -EIO;
@@ -513,7 +527,9 @@ static int attach_task(struct container
   get_task_struct(tsk);
  }
```

```
- retval = security_task_setscheduler(tsk, 0, NULL);
+#ifdef CONFIG_CPUSETS
+ retval = cpuset_can_attach_task(cont, tsk);
+#endif
  if (retval) {
   put_task_struct(tsk);
   return retval;
@@ -533,8 +549,16 @@ static int attach_task(struct container
  rcu_assign_pointer(tsk->container, cont);
  task_unlock(tsk);

+#ifdef CONFIG_CPUSETS
+ cpuset_attach_task(cont, tsk);
+#endif
+
  mutex_unlock(&callback_mutex);

+#ifdef CONFIG_CPUSETS
+ cpuset_post_attach_task(cont, oldcont, tsk);
+#endif
+
  put_task_struct(tsk);
  synchronize_rcu();
  if (atomic_dec_and_test(&oldcont->count))
@@ -549,6 +573,7 @@ typedef enum {
  FILE_DIR,
  FILE_NOTIFY_ON_RELEASE,
  FILE_TASKLIST,
+ FILE_RELEASE_AGENT,
 } container_filetype_t;

 static ssize_t container_common_file_write(struct container *cont,
@@ -562,8 +587,7 @@ static ssize_t container_common_file_wri
  char *pathbuf = NULL;
  int retval = 0;

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

  /* +1 for nul-terminator */
@@ -590,6 +614,20 @@ static ssize_t container_common_file_wri
  case FILE_TASKLIST:
   retval = attach_task(cont, buffer, &pathbuf);
   break;
+ case FILE_RELEASE_AGENT:
```

```
+ {
+  if (nbytes < sizeof(release_agent_path)) {
+   /* We never write anything other than '\0'
+    * into the last char of release_agent_path,
+    * so it always remains a NUL-terminated
+    * string */
+   strncpy(release_agent_path, buffer, nbytes);
+   release_agent_path[nbytes] = 0;
+  } else {
+   retval = -ENOSPC;
+  }
+  break;
+ }
  default:
   retval = -EINVAL;
   goto out2;
@@ -643,6 +681,17 @@ static ssize_t container_common_file_rea
  case FILE_NOTIFY_ON_RELEASE:
   *s++ = notify_on_release(cont) ? '1' : '0';
   break;
+ case FILE_RELEASE_AGENT:
+ {
+  size_t n;
+  container_manage_lock();
+  n = strnlen(release_agent_path, sizeof(release_agent_path));
+  n = min(n, (size_t) PAGE_SIZE);
+  strncpy(s, release_agent_path, n);
+  container_manage_unlock();
+  s += n;
+  break;
+ }
  default:
   retval = -EINVAL;
   goto out;
@@ -978,6 +1027,13 @@ static struct cftype cft_notify_on_relea
  .private = FILE_NOTIFY_ON_RELEASE,
 };

+static struct cftype cft_release_agent = {
+ .name = "release_agent",
+ .read = container_common_file_read,
+ .write = container_common_file_write,
+ .private = FILE_RELEASE_AGENT,
+};
+
 static int container_populate_dir(struct container *cont)
 {
  int err;
```

```
@@ -986,6 +1042,13 @@ static int container_populate_dir(struct
   return err;
  if ((err = container_add_file(cont, &cft_tasks)) < 0)
   return err;
+ if ((cont == &top_container) &&
+     (err = container_add_file(cont, &cft_release_agent)) < 0)
+  return err;
+#ifdef CONFIG_CPUSETS
+ if ((err = cpuset_populate_dir(cont)) < 0)
+  return err;
+#endif
  return 0;
 }

@@ -1017,6 +1080,12 @@ static long container_create(struct cont

  cont->parent = parent;

+#ifdef CONFIG_CPUSETS
+ err = cpuset_create(cont);
+ if (err)
+  goto err_unlock_free;
+#endif
+
  mutex_lock(&callback_mutex);
  list_add(&cont->sibling, &cont->parent->children);
  number_of_containers++;
@@ -1038,11 +1107,14 @@ static long container_create(struct cont
  return 0;

  err_remove:
+#ifdef CONFIG_CPUSETS
+ cpuset_destroy(cont);
+#endif
  mutex_lock(&callback_mutex);
  list_del(&cont->sibling);
  number_of_containers--;
  mutex_unlock(&callback_mutex);
-
+ err_unlock_free:
  mutex_unlock(&manage_mutex);
  kfree(cont);
  return err;
@@ -1097,6 +1169,9 @@ static int container_rmdir(struct inode
  dput(d);
  number_of_containers--;
  mutex_unlock(&callback_mutex);
+#ifdef CONFIG_CPUSETS
```

```
+ cpuset_destroy(cont);
+#endif
  if (list_empty(&parent->children))
   check_for_release(parent, &pathbuf);
  mutex_unlock(&manage_mutex);
@@ -1283,6 +1358,24 @@ void container_unlock(void)
  mutex_unlock(&callback_mutex);
 }

+void container_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 container_manage_unlock(void)
+{
+ mutex_unlock(&manage_mutex);
+}
+
+
+
 /*
  * proc_container_show()
  *  - Print tasks container path into seq_file.
Index: container-2.6.19-rc5/kernel/cpuset.c
 ===================================================================== =======
--- container-2.6.19-rc5.orig/kernel/cpuset.c
+++ container-2.6.19-rc5/kernel/cpuset.c
@@ -54,8 +54,6 @@
 #include <asm/atomic.h>
 #include <linux/mutex.h>

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

- /*
```

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

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

-static inline int is_removed(const struct cpuset *cs)
-{
- return test_bit(CS_REMOVED, &cs->flags);
-}
-
-static inline int notify_on_release(const struct cpuset *cs)
-{
- return test_bit(CS_NOTIFY_ON_RELEASE, &cs->flags);
-}
-
 static inline int is_memory_migrate(const struct cpuset *cs)
 {
  return test_bit(CS_MEMORY_MIGRATE, &cs->flags);
@@ -173,388 +147,32 @@ static struct cpuset top_cpuset = {
  .flags = ((1 << CS_CPU_EXCLUSIVE) | (1 << CS_MEM_EXCLUSIVE)),
  .cpus_allowed = CPU_MASK_ALL,
  .mems_allowed = NODE_MASK_ALL,
- .count = ATOMIC_INIT(0),
```

```
- .sibling = LIST_HEAD_INIT(top_cpuset.sibling),
- .children = LIST_HEAD_INIT(top_cpuset.children),
-};
-
-static struct vfsmount *cpuset_mount;
-static struct super_block *cpuset_sb;
-
-/*
- * We have two global cpuset mutexes below.  They can nest.
- * It is ok to first take manage_mutex, then nest callback_mutex.  We also
- * 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)
-{
- struct dentry *parent = dget(d->d_parent);
-
- d_delete(d);
- simple_rmdir(parent->d_inode, d);
- dput(parent);
-}
-
-/*
- * NOTE : the dentry must have been dget()'ed
- */
-static void cpuset_d_remove_dir(struct dentry *dentry)
-{
- struct list_head *node;
-
- spin_lock(&dcache_lock);
- node = dentry->d_subdirs.next;
- while (node != &dentry->d_subdirs) {
-  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
-  list_del_init(node);
-  if (d->d_inode) {
-   d = dget_locked(d);
-   spin_unlock(&dcache_lock);
-   d_delete(d);
-   simple_unlink(dentry->d_inode, d);
-   dput(d);
-   spin_lock(&dcache_lock);
-  }
-  node = dentry->d_subdirs.next;
- }
- list_del_init(&dentry->d_u.d_child);
- spin_unlock(&dcache_lock);
- remove_dir(dentry);
-}
-
-static struct super_operations cpuset_ops = {
- .statfs = simple_statfs,
- .drop_inode = generic_delete_inode,
 };

-static int cpuset_fill_super(struct super_block *sb, void *unused_data,
-       int unused_silent)
-{
- struct inode *inode;
- struct dentry *root;
-
```

```
- sb->s_blocksize = PAGE_CACHE_SIZE;
- sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
- sb->s_magic = CPUSET_SUPER_MAGIC;
- sb->s_op = &cpuset_ops;
- cpuset_sb = sb;
-
- inode = cpuset_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR);
- if (inode) {
-  inode->i_op = &simple_dir_inode_operations;
-  inode->i_fop = &simple_dir_operations;
-  /* directories start off with i_nlink == 2 (for "." entry) */
-  inc_nlink(inode);
- } else {
-  return -ENOMEM;
- }
-
- root = d_alloc_root(inode);
- if (!root) {
-  iput(inode);
-  return -ENOMEM;
- }
- sb->s_root = root;
- return 0;
-}
-
+/* This is ugly, but preserves the userspace API for existing cpuset
+ * users. If someone tries to mount the "cpuset" filesystem, we
+ * silently switch it to mount "container" instead */
 static int cpuset_get_sb(struct file_system_type *fs_type,
     int flags, const char *unused_dev_name,
     void *data, struct vfsmount *mnt)
 {
- return get_sb_single(fs_type, flags, data, cpuset_fill_super, mnt);
+ struct file_system_type *container_fs = get_fs_type("container");
+ int ret = -ENODEV;
+ container_set_release_agent_path("/sbin/cpuset_release_agent ");
+ if (container_fs) {
+  ret = container_fs->get_sb(container_fs, flags,
+        unused_dev_name,
+        data, mnt);
+  put_filesystem(container_fs);
+ }
+ return ret;
 }

 static struct file_system_type cpuset_fs_type = {
  .name = "cpuset",
  .get_sb = cpuset_get_sb,
```

```
- .kill_sb = kill_litter_super,
 };

-/* struct cftype:
- *
- * The files in the cpuset filesystem mostly have a very simple read/write
- * handling, some common function will take care of it. Nevertheless some cases
- * (read tasks) are special and therefore I define this structure for every
- * kind of file.
- *
- *
- * When reading/writing to a file:
- * - the cpuset to use in file->f_dentry->d_parent->d_fsdata
- * - the 'cftype' of the file is file->f_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 +270,20 @@ void cpuset_update_task_memory_state(voi
  struct task_struct *tsk = current;
  struct cpuset *cs;

- if (tsk->cpuset == &top_cpuset) {
+ if (tsk->container->cpuset == &top_cpuset) {
   /* Don't need rcu for top_cpuset.  It's never freed. */
   my_cpusets_mem_gen = top_cpuset.mems_generation;
  } else {
   rcu_read_lock();
-  cs = rcu_dereference(tsk->cpuset);
+  cs = rcu_dereference(tsk->container->cpuset);
   my_cpusets_mem_gen = cs->mems_generation;
   rcu_read_unlock();
  }

  if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
-  mutex_lock(&callback_mutex);
+  container_lock();
   task_lock(tsk);
-  cs = tsk->cpuset; /* Maybe changed when task not locked */
+  cs = tsk->container->cpuset; /* Maybe changed when task not locked */
   guarantee_online_mems(cs, &tsk->mems_allowed);
   tsk->cpuset_mems_generation = cs->mems_generation;
   if (is_spread_page(cs))
@@ -677,7 +295,7 @@ void cpuset_update_task_memory_state(voi
   else
    tsk->flags &= ~PF_SPREAD_SLAB;
   task_unlock(tsk);
-  mutex_unlock(&callback_mutex);
+  container_unlock();
   mpol_rebind_task(tsk, &tsk->mems_allowed);
  }
 }
@@ -720,10 +338,12 @@ static int is_cpuset_subset(const struct
```

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

  /* Each of our child cpusets must be a subset of us */
- list_for_each_entry(c, &cur->children, sibling) {
+ list_for_each_entry(cont, &cur->container->children, sibling) {
+  c = cont->cpuset;
   if (!is_cpuset_subset(c, trial))
    return -EBUSY;
  }
@@ -737,7 +357,8 @@ static int validate_change(const struct
   return -EACCES;

  /* If either I or some sibling (!= me) is exclusive, we can't overlap */
- list_for_each_entry(c, &par->children, sibling) {
+ list_for_each_entry(cont, &par->container->children, sibling) {
+  c = cont->cpuset;
   if ((is_cpu_exclusive(trial) || is_cpu_exclusive(c)) &&
       c != cur &&
       cpus_intersects(trial->cpus_allowed, c->cpus_allowed))
@@ -767,6 +388,7 @@ static int validate_change(const struct

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

@@ -778,7 +400,8 @@ static void update_cpu_domains(struct cp
   * children
   */
  pspan = par->cpus_allowed;
- list_for_each_entry(c, &par->children, sibling) {
+ list_for_each_entry(cont, &par->container->children, sibling) {
+  c = cont->cpuset;
   if (is_cpu_exclusive(c))
    cpus_andnot(pspan, pspan, c->cpus_allowed);
  }
@@ -795,7 +418,8 @@ static void update_cpu_domains(struct cp
    * Get all cpus from current cpuset's cpus_allowed not part
    * of exclusive children
    */
-  list_for_each_entry(c, &cur->children, sibling) {
+  list_for_each_entry(cont, &cur->container->children, sibling) {
+   c = cont->cpuset;
```

```
    if (is_cpu_exclusive(c))
     cpus_andnot(cspan, cspan, c->cpus_allowed);
   }
@@ -830,9 +454,9 @@ static int update_cpumask(struct cpuset
  if (retval < 0)
   return retval;
  cpus_unchanged = cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed);
- mutex_lock(&callback_mutex);
+ container_lock();
  cs->cpus_allowed = trialcs.cpus_allowed;
- mutex_unlock(&callback_mutex);
+ container_unlock();
  if (is_cpu_exclusive(cs) && !cpus_unchanged)
   update_cpu_domains(cs);
  return 0;
@@ -876,15 +500,15 @@ static void cpuset_migrate_mm(struct mm_

  cpuset_update_task_memory_state();

- mutex_lock(&callback_mutex);
+ container_lock();
  tsk->mems_allowed = *to;
- mutex_unlock(&callback_mutex);
+ container_unlock();

  do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);

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

 /*
@@ -911,12 +535,14 @@ static int update_nodemask(struct cpuset
  int migrate;
  int fudge;
  int retval;
+ struct container *cont;

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

  trialcs = *cs;
+ cont = cs->container;
```

```
  retval = nodelist_parse(buf, trialcs.mems_allowed);
  if (retval < 0)
   goto done;
@@ -934,10 +560,10 @@ static int update_nodemask(struct cpuset
  if (retval < 0)
   goto done;

- mutex_lock(&callback_mutex);
+ container_lock();
  cs->mems_allowed = trialcs.mems_allowed;
  cs->mems_generation = cpuset_mems_generation++;
- mutex_unlock(&callback_mutex);
+ container_unlock();

  set_cpuset_being_rebound(cs);  /* causes mpol_copy() rebind */

@@ -953,13 +579,13 @@ static int update_nodemask(struct cpuset
   * enough mmarray[] w/o using GFP_ATOMIC.
   */
  while (1) {
-  ntasks = atomic_read(&cs->count); /* guess */
+  ntasks = atomic_read(&cs->container->count);  /* guess */
   ntasks += fudge;
   mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
   if (!mmarray)
    goto done;
   write_lock_irq(&tasklist_lock);  /* block fork */
-  if (atomic_read(&cs->count) <= ntasks)
+  if (atomic_read(&cs->container->count) <= ntasks)
    break;    /* got enough */
   write_unlock_irq(&tasklist_lock); /* try again */
   kfree(mmarray);
@@ -976,7 +602,7 @@ static int update_nodemask(struct cpuset
     "Cpuset mempolicy rebind incomplete.\n");
    continue;
   }
-  if (p->cpuset != cs)
+  if (p->container != cont)
    continue;
   mm = get_task_mm(p);
   if (!mm)
@@ -1059,15 +685,15 @@ static int update_flag(cpuset_flagbits_t
   return err;
  cpu_exclusive_changed =
   (is_cpu_exclusive(cs) != is_cpu_exclusive(&trialcs));
- mutex_lock(&callback_mutex);
+ container_lock();
  if (turning_on)
```

```
   set_bit(bit, &cs->flags);
  else
   clear_bit(bit, &cs->flags);
- mutex_unlock(&callback_mutex);
+ container_unlock();

  if (cpu_exclusive_changed)
-                update_cpu_domains(cs);
+  update_cpu_domains(cs);
  return 0;
 }

@@ -1169,85 +795,35 @@ static int fmeter_getrate(struct fmeter
  return val;
 }

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

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

- if (pid) {
-  read_lock(&tasklist_lock);
-
-  tsk = find_task_by_pid(pid);
-  if (!tsk || tsk->flags & PF_EXITING) {
-   read_unlock(&tasklist_lock);
```

```
-   return -ESRCH;
-  }
-
-  get_task_struct(tsk);
-  read_unlock(&tasklist_lock);
-
-  if ((current->euid) && (current->euid != tsk->uid)
-      && (current->euid != tsk->suid)) {
-   put_task_struct(tsk);
-   return -EACCES;
-  }
- } else {
-  tsk = current;
-  get_task_struct(tsk);
- }
-
- retval = security_task_setscheduler(tsk, 0, NULL);
- if (retval) {
-  put_task_struct(tsk);
-  return retval;
- }
-
- mutex_lock(&callback_mutex);
-
- task_lock(tsk);
- oldcs = tsk->cpuset;
- /*
-  * After getting 'oldcs' cpuset ptr, be sure still not exiting.
-  * If 'oldcs' might be the top_cpuset due to the_top_cpuset_hack
-  * then fail this attach_task(), to avoid breaking top_cpuset.count.
-  */
- if (tsk->flags & PF_EXITING) {
-  task_unlock(tsk);
-  mutex_unlock(&callback_mutex);
-  put_task_struct(tsk);
-  return -ESRCH;
- }
- atomic_inc(&cs->count);
- rcu_assign_pointer(tsk->cpuset, cs);
- task_unlock(tsk);
+ return security_task_setscheduler(tsk, 0, NULL);
+}

+void cpuset_attach_task(struct container *cont, struct task_struct *tsk)
+{
+ cpumask_t cpus;
+ struct cpuset *cs = cont->cpuset;
  guarantee_online_cpus(cs, &cpus);
```

```
  set_cpus_allowed(tsk, cpus);
+}
+
+void cpuset_post_attach_task(struct container *cont,
+        struct container *oldcont,
+        struct task_struct *tsk)
+{
+ nodemask_t from, to;
+ struct mm_struct *mm;
+ struct cpuset *cs = cont->cpuset;
+ struct cpuset *oldcs = oldcont->cpuset;

  from = oldcs->mems_allowed;
  to = cs->mems_allowed;
-
- mutex_unlock(&callback_mutex);
-
  mm = get_task_mm(tsk);
  if (mm) {
   mpol_rebind_mm(mm, &to);
@@ -1256,43 +832,35 @@ static int attach_task(struct cpuset *cs
   mmput(mm);
  }

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

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

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

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

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

   /* +1 for nul-terminator */
@@ -1305,9 +873,9 @@ static ssize_t cpuset_common_file_write(
   }
   buffer[nbytes] = 0; /* nul-terminate */

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

- if (is_removed(cs)) {
+ if (container_is_removed(cont)) {
    retval = -ENODEV;
    goto out2;
   }
@@ -1325,9 +893,6 @@ 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 +910,6 @@ 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 +918,12 @@ static ssize_t cpuset_common_file_write(
  if (retval == 0)
   retval = nbytes;
 out2:
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
+ container_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_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
@@ -1396,9 +940,9 @@ static int cpuset_sprintf_cpulist(char *
 {
  cpumask_t mask;

- mutex_lock(&callback_mutex);
+ container_lock();
  mask = cs->cpus_allowed;
- mutex_unlock(&callback_mutex);
+ container_unlock();
```

```
  return cpulist_scnprintf(page, PAGE_SIZE, mask);
 }
@@ -1407,18 +951,20 @@ static int cpuset_sprintf_memlist(char *
 {
  nodemask_t mask;

- mutex_lock(&callback_mutex);
+ container_lock();
  mask = cs->mems_allowed;
- mutex_unlock(&callback_mutex);
+ container_unlock();

  return nodelist_scnprintf(page, PAGE_SIZE, mask);
 }

-static ssize_t cpuset_common_file_read(struct file *file, char __user *buf,
-    size_t nbytes, loff_t *ppos)
+static ssize_t cpuset_common_file_read(struct container *cont,
+           struct cftype *cft,
+           struct file *file,
+           char __user *buf,
+           size_t nbytes, loff_t *ppos)
 {
- struct cftype *cft = __d_cft(file->f_dentry);
- struct cpuset *cs = __d_cs(file->f_dentry->d_parent);
+ struct cpuset *cs = cont->cpuset;
  cpuset_filetype_t type = cft->private;
  char *page;
  ssize_t retval = 0;
@@ -1442,9 +988,6 @@ 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 +1015,97 @@ 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_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_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_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 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_dentry->d_parent);
- struct ctr_struct *ctr;
- pid_t *pidarray;
- int npids;
- char c;
-
- if (!(file->f_mode & FMODE_READ))
-  return 0;
-
- ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
- if (!ctr)
-  goto err0;
-
- /*
-  * If cpuset gets more users after we read count, we won't have
-  * enough space - tough.  This race is indistinguishable to the
```

```
-  * caller from the case that the additional cpuset users didn't
-  * show up until sometime later on.
-  */
- npids = atomic_read(&cs->count);
- pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
- if (!pidarray)
-  goto err1;
-
- npids = pid_array_load(pidarray, npids, cs);
- sort(pidarray, npids, sizeof(pid_t), cmppid, NULL);
-
- /* Call pid_array_to_buf() twice, first just to get bufsz */
- ctr->bufsz = pid_array_to_buf(&c, sizeof(c), pidarray, npids) + 1;
- ctr->buf = kmalloc(ctr->bufsz, GFP_KERNEL);
- if (!ctr->buf)
-  goto err2;
- ctr->bufsz = pid_array_to_buf(ctr->buf, ctr->bufsz, pidarray, npids);
-
- kfree(pidarray);
- file->private_data = ctr;
- return 0;
-
-err2:
- kfree(pidarray);
-err1:
- kfree(ctr);
-err0:
- return -ENOMEM;
-}
-
-static ssize_t cpuset_tasks_read(struct file *file, char __user *buf,
-      size_t nbytes, loff_t *ppos)
-{
- struct ctr_struct *ctr = file->private_data;
-
- if (*ppos + nbytes > ctr->bufsz)
-  nbytes = ctr->bufsz - *ppos;
- if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
-  return -EFAULT;
- *ppos += nbytes;
- return nbytes;
-}
-
-static int cpuset_tasks_release(struct inode *unused_inode, struct file *file)
-{
- struct ctr_struct *ctr;
-
- if (file->f_mode & FMODE_READ) {
```

```
-  ctr = file->private_data;
-  kfree(ctr->buf);
-  kfree(ctr);
- }
- return 0;
-}

 /*
  * for the common functions, 'private' gives the type of file
  */

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

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

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

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

-static struct cftype cft_notify_on_release = {
```

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

 static struct cftype cft_memory_pressure_enabled = {
  .name = "memory_pressure_enabled",
+ .read = cpuset_common_file_read,
+ .write = cpuset_common_file_write,
  .private = FILE_MEMORY_PRESSURE_ENABLED,
 };

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

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

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

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

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

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

@@ -1869,66 +1118,31 @@ static int cpuset_populate_dir(struct de
  * Must be called with the mutex on the parent inode held
  */

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

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

- mutex_lock(&manage_mutex);
  cpuset_update_task_memory_state();
  cs->flags = 0;
```

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

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

```
- /* the vfs holds inode->i_mutex already */
- return cpuset_create(c_parent, dentry->d_name.name, mode | S_IFDIR);
 }

 /*
@@ -1942,49 +1156,16 @@ static int cpuset_mkdir(struct inode *di
  * nesting would risk an ABBA deadlock.
  */

-static int cpuset_rmdir(struct inode *unused_dir, struct dentry *dentry)
+void cpuset_destroy(struct container *cont)
 {
- struct cpuset *cs = dentry->d_fsdata;
- struct dentry *d;
- struct cpuset *parent;
- char *pathbuf = NULL;
-
- /* the vfs holds both inode->i_mutex already */
+ struct cpuset *cs = cont->cpuset;

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

```
  number_of_cpusets--;
- mutex_unlock(&callback_mutex);
- if (list_empty(&parent->children))
-  check_for_release(parent, &pathbuf);
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
- return 0;
 }

 /*
@@ -1995,10 +1176,10 @@ static int cpuset_rmdir(struct inode *un

 int __init cpuset_init_early(void)
 {
- struct task_struct *tsk = current;
-
- tsk->cpuset = &top_cpuset;
- tsk->cpuset->mems_generation = cpuset_mems_generation++;
+ struct container *cont = current->container;
+ cont->cpuset = &top_cpuset;
+ top_cpuset.container = cont;
+ cont->cpuset->mems_generation = cpuset_mems_generation++;
  return 0;
 }

@@ -2010,39 +1191,19 @@ int __init cpuset_init_early(void)

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

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

- init_task.cpuset = &top_cpuset;
-
  err = register_filesystem(&cpuset_fs_type);
  if (err < 0)
-  goto out;
- cpuset_mount = kern_mount(&cpuset_fs_type);
- if (IS_ERR(cpuset_mount)) {
-  printk(KERN_ERR "cpuset: could not mount!\n");
-  err = PTR_ERR(cpuset_mount);
```

```
-  cpuset_mount = NULL;
-  goto out;
- }
- root = cpuset_mount->mnt_sb->s_root;
- root->d_fsdata = &top_cpuset;
- inc_nlink(root->d_inode);
- top_cpuset.dentry = root;
- root->d_inode->i_op = &cpuset_dir_inode_operations;
+  return err;
+
  number_of_cpusets = 1;
- err = cpuset_populate_dir(root);
- /* memory_pressure_enabled is in root cpuset only */
- if (err == 0)
-  err = cpuset_add_file(root, &cft_memory_pressure_enabled);
-out:
- return err;
+ return 0;
 }

 #if defined(CONFIG_HOTPLUG_CPU) || defined(CONFIG_MEMORY_HOTPLUG)
@@ -2069,10 +1230,12 @@ out:

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

  /* Each of our child cpusets mems must be online */
- list_for_each_entry(c, &cur->children, sibling) {
+ list_for_each_entry(c, &cur->container->children, sibling) {
+  c = container_cs(cont);
   guarantee_online_cpus_mems_in_subtree(c);
   if (!cpus_empty(c->cpus_allowed))
    guarantee_online_cpus(c, &c->cpus_allowed);
@@ -2099,15 +1262,15 @@ static void guarantee_online_cpus_mems_i

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

  guarantee_online_cpus_mems_in_subtree(&top_cpuset);
  top_cpuset.cpus_allowed = cpu_online_map;
  top_cpuset.mems_allowed = node_online_map;
```

```
- mutex_unlock(&callback_mutex);
- mutex_unlock(&manage_mutex);
+ container_unlock();
+ container_manage_unlock();
 }
 #endif

@@ -2158,111 +1321,6 @@ 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.
  *
@@ -2276,11 +1334,11 @@ cpumask_t cpuset_cpus_allowed(struct tas
 {
  cpumask_t mask;

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

  return mask;
 }
@@ -2304,11 +1362,11 @@ nodemask_t cpuset_mems_allowed(struct ta
 {
  nodemask_t mask;

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

```
- mutex_unlock(&callback_mutex);
+ container_unlock();

  return mask;
 }
@@ -2408,45 +1466,18 @@ int __cpuset_zone_allowed(struct zone *z
   return 1;

  /* Not hardwall and node outside mems_allowed: scan up cpusets */
- mutex_lock(&callback_mutex);
+ container_lock();

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

  allowed = node_isset(node, cs->mems_allowed);
- mutex_unlock(&callback_mutex);
+ container_unlock();
  return allowed;
 }

 /**
- * cpuset_lock - lock out any changes to cpuset structures
- *
- * The out of memory (oom) code needs to mutex_lock cpusets
- * from being changed while it scans the tasklist looking for a
- * task in an overlapping cpuset.  Expose callback_mutex via this
- * cpuset_lock() routine, so the oom code can lock it, before
- * locking the task list.  The tasklist_lock is a spinlock, so
- * must be taken inside callback_mutex.
- */
-
-void cpuset_lock(void)
-{
- mutex_lock(&callback_mutex);
-}
-
-/**
- * cpuset_unlock - release lock on cpuset changes
- *
- * Undo the lock taken in a previous cpuset_lock() call.
- */
-
-void cpuset_unlock(void)
-{
- mutex_unlock(&callback_mutex);
```

```
-}
-
-/**
  * cpuset_mem_spread_node() - On which node to begin search for a page
  *
  * If a task is marked PF_SPREAD_PAGE or PF_SPREAD_SLAB (as for
@@ -2506,7 +1537,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock(current);
   goto done;
  }
- cs1 = nearest_exclusive_ancestor(current->cpuset);
+ cs1 = nearest_exclusive_ancestor(current->container->cpuset);
  task_unlock(current);

  task_lock((struct task_struct *)p);
@@ -2514,7 +1545,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock((struct task_struct *)p);
   goto done;
  }
- cs2 = nearest_exclusive_ancestor(p->cpuset);
+ cs2 = nearest_exclusive_ancestor(p->container->cpuset);
  task_unlock((struct task_struct *)p);

  overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
@@ -2553,11 +1584,12 @@ void __cpuset_memory_pressure_bump(void)
  struct cpuset *cs;

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

+#ifdef CONFIG_PROC_PID_CPUSET
 /*
  * proc_cpuset_show()
  *  - Print tasks cpuset path into seq_file.
@@ -2588,15 +1620,15 @@ static int proc_cpuset_show(struct seq_f
   goto out_free;

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

- retval = cpuset_path(tsk->cpuset, buf, PAGE_SIZE);
+ retval = container_path(tsk->container, buf, PAGE_SIZE);
  if (retval < 0)
```

```
   goto out_unlock;
  seq_puts(m, buf);
  seq_putc(m, '\n');
 out_unlock:
- mutex_unlock(&manage_mutex);
+ container_manage_unlock();
  put_task_struct(tsk);
 out_free:
  kfree(buf);
@@ -2616,6 +1648,7 @@ struct file_operations proc_cpuset_opera
  .llseek  = seq_lseek,
  .release = single_release,
 };
+#endif /* CONFIG_PROC_PID_CPUSET */

 /* Display task cpus_allowed, mems_allowed in /proc/<pid>/status file. */
 char *cpuset_task_status_allowed(struct task_struct *task, char *buffer)
Index: container-2.6.19-rc5/init/Kconfig
 ===================================================================== =======
--- container-2.6.19-rc5.orig/init/Kconfig
+++ container-2.6.19-rc5/init/Kconfig
@@ -239,17 +239,24 @@ config IKCONFIG_PROC
    through /proc/config.gz.

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

-   Say N if unsure
+config MAX_CONTAINER_SUBSYS
+ int "Number of container subsystems to support"
+ depends on CONTAINERS
+ range 1 255
+ default 8
+
+config MAX_CONTAINER_HIERARCHIES
+ int "Number of container hierarchies to support"
+ depends on CONTAINERS
+ range 2 255
+ default 4

 config CPUSETS
  bool "Cpuset support"
  depends on SMP
```

```
+ select CONTAINERS
  help
    This option will let you create and manage CPUSETs which
    allow dynamically partitioning a system into sets of CPUs and
@@ -258,6 +265,10 @@ config CPUSETS

    Say N if unsure.

+config PROC_PID_CPUSET
+ bool "Include legacy /proc/<pid>/cpuset file"
+ depends on CPUSETS
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
Index: container-2.6.19-rc5/mm/oom_kill.c
 ===================================================================
--- container-2.6.19-rc5.orig/mm/oom_kill.c
+++ container-2.6.19-rc5/mm/oom_kill.c
@@ -395,7 +395,7 @@ void out_of_memory(struct zonelist *zone
   show_mem();
  }

- cpuset_lock();
+ container_lock();
  read_lock(&tasklist_lock);

  /*
@@ -429,7 +429,7 @@ retry:
   /* Found nothing?!?! Either we hang forever, or we panic. */
   if (!p) {
    read_unlock(&tasklist_lock);
-   cpuset_unlock();
+   container_unlock();
    panic("Out of memory and no killable processes...\n");
   }

@@ -441,7 +441,7 @@ retry:

 out:
  read_unlock(&tasklist_lock);
- cpuset_unlock();
+ container_unlock();

  /*
   * Give "p" a good chance of killing itself before we
Index: container-2.6.19-rc5/include/linux/sched.h
 ===================================================================
```

```
--- container-2.6.19-rc5.orig/include/linux/sched.h
+++ container-2.6.19-rc5/include/linux/sched.h
@@ -720,7 +720,6 @@ extern unsigned int max_cache_size;

 struct io_context;   /* See blkdev.h */
 struct container;
-struct cpuset;
 #define NGROUPS_SMALL  32
 #define NGROUPS_PER_BLOCK ((int)(PAGE_SIZE / sizeof(gid_t)))
 struct group_info {
@@ -1001,7 +1000,6 @@ 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;
@@ -1432,7 +1430,7 @@ static inline int thread_group_empty(str
 /*
  * Protects ->fs, ->files, ->mm, ->group_info, ->comm, keyring
  * subscriptions and synchronises with wait4().  Also used in procfs.  Also
- * pins the final release of task.io_context.  Also protects ->cpuset.
+ * pins the final release of task.io_context.  Also protects ->container.
  *
  * Nests both inside and outside of read_lock(&tasklist_lock).
  * It must not be nested with write_lock_irq(&tasklist_lock),
Index: container-2.6.19-rc5/Documentation/cpusets.txt
 ===================================================================== =======
--- container-2.6.19-rc5.orig/Documentation/cpusets.txt
+++ container-2.6.19-rc5/Documentation/cpusets.txt
@@ -7,6 +7,7 @@ Written by Simon.Derr@bull.net
 Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
 Modified by Paul Jackson <pj@sgi.com>
 Modified by Christoph Lameter <clameter@sgi.com>
+Modified by Paul Menage <menage@google.com>

 CONTENTS:
 =========
@@ -16,10 +17,9 @@ CONTENTS:
   1.2 Why are cpusets needed ?
   1.3 How are cpusets implemented ?
   1.4 What are exclusive cpusets ?
-  1.5 What does notify_on_release do ?
-  1.6 What is memory_pressure ?
-  1.7 What is memory spread ?
-  1.8 How do I use cpusets ?
+  1.5 What is memory_pressure ?
```

+ 1.6 What is memory spread ?
+ 1.7 How do I use cpusets ?
 2. Usage Examples and Syntax
   2.1 Basic Usage
   2.2 Adding/removing cpus
@@ -43,18 +43,19 @@ hierarchy visible in a virtual file syst
 hooks, beyond what is already present, required to manage dynamic
 job placement on large systems.

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

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

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

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

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

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

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

-1.5 What does notify_on_release do ?

-
-If the notify_on_release flag is enabled (1) in a cpuset, then whenever
-the last task in the cpuset leaves (exits or attaches to some other
-cpuset) and the last child cpuset of that cpuset is removed, then
-the kernel runs the command /sbin/cpuset_release_agent, supplying the

-pathname (relative to the mount point of the cpuset file system) of the
-abandoned cpuset. This enables automatic removal of abandoned cpusets.
-The default value of notify_on_release in the root cpuset at system
-boot is disabled (0). The default value of other cpusets at creation
-is the current value of their parents notify_on_release setting.
-
-
-1.6 What is memory_pressure ?
+1.5 What is memory_pressure ?

 The memory_pressure of a cpuset provides a simple per-cpuset metric
 of the rate that the tasks in a cpuset are attempting to free up in
@@ -324,7 +305,7 @@ the tasks in the cpuset, in units of rec
 times 1000.

-1.7 What is memory spread ?
+1.6 What is memory spread ?

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

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

 In order to minimize the impact of cpusets on critical kernel
@@ -485,7 +466,7 @@ than stress the kernel.
 To start a new job that is to be contained within a cpuset, the steps are:

   1) mkdir /dev/cpuset
- 2) mount -t cpuset none /dev/cpuset
+ 2) mount -t container none /dev/cpuset
   3) Create the new cpuset by doing mkdir's and write's (or echo's) in
      the /dev/cpuset virtual file system.
   4) Start a task that will be the "founding father" of the new job.
@@ -497,7 +478,7 @@ For example, the following sequence of c
 named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
 and then start a subshell 'sh' in that cpuset:

- mount -t cpuset none /dev/cpuset
+ mount -t container none /dev/cpuset
    cd /dev/cpuset
    mkdir Charlie
    cd Charlie

```
@@ -507,7 +488,7 @@ and then start a subshell 'sh' in that c
   sh
   # The subshell 'sh' is now running in cpuset Charlie
   # The next line should display '/Charlie'
-  cat /proc/self/cpuset
+  cat /proc/self/container

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

 To mount it, type:
-# mount -t cpuset none /dev/cpuset
+# mount -t container none /dev/cpuset

 Then under /dev/cpuset you can find a tree that corresponds to the
 tree of the cpusets in the system. For instance, /dev/cpuset
Index: container-2.6.19-rc5/fs/super.c
 ===================================================================
--- container-2.6.19-rc5.orig/fs/super.c
+++ container-2.6.19-rc5/fs/super.c
@@ -39,11 +39,6 @@
 #include <linux/mutex.h>
 #include <asm/uaccess.h>

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

Index: container-2.6.19-rc5/include/linux/fs.h
 ===================================================================
--- container-2.6.19-rc5.orig/include/linux/fs.h
+++ container-2.6.19-rc5/include/linux/fs.h
@@ -1875,6 +1875,8 @@ extern int vfs_fstat(unsigned int, struc

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

+extern void get_filesystem(struct file_system_type *fs);
+extern void put_filesystem(struct file_system_type *fs);
 extern struct file_system_type *get_fs_type(const char *name);
 extern struct super_block *get_super(struct block_device *);
 extern struct super_block *user_get_super(dev_t);
Index: container-2.6.19-rc5/include/linux/mempolicy.h
```

```
 ===================================================== =======
--- container-2.6.19-rc5.orig/include/linux/mempolicy.h
+++ container-2.6.19-rc5/include/linux/mempolicy.h
@@ -152,7 +152,7 @@ extern void mpol_fix_fork_child_flag(str

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

--
```

Subject: [PATCH 3/7] Add generic multi-subsystem API to containers
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:51 GMT
View Forum Message <> Reply to Message

This patch removes all cpuset-specific knowlege from the container system, replacing it with a generic API that can be used by multiple subsystems. Cpusets is adapted to be a container subsystem.

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

```
---
 Documentation/containers.txt |  233 ++++++++++++-
 include/linux/container.h    |  132 +++++++
 include/linux/cpuset.h       |   16
 include/linux/mempolicy.h    |   12
 include/linux/sched.h        |    4
 kernel/container.c           |  716 +++++++++++++++++++++++++++++++++++--------
 kernel/cpuset.c              |  165 ++++++---
 mm/mempolicy.c               |    2
 8 files changed, 1048 insertions(+), 232 deletions(-)

Index: container-2.6.19-rc6/include/linux/container.h
 ===================================================== =======
--- container-2.6.19-rc6.orig/include/linux/container.h
+++ container-2.6.19-rc6/include/linux/container.h
@@ -14,8 +14,6 @@

 #ifdef CONFIG_CONTAINERS

-extern int number_of_containers; /* How many containers are defined in system? */
-
 extern int container_init_early(void);
```

```
 extern int container_init(void);
 extern void container_init_smp(void);
@@ -30,6 +28,68 @@ extern void container_unlock(void);
 extern void container_manage_lock(void);
 extern void container_manage_unlock(void);

+struct containerfs_root;
+
+/* Per-subsystem/per-container state maintained by the system. */
+struct container_subsys_state {
+ /* The container that this subsystem is attached to. Useful
+  * for subsystems that want to know about the container
+  * hierarchy structure */
+ struct container *container;
+
+ /* State maintained by the container system to allow
+  * subsystems to be "busy". Should be accessed via css_get()
+  * and css_put() */
+ spinlock_t refcnt_lock;
+ atomic_t refcnt;
+};
+
+/*
+ * Call css_get() to hold a reference on the container; following a
+ * return of 0, this container subsystem state object is guaranteed
+ * not to be destroyed until css_put() is called on it.  A non-zero
+ * return code indicates that a reference could not be taken.
+ *
+ */
+
+static inline int css_get(struct container_subsys_state *css)
+{
+ int retval = 0;
+ unsigned long flags;
+ /* Synchronize with container_rmdir() */
+ spin_lock_irqsave(&css->refcnt_lock, flags);
+ if (atomic_read(&css->refcnt) >= 0) {
+  /* Container is still alive */
+  atomic_inc(&css->refcnt);
+ } else {
+  /* Container removal is in progress */
+  retval = -EINVAL;
+ }
+ spin_unlock_irqrestore(&css->refcnt_lock, flags);
+ return retval;
+}
+
+/*
```

```
+ * If you are holding current->alloc_lock then it's impossible for you
+ * to be moved out of your container, and hence it's impossible for
+ * your container to be destroyed. Therefore doing a simple
+ * atomic_inc() on a css is safe.
+ */
+
+static inline void css_get_current(struct container_subsys_state *css)
+{
+ atomic_inc(&css->refcnt);
+}
+
+/*
+ * css_put() should be called to release a reference taken by
+ * css_get() or css_get_current()
+ */
+
+static inline void css_put(struct container_subsys_state *css) {
+ atomic_dec(&css->refcnt);
+}
+
 struct container {
  unsigned long flags;  /* "unsigned long" so bitops work */

@@ -46,11 +106,15 @@ struct container {
  struct list_head children; /* my children */

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

-#ifdef CONFIG_CPUSETS
- struct cpuset *cpuset;
-#endif
+ /* Private pointers for each registered subsystem */
+ struct container_subsys_state *subsys[CONFIG_MAX_CONTAINER_SUBSYS];
+
+ int hierarchy;
+
+ struct containerfs_root *root;
+ struct container *top_container;
 };

 /* struct cftype:
@@ -67,8 +131,9 @@ struct container {
  */

 struct inode;
+#define MAX_CFTYPE_NAME 64
```

```
 struct cftype {
- char *name;
+ char name[MAX_CFTYPE_NAME];
  int private;
  int (*open) (struct inode *inode, struct file *file);
  ssize_t (*read) (struct container *cont, struct cftype *cft,
@@ -87,6 +152,59 @@ void container_set_release_agent_path(co

 int container_path(const struct container *cont, char *buf, int buflen);

+/* Container subsystem type. See Documentation/containers.txt for details */
+
+struct container_subsys {
+ int (*create)(struct container_subsys *ss,
+        struct container *cont);
+ void (*destroy)(struct container_subsys *ss, struct container *cont);
+ int (*can_attach)(struct container_subsys *ss,
+     struct container *cont, struct task_struct *tsk);
+ void (*attach)(struct container_subsys *ss, struct container *cont,
+   struct container *old_cont, struct task_struct *tsk);
+ void (*post_attach)(struct container_subsys *ss,
+       struct container *cont,
+       struct container *old_cont,
+       struct task_struct *tsk);
+ void (*fork)(struct container_subsys *ss, struct task_struct *task);
+ void (*exit)(struct container_subsys *ss, struct task_struct *task);
+ int (*populate)(struct container_subsys *ss,
+   struct container *cont);
+
+ int subsys_id;
+#define MAX_CONTAINER_TYPE_NAMELEN 32
+ const char *name;
+
+ /* Protected by RCU */
+ int hierarchy;
+
+ struct list_head sibling;
+};
+
+int container_register_subsys(struct container_subsys *subsys);
+
+static inline struct container_subsys_state *container_subsys_state(
+ struct container *cont,
+ struct container_subsys *ss)
+{
+ return cont->subsys[ss->subsys_id];
+}
+
```

```
+static inline struct container* task_container(struct task_struct *task,
+           struct container_subsys *ss)
+{
+ return rcu_dereference(task->container[ss->hierarchy]);
+}
+
+static inline struct container_subsys_state *task_subsys_state(
+ struct task_struct *task,
+ struct container_subsys *ss)
+{
+ return container_subsys_state(task_container(task, ss), ss);
+}
+
+int container_path(const struct container *cont, char *buf, int buflen);
+
 #else /* !CONFIG_CONTAINERS */

 static inline int container_init_early(void) { return 0; }
Index: container-2.6.19-rc6/include/linux/cpuset.h
 ========================================================== =======
--- container-2.6.19-rc6.orig/include/linux/cpuset.h
+++ container-2.6.19-rc6/include/linux/cpuset.h
@@ -60,16 +60,7 @@ static inline int cpuset_do_slab_mem_spr

 extern void cpuset_track_online_nodes(void);

-extern int cpuset_can_attach_task(struct container *cont,
-      struct task_struct *tsk);
-extern void cpuset_attach_task(struct container *cont,
-    struct task_struct *tsk);
-extern void cpuset_post_attach_task(struct container *cont,
-        struct container *oldcont,
-        struct task_struct *tsk);
-extern int cpuset_populate_dir(struct container *cont);
-extern int cpuset_create(struct container *cont);
-extern void cpuset_destroy(struct container *cont);
+extern int current_cpuset_is_being_rebound(void);

 #else /* !CONFIG_CPUSETS */

@@ -131,6 +122,11 @@ static inline int cpuset_do_slab_mem_spr

 static inline void cpuset_track_online_nodes(void) {}

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

```
+
 #endif /* !CONFIG_CPUSETS */

 #endif /* _LINUX_CPUSET_H */
Index: container-2.6.19-rc6/kernel/container.c
 ===================================================================== =======
--- container-2.6.19-rc6.orig/kernel/container.c
+++ container-2.6.19-rc6/kernel/container.c
@@ -55,7 +55,6 @@
 #include <linux/time.h>
 #include <linux/backing-dev.h>
 #include <linux/sort.h>
-#include <linux/cpuset.h>

 #include <asm/uaccess.h>
 #include <asm/atomic.h>
@@ -63,12 +62,47 @@

 #define CONTAINER_SUPER_MAGIC  0x27e0eb

-/*
- * Tracks how many containers are currently defined in system.
- * When there is only one container (the root container) we can
- * short circuit some hooks.
+static struct container_subsys *subsys[CONFIG_MAX_CONTAINER_SUBSYS];
+static int subsys_count = 0;
+
+struct containerfs_root {
+ struct super_block *sb;
+ unsigned long subsys_bits;
+ struct list_head subsys_list;
+ struct container top_container;
+ /*
+  * Tracks how many containers are currently defined in system.
+  * When there is only one container (the root container) we can
+  * short circuit some hooks.
+  */
+ int number_of_containers;
+ struct vfsmount *pin_mount;
+};
+
+/* The set of hierarchies in use. Hierarchy 0 is the "dummy
+ * container", reserved for the subsystems that are otherwise
+ * unattached - it never has more than a single container, and all
+ * tasks are part of that container. */
+
+static struct containerfs_root rootnode[CONFIG_MAX_CONTAINER_HIERARCHIES];
+
```

```
+/* dummytop is a shorthand for the dummy hierarchy's top container */
+#define dummytop (&rootnode[0].top_container)
+
+/* This flag indicates whether tasks in the fork and exit paths should
+ * take callback_mutex and check for fork/exit handlers to call. This
+ * avoids us having to take locks in the fork/exit path if none of the
+ * subsystems need to be called.
+ *
+ * It is protected via RCU, with the invariant that a process in an
+ * rcu_read_lock() section will never see this as 0 if there are
+ * actually registered subsystems with a fork or exit
+ * handler. (Sometimes it may be 1 without there being any registered
+ * subsystems with such a handler, but such periods are safe and of
+ * short duration).
  */
-int number_of_containers __read_mostly;
+
+static int need_forkexit_callback = 0;

 /* bits in struct container flags field */
 typedef enum {
@@ -87,11 +121,8 @@ static inline int notify_on_release(cons
  return test_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
 }

-static struct container top_container = {
- .count = ATOMIC_INIT(0),
- .sibling = LIST_HEAD_INIT(top_container.sibling),
- .children = LIST_HEAD_INIT(top_container.children),
-};
+#define for_each_subsys(_hierarchy, _ss) list_for_each_entry(_ss,
&rootnode[_hierarchy].subsys_list, sibling)
+

 /* The path to use for release notifications. No locking between
  * setting and use - so if userspace updates this while subcontainers
@@ -105,9 +136,6 @@ void container_set_release_agent_path(co
  container_manage_unlock();
 }

-static struct vfsmount *container_mount;
-static struct super_block *container_sb;
-
 /*
  * We have two global container mutexes below.  They can nest.
  * It is ok to first take manage_mutex, then nest callback_mutex.  We also
@@ -202,15 +230,18 @@ static DEFINE_MUTEX(callback_mutex);
```

```
 static int container_mkdir(struct inode *dir, struct dentry *dentry, int mode);
 static int container_rmdir(struct inode *unused_dir, struct dentry *dentry);
+static int container_populate_dir(struct container *cont);
+static struct inode_operations container_dir_inode_operations;
+struct file_operations proc_containerstats_operations;

 static struct backing_dev_info container_backing_dev_info = {
  .ra_pages = 0,  /* No readahead */
  .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
 };

-static struct inode *container_new_inode(mode_t mode)
+static struct inode *container_new_inode(mode_t mode, struct super_block *sb)
 {
- struct inode *inode = new_inode(container_sb);
+ struct inode *inode = new_inode(sb);

  if (inode) {
   inode->i_mode = mode;
@@ -282,32 +313,102 @@ static void container_d_remove_dir(struc
  remove_dir(dentry);
 }

+/*
+ * Release the last use of a hierarchy.  Will never be called when
+ * there are active subcontainers since each subcontainer bumps the
+ * value of sb->s_active.
+ */
+
+static void container_put_super(struct super_block *sb) {
+
+ struct containerfs_root *root = sb->s_fs_info;
+ int hierarchy = root->top_container.hierarchy;
+ int i;
+ struct container *cont = &root->top_container;
+ struct task_struct *g, *p;
+
+ root->sb = NULL;
+ sb->s_fs_info = NULL;
+
+ mutex_lock(&manage_mutex);
+
+ BUG_ON(root->number_of_containers != 1);
+ BUG_ON(!list_empty(&cont->children));
+ BUG_ON(!list_empty(&cont->sibling));
+ BUG_ON(!root->subsys_bits);
+
+ mutex_lock(&callback_mutex);
```

```
+
+ /* Remove all tasks from this container hierarchy */
+ read_lock(&tasklist_lock);
+ do_each_thread(g, p) {
+  task_lock(p);
+  BUG_ON(!p->container[hierarchy]);
+  BUG_ON(p->container[hierarchy] != cont);
+  rcu_assign_pointer(p->container[hierarchy], NULL);
+  task_unlock(p);
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+ atomic_set(&cont->count, 1);
+
+ /* Remove all subsystems from this hierarchy */
+ for (i = 0; i < subsys_count; i++) {
+  if (root->subsys_bits & (1 << i)) {
+   struct container_subsys *ss = subsys[i];
+   BUG_ON(cont->subsys[i] != dummytop->subsys[i]);
+   BUG_ON(cont->subsys[i]->container != cont);
+   dummytop->subsys[i]->container = dummytop;
+   cont->subsys[i] = NULL;
+   rcu_assign_pointer(subsys[i]->hierarchy, 0);
+   list_del(&ss->sibling);
+  } else {
+   BUG_ON(cont->subsys[i]);
+  }
+ }
+ root->subsys_bits = 0;
+ mutex_unlock(&callback_mutex);
+ synchronize_rcu();
+
+ mutex_unlock(&manage_mutex);
+}
+
+static int container_show_options(struct seq_file *seq, struct vfsmount *vfs) {
+ struct containerfs_root *root = vfs->mnt_sb->s_fs_info;
+ struct container_subsys *ss;
+ for_each_subsys(root->top_container.hierarchy, ss) {
+  seq_printf(seq, ",%s", ss->name);
+ }
+ return 0;
+}
+
 static struct super_operations container_ops = {
  .statfs = simple_statfs,
  .drop_inode = generic_delete_inode,
+ .put_super = container_put_super,
+ .show_options = container_show_options,
```

```
 };

-static int container_fill_super(struct super_block *sb, void *unused_data,
-       int unused_silent)
+static int container_fill_super(struct super_block *sb, void *options,
+    int unused_silent)
 {
  struct inode *inode;
  struct dentry *root;
+ struct containerfs_root *hroot = options;

  sb->s_blocksize = PAGE_CACHE_SIZE;
  sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
  sb->s_magic = CONTAINER_SUPER_MAGIC;
  sb->s_op = &container_ops;
- container_sb = sb;

- inode = container_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR);
- if (inode) {
-  inode->i_op = &simple_dir_inode_operations;
-  inode->i_fop = &simple_dir_operations;
-  /* directories start off with i_nlink == 2 (for "." entry) */
-  inode->i_nlink++;
- } else {
+ inode = container_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR, sb);
+ if (!inode)
   return -ENOMEM;
- }
+
+ inode->i_op = &simple_dir_inode_operations;
+ inode->i_fop = &simple_dir_operations;
+ inode->i_op = &container_dir_inode_operations;
+ /* directories start off with i_nlink == 2 (for "." entry) */
+ inc_nlink(inode);

  root = d_alloc_root(inode);
  if (!root) {
@@ -315,6 +416,12 @@ static int container_fill_super(struct s
   return -ENOMEM;
  }
  sb->s_root = root;
+ root->d_fsdata = &hroot->top_container;
+ hroot->top_container.dentry = root;
+
+ sb->s_fs_info = hroot;
+ hroot->sb = sb;
+
  return 0;
```

```
 }

@@ -322,7 +429,130 @@ static int container_get_sb(struct file_
     int flags, const char *unused_dev_name,
     void *data, struct vfsmount *mnt)
 {
- return get_sb_single(fs_type, flags, data, container_fill_super, mnt);
+ int i;
+ struct container_subsys *ss;
+ char *token, *o = data ?: "all";
+ unsigned long subsys_bits = 0;
+ int ret = 0;
+ struct containerfs_root *root = NULL;
+ int hierarchy;
+
+ mutex_lock(&manage_mutex);
+
+ /* First find the desired set of resource controllers */
+ while ((token = strsep(&o, ",")) != NULL) {
+  if (!*token) {
+   ret = -EINVAL;
+   goto out_unlock;
+  }
+  if (!strcmp(token, "all")) {
+   subsys_bits = (1 << subsys_count) - 1;
+  } else {
+   for (i = 0; i < subsys_count; i++) {
+    ss = subsys[i];
+    if (!strcmp(token, ss->name)) {
+     subsys_bits |= 1 << i;
+     break;
+    }
+   }
+   if (i == subsys_count) {
+    ret = -ENOENT;
+    goto out_unlock;
+   }
+  }
+ }
+
+ /* See if we already have a hierarchy containing this set */
+
+ for (i = 1; i < CONFIG_MAX_CONTAINER_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_CONTAINER_HIERARCHIES) {
+  /* No existing hierarchy matched this set - but we
+   * know that all the subsystems are free */
+  for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+   root = &rootnode[i];
+   if (!root->sb && !root->subsys_bits) break;
+  }
+ }
+
+ if (i == CONFIG_MAX_CONTAINER_HIERARCHIES) {
+  ret = -ENOSPC;
+  goto out_unlock;
+ }
+
+ hierarchy = i;
+
+ if (!root->sb) {
+  /* We need a new superblock for this container combination */
+  struct container *cont = &root->top_container;
+  struct container_subsys *ss;
+  struct task_struct *p, *g;
+
+  BUG_ON(root->subsys_bits);
+  root->subsys_bits = subsys_bits;
+  ret = get_sb_nodev(fs_type, flags, root,
+        container_fill_super, mnt);
+  if (ret)
+   goto out_unlock;
+
+  BUG_ON(!list_empty(&cont->sibling));
+  BUG_ON(!list_empty(&cont->children));
+  BUG_ON(root->number_of_containers != 1);
+
+  mutex_lock(&callback_mutex);
+
+  /* Add all tasks into this container hierarchy */
+  atomic_set(&cont->count, 1);
+  read_lock(&tasklist_lock);
+  do_each_thread(g, p) {
+   task_lock(p);
+   BUG_ON(p->container[hierarchy]);
+   rcu_assign_pointer(p->container[hierarchy], cont);
+   if (!(p->flags & PF_EXITING)) {
```

```
+    atomic_inc(&cont->count);
+   }
+   task_unlock(p);
+  } while_each_thread(g, p);
+  read_unlock(&tasklist_lock);
+
+  /* Move all the relevant subsystems into the hierarchy. */
+  for (i = 0; i < subsys_count; i++) {
+   if (!(subsys_bits & (1 << i))) continue;
+
+   ss = subsys[i];
+
+   BUG_ON(cont->subsys[i]);
+   BUG_ON(dummytop->subsys[i]->container != dummytop);
+   cont->subsys[i] = dummytop->subsys[i];
+   cont->subsys[i]->container = cont;
+   list_add(&ss->sibling, &root->subsys_list);
+   rcu_assign_pointer(subsys[i]->hierarchy,
+         hierarchy);
+  }
+  mutex_unlock(&callback_mutex);
+  synchronize_rcu();
+
+  container_populate_dir(cont);
+
+ } 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 container_fs_type = {
@@ -501,6 +731,8 @@ static int attach_task(struct container
  struct task_struct *tsk;
  struct container *oldcont;
  int retval = 0;
+ struct container_subsys *ss;
+ int h = cont->hierarchy;

  if (sscanf(pidbuf, "%d", &pid) != 1)
   return -EIO;
@@ -527,37 +759,45 @@ static int attach_task(struct container
```

```
   get_task_struct(tsk);
  }

-#ifdef CONFIG_CPUSETS
- retval = cpuset_can_attach_task(cont, tsk);
-#endif
- if (retval) {
-  put_task_struct(tsk);
-  return retval;
+ for_each_subsys(h, ss) {
+  if (ss->can_attach) {
+   retval = ss->can_attach(ss, cont, tsk);
+   if (retval) {
+    put_task_struct(tsk);
+    return retval;
+   }
+  }
  }

  mutex_lock(&callback_mutex);

  task_lock(tsk);
- oldcont = tsk->container;
+ oldcont = tsk->container[h];
  if (!oldcont) {
   task_unlock(tsk);
   mutex_unlock(&callback_mutex);
   put_task_struct(tsk);
   return -ESRCH;
  }
+        BUG_ON(oldcont == dummytop);
+
  atomic_inc(&cont->count);
- rcu_assign_pointer(tsk->container, cont);
+ rcu_assign_pointer(tsk->container[h], cont);
  task_unlock(tsk);

-#ifdef CONFIG_CPUSETS
- cpuset_attach_task(cont, tsk);
-#endif
+ for_each_subsys(h, ss) {
+  if (ss->attach) {
+   ss->attach(ss, cont, oldcont, tsk);
+  }
+ }

  mutex_unlock(&callback_mutex);
```

```
-#ifdef CONFIG_CPUSETS
- cpuset_post_attach_task(cont, oldcont, tsk);
-#endif
+ for_each_subsys(h, ss) {
+  if (ss->post_attach) {
+   ss->post_attach(ss, cont, oldcont, tsk);
+  }
+ }

  put_task_struct(tsk);
  synchronize_rcu();
@@ -780,7 +1020,7 @@ static struct inode_operations container
  .rename = container_rename,
 };

-static int container_create_file(struct dentry *dentry, int mode)
+static int container_create_file(struct dentry *dentry, int mode, struct super_block *sb)
 {
  struct inode *inode;

@@ -789,7 +1029,7 @@ static int container_create_file(struct
  if (dentry->d_inode)
   return -EEXIST;

- inode = container_new_inode(mode);
+ inode = container_new_inode(mode, sb);
  if (!inode)
   return -ENOMEM;

@@ -798,7 +1038,7 @@ static int container_create_file(struct
   inode->i_fop = &simple_dir_operations;

   /* start off with i_nlink == 2 (for "." entry) */
-  inode->i_nlink++;
+  inc_nlink(inode);
  } else if (S_ISREG(mode)) {
   inode->i_size = 0;
   inode->i_fop = &container_file_operations;
@@ -828,10 +1068,10 @@ static int container_create_dir(struct c
  dentry = container_get_dentry(parent, name);
  if (IS_ERR(dentry))
   return PTR_ERR(dentry);
- error = container_create_file(dentry, S_IFDIR | mode);
+ error = container_create_file(dentry, S_IFDIR | mode, cont->root->sb);
  if (!error) {
   dentry->d_fsdata = cont;
-  parent->d_inode->i_nlink++;
+  inc_nlink(parent->d_inode);
```

```
   cont->dentry = dentry;
  }
  dput(dentry);
@@ -848,7 +1088,7 @@ int container_add_file(struct container
  mutex_lock(&dir->d_inode->i_mutex);
  dentry = container_get_dentry(dir, cft->name);
  if (!IS_ERR(dentry)) {
-  error = container_create_file(dentry, 0644 | S_IFREG);
+  error = container_create_file(dentry, 0644 | S_IFREG, cont->root->sb);
   if (!error)
    dentry->d_fsdata = (void *)cft;
   dput(dentry);
@@ -894,7 +1134,7 @@ static int pid_array_load(pid_t *pidarra
  read_lock(&tasklist_lock);

  do_each_thread(g, p) {
-  if (p->container == cont) {
+  if (p->container[cont->hierarchy] == cont) {
    pidarray[n++] = p->pid;
    if (unlikely(n == npids))
     goto array_full;
@@ -1037,21 +1277,33 @@ static struct cftype cft_release_agent =
 static int container_populate_dir(struct container *cont)
 {
  int err;
+ struct container_subsys *ss;

  if ((err = container_add_file(cont, &cft_notify_on_release)) < 0)
   return err;
  if ((err = container_add_file(cont, &cft_tasks)) < 0)
   return err;
- if ((cont == &top_container) &&
+ if ((cont == cont->top_container) &&
      (err = container_add_file(cont, &cft_release_agent)) < 0)
   return err;
-#ifdef CONFIG_CPUSETS
- if ((err = cpuset_populate_dir(cont)) < 0)
-  return err;
-#endif
+
+ for_each_subsys(cont->hierarchy, ss) {
+  if (ss->populate && (err = ss->populate(ss, cont)) < 0)
+   return err;
+ }
+
  return 0;
 }
```

```
+static void init_container_css(struct container_subsys *ss,
+          struct container *cont)
+{
+ struct container_subsys_state *css = cont->subsys[ss->subsys_id];
+ css->container = cont;
+ spin_lock_init(&css->refcnt_lock);
+ atomic_set(&css->refcnt, 0);
+}
+
 /*
  * container_create - create a container
  * parent: container that will be parent of the new container.
@@ -1064,13 +1316,24 @@ static int container_populate_dir(struct
 static long container_create(struct container *parent, const char *name, int mode)
 {
  struct container *cont;
- int err;
+ struct containerfs_root *root = parent->root;
+ int err = 0;
+ struct container_subsys *ss;
+ struct super_block *sb = root->sb;

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

+ /* Grab a reference on the superblock so the hierarchy doesn't
+  * get deleted on unmount if there are child containers.  This
+  * can be done outside 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);
+
  cont->flags = 0;
  if (notify_on_release(parent))
   set_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
@@ -1079,16 +1342,18 @@ static long container_create(struct cont
  INIT_LIST_HEAD(&cont->children);

  cont->parent = parent;
+ cont->root = parent->root;
+ cont->hierarchy = parent->hierarchy;

-#ifdef CONFIG_CPUSETS
- err = cpuset_create(cont);
- if (err)
```

```
-  goto err_unlock_free;
-#endif
+ for_each_subsys(cont->hierarchy, ss) {
+  err = ss->create(ss, cont);
+  if (err) goto err_destroy;
+  init_container_css(ss, cont);
+ }

  mutex_lock(&callback_mutex);
  list_add(&cont->sibling, &cont->parent->children);
- number_of_containers++;
+ root->number_of_containers++;
  mutex_unlock(&callback_mutex);

  err = container_create_dir(cont, name, mode);
@@ -1107,15 +1372,23 @@ static long container_create(struct cont
  return 0;

  err_remove:
-#ifdef CONFIG_CPUSETS
- cpuset_destroy(cont);
-#endif
+
  mutex_lock(&callback_mutex);
  list_del(&cont->sibling);
- number_of_containers--;
+ root->number_of_containers--;
  mutex_unlock(&callback_mutex);
- err_unlock_free:
+
+ err_destroy:
+
+ for_each_subsys(cont->hierarchy, ss) {
+  if (cont->subsys[ss->subsys_id])
+   ss->destroy(ss, cont);
+ }
+
  mutex_unlock(&manage_mutex);
+
+ deactivate_super(sb);
+
  kfree(cont);
  return err;
 }
@@ -1145,6 +1418,11 @@ static int container_rmdir(struct inode
  struct dentry *d;
  struct container *parent;
  char *pathbuf = NULL;
```

```
+ struct container_subsys *ss;
+ struct super_block *sb;
+ struct containerfs_root *root;
+ unsigned long flags;
+ int css_busy = 0;

  /* the vfs holds both inode->i_mutex already */

@@ -1157,7 +1435,32 @@ static int container_rmdir(struct inode
   mutex_unlock(&manage_mutex);
   return -EBUSY;
  }
+
  parent = cont->parent;
+ root = cont->root;
+ sb = root->sb;
+
+ local_irq_save(flags);
+ /* Check each container, locking the refcnt lock and testing
+  * the refcnt. This will lock out any calls to css_get() */
+ for_each_subsys(root->top_container.hierarchy, ss) {
+  struct container_subsys_state *css;
+  css = cont->subsys[ss->subsys_id];
+  spin_lock(&css->refcnt_lock);
+  css_busy += atomic_read(&css->refcnt);
+ }
+ /* Go through and release all the locks; if we weren't busy,
+  * the set the refcount to -1 to prevent css_get() from adding
+  * a refcount */
+ for_each_subsys(root->top_container.hierarchy, ss) {
+  struct container_subsys_state *css;
+  css = cont->subsys[ss->subsys_id];
+  if (!css_busy) atomic_dec(&css->refcnt);
+  spin_unlock(&css->refcnt_lock);
+ }
+ local_irq_restore(flags);
+ if (css_busy) return -EBUSY;
+
  mutex_lock(&callback_mutex);
  set_bit(CONT_REMOVED, &cont->flags);
  list_del(&cont->sibling); /* delete my sibling from parent->children */
@@ -1165,67 +1468,142 @@ static int container_rmdir(struct inode
  d = dget(cont->dentry);
  cont->dentry = NULL;
  spin_unlock(&d->d_lock);
+
+ for_each_subsys(root->top_container.hierarchy, ss) {
+  ss->destroy(ss, cont);
```

```
+ }
  container_d_remove_dir(d);
  dput(d);
- number_of_containers--;
+ root->number_of_containers--;
  mutex_unlock(&callback_mutex);
-#ifdef CONFIG_CPUSETS
- cpuset_destroy(cont);
-#endif
+
  if (list_empty(&parent->children))
   check_for_release(parent, &pathbuf);
+
  mutex_unlock(&manage_mutex);
+ /* Drop the active superblock reference that we took when we
+  * created the container */
+ deactivate_super(sb);
  container_release_agent(pathbuf);
  return 0;
 }

-/*
- * container_init_early - probably not needed yet, but will be needed
- * once cpusets are hooked into this code
- */
+
+/**
+ * container_init_early - initialize containers at system boot
+ *
+ * Description: Initialize the container housekeeping structures
+ **/

 int __init container_init_early(void)
 {
- struct task_struct *tsk = current;
+ int i;
+
+ for (i = 0; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+  struct containerfs_root *root = &rootnode[i];
+  struct container *cont = &root->top_container;
+  INIT_LIST_HEAD(&root->subsys_list);
+  root->number_of_containers = 1;
+
+  cont->root = root;
+  cont->hierarchy = i;
+  INIT_LIST_HEAD(&cont->sibling);
+  INIT_LIST_HEAD(&cont->children);
+  cont->top_container = cont;
```

```
+  atomic_set(&cont->count, 1);
+ }
+ init_task.container[0] = &rootnode[0].top_container;

- tsk->container = &top_container;
  return 0;
 }

 /**
- * container_init - initialize containers at system boot
- *
- * Description: Initialize top_container and the container internal file system,
+ * container_init - register container filesystem and /proc file
  **/

 int __init container_init(void)
 {
- struct dentry *root;
  int err;
-
- init_task.container = &top_container;
+ struct proc_dir_entry *entry;

  err = register_filesystem(&container_fs_type);
  if (err < 0)
   goto out;
- container_mount = kern_mount(&container_fs_type);
- if (IS_ERR(container_mount)) {
-  printk(KERN_ERR "container: could not mount!\n");
-  err = PTR_ERR(container_mount);
-  container_mount = NULL;
-  goto out;
- }
- root = container_mount->mnt_sb->s_root;
- root->d_fsdata = &top_container;
- root->d_inode->i_nlink++;
- top_container.dentry = root;
- root->d_inode->i_op = &container_dir_inode_operations;
- number_of_containers = 1;
- err = container_populate_dir(&top_container);
+
+ entry = create_proc_entry("containers", 0, NULL);
+ if (entry)
+  entry->proc_fops = &proc_containerstats_operations;
+
 out:
  return err;
 }
```

```
+int container_register_subsys(struct container_subsys *new_subsys) {
+ int retval = 0;
+ int i;
+
+ BUG_ON(new_subsys->hierarchy);
+ mutex_lock(&manage_mutex);
+ if (subsys_count == CONFIG_MAX_CONTAINER_SUBSYS) {
+  retval = -ENOSPC;
+  goto out;
+ }
+ if (!new_subsys->name ||
+     (strlen(new_subsys->name) > MAX_CONTAINER_TYPE_NAMELEN) ||
+     !new_subsys->create || !new_subsys->destroy) {
+  retval = -EINVAL;
+  goto out;
+ }
+ for (i = 0; i < subsys_count; i++) {
+  if (!strcmp(subsys[i]->name, new_subsys->name)) {
+   retval = -EEXIST;
+   goto out;
+  }
+ }
+
+ new_subsys->subsys_id = subsys_count;
+ retval = new_subsys->create(new_subsys, dummytop);
+ if (retval) {
+  new_subsys->subsys_id = -1;
+  goto out;
+ }
+ init_container_css(new_subsys, dummytop);
+  mutex_lock(&callback_mutex);
+  /* If this is the first subsystem that requested a fork or
+   * exit callback, tell our fork/exit hooks that they need to
+   * grab callback_mutex on every invocation. If they are
+   * running concurrently with this code, they will either not
+   * see the change now and go straight on, or they will see it
+   * and grab callback_mutex, which will deschedule them. Either
+   * way once synchronize_rcu() returns we know that all current
+   * and future forks will make the callbacks. */
+  if (!need_forkexit_callback &&
+      (new_subsys->fork || new_subsys->exit)) {
+   need_forkexit_callback = 1;
+   synchronize_rcu();
+  }
+
+  /* If this subsystem requested that it be notified with fork
+   * events, we should send it one now for every process in the
```

```
+   * system */
+  if (new_subsys->fork) {
+   struct task_struct *g, *p;
+
+   read_lock(&tasklist_lock);
+   do_each_thread(g, p) {
+    new_subsys->fork(new_subsys, p);
+   } while_each_thread(g, p);
+   read_unlock(&tasklist_lock);
+  }
+
+ subsys[subsys_count++] = new_subsys;
+  mutex_unlock(&callback_mutex);
+ out:
+  mutex_unlock(&manage_mutex);
+  return retval;
+
++}
+
 /**
  * container_fork - attach newly forked task to its parents container.
  * @tsk: pointer to task_struct of forking parent process.
@@ -1246,10 +1624,38 @@ out:

 void container_fork(struct task_struct *child)
 {
+ int i, need_callback;
+
+ rcu_read_lock();
+ /* need_forkexit_callback will be true if we might need to do
+   * a callback */
+ need_callback = rcu_dereference(need_forkexit_callback);
+ if (need_callback) {
+  rcu_read_unlock();
+  mutex_lock(&callback_mutex);
+ }
  task_lock(current);
- child->container = current->container;
- atomic_inc(&child->container->count);
+        /* Skip hierarchy 0 since it's permanent */
+ for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+  struct container *cont = current->container[i];
+  if (!cont) continue;
+  child->container[i] = cont;
+  atomic_inc(&cont->count);
+ }
+ if (need_callback) {
+  for (i = 0; i < subsys_count; i++) {
```

```
+   struct container_subsys *ss = subsys[i];
+   if (ss->fork) {
+    ss->fork(ss, child);
+   }
+  }
+ }
  task_unlock(current);
+ if (need_callback) {
+  mutex_unlock(&callback_mutex);
+ } else {
+  rcu_read_unlock();
+ }
 }

 /**
@@ -1314,20 +1720,38 @@ void container_fork(struct task_struct *
 void container_exit(struct task_struct *tsk)
 {
  struct container *cont;
-
- cont = tsk->container;
- tsk->container = &top_container; /* the_top_container_hack - see above */
-
- if (notify_on_release(cont)) {
-  char *pathbuf = NULL;
-
-  mutex_lock(&manage_mutex);
-  if (atomic_dec_and_test(&cont->count))
-   check_for_release(cont, &pathbuf);
-  mutex_unlock(&manage_mutex);
-  container_release_agent(pathbuf);
+ int i;
+ rcu_read_lock();
+ if (rcu_dereference(need_forkexit_callback)) {
+  rcu_read_unlock();
+  mutex_lock(&callback_mutex);
+  for (i = 0; i < subsys_count; i++) {
+   struct container_subsys *ss = subsys[i];
+   if (ss->exit) {
+    ss->exit(ss, tsk);
+   }
+  }
+  mutex_unlock(&callback_mutex);
  } else {
-  atomic_dec(&cont->count);
+  rcu_read_unlock();
+ }
+
```

```
+ /* Skip hierarchy 0 since it's permanent */
+ for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+  cont = tsk->container[i];
+  if (!cont) continue;
+  /* the_top_container_hack - see above */
+  tsk->container[i] = cont->top_container;
+  if (notify_on_release(cont)) {
+   char *pathbuf = NULL;
+   mutex_lock(&manage_mutex);
+   if (atomic_dec_and_test(&cont->count))
+    check_for_release(cont, &pathbuf);
+   mutex_unlock(&manage_mutex);
+   container_release_agent(pathbuf);
+  } else {
+   atomic_dec(&cont->count);
+  }
  }
 }

@@ -1387,12 +1811,15 @@ void container_manage_unlock(void)
  *    the_top_container_hack in container_exit(), which sets an exiting tasks
  *    container to top_container.
  */
+
+/* TODO: Use a proper seq_file iterator */
 static int proc_container_show(struct seq_file *m, void *v)
 {
  struct pid *pid;
  struct task_struct *tsk;
  char *buf;
  int retval;
+ int i;

  retval = -ENOMEM;
  buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
@@ -1405,14 +1832,25 @@ static int proc_container_show(struct se
  if (!tsk)
   goto out_free;

- retval = -EINVAL;
+ retval = 0;
+
  mutex_lock(&manage_mutex);

- retval = container_path(tsk->container, buf, PAGE_SIZE);
- if (retval < 0)
-  goto out_unlock;
- seq_puts(m, buf);
```

```
- seq_putc(m, '\n');
+ for (i = 1; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+  struct containerfs_root *root = &rootnode[i];
+  struct container_subsys *ss;
+  int count = 0;
+  if (!root->subsys_bits) continue;
+  for_each_subsys(i, ss) {
+   seq_printf(m, "%s%s", count++ ? "," : "", ss->name);
+  }
+  seq_putc(m, ':');
+  retval = container_path(tsk->container[i], buf, PAGE_SIZE);
+  if (retval < 0)
+   goto out_unlock;
+  seq_puts(m, buf);
+  seq_putc(m, '\n');
+ }
 out_unlock:
  mutex_unlock(&manage_mutex);
  put_task_struct(tsk);
@@ -1434,3 +1872,47 @@ struct file_operations proc_container_op
  .llseek  = seq_lseek,
  .release = single_release,
 };
+
+static int proc_containerstats_show(struct seq_file *m, void *v)
+{
+ int i;
+ mutex_lock(&manage_mutex);
+ seq_puts(m, "Hierarchies:\n");
+ for (i = 0; i < CONFIG_MAX_CONTAINER_HIERARCHIES; i++) {
+  struct containerfs_root *root = &rootnode[i];
+  struct container_subsys *ss;
+  int first = 1;
+  seq_printf(m, "%d: topcount=%d bits=%lx containers=%d (",
+      i, atomic_read(&root->top_container.count),
+      root->subsys_bits, root->number_of_containers);
+  for_each_subsys(i, ss) {
+   seq_printf(m, "%s%s", first ? "" : ", ", ss->name);
+   first = false;
+  }
+  seq_putc(m, ')');
+  if (root->sb) {
+   seq_printf(m, " s_active=%d", atomic_read(&root->sb->s_active));
+  }
+  seq_putc(m, '\n');
+ }
+ seq_puts(m, "Subsystems:\n");
+ for (i = 0; i < subsys_count; i++) {
```

```
+  struct container_subsys *ss = subsys[i];
+  seq_printf(m, "%d: name=%s hierarchy=%d\n",
+       i, ss->name, ss->hierarchy);
+ }
+ mutex_unlock(&manage_mutex);
+ return 0;
+}
+
+static int containerstats_open(struct inode *inode, struct file *file)
+{
+ return single_open(file, proc_containerstats_show, 0);
+}
+
+struct file_operations proc_containerstats_operations = {
+ .open = containerstats_open,
+ .read = seq_read,
+ .llseek = seq_lseek,
+ .release = single_release,
+};
Index: container-2.6.19-rc6/kernel/cpuset.c
 ===================================================================== =======
--- container-2.6.19-rc6.orig/kernel/cpuset.c
+++ container-2.6.19-rc6/kernel/cpuset.c
@@ -5,6 +5,7 @@
  *
  *  Copyright (C) 2003 BULL SA.
  *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *  Copyright (C) 2006 Google, Inc
  *
  *  Portions derived from Patrick Mochel's sysfs code.
  *  sysfs is Copyright (c) 2001-3 Patrick Mochel
@@ -12,6 +13,7 @@
  *  2003-10-10 Written by Simon Derr.
  *  2003-10-22 Updates by Stephen Hemminger.
  *  2004 May-July Rework by Paul Jackson.
+ *  2006 Rework by Paul Menage to use generic containers
  *
  *  This file is subject to the terms and conditions of the GNU General Public
  *  License.  See the file COPYING in the main directory of the Linux
@@ -61,6 +63,10 @@
  */
 int number_of_cpusets __read_mostly;

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

```
 struct fmeter {
@@ -71,11 +77,12 @@ struct fmeter {
 };

 struct cpuset {
+ struct container_subsys_state css;
+
  unsigned long flags;  /* "unsigned long" so bitops work */
  cpumask_t cpus_allowed;  /* CPUs allowed to tasks in cpuset */
  nodemask_t mems_allowed; /* Memory Nodes allowed to tasks */

- struct container *container;    /* Task container */
  struct cpuset *parent;  /* my parent */

  /*
@@ -87,6 +94,26 @@ struct cpuset {
  struct fmeter fmeter;  /* memory_pressure filter */
 };

+/* Update the cpuset for a container */
+static inline void set_container_cs(struct container *cont, struct cpuset *cs)
+{
+ cont->subsys[cpuset_subsys.subsys_id] = &cs->css;
+}
+
+/* Retrieve the cpuset for a container */
+static inline struct cpuset *container_cs(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, &cpuset_subsys),
+       struct cpuset, css);
+}
+
+/* Retrieve the cpuset for a task */
+static inline struct cpuset *task_cs(struct task_struct *task)
+{
+ return container_cs(task_container(task, &cpuset_subsys));
+}
+
+
 /* bits in struct cpuset flags field */
 typedef enum {
  CS_CPU_EXCLUSIVE,
@@ -162,7 +189,7 @@ static int cpuset_get_sb(struct file_sys
  if (container_fs) {
   ret = container_fs->get_sb(container_fs, flags,
         unused_dev_name,
-        data, mnt);
```

```
+        "cpuset", mnt);
   put_filesystem(container_fs);
  }
  return ret;
@@ -270,20 +297,19 @@ void cpuset_update_task_memory_state(voi
  struct task_struct *tsk = current;
  struct cpuset *cs;

- if (tsk->container->cpuset == &top_cpuset) {
+ if (task_cs(tsk) == &top_cpuset) {
   /* Don't need rcu for top_cpuset.  It's never freed. */
   my_cpusets_mem_gen = top_cpuset.mems_generation;
  } else {
   rcu_read_lock();
-  cs = rcu_dereference(tsk->container->cpuset);
-  my_cpusets_mem_gen = cs->mems_generation;
+  my_cpusets_mem_gen = task_cs(current)->mems_generation;
   rcu_read_unlock();
  }

  if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
   container_lock();
   task_lock(tsk);
-  cs = tsk->container->cpuset; /* Maybe changed when task not locked */
+  cs = task_cs(tsk); /* Maybe changed when task not locked */
   guarantee_online_mems(cs, &tsk->mems_allowed);
   tsk->cpuset_mems_generation = cs->mems_generation;
   if (is_spread_page(cs))
@@ -342,9 +368,8 @@ static int validate_change(const struct
  struct cpuset *c, *par;

  /* Each of our child cpusets must be a subset of us */
- list_for_each_entry(cont, &cur->container->children, sibling) {
-  c = cont->cpuset;
-  if (!is_cpuset_subset(c, trial))
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+  if (!is_cpuset_subset(container_cs(cont), trial))
    return -EBUSY;
  }

@@ -357,8 +382,8 @@ static int validate_change(const struct
   return -EACCES;

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

```
   if ((is_cpu_exclusive(trial) || is_cpu_exclusive(c)) &&
       c != cur &&
       cpus_intersects(trial->cpus_allowed, c->cpus_allowed))
@@ -400,8 +425,8 @@ static void update_cpu_domains(struct cp
   * children
   */
  pspan = par->cpus_allowed;
- list_for_each_entry(cont, &par->container->children, sibling) {
-  c = cont->cpuset;
+ list_for_each_entry(cont, &par->css.container->children, sibling) {
+  c = container_cs(cont);
   if (is_cpu_exclusive(c))
    cpus_andnot(pspan, pspan, c->cpus_allowed);
  }
@@ -418,8 +443,8 @@ static void update_cpu_domains(struct cp
    * Get all cpus from current cpuset's cpus_allowed not part
    * of exclusive children
    */
-  list_for_each_entry(cont, &cur->container->children, sibling) {
-   c = cont->cpuset;
+  list_for_each_entry(cont, &cur->css.container->children, sibling) {
+   c = container_cs(cont);
    if (is_cpu_exclusive(c))
     cpus_andnot(cspan, cspan, c->cpus_allowed);
   }
@@ -507,7 +532,7 @@ static void cpuset_migrate_mm(struct mm_
  do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);

  container_lock();
- guarantee_online_mems(tsk->container->cpuset, &tsk->mems_allowed);
+ guarantee_online_mems(task_cs(tsk),&tsk->mems_allowed);
  container_unlock();
 }

@@ -525,6 +550,8 @@ 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;
@@ -542,7 +569,7 @@ static int update_nodemask(struct cpuset
   return -EACCES;

  trialcs = *cs;
- cont = cs->container;
```

```
+ cont = cs->css.container;
  retval = nodelist_parse(buf, trialcs.mems_allowed);
  if (retval < 0)
   goto done;
@@ -565,7 +592,7 @@ static int update_nodemask(struct cpuset
  cs->mems_generation = cpuset_mems_generation++;
  container_unlock();

- 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 */
@@ -579,13 +606,13 @@ static int update_nodemask(struct cpuset
   * enough mmarray[] w/o using GFP_ATOMIC.
   */
  while (1) {
-  ntasks = atomic_read(&cs->container->count);  /* guess */
+  ntasks = atomic_read(&cs->css.container->count);  /* guess */
   ntasks += fudge;
   mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
   if (!mmarray)
    goto done;
   write_lock_irq(&tasklist_lock);  /* block fork */
-  if (atomic_read(&cs->container->count) <= ntasks)
+  if (atomic_read(&cs->css.container->count) <= ntasks)
    break;    /* got enough */
   write_unlock_irq(&tasklist_lock); /* try again */
   kfree(mmarray);
@@ -602,7 +629,7 @@ static int update_nodemask(struct cpuset
     "Cpuset mempolicy rebind incomplete.\n");
    continue;
   }
-  if (p->container != cont)
+  if (task_cs(p) != cs)
    continue;
   mm = get_task_mm(p);
   if (!mm)
@@ -636,12 +663,17 @@ static int update_nodemask(struct cpuset

  /* We're done rebinding vma's to this cpusets new mems_allowed. */
  kfree(mmarray);
- set_cpuset_being_rebound(NULL);
+ cpuset_being_rebound = NULL;
  retval = 0;
 done:
  return retval;
 }
```

```
+int current_cpuset_is_being_rebound(void)
+{
+ return task_cs(current) == cpuset_being_rebound;
+}
+
 /*
  * Call with manage_mutex held.
  */
@@ -795,9 +827,10 @@ static int fmeter_getrate(struct fmeter
  return val;
 }

-int cpuset_can_attach_task(struct container *cont, struct task_struct *tsk)
+int cpuset_can_attach(struct container_subsys *ss,
+       struct container *cont, struct task_struct *tsk)
 {
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);

  if (cpus_empty(cs->cpus_allowed) || nodes_empty(cs->mems_allowed))
   return -ENOSPC;
@@ -805,22 +838,23 @@ int cpuset_can_attach_task(struct contai
  return security_task_setscheduler(tsk, 0, NULL);
 }

-void cpuset_attach_task(struct container *cont, struct task_struct *tsk)
+void cpuset_attach(struct container_subsys *ss, struct container *cont,
+     struct container *old_cont, struct task_struct *tsk)
 {
  cpumask_t cpus;
- struct cpuset *cs = cont->cpuset;
- guarantee_online_cpus(cs, &cpus);
+ guarantee_online_cpus(container_cs(cont), &cpus);
  set_cpus_allowed(tsk, cpus);
 }

-void cpuset_post_attach_task(struct container *cont,
-       struct container *oldcont,
-       struct task_struct *tsk)
+void cpuset_post_attach(struct container_subsys *ss,
+   struct container *cont,
+   struct container *oldcont,
+   struct task_struct *tsk)
 {
  nodemask_t from, to;
  struct mm_struct *mm;
- struct cpuset *cs = cont->cpuset;
```

```
- struct cpuset *oldcs = oldcont->cpuset;
+ struct cpuset *cs = container_cs(cont);
+ struct cpuset *oldcs = container_cs(oldcont);

  from = oldcs->mems_allowed;
  to = cs->mems_allowed;
@@ -854,7 +888,7 @@ static ssize_t cpuset_common_file_write(
      const char __user *userbuf,
      size_t nbytes, loff_t *unused_ppos)
 {
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);
  cpuset_filetype_t type = cft->private;
  char *buffer;
  int retval = 0;
@@ -964,7 +998,7 @@ static ssize_t cpuset_common_file_read(s
           char __user *buf,
           size_t nbytes, loff_t *ppos)
 {
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);
  cpuset_filetype_t type = cft->private;
  char *page;
  ssize_t retval = 0;
@@ -1083,7 +1117,7 @@ static struct cftype cft_spread_slab = {
  .private = FILE_SPREAD_SLAB,
 };

-int cpuset_populate_dir(struct container *cont)
+int cpuset_populate(struct container_subsys *ss, struct container *cont)
 {
  int err;

@@ -1118,11 +1152,19 @@ int cpuset_populate_dir(struct container
  * Must be called with the mutex on the parent inode held
  */

-int cpuset_create(struct container *cont)
+int cpuset_create(struct container_subsys *ss, struct container *cont)
 {
  struct cpuset *cs;
- struct cpuset *parent = cont->parent->cpuset;
+ struct cpuset *parent;

+ if (!cont->parent) {
+  /* This is early initialization for the top container */
+  set_container_cs(cont, &top_cpuset);
+  top_cpuset.css.container = cont;
```

```
+  top_cpuset.mems_generation = cpuset_mems_generation++;
+  return 0;
+ }
+ parent = container_cs(cont->parent);
  cs = kmalloc(sizeof(*cs), GFP_KERNEL);
  if (!cs)
   return -ENOMEM;
@@ -1139,8 +1181,8 @@ int cpuset_create(struct container *cont
  fmeter_init(&cs->fmeter);

  cs->parent = parent;
- cont->cpuset = cs;
- cs->container = cont;
+ set_container_cs(cont, cs);
+ cs->css.container = cont;
  number_of_cpusets++;
  return 0;
 }
@@ -1156,9 +1198,9 @@ int cpuset_create(struct container *cont
  * nesting would risk an ABBA deadlock.
  */

-void cpuset_destroy(struct container *cont)
+void cpuset_destroy(struct container_subsys *ss, struct container *cont)
 {
- struct cpuset *cs = cont->cpuset;
+ struct cpuset *cs = container_cs(cont);

  cpuset_update_task_memory_state();
  if (is_cpu_exclusive(cs)) {
@@ -1166,8 +1208,20 @@ void cpuset_destroy(struct container *co
   BUG_ON(retval);
  }
  number_of_cpusets--;
+ kfree(cs);
 }

+static struct container_subsys cpuset_subsys = {
+ .name = "cpuset",
+ .create = cpuset_create,
+ .destroy  = cpuset_destroy,
+ .can_attach = cpuset_can_attach,
+ .attach = cpuset_attach,
+ .post_attach = cpuset_post_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
@@ -1176,13 +1230,13 @@ void cpuset_destroy(struct container *co

 int __init cpuset_init_early(void)
 {
- struct container *cont = current->container;
- cont->cpuset = &top_cpuset;
- top_cpuset.container = cont;
- cont->cpuset->mems_generation = cpuset_mems_generation++;
+ if (container_register_subsys(&cpuset_subsys) < 0)
+  panic("Couldn't register cpuset subsystem");
+ top_cpuset.mems_generation = cpuset_mems_generation++;
  return 0;
 }

+
 /**
  * cpuset_init - initialize cpusets at system boot
  *
@@ -1192,6 +1246,7 @@ int __init cpuset_init_early(void)
 int __init cpuset_init(void)
 {
  int err = 0;
+
  top_cpuset.cpus_allowed = CPU_MASK_ALL;
  top_cpuset.mems_allowed = NODE_MASK_ALL;

@@ -1234,7 +1289,7 @@ static void guarantee_online_cpus_mems_i
  struct cpuset *c;

  /* Each of our child cpusets mems must be online */
- list_for_each_entry(c, &cur->container->children, sibling) {
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
   c = container_cs(cont);
   guarantee_online_cpus_mems_in_subtree(c);
   if (!cpus_empty(c->cpus_allowed))
@@ -1336,7 +1391,7 @@ cpumask_t cpuset_cpus_allowed(struct tas

  container_lock();
  task_lock(tsk);
- guarantee_online_cpus(tsk->container->cpuset, &mask);
+ guarantee_online_cpus(task_cs(tsk), &mask);
  task_unlock(tsk);
  container_unlock();

@@ -1364,7 +1419,7 @@ nodemask_t cpuset_mems_allowed(struct ta
```

```
  container_lock();
  task_lock(tsk);
- guarantee_online_mems(tsk->container->cpuset, &mask);
+ guarantee_online_mems(task_cs(tsk), &mask);
  task_unlock(tsk);
  container_unlock();

@@ -1469,7 +1524,7 @@ int __cpuset_zone_allowed(struct zone *z
  container_lock();

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

  allowed = node_isset(node, cs->mems_allowed);
@@ -1537,7 +1592,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock(current);
   goto done;
  }
- cs1 = nearest_exclusive_ancestor(current->container->cpuset);
+ cs1 = nearest_exclusive_ancestor(task_cs(current));
  task_unlock(current);

  task_lock((struct task_struct *)p);
@@ -1545,7 +1600,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock((struct task_struct *)p);
   goto done;
  }
- cs2 = nearest_exclusive_ancestor(p->container->cpuset);
+ cs2 = nearest_exclusive_ancestor(task_cs((struct task_struct *)p));
  task_unlock((struct task_struct *)p);

  overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
@@ -1581,11 +1636,8 @@ int cpuset_memory_pressure_enabled __rea

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

```
@@ -1622,7 +1674,8 @@ static int proc_cpuset_show(struct seq_f
  retval = -EINVAL;
  container_manage_lock();

- retval = container_path(tsk->container, buf, PAGE_SIZE);
+ retval = container_path(tsk->container[cpuset_subsys.hierarchy],
+    buf, PAGE_SIZE);
  if (retval < 0)
   goto out_unlock;
  seq_puts(m, buf);
Index: container-2.6.19-rc6/Documentation/containers.txt
 ========================================================== =======
--- container-2.6.19-rc6.orig/Documentation/containers.txt
+++ container-2.6.19-rc6/Documentation/containers.txt
@@ -17,12 +17,16 @@ CONTENTS:
   1.2 Why are containers needed ?
   1.3 How are containers implemented ?
   1.4 What does notify_on_release do ?
-  1.5 How do I use containers ?
+  1.5 What do the xxx_enabled files do ?
+  1.6 How do I use containers ?
 2. Usage Examples and Syntax
   2.1 Basic Usage
   2.2 Attaching processes
-3. Questions
-4. Contact
+3. Kernel API
+  3.1 Overview
+  3.2 Synchronization
+  3.3 Subsystem API
+4. Questions

 1. Containers
 ==========
@@ -31,13 +35,17 @@ CONTENTS:
 ----------------------

 Containers provide a mechanism for aggregating sets of tasks, and all
-their children, into hierarchical groups.
-
-Each task has a pointer to a container.  Multiple tasks may reference
-the same container. User level code may create and destroy containers
-by name in the container virtual file system, specify and query to
-which container a task is assigned, and list the task pids assigned to
-a container.
+their children, into hierarchical groups. A container associates a set
+of tasks with a set of parameters for one or more "subsystems"
+(typically resource controllers).
```

+
+At any one time there may be up to CONFIG_MAX_CONTAINER_HIERARCHIES
+active hierachies of tasks. Each task has a pointer to a container in
+each active hierarchy. Multiple tasks may reference the same
+container. User level code may create and destroy containers by name
+in an instance of the container virtual file system, specify and query
+to which container a task is assigned, and list the task pids assigned
+to a container.

 On their own, the only use for containers is for simple job
 tracking. The intention is that other subsystems, such as cpusets (see
@@ -67,27 +75,43 @@ desired.

 Containers extends the kernel as follows:

- - Each task in the system is attached to a container, via a pointer
-   in the task structure to a reference counted container structure.
- - The hierarchy of containers can be mounted at /dev/container (or
-   elsewhere), for browsing and manipulation from user space.
+ - Each task in the system has set of reference-counted container
+   pointers, one for each active hierarchy
+ - A container hierarchy filesystem can be mounted for browsing and
+   manipulation from user space.
  - You can list all the tasks (by pid) attached to any container.

 The implementation of containers requires a few, simple hooks
 into the rest of the kernel, none in performance critical paths:

- - in init/main.c, to initialize the root container at system boot.
- - in fork and exit, to attach and detach a task from its container.
+ - in init/main.c, to initialize the root containers at system boot.
+ - in fork and exit, to attach and detach a task from its containers.

 In addition a new file system, of type "container" may be mounted,
-typically at /dev/container, to enable browsing and modifying the containers
-presently known to the kernel. No new system calls are added for
-containers - all support for querying and modifying containers is via
-this container file system.
-
-Each task under /proc has an added file named 'container', displaying
-the container name, as the path relative to the root of the container file
-system.
+typically at /dev/container, to enable browsing and modifying the
+containers presently known to the kernel. When mounting a container
+hierarchy, you may specify a comma-separated list of subsystems to
+mount as the filesystem mount options. By default, mounting the
+container filesystem attempts to mount a hierarchy containing all
+registered subsystems.

+
+If an active hierarchy with exactly the same set of subsystems already
+exists, it will be reused for the new mount. If no existing hierarchy
+matches, and any of the requested subsystems are in use in an existing
+hierarchy, the mount will fail with -EBUSY. Otherwise, a new hierarchy
+is created, associated with the requested subsystems.
+
+When a container filesystem is unmounted, if there are any
+subcontainers created below the top-level container, that hierarchy
+will remain active even though unmounted; if there are no
+subcontainers then the hierarchy will be deactivated.
+
+No new system calls are added for containers - all support for
+querying and modifying containers is via this container file system.
+
+Each task under /proc has an added file named 'container' displaying,
+for each active hierarchy, the subsystem names and the container name
+as the path relative to the root of the container file system.

 Each container is represented by a directory in the container file system
 containing the following files describing that container:
@@ -129,7 +153,18 @@ The default value of notify_on_release i
 boot is disabled (0). The default value of other containers at creation
 is the current value of their parents notify_on_release setting.

-1.5 How do I use containers ?
+1.5 What do the xxx_enabled files do ?
+-------------------------------------
+
+In the top-level container directory there are a series of
+<subsys>_enabled files, one for each registered subsystem. Each of
+these files contains 0 or 1 to indicate whether the named container
+subsystem is enabled, and can only be modified when there are no
+subcontainers. Disabled container subsystems don't get new instances
+created when a subcontainer is created; the subsystem-specific state
+is simply inherited from the parent container.
+
+1.6 How do I use containers ?

 To start a new job that is to be contained within a container, the steps are:
@@ -214,8 +249,154 @@ If you have several tasks to attach, you
  ...
 # /bin/echo PIDn > tasks

+3. Kernel API
+=============
+

+3.1 Overview
+------------
+
+Each kernel subsystem that wants to hook into the generic container
+system needs to create a container_subsys object. This contains
+various methods, which are callbacks from the container system, along
+with a subsystem id which will be assigned by the container system.
+
+Other fields in the container_subsys object include:
+
+- subsys_id: a unique array index for the subsystem, indicating which
+ entry in container->subsys[] this subsystem should be
+ managing. Initialized by container_register_subsys(); prior to this
+ it should be initialized to -1
+
+- top_container: the subsystem-specific state representing the root
+ container in the system. This should always exist, even when the
+ subsystem isn't attached to a hierarchy.
+
+- hierarchy: an index indicating which hierarchy, if any, this
+ subsystem is currently attached to. If this is -1, then the
+ subsystem is not attached to any hierarchy, and all tasks should be
+ considered to be members of the subsystem's top_container. It should
+ be initialized to -1.
+
+- name: should be initialized to a unique subsystem name prior to
+ calling container_register_subsystem. Should be no longer than
+ MAX_CONTAINER_TYPE_NAMELEN
+
+Each container object created by the system has an array of pointers,
+indexed by subsystem id; this pointer is entirely managed by the
+subsystem; the generic container code will never touch this pointer.
+
+3.2 Synchronization
+-------------------
+
+There are two global mutexes used by the container system. The first
+is the manage_mutex, which should be taken by anything that wants to
+modify a container; The second if the callback_mutex, which should be
+taken by holders of the manage_mutex at the point when they actually
+make changes, and by callbacks from lower-level subsystems that want
+to ensure that no container changes occur. Note that memory
+allocations cannot be made while holding callback_mutex.
+
+The callback_mutex nests inside the manage_mutex.
+
+In general, the pattern of use is:
+

+1) take manage_mutex
+2) verify that the change is valid and do any necessary allocations\
+3) take callback_mutex
+4) make changes
+5) release callback_mutex
+6) release manage_mutex
+
+See kernel/container.c for more details.
+
+Subsystems can take/release the manage_mutex via the functions
+container_manage_lock()/container_manage_unlock(), and can
+take/release the callback_mutex via the functions
+container_lock()/container_unlock().
+
+Accessing a task's container pointer may be done in the following ways:
+- while holding manage_mutex
+- while holding callback_mutex
+- while holding the task's alloc_lock (via task_lock())
+- inside an rcu_read_lock() section via rcu_dereference()
+
+3.3 Subsystem API
+-------------------------
+
+Each subsystem should call container_register_subsys() with a pointer
+to its subsystem object. This will store the new subsystem id in the
+subsystem subsys_id field and return 0, or a negative error. There's
+currently no facility for deregestering a subsystem nor for
+registering a subsystem after any containers (other than the default
+"top_container") have been created.
+
+Each subsystem may export the following methods. The only mandatory
+methods are create/destroy. Any others that are null are presumed to
+be successful no-ops.
+
+int create(struct container *cont)
+LL=manage_mutex
+
+The subsystem should set its subsystem pointer for the passed
+container, returning 0 on success or a negative error code. On
+success, the subsystem pointer should point to a structure of type
+container_subsys_state (typically embedded in a larger
+subsystem-specific object), which will be initialized by the container
+system.
+
+void destroy(struct container *cont)
+LL=manage_mutex
+
+The container system is about to destroy the passed container; the

+subsystem should do any necessary cleanup
+
+int can_attach(struct container_subsys *ss, struct container *cont,
+ struct task_struct *task)
+LL=manage_mutex
+
+Called prior to moving a task into a container; if the subsystem
+returns an error, this will abort the attach operation. Note that
+this isn't called on a fork.
+
+void attach(struct container_subsys *ss, struct container *cont,
+ struct container *old_cont, struct task_struct *task)
+LL=manage_mutex & callback_mutex
+
+Called during the attach operation. The subsystem should do any
+necessary work that can be accomplished without memory allocations or
+sleeping.
+
+void post_attach(struct container_subsys *ss, struct container *cont,
+ struct container *old_cont, struct task_struct *task)
+LL=manage_mutex
+
+Called after the task has been attached to the container, to allow any
+post-attachment activity that requires memory allocations or blocking.
+
+void fork(struct container_subsy *ss, struct task_struct *task)
+LL=callback_mutex, maybe read_lock(tasklist_lock)
+
+Called when a task is forked into a container. Also called during
+registration for all existing tasks.
+
+void exit(struct container_subsys *ss, struct task_struct *task)
+LL=callback_mutex
+
+Called during task exit
+
+int populate(struct container_subsys *ss, struct container *cont)
+LL=none
+
+Called after creation of a container to allow a subsystem to populate
+the container directory with file entries. The subsystem should make
+calls to container_add_file() with objects of type cftype (see
+include/linux/container.h for details). Called during
+container_register_subsys() to populate the root container. Note that
+although this method can return an error code, the error code is
+currently not always handled well.
+

```
-3. Questions
+4. Questions
 ============

 Q: what's up with this '/bin/echo' ?
Index: container-2.6.19-rc6/include/linux/mempolicy.h
 ===================================================================
--- container-2.6.19-rc6.orig/include/linux/mempolicy.h
+++ container-2.6.19-rc6/include/linux/mempolicy.h
@@ -148,14 +148,6 @@ extern void mpol_rebind_task(struct task
      const nodemask_t *new);
 extern void mpol_rebind_mm(struct mm_struct *mm, nodemask_t *new);
 extern void mpol_fix_fork_child_flag(struct task_struct *p);
-#define set_cpuset_being_rebound(x) (cpuset_being_rebound = (x))
-
-#ifdef CONFIG_CPUSETS
-#define current_cpuset_is_being_rebound() \
- (cpuset_being_rebound == current->container->cpuset)
-#else
-#define current_cpuset_is_being_rebound() 0
-#endif

 extern struct mempolicy default_policy;
 extern struct zonelist *huge_zonelist(struct vm_area_struct *vma,
@@ -173,8 +165,6 @@ static inline void check_highest_zone(en
 int do_migrate_pages(struct mm_struct *mm,
  const nodemask_t *from_nodes, const nodemask_t *to_nodes, int flags);

-extern void *cpuset_being_rebound; /* Trigger mpol_copy vma rebind */
-
 #else

 struct mempolicy {};
@@ -253,8 +243,6 @@ static inline void mpol_fix_fork_child_f
 {
 }

-#define set_cpuset_being_rebound(x) do {} while (0)
-
 static inline struct zonelist *huge_zonelist(struct vm_area_struct *vma,
   unsigned long addr)
 {
Index: container-2.6.19-rc6/include/linux/sched.h
 ===================================================================
--- container-2.6.19-rc6.orig/include/linux/sched.h
+++ container-2.6.19-rc6/include/linux/sched.h
@@ -1005,7 +1005,7 @@ struct task_struct {
  int cpuset_mem_spread_rotor;
```

```
 #endif
 #ifdef CONFIG_CONTAINERS
- struct container *container;
+ struct container *container[CONFIG_MAX_CONTAINER_HIERARCHIES];
 #endif
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
@@ -1430,7 +1430,7 @@ static inline int thread_group_empty(str
 /*
  * Protects ->fs, ->files, ->mm, ->group_info, ->comm, keyring
  * subscriptions and synchronises with wait4().  Also used in procfs.  Also
- * pins the final release of task.io_context.  Also protects ->container.
+ * pins the final release of task.io_context.  Also protects ->container[].
  *
  * Nests both inside and outside of read_lock(&tasklist_lock).
  * It must not be nested with write_lock_irq(&tasklist_lock),
Index: container-2.6.19-rc6/mm/mempolicy.c
 ===================================================================== =======
--- container-2.6.19-rc6.orig/mm/mempolicy.c
+++ container-2.6.19-rc6/mm/mempolicy.c
@@ -1307,7 +1307,6 @@ EXPORT_SYMBOL(alloc_pages_current);
  * keeps mempolicies cpuset relative after its cpuset moves.  See
  * further kernel/cpuset.c update_nodemask().
  */
-void *cpuset_being_rebound;

 /* Slow path of a mempolicy copy */
 struct mempolicy *__mpol_copy(struct mempolicy *old)
@@ -1906,4 +1905,3 @@ out:
   m->version = (vma != priv->tail_vma) ? vma->vm_start : 0;
  return 0;
 }
-

--
```

Subject: [PATCH 5/7] Resource Groups over generic containers
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:53 GMT
View Forum Message <> Reply to Message

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

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

```
 include/linux/moduleparam.h  |   12 -
 include/linux/numtasks.h     |   28 ++
 include/linux/res_group.h    |   87 ++++++++
 include/linux/res_group_rc.h |   97 ++++++++
 init/Kconfig                 |   22 ++
 kernel/Makefile              |    1
 kernel/fork.c                |    7
 kernel/res_group/Makefile    |    2
 kernel/res_group/local.h     |   38 +++
 kernel/res_group/numtasks.c  |  467 ++++++++++++++++++++++++++++++++++++++++++
 kernel/res_group/res_group.c |  160 ++++++++++++++
 kernel/res_group/rgcs.c      |  302 ++++++++++++++++++++++++++
 kernel/res_group/shares.c    |  228 ++++++++++++++++++++
 13 files changed, 1447 insertions(+), 4 deletions(-)

Index: container-2.6.19-rc6/include/linux/moduleparam.h
 ===================================================================== =======
--- container-2.6.19-rc6.orig/include/linux/moduleparam.h
+++ container-2.6.19-rc6/include/linux/moduleparam.h
@@ -75,11 +75,17 @@ struct kparam_array
 /* Helper functions: type is byte, short, ushort, int, uint, long,
    ulong, charp, bool or invbool, or XXX if you define param_get_XXX,
    param_set_XXX and param_check_XXX. */
-#define module_param_named(name, value, type, perm)      \
- param_check_##type(name, &(value));       \
- module_param_call(name, param_set_##type, param_get_##type, &value, perm); \
+#define module_param_named_call(name, value, type, set, perm)  \
+ param_check_##type(name, &(value));    \
+ module_param_call(name, set, param_get_##type, &(value), perm); \
   __MODULE_PARM_TYPE(name, #type)

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

Index: container-2.6.19-rc6/include/linux/numtasks.h
```

```
 ===================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/include/linux/numtasks.h
@@ -0,0 +1,28 @@
+/* numtasks.h - No. of tasks resource controller for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *
+ * Provides No. of tasks resource controller for Resource Groups
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+#ifndef _LINUX_NUMTASKS_H
+#define _LINUX_NUMTASKS_H
+
+#ifdef CONFIG_RES_GROUPS_NUMTASKS
+#include <linux/res_group_rc.h>
+
+extern int numtasks_allow_fork(struct task_struct *);
+
+#else /* CONFIG_RES_GROUPS_NUMTASKS */
+
+#define numtasks_allow_fork(task) (0)
+
+#endif /* CONFIG_RES_GROUPS_NUMTASKS */
+#endif /* _LINUX_NUMTASKS_H */
Index: container-2.6.19-rc6/include/linux/res_group.h
 ===================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/include/linux/res_group.h
@@ -0,0 +1,87 @@
+/*
+ *  res_group.h - Header file to be used by Resource Groups
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003, 2004
+ *  (C) Shailabh Nagar,  IBM Corp. 2003, 2004
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *
+ * Provides data structures, macros and kernel APIs
+ *
+ * More details at http://ckrm.sf.net
+ *
```

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

```
+ int min_shares; /* minimun fraction of parent's resources allowed */
+ int max_shares; /* maximum fraction of parent's resources allowed */
+ int child_shares_divisor; /* >= 1, may not be DONT_CARE */
+
+ /*
+  * share values invisible to userspace.  adjusted when userspace
+  * sets shares
+  */
+ int unused_min_shares;
+  /* 0 <= unused_min_shares <= (child_shares_divisor -
+   *    Sum of min_shares of children)
+   */
+ int cur_max_shares; /* max(children's max_shares). need better name */
+
+ /* State maintained by container system - only relevant when
+  * this shares struct is the actual shares struct for a
+  * container */
+ struct container_subsys_state css;
+};
+
+/*
+ * Class is the grouping of tasks with shares of each resource that has
+ * registered a resource controller (see include/linux/res_group_rc.h).
+ */
+
+#define resource_group container
+
+#endif /* CONFIG_RES_GROUPS */
+#endif /* _LINUX_RES_GROUP_H */
Index: container-2.6.19-rc6/include/linux/res_group_rc.h
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc6/include/linux/res_group_rc.h
@@ -0,0 +1,97 @@
+/*
+ *  res_group_rc.h - Header file to be used by Resource controllers of
+ *        Resource Groups
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003
+ *  (C) Shailabh Nagar,  IBM Corp. 2003
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *  (C) Vivek Kashyap , IBM Corp. 2004
+ *
+ * Provides data structures, macros and kernel API of Resource Groups for
+ * resource controllers.
+ *
+ * More details at http://ckrm.sf.net
+ *
```

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

```
+
+ /*
+  * move_task is called when a task moves from one resource group to
+  * another. First parameter is the task that is moving, the second
+  * is the resource specific shares of the resource group the task
+  * was in, and the third is the shares of the resource group the
+  * task has moved to.
+  */
+ void (*move_task)(struct task_struct *, struct res_shares *,
+    struct res_shares *);
+};
+
+extern int register_controller(struct res_controller *);
+extern int unregister_controller(struct res_controller *);
+extern struct resource_group default_res_group;
+static inline int is_res_group_root(const struct resource_group *rgroup)
+{
+ return (rgroup->parent == NULL);
+}
+
+#define for_each_child(child, parent) \
+ list_for_each_entry(child, &parent->children, sibling)
+
+/* Get controller specific shares structure for the given resource group */
+static inline struct res_shares *get_controller_shares(
+ struct container *rgroup, struct res_controller *ctlr)
+{
+ if (rgroup && ctlr)
+  return container_of(rgroup->subsys[ctlr->subsys.subsys_id],
+        struct res_shares, css);
+ else
+  return NO_SHARE;
+}
+
+#endif /* _LINUX_RES_GROUP_RC_H */
Index: container-2.6.19-rc6/init/Kconfig
 ===================================================================== =======
--- container-2.6.19-rc6.orig/init/Kconfig
+++ container-2.6.19-rc6/init/Kconfig
@@ -311,6 +311,28 @@ config TASK_XACCT

    Say N if unsure.

+menu "Resource Groups"
+
+config RES_GROUPS
+ bool "Resource Groups"
+ depends on EXPERIMENTAL
```

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

Index: container-2.6.19-rc6/kernel/Makefile
 ===================================================================
--- container-2.6.19-rc6.orig/kernel/Makefile
+++ container-2.6.19-rc6/kernel/Makefile
@@ -53,6 +53,7 @@ obj-$(CONFIG_RELAY) += relay.o
 obj-$(CONFIG_UTS_NS) += utsname.o
 obj-$(CONFIG_TASK_DELAY_ACCT) += delayacct.o
 obj-$(CONFIG_TASKSTATS) += taskstats.o tsacct.o
+obj-$(CONFIG_RES_GROUPS) += res_group/

 ifneq ($(CONFIG_SCHED_NO_NO_OMIT_FRAME_POINTER),y)
 # According to Alan Modra <alan@linuxcare.com.au>, the -fno-omit-frame-pointer is
Index: container-2.6.19-rc6/kernel/fork.c
 ===================================================================
--- container-2.6.19-rc6.orig/kernel/fork.c
+++ container-2.6.19-rc6/kernel/fork.c
@@ -48,6 +48,7 @@
 #include <linux/delayacct.h>
 #include <linux/taskstats_kern.h>
 #include <linux/random.h>
+#include <linux/numtasks.h>

 #include <asm/pgtable.h>
 #include <asm/pgalloc.h>
@@ -1352,7 +1353,7 @@ long do_fork(unsigned long clone_flags,
       int __user *child_tidptr)
 {
  struct task_struct *p;
```

```
- int trace = 0;
+ int trace = 0, rc;
  struct pid *pid = alloc_pid();
  long nr;

@@ -1365,6 +1366,10 @@ long do_fork(unsigned long clone_flags,
    clone_flags |= CLONE_PTRACE;
  }

+ rc = numtasks_allow_fork(current);
+ if (rc)
+  return rc;
+
  p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr, child_tidptr, nr);
  /*
   * Do this prior waking up the new thread - the thread pointer
Index: container-2.6.19-rc6/kernel/res_group/Makefile
 ===================================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/kernel/res_group/Makefile
@@ -0,0 +1,2 @@
+obj-y = res_group.o shares.o rgcs.o
+obj-$(CONFIG_RES_GROUPS_NUMTASKS) += numtasks.o
Index: container-2.6.19-rc6/kernel/res_group/local.h
 ===================================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/kernel/res_group/local.h
@@ -0,0 +1,38 @@
+/*
+ * Contains function definitions that are local to the Resource Groups.
+ * NOT to be included by controllers.
+ */
+
+#include <linux/res_group_rc.h>
+
+extern struct res_controller *get_controller_by_name(const char *);
+extern struct res_controller *get_controller_by_id(unsigned int);
+extern void put_controller(struct res_controller *);
+extern struct resource_group *alloc_res_group(struct resource_group *,
+       const char *);
+extern int free_res_group(struct resource_group *);
+extern void release_res_group(struct kref *);
+extern int set_controller_shares(struct resource_group *,
+   struct res_controller *, const struct res_shares *);
+/* Set shares for the given resource group and resource to default values */
+extern void set_shares_to_default(struct resource_group *,
+      struct res_controller *);
+extern void res_group_teardown(void);
```

```
+extern int set_res_group(pid_t, struct resource_group *);
+extern void move_tasks_to_parent(struct resource_group *);
+
+ssize_t res_group_file_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       char __user *buf,
+       size_t nbytes, loff_t *ppos);
+ssize_t res_group_file_write(struct container *cont,
+        struct cftype *cft,
+        struct file *file,
+        const char __user *userbuf,
+        size_t nbytes, loff_t *ppos);
+
+enum {
+ RG_FILE_SHARES,
+ RG_FILE_STATS,
+};
Index: container-2.6.19-rc6/kernel/res_group/numtasks.c
 ===================================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/kernel/res_group/numtasks.c
@@ -0,0 +1,467 @@
+/* numtasks.c - "Number of tasks" resource controller for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman,  IBM Corp. 2003-2006
+ *       (C) Matt Helsley, IBM Corp. 2006
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+/*
+ * Resource controller for tracking number of tasks in a resource group.
+ */
+#include <linux/module.h>
+#include <linux/res_group_rc.h>
+#include <linux/numtasks.h>
+
+static const char res_ctlr_name[] = "numtasks";
+
+#define UNLIMITED INT_MAX
+#define DEF_TOTAL_NUM_TASKS UNLIMITED
```

```
+static int total_numtasks __read_mostly = DEF_TOTAL_NUM_TASKS;
+
+static struct resource_group *root_rgroup;
+static int total_cnt_alloc = 0;
+
+#define DEF_FORKRATE UNLIMITED
+#define DEF_FORKRATE_INTERVAL (1)
+static int forkrate __read_mostly = DEF_FORKRATE;
+static int forkrate_interval __read_mostly = DEF_FORKRATE_INTERVAL;
+
+struct numtasks {
+ struct res_shares shares;
+ int cnt_min_shares; /* num_tasks min_shares in local units */
+ int cnt_unused;  /* has to borrow if more than this is needed */
+ int cnt_max_shares; /* no tasks over this limit. */
+    /* Three above cnt_* fields are protected
+     * by resource group's group_lock */
+ atomic_t cnt_cur_alloc; /* current alloc from self */
+ atomic_t cnt_borrowed; /* borrowed from the parent */
+
+ /* stats */
+ int successes;
+ int failures;
+ int forkrate_failures;
+
+ /* Fork rate fields */
+ int forks_in_period;
+ unsigned long period_start;
+};
+
+struct res_controller numtasks_ctlr;
+
+static struct numtasks *get_shares_numtasks(struct res_shares *shares)
+{
+ if (shares)
+  return container_of(shares, struct numtasks, shares);
+ return NULL;
+}
+
+static struct numtasks *get_numtasks(struct resource_group *rgroup)
+{
+ return get_shares_numtasks(get_controller_shares(rgroup,
+      &numtasks_ctlr));
+}
+
+static struct resource_group *numtasks_rgroup(struct numtasks *nt)
+{
+ return nt->shares.css.container;
```

```
+}
+
+static inline int check_forkrate(struct numtasks *res)
+{
+ if (time_after(jiffies, res->period_start + forkrate_interval * HZ)) {
+  res->period_start = jiffies;
+  res->forks_in_period = 0;
+ }
+
+ if (res->forks_in_period >= forkrate) {
+   res->forkrate_failures++;
+  return -ENOSPC;
+ }
+ res->forks_in_period++;
+ return 0;
+}
+
+int numtasks_allow_fork(struct task_struct *task)
+{
+ int rc = 0;
+ struct numtasks *res;
+
+ /* task->container won't change during an RCU critical section */
+ rcu_read_lock();
+
+ /* controller is not registered; no resource group is given */
+ if (numtasks_ctlr.ctlr_id == NO_RES_ID)
+  goto out;
+ res = get_numtasks(task_container(task, &numtasks_ctlr.subsys));
+
+ /* numtasks not available for this resource group */
+ if (!res)
+  goto out;
+
+ /* Check forkrate before checking resource group's usage */
+ rc = check_forkrate(res);
+ if (rc)
+  goto out;
+
+ if (res->cnt_max_shares == SHARE_DONT_CARE)
+  goto out;
+
+ /* Over the limit ? */
+ if (atomic_read(&res->cnt_cur_alloc) >= res->cnt_max_shares) {
+  res->failures++;
+  rc = -ENOSPC;
+  goto out;
+ }
```

```
+ out:
+ rcu_read_unlock();
+ return rc;
+}
+
+static void inc_usage_count(struct numtasks *res)
+{
+ struct resource_group *rgroup = numtasks_rgroup(res);
+ atomic_inc(&res->cnt_cur_alloc);
+
+ if (is_res_group_root(rgroup)) {
+  total_cnt_alloc++;
+  res->successes++;
+  return;
+ }
+ /* Do we need to borrow from our parent ? */
+ if ((res->cnt_unused == SHARE_DONT_CARE) ||
+   (atomic_read(&res->cnt_cur_alloc) > res->cnt_unused)) {
+  inc_usage_count(get_numtasks(rgroup->parent));
+  atomic_inc(&res->cnt_borrowed);
+ } else {
+  total_cnt_alloc++;
+    res->successes++;
+ }
+}
+
+static void dec_usage_count(struct numtasks *res)
+{
+ if (atomic_read(&res->cnt_cur_alloc) == 0)
+  return;
+ atomic_dec(&res->cnt_cur_alloc);
+ if (atomic_read(&res->cnt_borrowed) > 0) {
+  atomic_dec(&res->cnt_borrowed);
+  dec_usage_count(get_numtasks(numtasks_rgroup(res)->parent));
+ } else
+  total_cnt_alloc--;
+
+}
+
+static void numtasks_move_task(struct task_struct *task,
+  struct res_shares *old, struct res_shares *new)
+{
+ struct numtasks *oldres, *newres;
+
+ if (old == new)
+  return;
+
+ /* Decrement usage count of old resource group */
```

```
+ oldres = get_shares_numtasks(old);
+ if (oldres)
+  dec_usage_count(oldres);
+
+ /* Increment usage count of new resource group */
+ newres = get_shares_numtasks(new);
+ if (newres)
+  inc_usage_count(newres);
+}
+
+/* Initialize share struct values */
+static void numtasks_res_init_one(struct numtasks *numtasks_res)
+{
+ numtasks_res->shares.min_shares = SHARE_DONT_CARE;
+ numtasks_res->shares.max_shares = SHARE_DONT_CARE;
+ numtasks_res->shares.child_shares_divisor = SHARE_DEFAULT_DIVISOR;
+ numtasks_res->shares.unused_min_shares = SHARE_DEFAULT_DIVISOR;
+
+ numtasks_res->cnt_min_shares = SHARE_DONT_CARE;
+ numtasks_res->cnt_unused = SHARE_DONT_CARE;
+ numtasks_res->cnt_max_shares = SHARE_DONT_CARE;
+ numtasks_res->period_start = jiffies;
+}
+
+static struct res_shares *numtasks_alloc_shares_struct(
+     struct resource_group *rgroup)
+{
+ struct numtasks *res;
+
+ res = kzalloc(sizeof(struct numtasks), GFP_KERNEL);
+ if (!res)
+  return NULL;
+ numtasks_res_init_one(res);
+ if (is_res_group_root(rgroup))
+  root_rgroup = rgroup; /* store root's resource group. */
+ return &res->shares;
+}
+
+/*
+ * No locking of this resource group object necessary as we are not
+ * supposed to be assigned (or used) when/after this function is called.
+ */
+static void numtasks_free_shares_struct(struct res_shares *my_res)
+{
+ struct numtasks *res, *parres;
+ int i, borrowed;
+ struct resource_group *rgroup;
+
```

```
+ res = get_shares_numtasks(my_res);
+ rgroup = numtasks_rgroup(res);
+ if (!is_res_group_root(rgroup)) {
+  parres = get_numtasks(rgroup->parent);
+  borrowed = atomic_read(&res->cnt_borrowed);
+  for (i = 0; i < borrowed; i++)
+   dec_usage_count(parres);
+ }
+ kfree(res);
+}
+
+static int recalc_shares(int self_shares, int parent_shares, int parent_divisor)
+{
+ u64 numerator;
+
+ if ((self_shares == SHARE_DONT_CARE) ||
+     (parent_shares == SHARE_DONT_CARE))
+  return SHARE_DONT_CARE;
+ if (parent_divisor == 0)
+  return 0;
+ numerator = (u64) self_shares * parent_shares;
+ do_div(numerator, parent_divisor);
+ return numerator;
+}
+
+static int recalc_unused_shares(int self_cnt_min_shares,
+     int self_unused_min_shares, int self_divisor)
+{
+ u64 numerator;
+
+ if (self_cnt_min_shares == SHARE_DONT_CARE)
+  return SHARE_DONT_CARE;
+ if (self_divisor == 0)
+  return 0;
+ numerator = (u64) self_unused_min_shares * self_cnt_min_shares;
+ do_div(numerator, self_divisor);
+ return numerator;
+}
+
+static void recalc_self(struct numtasks *res,
+     struct numtasks *parres)
+{
+ struct res_shares *par = &parres->shares;
+ struct res_shares *self = &res->shares;
+
+ res->cnt_min_shares = recalc_shares(self->min_shares,
+       parres->cnt_min_shares,
+       par->child_shares_divisor);
```

```
+ res->cnt_max_shares = recalc_shares(self->max_shares,
+      parres->cnt_max_shares,
+      par->child_shares_divisor);
+
+ /*
+  * Now that we know the new cnt_min/cnt_max boundaries we can update
+  * the unused quantity.
+  */
+ res->cnt_unused = recalc_unused_shares(res->cnt_min_shares,
+      self->unused_min_shares,
+      self->child_shares_divisor);
+}
+
+
+/*
+ * Recalculate the min_shares and max_shares in real units and propagate the
+ * same to children.
+ * Called with container_manage_lock() held.
+ */
+static void recalc_and_propagate(struct numtasks *res,
+    struct numtasks *parres)
+{
+ struct resource_group *child = NULL;
+ struct numtasks *childres;
+
+ if (parres)
+  recalc_self(res, parres);
+
+ /* propagate to children */
+ for_each_child(child, numtasks_rgroup(res)) {
+  childres = get_numtasks(child);
+  BUG_ON(!childres);
+  recalc_and_propagate(childres, res);
+ }
+}
+
+static void numtasks_shares_changed(struct res_shares *my_res)
+{
+ struct numtasks *parres, *res;
+ struct res_shares *cur, *par;
+ struct resource_group *rgroup;
+
+ res = get_shares_numtasks(my_res);
+ if (!res)
+  return;
+ rgroup = numtasks_rgroup(res);
+ cur = &res->shares;
+
```

```
+ if (!is_res_group_root(rgroup)) {
+  parres = get_numtasks(rgroup->parent);
+  par = &parres->shares;
+ } else {
+  parres = NULL;
+  par = NULL;
+ }
+ if (parres)
+  parres->cnt_unused = recalc_unused_shares(
+      parres->cnt_min_shares,
+      par->unused_min_shares,
+      par->child_shares_divisor);
+ recalc_and_propagate(res, parres);
+}
+
+static ssize_t numtasks_show_stats(struct res_shares *my_res,
+     char *buf, size_t buf_size)
+{
+ ssize_t i, j = 0;
+ struct numtasks *res;
+
+ res = get_shares_numtasks(my_res);
+ if (!res)
+  return -EINVAL;
+
+ i = snprintf(buf, buf_size, "%s: Current usage %d\n",
+     res_ctlr_name,
+     atomic_read(&res->cnt_cur_alloc));
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of successes %d\n",
+     res_ctlr_name, res->successes);
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of failures %d\n",
+     res_ctlr_name, res->failures);
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of forkrate failures %d\n",
+     res_ctlr_name, res->forkrate_failures);
+ j += i;
+ return j;
+}
+
+struct res_controller numtasks_ctlr = {
+ .name = res_ctlr_name,
+ .ctlr_id = NO_RES_ID,
+ .alloc_shares_struct = numtasks_alloc_shares_struct,
+ .free_shares_struct = numtasks_free_shares_struct,
+ .move_task = numtasks_move_task,
+ .shares_changed = numtasks_shares_changed,
```

```
+ .show_stats = numtasks_show_stats,
+};
+
+/*
+ * Writeable module parameters use these set_<parameter> functions to respond
+ * to changes. Otherwise the values can be read and used any time.
+ */
+static int set_numtasks_config_val(int *var, int old_value, const char *val,
+    struct kernel_param *kp)
+{
+ int rc = param_set_int(val, kp);
+
+ if (rc < 0)
+  return rc;
+ if (*var < 1) {
+  *var = old_value;
+  return -EINVAL;
+ }
+ return 0;
+}
+
+static int set_total_numtasks(const char *val, struct kernel_param *kp)
+{
+ int prev = total_numtasks;
+ int rc = set_numtasks_config_val(&total_numtasks, prev, val, kp);
+ struct numtasks *res = NULL;
+
+ if (!root_rgroup)
+  return 0;
+ if (rc < 0)
+  return rc;
+ if (total_numtasks <= total_cnt_alloc) {
+  total_numtasks = prev;
+  return -EINVAL;
+ }
+ container_lock();
+ res = get_numtasks(root_rgroup);
+ res->cnt_min_shares = total_numtasks;
+ res->cnt_unused = total_numtasks;
+ res->cnt_max_shares = total_numtasks;
+ recalc_and_propagate(res, NULL);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(total_numtasks, int, set_total_numtasks,
+   S_IRUGO | S_IWUSR);
+
+static void reset_forkrates(struct resource_group *rgroup, unsigned long now)
```

```
+{
+ struct numtasks *res;
+ struct resource_group *child = NULL;
+
+ res = get_numtasks(rgroup);
+ if (!res)
+  return;
+ res->forks_in_period = 0;
+ res->period_start = now;
+
+ for_each_child(child, rgroup)
+  reset_forkrates(child, now);
+}
+
+static int set_forkrate(const char *val, struct kernel_param *kp)
+{
+ int prev = forkrate;
+ int rc = set_numtasks_config_val(&forkrate, prev, val, kp);
+ if (rc < 0)
+  return rc;
+ container_lock();
+ reset_forkrates(root_rgroup, jiffies);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(forkrate, int, set_forkrate, S_IRUGO | S_IWUSR);
+
+static int set_forkrate_interval(const char *val, struct kernel_param *kp)
+{
+ int prev = forkrate_interval;
+ int rc = set_numtasks_config_val(&forkrate_interval, prev, val, kp);
+ if (rc < 0)
+  return rc;
+ container_lock();
+ reset_forkrates(root_rgroup, jiffies);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(forkrate_interval, int, set_forkrate_interval,
+   S_IRUGO | S_IWUSR);
+
+int __init init_numtasks_res(void)
+{
+ if (numtasks_ctlr.ctlr_id != NO_RES_ID)
+  return -EBUSY; /* already registered */
+ return register_controller(&numtasks_ctlr);
+}
+
```

```
+void __exit exit_numtasks_res(void)
+{
+ int rc;
+ do {
+  rc = unregister_controller(&numtasks_ctlr);
+ } while (rc == -EBUSY);
+ BUG_ON(rc != 0);
+}
+module_init(init_numtasks_res)
+module_exit(exit_numtasks_res)
Index: container-2.6.19-rc6/kernel/res_group/res_group.c
 ===================================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/kernel/res_group/res_group.c
@@ -0,0 +1,160 @@
+/* res_group.c - Resource Groups: Resource management through grouping of
+ *    unrelated tasks.
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003, 2004
+ *  (C) Shailabh Nagar, IBM Corp. 2003, 2004
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *  (C) Vivek Kashyap, IBM Corp. 2004
+ *  (C) Matt Helsley, IBM Corp. 2006
+ *
+ * Provides kernel API of Resource Groups for in-kernel,per-resource
+ * controllers (one each for cpu, memory and io).
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+#include <linux/module.h>
+#include <asm/uaccess.h>
+#include <linux/fs.h>
+#include "local.h"
+
+
+static int res_group_create(struct container_subsys *ss,
+       struct container *cont)
+{
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ struct res_shares *shares = ctlr->alloc_shares_struct(cont);
+ cont->subsys[ss->subsys_id] = &shares->css;
```

```
+ return 0;
+}
+
+static void res_group_destroy(struct container_subsys *ss,
+        struct container *cont)
+{
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ struct res_shares *shares = get_controller_shares(cont, ctlr);
+ ctlr->free_shares_struct(shares);
+}
+
+static int res_group_populate(struct container_subsys *ss,
+         struct container *cont) {
+ int err;
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ if ((err = container_add_file(cont, &ctlr->shares_cft.cft)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &ctlr->stats_cft.cft)) < 0)
+  return err;
+
+ return 0;
+}
+
+static void res_group_attach(struct container_subsys *ss,
+        struct container *cont,
+        struct container *old_cont,
+        struct task_struct *tsk) {
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ struct res_shares *oldshares = get_controller_shares(old_cont, ctlr);
+ struct res_shares *newshares = get_controller_shares(cont, ctlr);
+
+ if (ctlr->move_task) {
+  ctlr->move_task(tsk, oldshares, newshares);
+ }
+}
+
+static void res_group_fork(struct container_subsys *ss,
+    struct task_struct *task) {
+ struct res_controller *ctlr =
+  container_of(ss, struct res_controller, subsys);
+ struct res_shares *shares =
+  get_controller_shares(task_container(task, ss), ctlr);
+ if (ctlr->move_task) {
+  ctlr->move_task(task, NULL, shares);
+ }
+}
+
+static void res_group_exit(struct container_subsys *ss,
```

```
+      struct task_struct *task) {
+ struct res_controller *ctlr =
+  container_of(ss, struct res_controller, subsys);
+ struct res_shares *shares =
+  get_controller_shares(task_container(task, ss), ctlr);
+ if (ctlr->move_task) {
+  ctlr->move_task(task, shares, NULL);
+ }
+}
+
+/*
+ * Interface for registering a resource controller.
+ *
+ * Returns the 0 on success, -errno for failure.
+ * Fills ctlr->ctlr_id with a valid controller id on success.
+ */
+int register_controller(struct res_controller *ctlr)
+{
+ int ret;
+
+ struct container_subsys *ss = &ctlr->subsys;
+
+ if (!ctlr)
+  return -EINVAL;
+
+ /* Make sure there is an alloc and a free */
+ if (!ctlr->alloc_shares_struct || !ctlr->free_shares_struct)
+  return -EINVAL;
+
+ ss->create = res_group_create;
+ ss->destroy = res_group_destroy;
+ ss->populate = res_group_populate;
+ if (ctlr->move_task) {
+  ss->attach = res_group_attach;
+  ss->fork = res_group_fork;
+  ss->exit = res_group_exit;
+ }
+
+ ctlr->shares_cft.ctlr = ctlr;
+ strcpy(ctlr->shares_cft.cft.name, ctlr->name);
+ strcat(ctlr->shares_cft.cft.name, ".shares");
+ ctlr->shares_cft.cft.private = RG_FILE_SHARES;
+ ctlr->shares_cft.cft.read = res_group_file_read;
+ ctlr->shares_cft.cft.write = res_group_file_write;
+
+ ctlr->stats_cft.ctlr = ctlr;
+ strcpy(ctlr->stats_cft.cft.name, ctlr->name);
+ strcat(ctlr->stats_cft.cft.name, ".stats");
```

```
+ ctlr->stats_cft.cft.private = RG_FILE_STATS;
+ ctlr->stats_cft.cft.read = res_group_file_read;
+ ctlr->stats_cft.cft.write = res_group_file_write;
+
+ ss->name = ctlr->name;
+
+ ret = container_register_subsys(ss);
+
+ if (ret < 0)
+  return ret;
+
+ ctlr->ctlr_id = ss->subsys_id;
+
+ return 0;
+}
+
+/*
+ * Unregistering resource controller.
+ *
+ * Returns 0 on success -errno for failure.
+ */
+int unregister_controller(struct res_controller *ctlr)
+{
+ BUG();
+ return 0;
+}
+
+
+EXPORT_SYMBOL_GPL(register_controller);
+EXPORT_SYMBOL_GPL(unregister_controller);
+EXPORT_SYMBOL_GPL(set_controller_shares);
Index: container-2.6.19-rc6/kernel/res_group/rgcs.c
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc6/kernel/res_group/rgcs.c
@@ -0,0 +1,302 @@
+/*
+ * kernel/res_group/rgcs.c
+ *
+ * Copyright (C) Shailabh Nagar,  IBM Corp. 2005
+ *          Chandra Seetharaman,   IBM Corp. 2005, 2006
+ *
+ * Resource Group Configfs Subsystem (rgcs) provides the user interface
+ * for Resource groups.
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
```

```
+ * it under the terms of version 2 of the  GNU General Public License
+ * as published by the Free Software Foundation.
+ *
+ */
+#include <linux/ctype.h>
+#include <linux/module.h>
+#include <linux/configfs.h>
+#include <linux/parser.h>
+#include <linux/fs.h>
+#include <asm/uaccess.h>
+
+#include "local.h"
+
+#define RES_STRING "res"
+#define MIN_SHARES_STRING "min_shares"
+#define MAX_SHARES_STRING "max_shares"
+#define CHILD_SHARES_DIVISOR_STRING "child_shares_divisor"
+
+static ssize_t show_stats(struct resource_group *rgroup,
+     struct res_controller *ctlr,
+     char *buf)
+{
+ int j = 0, rc = 0;
+ size_t buf_size = PAGE_SIZE-1; /* allow only PAGE_SIZE # of bytes */
+ struct res_shares *shares;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares && ctlr->show_stats)
+  j = ctlr->show_stats(shares, buf, buf_size);
+ rc += j;
+ buf += j;
+ buf_size -= j;
+ return rc;
+}
+
+enum parse_token_t {
+ parse_res_type, parse_err
+};
+
+static match_table_t parse_tokens = {
+ {parse_res_type, RES_STRING"=%s"},
+ {parse_err, NULL}
+};
+
+static int stats_parse(const char *options,
+    char **resname, char **remaining_line)
+{
+ char *p, *str;
```

```
+ int rc = -EINVAL;
+
+ if (!options)
+  return -EINVAL;
+
+ while ((p = strsep((char **)&options, ",")) != NULL) {
+  substring_t args[MAX_OPT_ARGS];
+  int token;
+
+  if (!*p)
+   continue;
+  token = match_token(p, parse_tokens, args);
+  if (token == parse_res_type) {
+   *resname = match_strdup(args);
+   str = p + strlen(p) + 1;
+   *remaining_line = kmalloc(strlen(str) + 1, GFP_KERNEL);
+   if (*remaining_line == NULL) {
+    kfree(*resname);
+    *resname = NULL;
+    rc = -ENOMEM;
+   } else {
+    strcpy(*remaining_line, str);
+    rc = 0;
+   }
+   break;
+  }
+ }
+ return rc;
+}
+
+static int reset_stats(struct resource_group *rgroup, struct res_controller *ctlr, const char *str)
+{
+ int rc;
+ char *resname = NULL, *statstr = NULL;
+ struct res_shares *shares;
+
+ rc = stats_parse(str, &resname, &statstr);
+ if (rc)
+  return rc;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares && ctlr->reset_stats)
+  rc = ctlr->reset_stats(shares, statstr);
+
+ kfree(resname);
+ kfree(statstr);
+ return rc;
+}
```

```
+
+
+enum share_token_t {
+ MIN_SHARES_TOKEN,
+ MAX_SHARES_TOKEN,
+ CHILD_SHARES_DIVISOR_TOKEN,
+ RESOURCE_TYPE_TOKEN,
+ ERROR_TOKEN
+};
+
+/* Token matching for parsing input to this magic file */
+static match_table_t shares_tokens = {
+ {RESOURCE_TYPE_TOKEN, RES_STRING"=%s"},
+ {MIN_SHARES_TOKEN, MIN_SHARES_STRING"=%d"},
+ {MAX_SHARES_TOKEN, MAX_SHARES_STRING"=%d"},
+ {CHILD_SHARES_DIVISOR_TOKEN, CHILD_SHARES_DIVISOR_STRING"=%d"},
+ {ERROR_TOKEN, NULL}
+};
+
+static int shares_parse(const char *options, char **resname,
+     struct res_shares *shares)
+{
+ char *p;
+ int option, rc = -EINVAL;
+
+ *resname = NULL;
+ if (!options)
+  goto done;
+
+ while ((p = strsep((char **)&options, ",")) != NULL) {
+  substring_t args[MAX_OPT_ARGS];
+  int token;
+
+  if (!*p)
+   continue;
+
+  token = match_token(p, shares_tokens, args);
+  switch (token) {
+  case RESOURCE_TYPE_TOKEN:
+   if (*resname)
+    goto done;
+   *resname = match_strdup(args);
+   break;
+  case MIN_SHARES_TOKEN:
+   if (match_int(args, &option))
+    goto done;
+   shares->min_shares = option;
+   break;
```

```
+  case MAX_SHARES_TOKEN:
+   if (match_int(args, &option))
+    goto done;
+   shares->max_shares = option;
+   break;
+  case CHILD_SHARES_DIVISOR_TOKEN:
+   if (match_int(args, &option))
+    goto done;
+   shares->child_shares_divisor = option;
+   break;
+  default:
+   goto done;
+  }
+ }
+ rc = 0;
+done:
+ if (rc) {
+  kfree(*resname);
+  *resname = NULL;
+ }
+ return rc;
+}
+
+static int set_shares(struct resource_group *rgroup,
+        struct res_controller *ctlr,
+        const char *str)
+{
+ char *resname = NULL;
+ int rc;
+ struct res_shares shares = {
+  .min_shares = SHARE_UNCHANGED,
+  .max_shares = SHARE_UNCHANGED,
+  .child_shares_divisor = SHARE_UNCHANGED,
+ };
+
+ rc = shares_parse(str, &resname, &shares);
+ if (!rc) {
+  rc = set_controller_shares(rgroup, ctlr, &shares);
+  kfree(resname);
+ }
+ return rc;
+}
+
+static ssize_t show_shares(struct resource_group *rgroup,
+      struct res_controller *ctlr,
+      char *buf)
+{
+ ssize_t j, rc = 0, bufsize = PAGE_SIZE;
```

```
+ struct res_shares *shares;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares) {
+  j = snprintf(buf, bufsize, "%s=%s,%s=%d,%s=%d,%s=%d\n",
+        RES_STRING, ctlr->name,
+        MIN_SHARES_STRING, shares->min_shares,
+        MAX_SHARES_STRING, shares->max_shares,
+        CHILD_SHARES_DIVISOR_STRING,
+        shares->child_shares_divisor);
+  rc += j; buf += j; bufsize -= j;
+ }
+ return rc;
+}
+
+ssize_t res_group_file_write(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       const char __user *userbuf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct res_group_cft *rgcft = container_of(cft, struct res_group_cft, cft);
+ struct res_controller *ctlr = rgcft->ctlr;
+
+ char *buf;
+ ssize_t retval;
+ int filetype = cft->private;
+
+ if (nbytes >= PAGE_SIZE)
+  return -E2BIG;
+
+ buf = kmalloc(nbytes + 1, GFP_USER);
+ if (!buf) return -ENOMEM;
+ if (copy_from_user(buf, userbuf, nbytes)) {
+  retval = -EFAULT;
+  goto out1;
+ }
+ buf[nbytes] = 0; /* nul-terminate */
+
+ container_manage_lock();
+
+ if (container_is_removed(cont)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch(filetype) {
+ case RG_FILE_SHARES:
```

```
+  retval = set_shares(cont, ctlr, buf);
+  break;
+ case RG_FILE_STATS:
+  retval = reset_stats(cont, ctlr, buf);
+  break;
+ default:
+  retval = -EINVAL;
+ }
+ if (!retval) retval = nbytes;
+
+ out2:
+ container_manage_unlock();
+ out1:
+ kfree(buf);
+ return retval;
+}
+
+ssize_t res_group_file_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct res_group_cft *rgcft = container_of(cft, struct res_group_cft, cft);
+ struct res_controller *ctlr = rgcft->ctlr;
+
+ char *page = kmalloc(PAGE_SIZE, GFP_USER);
+ ssize_t retval;
+ int filetype = cft->private;
+
+ if (!page) return -ENOMEM;
+
+ switch(filetype) {
+ case RG_FILE_SHARES:
+  retval = show_shares(cont, ctlr, page);
+  break;
+ case RG_FILE_STATS:
+  retval = show_stats(cont, ctlr, page);
+  break;
+ default:
+  retval = -EINVAL;
+ }
+
+ if (retval >= 0) {
+  retval = simple_read_from_buffer(buf, nbytes,
+       ppos, page, retval);
+ }
+ kfree(page);
```

```
+ return retval;
+}
Index: container-2.6.19-rc6/kernel/res_group/shares.c
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc6/kernel/res_group/shares.c
@@ -0,0 +1,228 @@
+/*
+ * shares.c - Share management functions for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman,  IBM Corp. 2003, 2004, 2005, 2006
+ *  (C) Hubertus Franke,  IBM Corp. 2004
+ *  (C) Matt Helsley,  IBM Corp. 2006
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License version 2 as
+ * published by the Free Software Foundation.
+ */
+
+#include <linux/errno.h>
+#include <linux/res_group_rc.h>
+#include <linux/container.h>
+
+/*
+ * Share values can be quantitative (quantity of memory for instance) or
+ * symbolic. The symbolic value DONT_CARE allows for any quantity of a resource
+ * to be substituted in its place. The symbolic value UNCHANGED is only used
+ * when setting share values and means that the old value should be used.
+ */
+
+/* Is the share a quantity (as opposed to "symbols" DONT_CARE or UNCHANGED) */
+static inline int is_share_quantitative(int share)
+{
+ return (share >= 0);
+}
+
+static inline int is_share_symbolic(int share)
+{
+ return !is_share_quantitative(share);
+}
+
+static inline int is_share_valid(int share)
+{
+ return ((share == SHARE_DONT_CARE) ||
+   (share == SHARE_UNSUPPORTED) ||
+   is_share_quantitative(share));
```

```
+}
+
+static inline int did_share_change(int share)
+{
+ return (share != SHARE_UNCHANGED);
+}
+
+static inline int change_supported(int share)
+{
+ return (share != SHARE_UNSUPPORTED);
+}
+
+/*
+ * Caller is responsible for protecting 'parent'
+ * Caller is responsible for making sure that the sum of sibling min_shares
+ * doesn't exceed parent's total min_shares.
+ */
+static inline void child_min_shares_changed(struct res_shares *parent,
+       int child_cur_min_shares,
+       int child_new_min_shares)
+{
+ if (is_share_quantitative(child_new_min_shares))
+  parent->unused_min_shares -= child_new_min_shares;
+ if (is_share_quantitative(child_cur_min_shares))
+  parent->unused_min_shares += child_cur_min_shares;
+}
+
+/*
+ * Set parent's cur_max_shares to the largest 'max_shares' of all
+ * of its children.
+ */
+static inline void set_cur_max_shares(struct resource_group *parent,
+         struct res_controller *ctlr)
+{
+ int max_shares = 0;
+ struct resource_group *child = NULL;
+ struct res_shares *child_shares, *parent_shares;
+
+ for_each_child(child, parent) {
+  child_shares = get_controller_shares(child, ctlr);
+  max_shares = max(max_shares, child_shares->max_shares);
+ }
+
+ parent_shares = get_controller_shares(parent, ctlr);
+ parent_shares->cur_max_shares = max_shares;
+}
+
+/*
```

```
+ * Return -EINVAL if the child's shares violate self-consistency or
+ * parent-imposed restrictions. Otherwise return 0.
+ *
+ * This involves checking shares between the child and its parent;
+ * the child and itself (userspace can't be trusted).
+ */
+static inline int are_shares_valid(struct res_shares *child,
+       struct res_shares *parent,
+       int current_usage,
+       int min_shares_increase)
+{
+ /*
+  * CHILD <-> PARENT validation
+  * Increases in child's min_shares or max_shares can't exceed
+  * limitations imposed by the parent resource group.
+  * Only validate this if we have a parent.
+  */
+ if (parent &&
+     ((is_share_quantitative(child->min_shares) &&
+       (min_shares_increase > parent->unused_min_shares)) ||
+      (is_share_quantitative(child->max_shares) &&
+       (child->max_shares > parent->child_shares_divisor))))
+  return -EINVAL;
+
+ /* CHILD validation: is min valid */
+ if (!is_share_valid(child->min_shares))
+  return -EINVAL;
+
+ /* CHILD validation: is max valid */
+ if (!is_share_valid(child->max_shares))
+  return -EINVAL;
+
+ /*
+  * CHILD validation: is divisor quantitative & current_usage
+  * is not more than the new divisor
+  */
+ if (!is_share_quantitative(child->child_shares_divisor) ||
+   (current_usage > child->child_shares_divisor))
+  return -EINVAL;
+
+ /*
+  * CHILD validation: is the new child_shares_divisor large
+  * enough to accomodate largest max_shares of any of my child
+  */
+ if (child->child_shares_divisor < child->cur_max_shares)
+  return -EINVAL;
+
+ /* CHILD validation: min <= max */
```

```
+ if (is_share_quantitative(child->min_shares) &&
+   is_share_quantitative(child->max_shares) &&
+   (child->min_shares > child->max_shares))
+  return -EINVAL;
+
+ return 0;
+}
+
+/*
+ * Set the resource shares of a child resource group given the new shares
+ * specified by userspace, the child's current shares, and the parent
+ * resource group's shares.
+ *
+ * Caller is responsible for holding group_lock of child and parent
+ * resource groups to protect the shares structures passed to this function.
+ */
+static int set_shares(const struct res_shares *new,
+       struct res_shares *child_shares,
+           struct res_shares *parent_shares)
+{
+ int rc, current_usage, min_shares_increase;
+ struct res_shares final_shares;
+
+ BUG_ON(!new || !child_shares);
+
+ final_shares = *child_shares;
+ if (did_share_change(new->child_shares_divisor) &&
+    change_supported(child_shares->child_shares_divisor))
+  final_shares.child_shares_divisor = new->child_shares_divisor;
+ if (did_share_change(new->min_shares) &&
+    change_supported(child_shares->min_shares))
+  final_shares.min_shares = new->min_shares;
+ if (did_share_change(new->max_shares) &&
+    change_supported(child_shares->max_shares))
+  final_shares.max_shares = new->max_shares;
+
+ current_usage = child_shares->child_shares_divisor -
+         child_shares->unused_min_shares;
+ min_shares_increase = final_shares.min_shares;
+ if (is_share_quantitative(child_shares->min_shares))
+  min_shares_increase -= child_shares->min_shares;
+
+ rc = are_shares_valid(&final_shares, parent_shares, current_usage,
+             min_shares_increase);
+ if (rc)
+  return rc; /* new shares would violate restrictions */
+
+ if (did_share_change(new->child_shares_divisor))
```

```
+  final_shares.unused_min_shares =
+   (final_shares.child_shares_divisor - current_usage);
+ *child_shares = final_shares;
+ return 0;
+}
+
+int set_controller_shares(struct resource_group *rgroup,
+     struct res_controller *ctlr,
+     const struct res_shares *new_shares)
+{
+ struct res_shares *shares, *parent_shares;
+ int prev_min, prev_max, rc;
+
+ if (!ctlr->shares_changed)
+  return -EINVAL;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (!shares)
+  return -EINVAL;
+
+ prev_min = shares->min_shares;
+ prev_max = shares->max_shares;
+
+ container_lock(); /* XXX */
+ //spin_lock(&rgroup->group_lock);
+ parent_shares = get_controller_shares(rgroup->parent, ctlr);
+ rc = set_shares(new_shares, shares, parent_shares);
+
+ if (rc || is_res_group_root(rgroup))
+  goto done;
+
+ /* Notify parent about changes in my shares */
+ child_min_shares_changed(parent_shares, prev_min,
+          shares->min_shares);
+ if (prev_max != shares->max_shares)
+  set_cur_max_shares(rgroup->parent, ctlr);
+
+done:
+ container_unlock(); /* XXX */
+ if (!rc)
+  ctlr->shares_changed(shares);
+ return rc;
+}

--
```

Subject: [PATCH 6/7] Split Cpusets into Cpusets and Memsets
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:54 GMT
View Forum Message <> Reply to Message

This patch splits the Cpusets container subsystem into two independent subsystems; currently CPUsets are the cpu and memory node control functionality in Cpusets are pretty much disjoint and unrelated; now that the common process container abstraction has been moved out, there's no particular reason to keep them together in the same subsystem.

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

```
 fs/proc/array.c           |    2
 include/linux/cpuset.h    |   75 --
 include/linux/mempolicy.h |    2
 include/linux/memset.h    |  125 ++++
 include/linux/sched.h     |   10
 init/Kconfig              |   14
 init/main.c               |    3
 kernel/Makefile           |    1
 kernel/cpuset.c           |  994 +-------------------------------
 kernel/memset.c           | 1352 ++++++++++++++++++++++++++++++++++++++++++++++
 mm/filemap.c              |    6
 mm/hugetlb.c              |    4
 mm/memory_hotplug.c       |    4
 mm/mempolicy.c            |   36 -
 mm/migrate.c              |    4
 mm/oom_kill.c             |   14
 mm/page_alloc.c           |   26
 mm/slab.c                 |   12
 mm/vmscan.c               |   10
 19 files changed, 1593 insertions(+), 1101 deletions(-)

Index: container-2.6.19-rc6/include/linux/cpuset.h
===================================================================
--- container-2.6.19-rc6.orig/include/linux/cpuset.h
+++ container-2.6.19-rc6/include/linux/cpuset.h
@@ -10,58 +10,22 @@

 #include <linux/sched.h>
 #include <linux/cpumask.h>
-#include <linux/nodemask.h>
 #include <linux/container.h>

 #ifdef CONFIG_CPUSETS
```

```
-extern int number_of_cpusets;  /* How many cpusets are defined in system? */
-
 extern int cpuset_init_early(void);
 extern int cpuset_init(void);
 extern void cpuset_init_smp(void);
 extern cpumask_t cpuset_cpus_allowed(struct task_struct *p);
-extern nodemask_t cpuset_mems_allowed(struct task_struct *p);
-void cpuset_init_current_mems_allowed(void);
-void cpuset_update_task_memory_state(void);
-#define cpuset_nodes_subset_current_mems_allowed(nodes) \
-  nodes_subset((nodes), current->mems_allowed)
-int cpuset_zonelist_valid_mems_allowed(struct zonelist *zl);
-
-extern int __cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask);
-static int inline cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask)
-{
- return number_of_cpusets <= 1 || __cpuset_zone_allowed(z, gfp_mask);
-}

 extern int cpuset_excl_nodes_overlap(const struct task_struct *p);

-#define cpuset_memory_pressure_bump()     \
- do {       \
-  if (cpuset_memory_pressure_enabled)  \
-   __cpuset_memory_pressure_bump(); \
- } while (0)
-extern int cpuset_memory_pressure_enabled;
-extern void __cpuset_memory_pressure_bump(void);
-
 extern struct file_operations proc_cpuset_operations;
 extern char *cpuset_task_status_allowed(struct task_struct *task, char *buffer);
-extern int cpuset_mem_spread_node(void);
-
-static inline int cpuset_do_page_mem_spread(void)
-{
- return current->flags & PF_SPREAD_PAGE;
-}
-
-static inline int cpuset_do_slab_mem_spread(void)
-{
- return current->flags & PF_SPREAD_SLAB;
-}

 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; }
@@ -73,60 +37,21 @@ static inline cpumask_t cpuset_cpus_allo
  return cpu_possible_map;
 }

-static inline nodemask_t cpuset_mems_allowed(struct task_struct *p)
-{
- return node_possible_map;
-}
-
-static inline void cpuset_init_current_mems_allowed(void) {}
-static inline void cpuset_update_task_memory_state(void) {}
-#define cpuset_nodes_subset_current_mems_allowed(nodes) (1)
-
-static inline int cpuset_zonelist_valid_mems_allowed(struct zonelist *zl)
-{
- return 1;
-}
-
-static inline int cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask)
-{
- return 1;
-}

 static inline int cpuset_excl_nodes_overlap(const struct task_struct *p)
 {
  return 1;
 }

-static inline void cpuset_memory_pressure_bump(void) {}
-
 static inline char *cpuset_task_status_allowed(struct task_struct *task,
        char *buffer)
 {
  return buffer;
 }

-static inline int cpuset_mem_spread_node(void)
-{
- return 0;
-}
-
-static inline int cpuset_do_page_mem_spread(void)
-{
- return 0;
-}
-
```

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

 static inline void cpuset_track_online_nodes(void) {}

-static inline int current_cpuset_is_being_rebound(void)
-{
- return 0;
-}
-
 #endif /* !CONFIG_CPUSETS */

 #endif /* _LINUX_CPUSET_H */
Index: container-2.6.19-rc6/include/linux/memset.h
===================================================================
--- /dev/null
+++ container-2.6.19-rc6/include/linux/memset.h
@@ -0,0 +1,125 @@
+#ifndef _LINUX_MEMSET_H
+#define _LINUX_MEMSET_H
+/*
+ *  memset interface
+ *
+ *  Copyright (C) 2003 BULL SA
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ */
+
+#include <linux/sched.h>
+#include <linux/nodemask.h>
+#include <linux/container.h>
+
+#ifdef CONFIG_MEMSETS
+
+extern int number_of_memsets;  /* How many memsets are defined in system? */
+
+extern int memset_init_early(void);
+extern int memset_init(void);
+extern void memset_init_smp(void);
+extern nodemask_t memset_mems_allowed(struct task_struct *p);
+void memset_init_current_mems_allowed(void);
+void memset_update_task_memory_state(void);
+#define memset_nodes_subset_current_mems_allowed(nodes) \
+  nodes_subset((nodes), current->mems_allowed)
+int memset_zonelist_valid_mems_allowed(struct zonelist *zl);
+
```

```
+extern int __memset_zone_allowed(struct zone *z, gfp_t gfp_mask);
+static int inline memset_zone_allowed(struct zone *z, gfp_t gfp_mask)
+{
+ return number_of_memsets <= 1 || __memset_zone_allowed(z, gfp_mask);
+}
+
+extern int memset_excl_nodes_overlap(const struct task_struct *p);
+
+#define memset_memory_pressure_bump()     \
+ do {       \
+  if (memset_memory_pressure_enabled)  \
+   __memset_memory_pressure_bump(); \
+ } while (0)
+extern int memset_memory_pressure_enabled;
+extern void __memset_memory_pressure_bump(void);
+
+extern struct file_operations proc_memset_operations;
+extern char *memset_task_status_allowed(struct task_struct *task, char *buffer);
+extern int memset_mem_spread_node(void);
+
+static inline int memset_do_page_mem_spread(void)
+{
+ return current->flags & PF_SPREAD_PAGE;
+}
+
+static inline int memset_do_slab_mem_spread(void)
+{
+ return current->flags & PF_SPREAD_SLAB;
+}
+
+extern void memset_track_online_nodes(void);
+
+extern int current_memset_is_being_rebound(void);
+
+#else /* !CONFIG_MEMSETS */
+
+static inline int memset_init_early(void) { return 0; }
+static inline int memset_init(void) { return 0; }
+static inline void memset_init_smp(void) {}
+
+static inline nodemask_t memset_mems_allowed(struct task_struct *p)
+{
+ return node_possible_map;
+}
+
+static inline void memset_init_current_mems_allowed(void) {}
+static inline void memset_update_task_memory_state(void) {}
+#define memset_nodes_subset_current_mems_allowed(nodes) (1)
```

```
+
+static inline int memset_zonelist_valid_mems_allowed(struct zonelist *zl)
+{
+ return 1;
+}
+
+static inline int memset_zone_allowed(struct zone *z, gfp_t gfp_mask)
+{
+ return 1;
+}
+
+static inline int memset_excl_nodes_overlap(const struct task_struct *p)
+{
+ return 1;
+}
+
+static inline void memset_memory_pressure_bump(void) {}
+
+static inline char *memset_task_status_allowed(struct task_struct *task,
+       char *buffer)
+{
+ return buffer;
+}
+
+static inline int memset_mem_spread_node(void)
+{
+ return 0;
+}
+
+static inline int memset_do_page_mem_spread(void)
+{
+ return 0;
+}
+
+static inline int memset_do_slab_mem_spread(void)
+{
+ return 0;
+}
+
+static inline void memset_track_online_nodes(void) {}
+
+static inline int current_memset_is_being_rebound(void)
+{
+ return 0;
+}
+
+#endif /* !CONFIG_MEMSETS */
+
```

```
+#endif /* _LINUX_MEMSET_H */
Index: container-2.6.19-rc6/kernel/cpuset.c
===================================================================
--- container-2.6.19-rc6.orig/kernel/cpuset.c
+++ container-2.6.19-rc6/kernel/cpuset.c
@@ -32,7 +32,6 @@
 #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>
@@ -56,42 +55,18 @@
 #include <asm/atomic.h>
 #include <linux/mutex.h>

-/*
- * Tracks how many cpusets are currently defined in system.
- * When there is only one cpuset (the root cpuset) we can
- * short circuit some hooks.
- */
-int number_of_cpusets __read_mostly;
-
 /* Retrieve the cpuset from a container */
 static struct container_subsys cpuset_subsys;
 struct cpuset;

-/* See "Frequency meter" comments, below. */
-
-struct fmeter {
- int cnt;  /* unprocessed events count */
- int val;  /* most recent output value */
- time_t time;  /* clock (secs) when val computed */
- spinlock_t lock; /* guards read or write of above */
-};
-
 struct cpuset {
  struct container_subsys_state css;

  unsigned long flags;  /* "unsigned long" so bitops work */
  cpumask_t cpus_allowed;  /* CPUs allowed to tasks in cpuset */
- nodemask_t mems_allowed; /* Memory Nodes allowed to tasks */

  struct cpuset *parent;  /* my parent */

- /*
-  * Copy of global cpuset_mems_generation as of the most
```

```
-  * recent time this cpuset changed its mems_allowed.
-  */
- int mems_generation;
-
- struct fmeter fmeter;  /* memory_pressure filter */
 };

 /* Update the cpuset for a container */
@@ -117,10 +92,6 @@ static inline struct cpuset *task_cs(str
 /* bits in struct cpuset flags field */
 typedef enum {
  CS_CPU_EXCLUSIVE,
- CS_MEM_EXCLUSIVE,
- CS_MEMORY_MIGRATE,
- CS_SPREAD_PAGE,
- CS_SPREAD_SLAB,
 } cpuset_flagbits_t;

 /* convenient tests for these bits */
@@ -129,51 +100,10 @@ static inline int is_cpu_exclusive(const
  return test_bit(CS_CPU_EXCLUSIVE, &cs->flags);
 }

-static inline int is_mem_exclusive(const struct cpuset *cs)
-{
- return test_bit(CS_MEM_EXCLUSIVE, &cs->flags);
-}
-
-static inline int is_memory_migrate(const struct cpuset *cs)
-{
- return test_bit(CS_MEMORY_MIGRATE, &cs->flags);
-}
-
-static inline int is_spread_page(const struct cpuset *cs)
-{
- return test_bit(CS_SPREAD_PAGE, &cs->flags);
-}
-
-static inline int is_spread_slab(const struct cpuset *cs)
-{
- return test_bit(CS_SPREAD_SLAB, &cs->flags);
-}
-
-/*
- * Increment this integer everytime any cpuset changes its
- * mems_allowed value.  Users of cpusets can track this generation
- * number, and avoid having to lock and reload mems_allowed unless
- * the cpuset they're using changes generation.
```

```
- *
- * A single, global generation is needed because attach_task() could
- * reattach a task to a different cpuset, which must not have its
- * generation numbers aliased with those of that tasks previous cpuset.
- *
- * Generations are needed for mems_allowed because one task cannot
- * modify anothers memory placement.  So we must enable every task,
- * on every visit to __alloc_pages(), to efficiently check whether
- * its current->cpuset->mems_allowed has changed, requiring an update
- * of its current->mems_allowed.
- *
- * Since cpuset_mems_generation is guarded by manage_mutex,
- * there is no need to mark it atomic.
- */
-static int cpuset_mems_generation;

 static struct cpuset top_cpuset = {
- .flags = ((1 << CS_CPU_EXCLUSIVE) | (1 << CS_MEM_EXCLUSIVE)),
+ .flags = 1 << CS_CPU_EXCLUSIVE,
  .cpus_allowed = CPU_MASK_ALL,
- .mems_allowed = NODE_MASK_ALL,
 };

 /* This is ugly, but preserves the userspace API for existing cpuset
@@ -189,7 +119,7 @@ static int cpuset_get_sb(struct file_sys
  if (container_fs) {
   ret = container_fs->get_sb(container_fs, flags,
         unused_dev_name,
-        "cpuset", mnt);
+        "cpuset,memset", mnt);
   put_filesystem(container_fs);
  }
  return ret;
@@ -226,120 +156,17 @@ static void guarantee_online_cpus(const
 }

 /*
- * Return in *pmask the portion of a cpusets's mems_allowed that
- * are online.  If none are online, walk up the cpuset hierarchy
- * until we find one that does have some online mems.  If we get
- * all the way to the top and still haven't found any online mems,
- * return node_online_map.
- *
- * One way or another, we guarantee to return some non-empty subset
- * of node_online_map.
- *
- * Call with callback_mutex held.
- */
```

```
-
-static void guarantee_online_mems(const struct cpuset *cs, nodemask_t *pmask)
-{
- while (cs && !nodes_intersects(cs->mems_allowed, node_online_map))
-  cs = cs->parent;
- if (cs)
-  nodes_and(*pmask, cs->mems_allowed, node_online_map);
- else
-  *pmask = node_online_map;
- BUG_ON(!nodes_intersects(*pmask, node_online_map));
-}
-
-/**
- * cpuset_update_task_memory_state - update task memory placement
- *
- * If the current tasks cpusets mems_allowed changed behind our
- * backs, update current->mems_allowed, mems_generation and task NUMA
- * mempolicy to the new value.
- *
- * Task mempolicy is updated by rebinding it relative to the
- * current->cpuset if a task has its memory placement changed.
- * Do not call this routine if in_interrupt().
- *
- * Call without callback_mutex or task_lock() held.  May be
- * called with or without manage_mutex held.  Thanks in part to
- * 'the_top_cpuset_hack', the tasks cpuset pointer will never
- * be NULL.  This routine also might acquire callback_mutex and
- * current->mm->mmap_sem during call.
- *
- * Reading current->cpuset->mems_generation doesn't need task_lock
- * to guard the current->cpuset derefence, because it is guarded
- * from concurrent freeing of current->cpuset by attach_task(),
- * using RCU.
- *
- * The rcu_dereference() is technically probably not needed,
- * as I don't actually mind if I see a new cpuset pointer but
- * an old value of mems_generation.  However this really only
- * matters on alpha systems using cpusets heavily.  If I dropped
- * that rcu_dereference(), it would save them a memory barrier.
- * For all other arch's, rcu_dereference is a no-op anyway, and for
- * alpha systems not using cpusets, another planned optimization,
- * avoiding the rcu critical section for tasks in the root cpuset
- * which is statically allocated, so can't vanish, will make this
- * irrelevant.  Better to use RCU as intended, than to engage in
- * some cute trick to save a memory barrier that is impossible to
- * test, for alpha systems using cpusets heavily, which might not
- * even exist.
- *
```

```
- * This routine is needed to update the per-task mems_allowed data,
- * within the tasks context, when it is trying to allocate memory
- * (in various mm/mempolicy.c routines) and notices that some other
- * task has been modifying its cpuset.
- */
-
-void cpuset_update_task_memory_state(void)
-{
- int my_cpusets_mem_gen;
- struct task_struct *tsk = current;
- struct cpuset *cs;
-
- 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();
-  my_cpusets_mem_gen = task_cs(current)->mems_generation;
-  rcu_read_unlock();
- }
-
- if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
-  container_lock();
-  task_lock(tsk);
-  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))
-   tsk->flags |= PF_SPREAD_PAGE;
-  else
-   tsk->flags &= ~PF_SPREAD_PAGE;
-  if (is_spread_slab(cs))
-   tsk->flags |= PF_SPREAD_SLAB;
-  else
-   tsk->flags &= ~PF_SPREAD_SLAB;
-  task_unlock(tsk);
-  container_unlock();
-  mpol_rebind_task(tsk, &tsk->mems_allowed);
- }
-}
-
-/*
  * is_cpuset_subset(p, q) - Is cpuset p a subset of cpuset q?
  *
- * One cpuset is a subset of another if all its allowed CPUs and
- * Memory Nodes are a subset of the other, and its exclusive flags
- * are only set if the other's are set.  Call holding manage_mutex.
+ * One cpuset is a subset of another if all its allowed CPUs are a
```

```
+ * subset of the other, and its exclusive flags are only set if the
+ * other's are set.  Call holding manage_mutex.
  */

 static int is_cpuset_subset(const struct cpuset *p, const struct cpuset *q)
 {
  return cpus_subset(p->cpus_allowed, q->cpus_allowed) &&
-  nodes_subset(p->mems_allowed, q->mems_allowed) &&
-  is_cpu_exclusive(p) <= is_cpu_exclusive(q) &&
-  is_mem_exclusive(p) <= is_mem_exclusive(q);
+  is_cpu_exclusive(p) <= is_cpu_exclusive(q);
 }

 /*
@@ -356,7 +183,7 @@ static int is_cpuset_subset(const struct
  * cpuset in the list must use cur below, not trial.
  *
  * 'trial' is the address of bulk structure copy of cur, with
- * perhaps one or more of the fields cpus_allowed, mems_allowed,
+ * perhaps one or more of the fields cpus_allowed,
  * or flags changed to new, trial values.
  *
  * Return 0 if valid, -errno if not.
@@ -388,10 +215,6 @@ static int validate_change(const struct
       c != cur &&
       cpus_intersects(trial->cpus_allowed, c->cpus_allowed))
    return -EINVAL;
-  if ((is_mem_exclusive(trial) || is_mem_exclusive(c)) &&
-      c != cur &&
-      nodes_intersects(trial->mems_allowed, c->mems_allowed))
-   return -EINVAL;
  }

  return 0;
@@ -487,211 +310,10 @@ static int update_cpumask(struct cpuset
  return 0;
 }

-/*
- * cpuset_migrate_mm
- *
- *    Migrate memory region from one set of nodes to another.
- *
- *    Temporarilly set tasks mems_allowed to target nodes of migration,
- *    so that the migration code can allocate pages on these nodes.
- *
- *    Call holding manage_mutex, so our current->cpuset won't change
- *    during this call, as manage_mutex holds off any attach_task()
```

```
- *    calls.  Therefore we don't need to take task_lock around the
- *    call to guarantee_online_mems(), as we know no one is changing
- *    our tasks cpuset.
- *
- *    Hold callback_mutex around the two modifications of our tasks
- *    mems_allowed to synchronize with cpuset_mems_allowed().
- *
- *    While the mm_struct we are migrating is typically from some
- *    other task, the task_struct mems_allowed that we are hacking
- *    is for our current task, which must allocate new pages for that
- *    migrating memory region.
- *
- *    We call cpuset_update_task_memory_state() before hacking
- *    our tasks mems_allowed, so that we are assured of being in
- *    sync with our tasks cpuset, and in particular, callbacks to
- *    cpuset_update_task_memory_state() from nested page allocations
- *    won't see any mismatch of our cpuset and task mems_generation
- *    values, so won't overwrite our hacked tasks mems_allowed
- *    nodemask.
- */
-
-static void cpuset_migrate_mm(struct mm_struct *mm, const nodemask_t *from,
-       const nodemask_t *to)
-{
- struct task_struct *tsk = current;
-
- cpuset_update_task_memory_state();
-
- container_lock();
- tsk->mems_allowed = *to;
- container_unlock();
-
- do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);
-
- container_lock();
- guarantee_online_mems(task_cs(tsk),&tsk->mems_allowed);
- container_unlock();
-}
-
-/*
- * Handle user request to change the 'mems' memory placement
- * of a cpuset.  Needs to validate the request, update the
- * cpusets mems_allowed and mems_generation, and for each
- * task in the cpuset, rebind any vma mempolicies and if
- * the cpuset is marked 'memory_migrate', migrate the tasks
- * pages to the new memory.
- *
- * Call with manage_mutex held.  May take callback_mutex during call.
```

```
- * Will take tasklist_lock, scan tasklist for tasks in cpuset cs,
- * lock each such tasks mm->mmap_sem, scan its vma's and rebind
- * 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;
- nodemask_t oldmem;
- struct task_struct *g, *p;
- struct mm_struct **mmarray;
- int i, n, ntasks;
- int migrate;
- int fudge;
- int retval;
- struct container *cont;
-
- /* top_cpuset.mems_allowed tracks node_online_map; it's read-only */
- if (cs == &top_cpuset)
-  return -EACCES;
-
- trialcs = *cs;
- cont = cs->css.container;
- retval = nodelist_parse(buf, trialcs.mems_allowed);
- if (retval < 0)
-  goto done;
- nodes_and(trialcs.mems_allowed, trialcs.mems_allowed, node_online_map);
- oldmem = cs->mems_allowed;
- if (nodes_equal(oldmem, trialcs.mems_allowed)) {
-  retval = 0;  /* Too easy - nothing to do */
-  goto done;
- }
- if (nodes_empty(trialcs.mems_allowed)) {
-  retval = -ENOSPC;
-  goto done;
- }
- retval = validate_change(cs, &trialcs);
- if (retval < 0)
-  goto done;
-
- container_lock();
- cs->mems_allowed = trialcs.mems_allowed;
- cs->mems_generation = cpuset_mems_generation++;
- container_unlock();
-
- cpuset_being_rebound = cs;  /* causes mpol_copy() rebind */
```

```
-
- fudge = 10;    /* spare mmarray[] slots */
- fudge += cpus_weight(cs->cpus_allowed); /* imagine one fork-bomb/cpu */
- retval = -ENOMEM;
-
- /*
-  * Allocate mmarray[] to hold mm reference for each task
-  * in cpuset cs.  Can't kmalloc GFP_KERNEL while holding
-  * tasklist_lock.  We could use GFP_ATOMIC, but with a
-  * few more lines of code, we can retry until we get a big
-  * enough mmarray[] w/o using GFP_ATOMIC.
-  */
- while (1) {
-  ntasks = atomic_read(&cs->css.container->count);  /* guess */
-  ntasks += fudge;
-  mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
-  if (!mmarray)
-   goto done;
-  write_lock_irq(&tasklist_lock);  /* block fork */
-  if (atomic_read(&cs->css.container->count) <= ntasks)
-   break;    /* got enough */
-  write_unlock_irq(&tasklist_lock); /* try again */
-  kfree(mmarray);
- }
-
- n = 0;
-
- /* Load up mmarray[] with mm reference for each task in cpuset. */
- do_each_thread(g, p) {
-  struct mm_struct *mm;
-
-  if (n >= ntasks) {
-   printk(KERN_WARNING
-    "Cpuset mempolicy rebind incomplete.\n");
-   continue;
-  }
-  if (task_cs(p) != cs)
-   continue;
-  mm = get_task_mm(p);
-  if (!mm)
-   continue;
-  mmarray[n++] = mm;
- } while_each_thread(g, p);
- write_unlock_irq(&tasklist_lock);
-
- /*
-  * Now that we've dropped the tasklist spinlock, we can
-  * rebind the vma mempolicies of each mm in mmarray[] to their
```

```
-  * new cpuset, and release that mm.  The mpol_rebind_mm()
-  * call takes mmap_sem, which we couldn't take while holding
-  * tasklist_lock.  Forks can happen again now - the mpol_copy()
-  * cpuset_being_rebound check will catch such forks, and rebind
-  * their vma mempolicies too.  Because we still hold the global
-  * cpuset manage_mutex, we know that no other rebind effort will
-  * be contending for the global variable cpuset_being_rebound.
-  * It's ok if we rebind the same mm twice; mpol_rebind_mm()
-  * is idempotent.  Also migrate pages in each mm to new nodes.
-  */
- migrate = is_memory_migrate(cs);
- for (i = 0; i < n; i++) {
-  struct mm_struct *mm = mmarray[i];
-
-  mpol_rebind_mm(mm, &cs->mems_allowed);
-  if (migrate)
-   cpuset_migrate_mm(mm, &oldmem, &cs->mems_allowed);
-  mmput(mm);
- }
-
- /* We're done rebinding vma's to this cpusets new mems_allowed. */
- kfree(mmarray);
- 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.
- */
-
-static int update_memory_pressure_enabled(struct cpuset *cs, char *buf)
-{
- if (simple_strtoul(buf, NULL, 10) != 0)
-  cpuset_memory_pressure_enabled = 1;
- else
-  cpuset_memory_pressure_enabled = 0;
- return 0;
-}

 /*
  * update_flag - read a 0 or a 1 in a file and update associated flag
```

```
- * bit: the bit to update (CS_CPU_EXCLUSIVE, CS_MEM_EXCLUSIVE,
- *    CS_NOTIFY_ON_RELEASE, CS_MEMORY_MIGRATE,
- *    CS_SPREAD_PAGE, CS_SPREAD_SLAB)
+ * bit: the bit to update (CS_CPU_EXCLUSIVE)
  * cs: the cpuset to update
  * buf: the buffer where we read the 0 or 1
  *
@@ -729,110 +351,12 @@ static int update_flag(cpuset_flagbits_t
  return 0;
 }

-/*
- * Frequency meter - How fast is some event occurring?
- *
- * These routines manage a digitally filtered, constant time based,
- * event frequency meter.  There are four routines:
- *   fmeter_init() - initialize a frequency meter.
- *   fmeter_markevent() - called each time the event happens.
- *   fmeter_getrate() - returns the recent rate of such events.
- *   fmeter_update() - internal routine used to update fmeter.
- *
- * A common data structure is passed to each of these routines,
- * which is used to keep track of the state required to manage the
- * frequency meter and its digital filter.
- *
- * The filter works on the number of events marked per unit time.
- * The filter is single-pole low-pass recursive (IIR).  The time unit
- * is 1 second.  Arithmetic is done using 32-bit integers scaled to
- * simulate 3 decimal digits of precision (multiplied by 1000).
- *
- * With an FM_COEF of 933, and a time base of 1 second, the filter
- * has a half-life of 10 seconds, meaning that if the events quit
- * happening, then the rate returned from the fmeter_getrate()
- * will be cut in half each 10 seconds, until it converges to zero.
- *
- * It is not worth doing a real infinitely recursive filter.  If more
- * than FM_MAXTICKS ticks have elapsed since the last filter event,
- * just compute FM_MAXTICKS ticks worth, by which point the level
- * will be stable.
- *
- * Limit the count of unprocessed events to FM_MAXCNT, so as to avoid
- * arithmetic overflow in the fmeter_update() routine.
- *
- * Given the simple 32 bit integer arithmetic used, this meter works
- * best for reporting rates between one per millisecond (msec) and
- * one per 32 (approx) seconds.  At constant rates faster than one
- * per msec it maxes out at values just under 1,000,000.  At constant
- * rates between one per msec, and one per second it will stabilize
```

```
- * to a value N*1000, where N is the rate of events per second.
- * At constant rates between one per second and one per 32 seconds,
- * it will be choppy, moving up on the seconds that have an event,
- * and then decaying until the next event.  At rates slower than
- * about one in 32 seconds, it decays all the way back to zero between
- * each event.
- */
-
-#define FM_COEF 933  /* coefficient for half-life of 10 secs */
-#define FM_MAXTICKS ((time_t)99) /* useless computing more ticks than this */
-#define FM_MAXCNT 1000000 /* limit cnt to avoid overflow */
-#define FM_SCALE 1000  /* faux fixed point scale */
-
-/* Initialize a frequency meter */
-static void fmeter_init(struct fmeter *fmp)
-{
- fmp->cnt = 0;
- fmp->val = 0;
- fmp->time = 0;
- spin_lock_init(&fmp->lock);
-}
-
-/* Internal meter update - process cnt events and update value */
-static void fmeter_update(struct fmeter *fmp)
-{
- time_t now = get_seconds();
- time_t ticks = now - fmp->time;
-
- if (ticks == 0)
-  return;
-
- ticks = min(FM_MAXTICKS, ticks);
- while (ticks-- > 0)
-  fmp->val = (FM_COEF * fmp->val) / FM_SCALE;
- fmp->time = now;
-
- fmp->val += ((FM_SCALE - FM_COEF) * fmp->cnt) / FM_SCALE;
- fmp->cnt = 0;
-}
-
-/* Process any previous ticks, then bump cnt by one (times scale). */
-static void fmeter_markevent(struct fmeter *fmp)
-{
- spin_lock(&fmp->lock);
- fmeter_update(fmp);
- fmp->cnt = min(FM_MAXCNT, fmp->cnt + FM_SCALE);
- spin_unlock(&fmp->lock);
-}
```

```
-
-/* Process any previous ticks, then return current value. */
-static int fmeter_getrate(struct fmeter *fmp)
-{
- int val;
-
- spin_lock(&fmp->lock);
- fmeter_update(fmp);
- val = fmp->val;
- spin_unlock(&fmp->lock);
- return val;
-}
-
 int cpuset_can_attach(struct container_subsys *ss,
         struct container *cont, struct task_struct *tsk)
 {
  struct cpuset *cs = container_cs(cont);

- if (cpus_empty(cs->cpus_allowed) || nodes_empty(cs->mems_allowed))
+ if (cpus_empty(cs->cpus_allowed))
    return -ENOSPC;

  return security_task_setscheduler(tsk, 0, NULL);
@@ -846,40 +370,11 @@ void cpuset_attach(struct container_subs
  set_cpus_allowed(tsk, cpus);
 }

-void cpuset_post_attach(struct container_subsys *ss,
-    struct container *cont,
-    struct container *oldcont,
-    struct task_struct *tsk)
-{
- nodemask_t from, to;
- struct mm_struct *mm;
- struct cpuset *cs = container_cs(cont);
- struct cpuset *oldcs = container_cs(oldcont);
-
- from = oldcs->mems_allowed;
- to = cs->mems_allowed;
- mm = get_task_mm(tsk);
- if (mm) {
-  mpol_rebind_mm(mm, &to);
-  if (is_memory_migrate(cs))
-   cpuset_migrate_mm(mm, &from, &to);
-  mmput(mm);
- }
-
-}
```

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

 typedef enum {
- FILE_MEMORY_MIGRATE,
  FILE_CPULIST,
- FILE_MEMLIST,
  FILE_CPU_EXCLUSIVE,
- FILE_MEM_EXCLUSIVE,
- FILE_MEMORY_PRESSURE_ENABLED,
- FILE_MEMORY_PRESSURE,
- FILE_SPREAD_PAGE,
- FILE_SPREAD_SLAB,
 } cpuset_filetype_t;

 static ssize_t cpuset_common_file_write(struct container *cont,
@@ -894,7 +389,7 @@ static ssize_t cpuset_common_file_write(
  int retval = 0;

  /* Crude upper limit on largest legitimate list user might write. */
- if (nbytes > 100 + 6 * max(NR_CPUS, MAX_NUMNODES))
+ if (nbytes > 100 + 6 * NR_CPUS)
   return -E2BIG;

  /* +1 for nul-terminator */
@@ -918,32 +413,9 @@ static ssize_t cpuset_common_file_write(
  case FILE_CPULIST:
   retval = update_cpumask(cs, buffer);
   break;
- case FILE_MEMLIST:
-  retval = update_nodemask(cs, buffer);
-  break;
  case FILE_CPU_EXCLUSIVE:
   retval = update_flag(CS_CPU_EXCLUSIVE, cs, buffer);
   break;
- case FILE_MEM_EXCLUSIVE:
-  retval = update_flag(CS_MEM_EXCLUSIVE, cs, buffer);
-  break;
- case FILE_MEMORY_MIGRATE:
-  retval = update_flag(CS_MEMORY_MIGRATE, cs, buffer);
-  break;
- case FILE_MEMORY_PRESSURE_ENABLED:
-  retval = update_memory_pressure_enabled(cs, buffer);
-  break;
- case FILE_MEMORY_PRESSURE:
-  retval = -EACCES;
-  break;
- case FILE_SPREAD_PAGE:
```

```
-  retval = update_flag(CS_SPREAD_PAGE, cs, buffer);
-  cs->mems_generation = cpuset_mems_generation++;
-  break;
- case FILE_SPREAD_SLAB:
-  retval = update_flag(CS_SPREAD_SLAB, cs, buffer);
-  cs->mems_generation = cpuset_mems_generation++;
-  break;
   default:
    retval = -EINVAL;
    goto out2;
@@ -981,17 +453,6 @@ static int cpuset_sprintf_cpulist(char *
   return cpulist_scnprintf(page, PAGE_SIZE, mask);
 }

-static int cpuset_sprintf_memlist(char *page, struct cpuset *cs)
-{
- nodemask_t mask;
-
- container_lock();
- mask = cs->mems_allowed;
- container_unlock();
-
- return nodelist_scnprintf(page, PAGE_SIZE, mask);
-}
-
 static ssize_t cpuset_common_file_read(struct container *cont,
            struct cftype *cft,
            struct file *file,
@@ -1013,30 +474,9 @@ static ssize_t cpuset_common_file_read(s
   case FILE_CPULIST:
    s += cpuset_sprintf_cpulist(s, cs);
    break;
- case FILE_MEMLIST:
-  s += cpuset_sprintf_memlist(s, cs);
-  break;
   case FILE_CPU_EXCLUSIVE:
    *s++ = is_cpu_exclusive(cs) ? '1' : '0';
    break;
- case FILE_MEM_EXCLUSIVE:
-  *s++ = is_mem_exclusive(cs) ? '1' : '0';
-  break;
- case FILE_MEMORY_MIGRATE:
-  *s++ = is_memory_migrate(cs) ? '1' : '0';
-  break;
- case FILE_MEMORY_PRESSURE_ENABLED:
-  *s++ = cpuset_memory_pressure_enabled ? '1' : '0';
-  break;
- case FILE_MEMORY_PRESSURE:
```

```
-  s += sprintf(s, "%d", fmeter_getrate(&cs->fmeter));
-  break;
- case FILE_SPREAD_PAGE:
-  *s++ = is_spread_page(cs) ? '1' : '0';
-  break;
- case FILE_SPREAD_SLAB:
-  *s++ = is_spread_slab(cs) ? '1' : '0';
-  break;
   default:
    retval = -EINVAL;
    goto out;
@@ -1061,13 +501,6 @@ static struct cftype cft_cpus = {
   .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,
@@ -1075,71 +508,14 @@ static struct cftype cft_cpu_exclusive =
   .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_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,
-};
-
 int cpuset_populate(struct container_subsys *ss, struct container *cont)
 {
  int err;

  if ((err = container_add_file(cont, &cft_cpus)) < 0)
   return err;
- if ((err = container_add_file(cont, &cft_mems)) < 0)
-  return err;
  if ((err = container_add_file(cont, &cft_cpu_exclusive)) < 0)
   return err;
- if ((err = container_add_file(cont, &cft_mem_exclusive)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_memory_migrate)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_memory_pressure)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_spread_page)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_spread_slab)) < 0)
-  return err;
- /* memory_pressure_enabled is in root cpuset only */
- if (err == 0 && !cont->parent)
-  err = container_add_file(cont, &cft_memory_pressure_enabled);
  return 0;
 }
```

```
@@ -1161,7 +537,6 @@ int cpuset_create(struct container_subsy
   /* This is early initialization for the top container */
   set_container_cs(cont, &top_cpuset);
   top_cpuset.css.container = cont;
-  top_cpuset.mems_generation = cpuset_mems_generation++;
   return 0;
  }
  parent = container_cs(cont->parent);
@@ -1169,21 +544,12 @@ int cpuset_create(struct container_subsy
  if (!cs)
   return -ENOMEM;

- cpuset_update_task_memory_state();
  cs->flags = 0;
- if (is_spread_page(parent))
-  set_bit(CS_SPREAD_PAGE, &cs->flags);
- if (is_spread_slab(parent))
-  set_bit(CS_SPREAD_SLAB, &cs->flags);
  cs->cpus_allowed = CPU_MASK_NONE;
- cs->mems_allowed = NODE_MASK_NONE;
- cs->mems_generation = cpuset_mems_generation++;
- fmeter_init(&cs->fmeter);

  cs->parent = parent;
  set_container_cs(cont, cs);
  cs->css.container = cont;
- number_of_cpusets++;
  return 0;
 }

@@ -1202,12 +568,10 @@ void cpuset_destroy(struct container_sub
 {
  struct cpuset *cs = container_cs(cont);

- cpuset_update_task_memory_state();
  if (is_cpu_exclusive(cs)) {
   int retval = update_flag(CS_CPU_EXCLUSIVE, cs, "0");
   BUG_ON(retval);
  }
- number_of_cpusets--;
  kfree(cs);
 }

@@ -1217,7 +581,6 @@ static struct container_subsys cpuset_su
  .destroy  = cpuset_destroy,
  .can_attach = cpuset_can_attach,
  .attach = cpuset_attach,
```

```
- .post_attach = cpuset_post_attach,
  .populate = cpuset_populate,
  .subsys_id = -1,
 };
@@ -1232,7 +595,6 @@ int __init cpuset_init_early(void)
 {
  if (container_register_subsys(&cpuset_subsys) < 0)
   panic("Couldn't register cpuset subsystem");
- top_cpuset.mems_generation = cpuset_mems_generation++;
  return 0;
 }

@@ -1248,119 +610,76 @@ int __init cpuset_init(void)
  int err = 0;

  top_cpuset.cpus_allowed = CPU_MASK_ALL;
- top_cpuset.mems_allowed = NODE_MASK_ALL;
-
- fmeter_init(&top_cpuset.fmeter);
- top_cpuset.mems_generation = cpuset_mems_generation++;

  err = register_filesystem(&cpuset_fs_type);
  if (err < 0)
   return err;

- number_of_cpusets = 1;
  return 0;
 }

-#if defined(CONFIG_HOTPLUG_CPU) || defined(CONFIG_MEMORY_HOTPLUG)
+#if defined(CONFIG_HOTPLUG_CPU)
 /*
- * If common_cpu_mem_hotplug_unplug(), below, unplugs any CPUs
- * or memory nodes, we need to walk over the cpuset hierarchy,
- * removing that CPU or node from all cpusets.  If this removes the
- * last CPU or node from a cpuset, then the guarantee_online_cpus()
- * or guarantee_online_mems() code will use that emptied cpusets
- * parent online CPUs or nodes.  Cpusets that were already empty of
- * CPUs or nodes are left empty.
+ * If we unplug any CPUs or memory nodes, we need to walk over the
+ * cpuset hierarchy, removing that CPU from all cpusets.  If this
+ * removes the last CPU from a cpuset, then the
+ * guarantee_online_cpus() code will use that emptied cpusets parent
+ * online CPUs.  Cpusets that were already empty of CPUs are left
+ * empty.
  *
- * This routine is intentionally inefficient in a couple of regards.
- * It will check all cpusets in a subtree even if the top cpuset of
```

```
- * the subtree has no offline CPUs or nodes.  It checks both CPUs and
- * nodes, even though the caller could have been coded to know that
- * only one of CPUs or nodes needed to be checked on a given call.
- * This was done to minimize text size rather than cpu cycles.
+ * This routine will check all cpusets in a subtree even if the top
+ * cpuset of the subtree has no offline CPUs.
  *
  * Call with both manage_mutex and callback_mutex held.
  *
  * Recursive, on depth of cpuset subtree.
  */

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

- /* Each of our child cpusets mems must be online */
+ /* Each of our child cpusets cpus must be online */
  list_for_each_entry(cont, &cur->css.container->children, sibling) {
   c = container_cs(cont);
-  guarantee_online_cpus_mems_in_subtree(c);
+  guarantee_online_cpus_in_subtree(c);
   if (!cpus_empty(c->cpus_allowed))
    guarantee_online_cpus(c, &c->cpus_allowed);
-  if (!nodes_empty(c->mems_allowed))
-   guarantee_online_mems(c, &c->mems_allowed);
  }
 }

 /*
- * The cpus_allowed and mems_allowed nodemasks in the top_cpuset track
- * cpu_online_map and node_online_map.  Force the top cpuset to track
- * whats online after any CPU or memory node hotplug or unplug event.
+ * The top_cpuset tracks what CPUs are online,
+ * period.  This is necessary in order to make cpusets transparent
+ * (of no affect) on systems that are actively using CPU hotplug
+ * but making no active use of cpusets.
  *
- * To ensure that we don't remove a CPU or node from the top cpuset
+ * To ensure that we don't remove a CPU from the top cpuset
  * that is currently in use by a child cpuset (which would violate
  * the rule that cpusets must be subsets of their parent), we first
- * call the recursive routine guarantee_online_cpus_mems_in_subtree().
+ * call the recursive routine guarantee_online_cpus_in_subtree().
  *
- * Since there are two callers of this routine, one for CPU hotplug
```

```
- * events and one for memory node hotplug events, we could have coded
- * two separate routines here.  We code it as a single common routine
- * in order to minimize text size.
+ * This routine ensures that top_cpuset.cpus_allowed tracks
+ * cpu_online_map on each CPU hotplug (cpuhp) event.
  */

-static void common_cpu_mem_hotplug_unplug(void)
+static int cpuset_handle_cpuhp(struct notifier_block *nb,
+    unsigned long phase, void *cpu)
 {
  container_manage_lock();
  container_lock();

- guarantee_online_cpus_mems_in_subtree(&top_cpuset);
+ guarantee_online_cpus_in_subtree(&top_cpuset);
  top_cpuset.cpus_allowed = cpu_online_map;
- top_cpuset.mems_allowed = node_online_map;

  container_unlock();
  container_manage_unlock();
-}
-#endif

-#ifdef CONFIG_HOTPLUG_CPU
-/*
- * The top_cpuset tracks what CPUs and Memory Nodes are online,
- * period.  This is necessary in order to make cpusets transparent
- * (of no affect) on systems that are actively using CPU hotplug
- * but making no active use of cpusets.
- *
- * This routine ensures that top_cpuset.cpus_allowed tracks
- * cpu_online_map on each CPU hotplug (cpuhp) event.
- */
-
-static int cpuset_handle_cpuhp(struct notifier_block *nb,
-    unsigned long phase, void *cpu)
-{
- common_cpu_mem_hotplug_unplug();
  return 0;
 }
 #endif

-#ifdef CONFIG_MEMORY_HOTPLUG
-/*
- * Keep top_cpuset.mems_allowed tracking node_online_map.
- * Call this routine anytime after you change node_online_map.
- * See also the previous routine cpuset_handle_cpuhp().
```

```
- */
-
-void cpuset_track_online_nodes(void)
-{
- common_cpu_mem_hotplug_unplug();
-}
-#endif
-
 /**
  * cpuset_init_smp - initialize cpus_allowed
  *
@@ -1370,7 +689,6 @@ void cpuset_track_online_nodes(void)
 void __init cpuset_init_smp(void)
 {
  top_cpuset.cpus_allowed = cpu_online_map;
- top_cpuset.mems_allowed = node_online_map;

  hotcpu_notifier(cpuset_handle_cpuhp, 0);
 }
@@ -1398,249 +716,6 @@ cpumask_t cpuset_cpus_allowed(struct tas
  return mask;
 }

-void cpuset_init_current_mems_allowed(void)
-{
- current->mems_allowed = NODE_MASK_ALL;
-}
-
-/**
- * cpuset_mems_allowed - return mems_allowed mask from a tasks cpuset.
- * @tsk: pointer to task_struct from which to obtain cpuset->mems_allowed.
- *
- * Description: Returns the nodemask_t mems_allowed of the cpuset
- * attached to the specified @tsk.  Guaranteed to return some non-empty
- * subset of node_online_map, even if this means going outside the
- * tasks cpuset.
- **/
-
-nodemask_t cpuset_mems_allowed(struct task_struct *tsk)
-{
- nodemask_t mask;
-
- container_lock();
- task_lock(tsk);
- guarantee_online_mems(task_cs(tsk), &mask);
- task_unlock(tsk);
- container_unlock();
-
```

```
- return mask;
-}
-
-/**
- * cpuset_zonelist_valid_mems_allowed - check zonelist vs. curremt mems_allowed
- * @zl: the zonelist to be checked
- *
- * Are any of the nodes on zonelist zl allowed in current->mems_allowed?
- */
-int cpuset_zonelist_valid_mems_allowed(struct zonelist *zl)
-{
- int i;
-
- for (i = 0; zl->zones[i]; i++) {
-  int nid = zone_to_nid(zl->zones[i]);
-
-  if (node_isset(nid, current->mems_allowed))
-   return 1;
- }
- return 0;
-}
-
-/*
- * nearest_exclusive_ancestor() - Returns the nearest mem_exclusive
- * ancestor to the specified cpuset.  Call holding callback_mutex.
- * If no ancestor is mem_exclusive (an unusual configuration), then
- * returns the root cpuset.
- */
-static const struct cpuset *nearest_exclusive_ancestor(const struct cpuset *cs)
-{
- while (!is_mem_exclusive(cs) && cs->parent)
-  cs = cs->parent;
- return cs;
-}
-
-/**
- * cpuset_zone_allowed - Can we allocate memory on zone z's memory node?
- * @z: is this zone on an allowed node?
- * @gfp_mask: memory allocation flags (we use __GFP_HARDWALL)
- *
- * If we're in interrupt, yes, we can always allocate.  If zone
- * z's node is in our tasks mems_allowed, yes.  If it's not a
- * __GFP_HARDWALL request and this zone's nodes is in the nearest
- * mem_exclusive cpuset ancestor to this tasks cpuset, yes.
- * Otherwise, no.
- *
- * GFP_USER allocations are marked with the __GFP_HARDWALL bit,
- * and do not allow allocations outside the current tasks cpuset.
```

```
- * GFP_KERNEL allocations are not so marked, so can escape to the
- * nearest mem_exclusive ancestor cpuset.
- *
- * Scanning up parent cpusets requires callback_mutex.  The __alloc_pages()
- * routine only calls here with __GFP_HARDWALL bit _not_ set if
- * it's a GFP_KERNEL allocation, and all nodes in the current tasks
- * mems_allowed came up empty on the first pass over the zonelist.
- * So only GFP_KERNEL allocations, if all nodes in the cpuset are
- * short of memory, might require taking the callback_mutex mutex.
- *
- * The first call here from mm/page_alloc:get_page_from_freelist()
- * has __GFP_HARDWALL set in gfp_mask, enforcing hardwall cpusets, so
- * no allocation on a node outside the cpuset is allowed (unless in
- * interrupt, of course).
- *
- * The second pass through get_page_from_freelist() doesn't even call
- * here for GFP_ATOMIC calls.  For those calls, the __alloc_pages()
- * variable 'wait' is not set, and the bit ALLOC_CPUSET is not set
- * in alloc_flags.  That logic and the checks below have the combined
- * affect that:
- * in_interrupt - any node ok (current task context irrelevant)
- * GFP_ATOMIC   - any node ok
- * GFP_KERNEL   - any node in enclosing mem_exclusive cpuset ok
- * GFP_USER     - only nodes in current tasks mems allowed ok.
- *
- * Rule:
- *    Don't call cpuset_zone_allowed() if you can't sleep, unless you
- *    pass in the __GFP_HARDWALL flag set in gfp_flag, which disables
- *    the code that might scan up ancestor cpusets and sleep.
- **/
-
-int __cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask)
-{
- int node;   /* node that zone z is on */
- const struct cpuset *cs; /* current cpuset ancestors */
- int allowed;   /* is allocation in zone z allowed? */
-
- if (in_interrupt() || (gfp_mask & __GFP_THISNODE))
-  return 1;
- node = zone_to_nid(z);
- might_sleep_if(!(gfp_mask & __GFP_HARDWALL));
- if (node_isset(node, current->mems_allowed))
-  return 1;
- if (gfp_mask & __GFP_HARDWALL) /* If hardwall request, stop here */
-  return 0;
-
- if (current->flags & PF_EXITING) /* Let dying task have memory */
-  return 1;
```

-
- /* Not hardwall and node outside mems_allowed: scan up cpusets */
- container_lock();
-
- task_lock(current);
- cs = nearest_exclusive_ancestor(task_cs(current));
- task_unlock(current);
-
- allowed = node_isset(node, cs->mems_allowed);
- container_unlock();
- return allowed;
-}
-
-/**
- * cpuset_mem_spread_node() - On which node to begin search for a page
- *
- * If a task is marked PF_SPREAD_PAGE or PF_SPREAD_SLAB (as for
- * tasks in a cpuset with is_spread_page or is_spread_slab set),
- * and if the memory allocation used cpuset_mem_spread_node()
- * to determine on which node to start looking, as it will for
- * certain page cache or slab cache pages such as used for file
- * system buffers and inode caches, then instead of starting on the
- * local node to look for a free page, rather spread the starting
- * node around the tasks mems_allowed nodes.
- *
- * We don't have to worry about the returned node being offline
- * because "it can't happen", and even if it did, it would be ok.
- *
- * The routines calling guarantee_online_mems() are careful to
- * only set nodes in task->mems_allowed that are online. So it
- * should not be possible for the following code to return an
- * offline node. But if it did, that would be ok, as this routine
- * is not returning the node where the allocation must be, only
- * the node where the search should start. The zonelist passed to
- * __alloc_pages() will include all nodes. If the slab allocator
- * is passed an offline node, it will fall back to the local node.
- * See kmem_cache_alloc_node().
- */
-
-int cpuset_mem_spread_node(void)
-{
- int node;
-
- node = next_node(current->cpuset_mem_spread_rotor, current->mems_allowed);
- if (node == MAX_NUMNODES)
- node = first_node(current->mems_allowed);
- current->cpuset_mem_spread_rotor = node;
- return node;

```
-}
-EXPORT_SYMBOL_GPL(cpuset_mem_spread_node);
-
-/**
- * cpuset_excl_nodes_overlap - Do we overlap @p's mem_exclusive ancestors?
- * @p: pointer to task_struct of some other task.
- *
- * Description: Return true if the nearest mem_exclusive ancestor
- * cpusets of tasks @p and current overlap.  Used by oom killer to
- * determine if task @p's memory usage might impact the memory
- * available to the current task.
- *
- * Call while holding callback_mutex.
- **/
-
-int cpuset_excl_nodes_overlap(const struct task_struct *p)
-{
- const struct cpuset *cs1, *cs2; /* my and p's cpuset ancestors */
- int overlap = 1;  /* do cpusets overlap? */
-
- task_lock(current);
- if (current->flags & PF_EXITING) {
-  task_unlock(current);
-  goto done;
- }
- cs1 = nearest_exclusive_ancestor(task_cs(current));
- task_unlock(current);
-
- task_lock((struct task_struct *)p);
- if (p->flags & PF_EXITING) {
-  task_unlock((struct task_struct *)p);
-  goto done;
- }
- cs2 = nearest_exclusive_ancestor(task_cs((struct task_struct *)p));
- task_unlock((struct task_struct *)p);
-
- overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
-done:
- return overlap;
-}
-
-/*
- * Collection of memory_pressure is suppressed unless
- * this flag is enabled by writing "1" to the special
- * cpuset file 'memory_pressure_enabled' in the root cpuset.
- */
-
-int cpuset_memory_pressure_enabled __read_mostly;
```

```
-
-/**
- * cpuset_memory_pressure_bump - keep stats of per-cpuset reclaims.
- *
- * Keep a running average of the rate of synchronous (direct)
- * page reclaim efforts initiated by tasks in each cpuset.
- *
- * This represents the rate at which some task in the cpuset
- * ran low on memory on all nodes it was allowed to use, and
- * had to enter the kernels page reclaim code in an effort to
- * create more free memory by tossing clean pages or swapping
- * or writing dirty pages.
- *
- * Display to user space in the per-cpuset read-only file
- * "memory_pressure".  Value displayed is an integer
- * representing the recent rate of entry into the synchronous
- * (direct) page reclaim by any task attached to the cpuset.
- **/
-
-void __cpuset_memory_pressure_bump(void)
-{
- task_lock(current);
- fmeter_markevent(&task_cs(current)->fmeter);
- task_unlock(current);
-}
-
 #ifdef CONFIG_PROC_PID_CPUSET
 /*
  * proc_cpuset_show()
@@ -1709,8 +784,5 @@ char *cpuset_task_status_allowed(struct
  buffer += sprintf(buffer, "Cpus_allowed:\t");
  buffer += cpumask_scnprintf(buffer, PAGE_SIZE, task->cpus_allowed);
  buffer += sprintf(buffer, "\n");
- buffer += sprintf(buffer, "Mems_allowed:\t");
- buffer += nodemask_scnprintf(buffer, PAGE_SIZE, task->mems_allowed);
- buffer += sprintf(buffer, "\n");
  return buffer;
 }
Index: container-2.6.19-rc6/kernel/memset.c
===================================================================
--- /dev/null
+++ container-2.6.19-rc6/kernel/memset.c
@@ -0,0 +1,1352 @@
+/*
+ *  kernel/memset.c
+ *
+ *  Processor and Memory placement constraints for sets of tasks.
+ *
```

```
+ *  Copyright (C) 2003 BULL SA.
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *  Copyright (C) 2006 Google, Inc
+ *
+ *  Portions derived from Patrick Mochel's sysfs code.
+ *  sysfs is Copyright (c) 2001-3 Patrick Mochel
+ *
+ *  2003-10-10 Written by Simon Derr.
+ *  2003-10-22 Updates by Stephen Hemminger.
+ *  2004 May-July Rework by Paul Jackson.
+ *  2006 Rework by Paul Menage to use generic containers
+ *
+ *  This file is subject to the terms and conditions of the GNU General Public
+ *  License.  See the file COPYING in the main directory of the Linux
+ *  distribution for more details.
+ */
+
+#include <linux/cpu.h>
+#include <linux/cpumask.h>
+#include <linux/memset.h>
+#include <linux/err.h>
+#include <linux/errno.h>
+#include <linux/file.h>
+#include <linux/fs.h>
+#include <linux/init.h>
+#include <linux/interrupt.h>
+#include <linux/kernel.h>
+#include <linux/kmod.h>
+#include <linux/list.h>
+#include <linux/mempolicy.h>
+#include <linux/mm.h>
+#include <linux/module.h>
+#include <linux/mount.h>
+#include <linux/namei.h>
+#include <linux/pagemap.h>
+#include <linux/proc_fs.h>
+#include <linux/rcupdate.h>
+#include <linux/sched.h>
+#include <linux/seq_file.h>
+#include <linux/security.h>
+#include <linux/slab.h>
+#include <linux/smp_lock.h>
+#include <linux/spinlock.h>
+#include <linux/stat.h>
+#include <linux/string.h>
+#include <linux/time.h>
+#include <linux/backing-dev.h>
+#include <linux/sort.h>
```

```
+
+#include <asm/uaccess.h>
+#include <asm/atomic.h>
+#include <linux/mutex.h>
+
+/*
+ * Tracks how many memsets are currently defined in system.
+ * When there is only one memset (the root memset) we can
+ * short circuit some hooks.
+ */
+int number_of_memsets __read_mostly;
+
+/* Retrieve the memset from a container */
+static struct container_subsys memset_subsys;
+struct memset;
+
+/* See "Frequency meter" comments, below. */
+
+struct fmeter {
+ int cnt;  /* unprocessed events count */
+ int val;  /* most recent output value */
+ time_t time;  /* clock (secs) when val computed */
+ spinlock_t lock; /* guards read or write of above */
+};
+
+struct memset {
+ struct container_subsys_state css;
+
+ unsigned long flags;  /* "unsigned long" so bitops work */
+ nodemask_t mems_allowed; /* Memory Nodes allowed to tasks */
+
+ struct memset *parent;  /* my parent */
+
+ /*
+  * Copy of global memset_mems_generation as of the most
+  * recent time this memset changed its mems_allowed.
+  */
+ int mems_generation;
+
+ struct fmeter fmeter;  /* memory_pressure filter */
+};
+
+/* Update the memset for a container */
+static inline void set_container_ms(struct container *cont, struct memset *ms)
+{
+ cont->subsys[memset_subsys.subsys_id] = &ms->css;
+}
+
```

```
+/* Retrieve the memset for a container */
+static inline struct memset *container_ms(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, &memset_subsys),
+       struct memset, css);
+}
+
+/* Retrieve the memset for a task */
+static inline struct memset *task_ms(struct task_struct *task)
+{
+ return container_ms(task_container(task, &memset_subsys));
+}
+
+
+/* bits in struct memset flags field */
+typedef enum {
+ MS_CPU_EXCLUSIVE,
+ MS_MEM_EXCLUSIVE,
+ MS_MEMORY_MIGRATE,
+ MS_SPREAD_PAGE,
+ MS_SPREAD_SLAB,
+} memset_flagbits_t;
+
+/* convenient tests for these bits */
+static inline int is_cpu_exclusive(const struct memset *ms)
+{
+ return test_bit(MS_CPU_EXCLUSIVE, &ms->flags);
+}
+
+static inline int is_mem_exclusive(const struct memset *ms)
+{
+ return test_bit(MS_MEM_EXCLUSIVE, &ms->flags);
+}
+
+static inline int is_memory_migrate(const struct memset *ms)
+{
+ return test_bit(MS_MEMORY_MIGRATE, &ms->flags);
+}
+
+static inline int is_spread_page(const struct memset *ms)
+{
+ return test_bit(MS_SPREAD_PAGE, &ms->flags);
+}
+
+static inline int is_spread_slab(const struct memset *ms)
+{
+ return test_bit(MS_SPREAD_SLAB, &ms->flags);
+}
```

```
+
+/*
+ * Increment this integer everytime any memset changes its
+ * mems_allowed value.  Users of memsets can track this generation
+ * number, and avoid having to lock and reload mems_allowed unless
+ * the memset they're using changes generation.
+ *
+ * A single, global generation is needed because attach_task() could
+ * reattach a task to a different memset, which must not have its
+ * generation numbers aliased with those of that tasks previous memset.
+ *
+ * Generations are needed for mems_allowed because one task cannot
+ * modify anothers memory placement.  So we must enable every task,
+ * on every visit to __alloc_pages(), to efficiently check whether
+ * its current->memset->mems_allowed has changed, requiring an update
+ * of its current->mems_allowed.
+ *
+ * Since memset_mems_generation is guarded by manage_mutex,
+ * there is no need to mark it atomic.
+ */
+static int memset_mems_generation;
+
+static struct memset top_memset = {
+ .flags = 1 << MS_MEM_EXCLUSIVE,
+ .mems_allowed = NODE_MASK_ALL,
+};
+
+/*
+ * Return in *pmask the portion of a memsets's mems_allowed that
+ * are online.  If none are online, walk up the memset hierarchy
+ * until we find one that does have some online mems.  If we get
+ * all the way to the top and still haven't found any online mems,
+ * return node_online_map.
+ *
+ * One way or another, we guarantee to return some non-empty subset
+ * of node_online_map.
+ *
+ * Call with callback_mutex held.
+ */
+
+static void guarantee_online_mems(const struct memset *ms, nodemask_t *pmask)
+{
+ while (ms && !nodes_intersects(ms->mems_allowed, node_online_map))
+  ms = ms->parent;
+ if (ms)
+  nodes_and(*pmask, ms->mems_allowed, node_online_map);
+ else
+  *pmask = node_online_map;
```

```
+ BUG_ON(!nodes_intersects(*pmask, node_online_map));
+}
+
+/**
+ * memset_update_task_memory_state - update task memory placement
+ *
+ * If the current tasks memsets mems_allowed changed behind our
+ * backs, update current->mems_allowed, mems_generation and task NUMA
+ * mempolicy to the new value.
+ *
+ * Task mempolicy is updated by rebinding it relative to the
+ * current->memset if a task has its memory placement changed.
+ * Do not call this routine if in_interrupt().
+ *
+ * Call without callback_mutex or task_lock() held.  May be
+ * called with or without manage_mutex held.  Thanks in part to
+ * 'the_top_memset_hack', the tasks memset pointer will never
+ * be NULL.  This routine also might acquire callback_mutex and
+ * current->mm->mmap_sem during call.
+ *
+ * Reading current->memset->mems_generation doesn't need task_lock
+ * to guard the current->memset derefence, because it is guarded
+ * from concurrent freeing of current->memset by attach_task(),
+ * using RCU.
+ *
+ * The rcu_dereference() is technically probably not needed,
+ * as I don't actually mind if I see a new memset pointer but
+ * an old value of mems_generation.  However this really only
+ * matters on alpha systems using memsets heavily.  If I dropped
+ * that rcu_dereference(), it would save them a memory barrier.
+ * For all other arch's, rcu_dereference is a no-op anyway, and for
+ * alpha systems not using memsets, another planned optimization,
+ * avoiding the rcu critical section for tasks in the root memset
+ * which is statically allocated, so can't vanish, will make this
+ * irrelevant.  Better to use RCU as intended, than to engage in
+ * some cute trick to save a memory barrier that is impossible to
+ * test, for alpha systems using memsets heavily, which might not
+ * even exist.
+ *
+ * This routine is needed to update the per-task mems_allowed data,
+ * within the tasks context, when it is trying to allocate memory
+ * (in various mm/mempolicy.c routines) and notices that some other
+ * task has been modifying its memset.
+ */
+
+void memset_update_task_memory_state(void)
+{
+ int my_memsets_mem_gen;
```

```
+ struct task_struct *tsk = current;
+ struct memset *ms;
+
+ if (task_ms(tsk) == &top_memset) {
+  /* Don't need rcu for top_memset.  It's never freed. */
+  my_memsets_mem_gen = top_memset.mems_generation;
+ } else {
+  rcu_read_lock();
+  my_memsets_mem_gen = task_ms(current)->mems_generation;
+  rcu_read_unlock();
+ }
+
+ if (my_memsets_mem_gen != tsk->memset_mems_generation) {
+  container_lock();
+  task_lock(tsk);
+  ms = task_ms(tsk); /* Maybe changed when task not locked */
+  guarantee_online_mems(ms, &tsk->mems_allowed);
+  tsk->memset_mems_generation = ms->mems_generation;
+  if (is_spread_page(ms))
+   tsk->flags |= PF_SPREAD_PAGE;
+  else
+   tsk->flags &= ~PF_SPREAD_PAGE;
+  if (is_spread_slab(ms))
+   tsk->flags |= PF_SPREAD_SLAB;
+  else
+   tsk->flags &= ~PF_SPREAD_SLAB;
+  task_unlock(tsk);
+  container_unlock();
+  mpol_rebind_task(tsk, &tsk->mems_allowed);
+ }
+}
+
+/*
+ * is_memset_subset(p, q) - Is memset p a subset of memset q?
+ *
+ * One memset is a subset of another if all its allowed
+ * Memory Nodes are a subset of the other, and its exclusive flags
+ * are only set if the other's are set.  Call holding manage_mutex.
+ */
+
+static int is_memset_subset(const struct memset *p, const struct memset *q)
+{
+ return nodes_subset(p->mems_allowed, q->mems_allowed) &&
+  is_mem_exclusive(p) <= is_mem_exclusive(q);
+}
+
+/*
+ * validate_change() - Used to validate that any proposed memset change
```

```
+ *        follows the structural rules for memsets.
+ *
+ * If we replaced the flag and mask values of the current memset
+ * (cur) with those values in the trial memset (trial), would
+ * our various subset and exclusive rules still be valid?  Presumes
+ * manage_mutex held.
+ *
+ * 'cur' is the address of an actual, in-use memset.  Operations
+ * such as list traversal that depend on the actual address of the
+ * memset in the list must use cur below, not trial.
+ *
+ * 'trial' is the address of bulk structure copy of cur, with
+ * perhaps one or more of the fields mems_allowed,
+ * or flags changed to new, trial values.
+ *
+ * Return 0 if valid, -errno if not.
+ */
+
+static int validate_change(const struct memset *cur, const struct memset *trial)
+{
+ struct container *cont;
+ struct memset *c, *par;
+
+ /* Each of our child memsets must be a subset of us */
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+  if (!is_memset_subset(container_ms(cont), trial))
+   return -EBUSY;
+ }
+
+ /* Remaining checks don't apply to root memset */
+ if ((par = cur->parent) == NULL)
+  return 0;
+
+ /* We must be a subset of our parent memset */
+ if (!is_memset_subset(trial, par))
+  return -EACCES;
+
+ /* If either I or some sibling (!= me) is exclusive, we can't overlap */
+ list_for_each_entry(cont, &par->css.container->children, sibling) {
+  c = container_ms(cont);
+  if ((is_mem_exclusive(trial) || is_mem_exclusive(c)) &&
+      c != cur &&
+      nodes_intersects(trial->mems_allowed, c->mems_allowed))
+   return -EINVAL;
+ }
+
+ return 0;
+}
```

```
+
+/*
+ * memset_migrate_mm
+ *
+ *    Migrate memory region from one set of nodes to another.
+ *
+ *    Temporarilly set tasks mems_allowed to target nodes of migration,
+ *    so that the migration code can allocate pages on these nodes.
+ *
+ *    Call holding manage_mutex, so our current->memset won't change
+ *    during this call, as manage_mutex holds off any attach_task()
+ *    calls.  Therefore we don't need to take task_lock around the
+ *    call to guarantee_online_mems(), as we know no one is changing
+ *    our tasks memset.
+ *
+ *    Hold callback_mutex around the two modifications of our tasks
+ *    mems_allowed to synchronize with memset_mems_allowed().
+ *
+ *    While the mm_struct we are migrating is typically from some
+ *    other task, the task_struct mems_allowed that we are hacking
+ *    is for our current task, which must allocate new pages for that
+ *    migrating memory region.
+ *
+ *    We call memset_update_task_memory_state() before hacking
+ *    our tasks mems_allowed, so that we are assured of being in
+ *    sync with our tasks memset, and in particular, callbacks to
+ *    memset_update_task_memory_state() from nested page allocations
+ *    won't see any mismatch of our memset and task mems_generation
+ *    values, so won't overwrite our hacked tasks mems_allowed
+ *    nodemask.
+ */
+
+static void memset_migrate_mm(struct mm_struct *mm, const nodemask_t *from,
+       const nodemask_t *to)
+{
+ struct task_struct *tsk = current;
+
+ memset_update_task_memory_state();
+
+ container_lock();
+ tsk->mems_allowed = *to;
+ container_unlock();
+
+ do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);
+
+ container_lock();
+ guarantee_online_mems(task_ms(tsk),&tsk->mems_allowed);
+ container_unlock();
```

```
+}
+
+/*
+ * Handle user request to change the 'mems' memory placement
+ * of a memset.  Needs to validate the request, update the
+ * memsets mems_allowed and mems_generation, and for each
+ * task in the memset, rebind any vma mempolicies and if
+ * the memset is marked 'memory_migrate', migrate the tasks
+ * pages to the new memory.
+ *
+ * Call with manage_mutex held.  May take callback_mutex during call.
+ * Will take tasklist_lock, scan tasklist for tasks in memset ms,
+ * lock each such tasks mm->mmap_sem, scan its vma's and rebind
+ * their mempolicies to the memsets new mems_allowed.
+ */
+
+static void *memset_being_rebound;
+
+static int update_nodemask(struct memset *ms, char *buf)
+{
+ struct memset trialms;
+ nodemask_t oldmem;
+ struct task_struct *g, *p;
+ struct mm_struct **mmarray;
+ int i, n, ntasks;
+ int migrate;
+ int fudge;
+ int retval;
+ struct container *cont;
+
+ /* top_memset.mems_allowed tracks node_online_map; it's read-only */
+ if (ms == &top_memset)
+  return -EACCES;
+
+ trialms = *ms;
+ cont = ms->css.container;
+ retval = nodelist_parse(buf, trialms.mems_allowed);
+ if (retval < 0)
+  goto done;
+ nodes_and(trialms.mems_allowed, trialms.mems_allowed, node_online_map);
+ oldmem = ms->mems_allowed;
+ if (nodes_equal(oldmem, trialms.mems_allowed)) {
+  retval = 0;  /* Too easy - nothing to do */
+  goto done;
+ }
+ if (nodes_empty(trialms.mems_allowed)) {
+  retval = -ENOSPC;
+  goto done;
```

```
+ }
+ retval = validate_change(ms, &trialms);
+ if (retval < 0)
+  goto done;
+
+ container_lock();
+ ms->mems_allowed = trialms.mems_allowed;
+ ms->mems_generation = memset_mems_generation++;
+ container_unlock();
+
+ memset_being_rebound = ms;  /* causes mpol_copy() rebind */
+
+ fudge = 10;    /* spare mmarray[] slots */
+ fudge += cpus_weight(cpu_online_map); /* imagine one fork-bomb/cpu */
+ retval = -ENOMEM;
+
+ /*
+  * Allocate mmarray[] to hold mm reference for each task
+  * in memset ms.  Can't kmalloc GFP_KERNEL while holding
+  * tasklist_lock.  We could use GFP_ATOMIC, but with a
+  * few more lines of code, we can retry until we get a big
+  * enough mmarray[] w/o using GFP_ATOMIC.
+  */
+ while (1) {
+  ntasks = atomic_read(&ms->css.container->count);  /* guess */
+  ntasks += fudge;
+  mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
+  if (!mmarray)
+   goto done;
+  write_lock_irq(&tasklist_lock);  /* block fork */
+  if (atomic_read(&ms->css.container->count) <= ntasks)
+   break;    /* got enough */
+  write_unlock_irq(&tasklist_lock); /* try again */
+  kfree(mmarray);
+ }
+
+ n = 0;
+
+ /* Load up mmarray[] with mm reference for each task in memset. */
+ do_each_thread(g, p) {
+  struct mm_struct *mm;
+
+  if (n >= ntasks) {
+   printk(KERN_WARNING
+    "Memset mempolicy rebind incomplete.\n");
+   continue;
+  }
+  if (task_ms(p) != ms)
```

```
+   continue;
+  mm = get_task_mm(p);
+  if (!mm)
+   continue;
+  mmarray[n++] = mm;
+ } while_each_thread(g, p);
+ write_unlock_irq(&tasklist_lock);
+
+ /*
+  * Now that we've dropped the tasklist spinlock, we can
+  * rebind the vma mempolicies of each mm in mmarray[] to their
+  * new memset, and release that mm.  The mpol_rebind_mm()
+  * call takes mmap_sem, which we couldn't take while holding
+  * tasklist_lock.  Forks can happen again now - the mpol_copy()
+  * memset_being_rebound check will catch such forks, and rebind
+  * their vma mempolicies too.  Because we still hold the global
+  * memset manage_mutex, we know that no other rebind effort will
+  * be contending for the global variable memset_being_rebound.
+  * It's ok if we rebind the same mm twice; mpol_rebind_mm()
+  * is idempotent.  Also migrate pages in each mm to new nodes.
+  */
+ migrate = is_memory_migrate(ms);
+ for (i = 0; i < n; i++) {
+  struct mm_struct *mm = mmarray[i];
+
+  mpol_rebind_mm(mm, &ms->mems_allowed);
+  if (migrate)
+   memset_migrate_mm(mm, &oldmem, &ms->mems_allowed);
+  mmput(mm);
+ }
+
+ /* We're done rebinding vma's to this memsets new mems_allowed. */
+ kfree(mmarray);
+ memset_being_rebound = NULL;
+ retval = 0;
+done:
+ return retval;
+}
+
+int current_memset_is_being_rebound(void)
+{
+ return task_ms(current) == memset_being_rebound;
+}
+
+/*
+ * Call with manage_mutex held.
+ */
+
```

```
+static int update_memory_pressure_enabled(struct memset *ms, char *buf)
+{
+ if (simple_strtoul(buf, NULL, 10) != 0)
+  memset_memory_pressure_enabled = 1;
+ else
+  memset_memory_pressure_enabled = 0;
+ return 0;
+}
+
+/*
+ * update_flag - read a 0 or a 1 in a file and update associated flag
+ * bit: the bit to update (MS_MEM_EXCLUSIVE,
+ *    MS_NOTIFY_ON_RELEASE, MS_MEMORY_MIGRATE,
+ *    MS_SPREAD_PAGE, MS_SPREAD_SLAB)
+ * ms: the memset to update
+ * buf: the buffer where we read the 0 or 1
+ *
+ * Call with manage_mutex held.
+ */
+
+static int update_flag(memset_flagbits_t bit, struct memset *ms, char *buf)
+{
+ int turning_on;
+ struct memset trialms;
+ int err;
+
+ turning_on = (simple_strtoul(buf, NULL, 10) != 0);
+
+ trialms = *ms;
+ if (turning_on)
+  set_bit(bit, &trialms.flags);
+ else
+  clear_bit(bit, &trialms.flags);
+
+ err = validate_change(ms, &trialms);
+ if (err < 0)
+  return err;
+ container_lock();
+ if (turning_on)
+  set_bit(bit, &ms->flags);
+ else
+  clear_bit(bit, &ms->flags);
+ container_unlock();
+
+ return 0;
+}
+
+/*
```

```
+ * Frequency meter - How fast is some event occurring?
+ *
+ * These routines manage a digitally filtered, constant time based,
+ * event frequency meter.  There are four routines:
+ *   fmeter_init() - initialize a frequency meter.
+ *   fmeter_markevent() - called each time the event happens.
+ *   fmeter_getrate() - returns the recent rate of such events.
+ *   fmeter_update() - internal routine used to update fmeter.
+ *
+ * A common data structure is passed to each of these routines,
+ * which is used to keep track of the state required to manage the
+ * frequency meter and its digital filter.
+ *
+ * The filter works on the number of events marked per unit time.
+ * The filter is single-pole low-pass recursive (IIR).  The time unit
+ * is 1 second.  Arithmetic is done using 32-bit integers scaled to
+ * simulate 3 decimal digits of precision (multiplied by 1000).
+ *
+ * With an FM_COEF of 933, and a time base of 1 second, the filter
+ * has a half-life of 10 seconds, meaning that if the events quit
+ * happening, then the rate returned from the fmeter_getrate()
+ * will be cut in half each 10 seconds, until it converges to zero.
+ *
+ * It is not worth doing a real infinitely recursive filter.  If more
+ * than FM_MAXTICKS ticks have elapsed since the last filter event,
+ * just compute FM_MAXTICKS ticks worth, by which point the level
+ * will be stable.
+ *
+ * Limit the count of unprocessed events to FM_MAXCNT, so as to avoid
+ * arithmetic overflow in the fmeter_update() routine.
+ *
+ * Given the simple 32 bit integer arithmetic used, this meter works
+ * best for reporting rates between one per millisecond (msec) and
+ * one per 32 (approx) seconds.  At constant rates faster than one
+ * per msec it maxes out at values just under 1,000,000.  At constant
+ * rates between one per msec, and one per second it will stabilize
+ * to a value N*1000, where N is the rate of events per second.
+ * At constant rates between one per second and one per 32 seconds,
+ * it will be choppy, moving up on the seconds that have an event,
+ * and then decaying until the next event.  At rates slower than
+ * about one in 32 seconds, it decays all the way back to zero between
+ * each event.
+ */
+
+#define FM_COEF 933  /* coefficient for half-life of 10 secs */
+#define FM_MAXTICKS ((time_t)99) /* useless computing more ticks than this */
+#define FM_MAXCNT 1000000 /* limit cnt to avoid overflow */
+#define FM_SCALE 1000  /* faux fixed point scale */
```

```
+
+/* Initialize a frequency meter */
+static void fmeter_init(struct fmeter *fmp)
+{
+ fmp->cnt = 0;
+ fmp->val = 0;
+ fmp->time = 0;
+ spin_lock_init(&fmp->lock);
+}
+
+/* Internal meter update - process cnt events and update value */
+static void fmeter_update(struct fmeter *fmp)
+{
+ time_t now = get_seconds();
+ time_t ticks = now - fmp->time;
+
+ if (ticks == 0)
+  return;
+
+ ticks = min(FM_MAXTICKS, ticks);
+ while (ticks-- > 0)
+  fmp->val = (FM_COEF * fmp->val) / FM_SCALE;
+ fmp->time = now;
+
+ fmp->val += ((FM_SCALE - FM_COEF) * fmp->cnt) / FM_SCALE;
+ fmp->cnt = 0;
+}
+
+/* Process any previous ticks, then bump cnt by one (times scale). */
+static void fmeter_markevent(struct fmeter *fmp)
+{
+ spin_lock(&fmp->lock);
+ fmeter_update(fmp);
+ fmp->cnt = min(FM_MAXCNT, fmp->cnt + FM_SCALE);
+ spin_unlock(&fmp->lock);
+}
+
+/* Process any previous ticks, then return current value. */
+static int fmeter_getrate(struct fmeter *fmp)
+{
+ int val;
+
+ spin_lock(&fmp->lock);
+ fmeter_update(fmp);
+ val = fmp->val;
+ spin_unlock(&fmp->lock);
+ return val;
+}
```

```
+
+int memset_can_attach(struct container_subsys *ss,
+        struct container *cont, struct task_struct *tsk)
+{
+ struct memset *ms = container_ms(cont);
+
+ if (nodes_empty(ms->mems_allowed))
+  return -ENOSPC;
+ return 0;
+}
+
+void memset_post_attach(struct container_subsys *ss,
+   struct container *cont,
+   struct container *oldcont,
+   struct task_struct *tsk)
+{
+ nodemask_t from, to;
+ struct mm_struct *mm;
+ struct memset *ms = container_ms(cont);
+ struct memset *oldms = container_ms(oldcont);
+
+ from = oldms->mems_allowed;
+ to = ms->mems_allowed;
+ mm = get_task_mm(tsk);
+ if (mm) {
+  mpol_rebind_mm(mm, &to);
+  if (is_memory_migrate(ms))
+   memset_migrate_mm(mm, &from, &to);
+  mmput(mm);
+ }
+
+}
+
+/* The various types of files and directories in a memset file system */
+
+typedef enum {
+ FILE_MEMORY_MIGRATE,
+ FILE_MEMLIST,
+ FILE_MEM_EXCLUSIVE,
+ FILE_MEMORY_PRESSURE_ENABLED,
+ FILE_MEMORY_PRESSURE,
+ FILE_SPREAD_PAGE,
+ FILE_SPREAD_SLAB,
+} memset_filetype_t;
+
+static ssize_t memset_common_file_write(struct container *cont,
+     struct cftype *cft,
+     struct file *file,
```

```
+     const char __user *userbuf,
+     size_t nbytes, loff_t *unused_ppos)
+{
+ struct memset *ms = container_ms(cont);
+ memset_filetype_t type = cft->private;
+ char *buffer;
+ int retval = 0;
+
+ /* Crude upper limit on largest legitimate list user might write. */
+ if (nbytes > 100 + 6 * MAX_NUMNODES)
+  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 */
+
+ container_manage_lock();
+
+ if (container_is_removed(cont)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch (type) {
+ case FILE_MEMLIST:
+  retval = update_nodemask(ms, buffer);
+  break;
+ case FILE_MEM_EXCLUSIVE:
+  retval = update_flag(MS_MEM_EXCLUSIVE, ms, buffer);
+  break;
+ case FILE_MEMORY_MIGRATE:
+  retval = update_flag(MS_MEMORY_MIGRATE, ms, buffer);
+  break;
+ case FILE_MEMORY_PRESSURE_ENABLED:
+  retval = update_memory_pressure_enabled(ms, buffer);
+  break;
+ case FILE_MEMORY_PRESSURE:
+  retval = -EACCES;
+  break;
+ case FILE_SPREAD_PAGE:
+  retval = update_flag(MS_SPREAD_PAGE, ms, buffer);
+  ms->mems_generation = memset_mems_generation++;
```

```
+  break;
+ case FILE_SPREAD_SLAB:
+  retval = update_flag(MS_SPREAD_SLAB, ms, buffer);
+  ms->mems_generation = memset_mems_generation++;
+  break;
+ default:
+  retval = -EINVAL;
+  goto out2;
+ }
+
+ if (retval == 0)
+  retval = nbytes;
+out2:
+ container_manage_unlock();
+out1:
+ kfree(buffer);
+ 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
+ * chunks, there is no guarantee of atomicity.  Since the display format
+ * used, list of ranges of sequential numbers, is variable length,
+ * and since these maps can change value dynamically, one could read
+ * gibberish by doing partial reads while a list was changing.
+ * A single large read to a buffer that crosses a page boundary is
+ * ok, because the result being copied to user land is not recomputed
+ * across a page fault.
+ */
+
+static int memset_sprintf_memlist(char *page, struct memset *ms)
+{
+ nodemask_t mask;
+
+ container_lock();
+ mask = ms->mems_allowed;
+ container_unlock();
+
+ return nodelist_scnprintf(page, PAGE_SIZE, mask);
+}
+
+static ssize_t memset_common_file_read(struct container *cont,
+           struct cftype *cft,
+           struct file *file,
+           char __user *buf,
+           size_t nbytes, loff_t *ppos)
+{
```

```
+ struct memset *ms = container_ms(cont);
+ memset_filetype_t type = cft->private;
+ char *page;
+ ssize_t retval = 0;
+ char *s;
+
+ if (!(page = (char *)__get_free_page(GFP_KERNEL)))
+  return -ENOMEM;
+
+ s = page;
+
+ switch (type) {
+ case FILE_MEMLIST:
+  s += memset_sprintf_memlist(s, ms);
+  break;
+ case FILE_MEM_EXCLUSIVE:
+  *s++ = is_mem_exclusive(ms) ? '1' : '0';
+  break;
+ case FILE_MEMORY_MIGRATE:
+  *s++ = is_memory_migrate(ms) ? '1' : '0';
+  break;
+ case FILE_MEMORY_PRESSURE_ENABLED:
+  *s++ = memset_memory_pressure_enabled ? '1' : '0';
+  break;
+ case FILE_MEMORY_PRESSURE:
+  s += sprintf(s, "%d", fmeter_getrate(&ms->fmeter));
+  break;
+ case FILE_SPREAD_PAGE:
+  *s++ = is_spread_page(ms) ? '1' : '0';
+  break;
+ case FILE_SPREAD_SLAB:
+  *s++ = is_spread_slab(ms) ? '1' : '0';
+  break;
+ default:
+  retval = -EINVAL;
+  goto out;
+ }
+ *s++ = '\n';
+
+ retval = simple_read_from_buffer(buf, nbytes, ppos, page, s - page);
+out:
+ free_page((unsigned long)page);
+ return retval;
+}
+
+
+/*
+ * for the common functions, 'private' gives the type of file
```

```
+ */
+
+static struct cftype cft_mems = {
+ .name = "mems",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMLIST,
+};
+
+static struct cftype cft_mem_exclusive = {
+ .name = "mem_exclusive",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEM_EXCLUSIVE,
+};
+
+static struct cftype cft_memory_migrate = {
+ .name = "memory_migrate",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMORY_MIGRATE,
+};
+
+static struct cftype cft_memory_pressure_enabled = {
+ .name = "memory_pressure_enabled",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMORY_PRESSURE_ENABLED,
+};
+
+static struct cftype cft_memory_pressure = {
+ .name = "memory_pressure",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMORY_PRESSURE,
+};
+
+static struct cftype cft_spread_page = {
+ .name = "memory_spread_page",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_SPREAD_PAGE,
+};
+
+static struct cftype cft_spread_slab = {
+ .name = "memory_spread_slab",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
```

```
+ .private = FILE_SPREAD_SLAB,
+};
+
+int memset_populate(struct container_subsys *ss, struct container *cont)
+{
+ int err;
+
+ if ((err = container_add_file(cont, &cft_mems)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_mem_exclusive)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_memory_migrate)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_memory_pressure)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_spread_page)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_spread_slab)) < 0)
+  return err;
+ /* memory_pressure_enabled is in root memset only */
+ if (err == 0 && !cont->parent)
+  err = container_add_file(cont, &cft_memory_pressure_enabled);
+ return 0;
+}
+
+/*
+ * memset_create - create a memset
+ * parent: memset that will be parent of the new memset.
+ * name:  name of the new memset. Will be strcpy'ed.
+ * mode:  mode to set on new inode
+ *
+ * Must be called with the mutex on the parent inode held
+ */
+
+int memset_create(struct container_subsys *ss, struct container *cont)
+{
+ struct memset *ms;
+ struct memset *parent;
+
+ if (!cont->parent) {
+  /* This is early initialization for the top container */
+  set_container_ms(cont, &top_memset);
+  top_memset.css.container = cont;
+  top_memset.mems_generation = memset_mems_generation++;
+  return 0;
+ }
+ parent = container_ms(cont->parent);
+ ms = kmalloc(sizeof(*ms), GFP_KERNEL);
```

```
+ if (!ms)
+  return -ENOMEM;
+
+ memset_update_task_memory_state();
+ ms->flags = 0;
+ if (is_spread_page(parent))
+  set_bit(MS_SPREAD_PAGE, &ms->flags);
+ if (is_spread_slab(parent))
+  set_bit(MS_SPREAD_SLAB, &ms->flags);
+ ms->mems_allowed = NODE_MASK_NONE;
+ ms->mems_generation = memset_mems_generation++;
+ fmeter_init(&ms->fmeter);
+
+ ms->parent = parent;
+ set_container_ms(cont, ms);
+ ms->css.container = cont;
+ number_of_memsets++;
+ return 0;
+}
+
+void memset_reparent(struct container_subsys *ss, struct container *cont,
+       void *state)
+{
+ struct memset *ms = state;
+ ms->css.container = cont;
+}
+
+void memset_destroy(struct container_subsys *ss, struct container *cont)
+{
+ struct memset *ms = container_ms(cont);
+ memset_update_task_memory_state();
+ number_of_memsets--;
+ kfree(ms);
+}
+
+static struct container_subsys memset_subsys = {
+ .name = "memset",
+ .create = memset_create,
+ .destroy  = memset_destroy,
+ .can_attach = memset_can_attach,
+ .post_attach = memset_post_attach,
+ .populate = memset_populate,
+ .subsys_id = -1,
+};
+
+/*
+ * memset_init_early - just enough so that the calls to
+ * memset_update_task_memory_state() in early init code
```

```
+ * are harmless.
+ */
+
+int __init memset_init_early(void)
+{
+ if (container_register_subsys(&memset_subsys) < 0)
+  panic("Couldn't register memset subsystem");
+ top_memset.mems_generation = memset_mems_generation++;
+ return 0;
+}
+
+
+/**
+ * memset_init - initialize memsets at system boot
+ *
+ * Description: Initialize top_memset and the memset internal file system,
+ **/
+
+int __init memset_init(void)
+{
+ top_memset.mems_allowed = NODE_MASK_ALL;
+
+ fmeter_init(&top_memset.fmeter);
+ top_memset.mems_generation = memset_mems_generation++;
+
+ number_of_memsets = 1;
+ return 0;
+}
+
+#if defined(CONFIG_MEMORY_HOTPLUG)
+/*
+ * If memset_track_online_nodes(), below, unplugs any memory nodes, we
+ * need to walk over the memset hierarchy, removing that node from all
+ * memsets.  If this removes the last node from a memset, then
+ * guarantee_online_mems() will use that emptied memsets parent online
+ * nodes.  Memsets that were already empty of nodes are left empty.
+ *
+ * This routine will check all memsets in a subtree even if the top
+ * memset of the subtree has no offline nodes.
+ *
+ * Call with both manage_mutex and callback_mutex held.
+ *
+ * Recursive, on depth of memset subtree.
+ */
+
+static void guarantee_online_mems_in_subtree(const struct memset *cur)
+{
+ struct container *cont;
```

```
+ struct memset *c;
+
+ /* Each of our child memsets mems must be online */
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+  c = container_ms(cont);
+  guarantee_online_mems_in_subtree(c);
+  if (!nodes_empty(c->mems_allowed))
+   guarantee_online_mems(c, &c->mems_allowed);
+ }
+}
+
+/*
+ * Keep top_memset.mems_allowed tracking node_online_map.
+ * Call this routine anytime after you change node_online_map.
+ */
+
+void memset_track_online_nodes(void)
+{
+ container_manage_lock();
+ container_lock();
+
+ guarantee_online_mems_in_subtree(&top_memset);
+ top_memset.mems_allowed = node_online_map;
+
+ container_unlock();
+ container_manage_unlock();
+}
+#endif
+
+/**
+ * memset_init_smp - initialize mems_allowed
+ *
+ * Description: Finish top memset after cpu, node maps are initialized
+ **/
+
+void __init memset_init_smp(void)
+{
+ top_memset.mems_allowed = node_online_map;
+}
+
+void memset_init_current_mems_allowed(void)
+{
+ current->mems_allowed = NODE_MASK_ALL;
+}
+
+/**
+ * memset_mems_allowed - return mems_allowed mask from a tasks memset.
+ * @tsk: pointer to task_struct from which to obtain memset->mems_allowed.
```

```
+ *
+ * Description: Returns the nodemask_t mems_allowed of the memset
+ * attached to the specified @tsk.  Guaranteed to return some non-empty
+ * subset of node_online_map, even if this means going outside the
+ * tasks memset.
+ **/
+
+nodemask_t memset_mems_allowed(struct task_struct *tsk)
+{
+ nodemask_t mask;
+
+ container_lock();
+ task_lock(tsk);
+ guarantee_online_mems(task_ms(tsk), &mask);
+ task_unlock(tsk);
+ container_unlock();
+
+ return mask;
+}
+
+/**
+ * memset_zonelist_valid_mems_allowed - check zonelist vs. curremt mems_allowed
+ * @zl: the zonelist to be checked
+ *
+ * Are any of the nodes on zonelist zl allowed in current->mems_allowed?
+ */
+int memset_zonelist_valid_mems_allowed(struct zonelist *zl)
+{
+ int i;
+
+ for (i = 0; zl->zones[i]; i++) {
+  int nid = zone_to_nid(zl->zones[i]);
+
+  if (node_isset(nid, current->mems_allowed))
+   return 1;
+ }
+ return 0;
+}
+
+/*
+ * nearest_exclusive_ancestor() - Returns the nearest mem_exclusive
+ * ancestor to the specified memset.  Call holding callback_mutex.
+ * If no ancestor is mem_exclusive (an unusual configuration), then
+ * returns the root memset.
+ */
+static const struct memset *nearest_exclusive_ancestor(const struct memset *ms)
+{
+ while (!is_mem_exclusive(ms) && ms->parent)
```

```
+  ms = ms->parent;
+ return ms;
+}
+
+/**
+ * memset_zone_allowed - Can we allocate memory on zone z's memory node?
+ * @z: is this zone on an allowed node?
+ * @gfp_mask: memory allocation flags (we use __GFP_HARDWALL)
+ *
+ * If we're in interrupt, yes, we can always allocate.  If zone
+ * z's node is in our tasks mems_allowed, yes.  If it's not a
+ * __GFP_HARDWALL request and this zone's nodes is in the nearest
+ * mem_exclusive memset ancestor to this tasks memset, yes.
+ * Otherwise, no.
+ *
+ * GFP_USER allocations are marked with the __GFP_HARDWALL bit,
+ * and do not allow allocations outside the current tasks memset.
+ * GFP_KERNEL allocations are not so marked, so can escape to the
+ * nearest mem_exclusive ancestor memset.
+ *
+ * Scanning up parent memsets requires callback_mutex.  The __alloc_pages()
+ * routine only calls here with __GFP_HARDWALL bit _not_ set if
+ * it's a GFP_KERNEL allocation, and all nodes in the current tasks
+ * mems_allowed came up empty on the first pass over the zonelist.
+ * So only GFP_KERNEL allocations, if all nodes in the memset are
+ * short of memory, might require taking the callback_mutex mutex.
+ *
+ * The first call here from mm/page_alloc:get_page_from_freelist()
+ * has __GFP_HARDWALL set in gfp_mask, enforcing hardwall memsets, so
+ * no allocation on a node outside the memset is allowed (unless in
+ * interrupt, of course).
+ *
+ * The second pass through get_page_from_freelist() doesn't even call
+ * here for GFP_ATOMIC calls.  For those calls, the __alloc_pages()
+ * variable 'wait' is not set, and the bit ALLOC_MEMSET is not set
+ * in alloc_flags.  That logic and the checks below have the combined
+ * affect that:
+ * in_interrupt - any node ok (current task context irrelevant)
+ * GFP_ATOMIC   - any node ok
+ * GFP_KERNEL   - any node in enclosing mem_exclusive memset ok
+ * GFP_USER     - only nodes in current tasks mems allowed ok.
+ *
+ * Rule:
+ *    Don't call memset_zone_allowed() if you can't sleep, unless you
+ *    pass in the __GFP_HARDWALL flag set in gfp_flag, which disables
+ *    the code that might scan up ancestor memsets and sleep.
+ **/
+
```

```
+int __memset_zone_allowed(struct zone *z, gfp_t gfp_mask)
+{
+ int node;   /* node that zone z is on */
+ const struct memset *ms; /* current memset ancestors */
+ int allowed;   /* is allocation in zone z allowed? */
+
+ if (in_interrupt() || (gfp_mask & __GFP_THISNODE))
+  return 1;
+ node = zone_to_nid(z);
+ might_sleep_if(!(gfp_mask & __GFP_HARDWALL));
+ if (node_isset(node, current->mems_allowed))
+  return 1;
+ if (gfp_mask & __GFP_HARDWALL) /* If hardwall request, stop here */
+  return 0;
+
+ if (current->flags & PF_EXITING) /* Let dying task have memory */
+  return 1;
+
+ /* Not hardwall and node outside mems_allowed: scan up memsets */
+ container_lock();
+
+ task_lock(current);
+ ms = nearest_exclusive_ancestor(task_ms(current));
+ task_unlock(current);
+
+ allowed = node_isset(node, ms->mems_allowed);
+ container_unlock();
+ return allowed;
+}
+
+/**
+ * memset_mem_spread_node() - On which node to begin search for a page
+ *
+ * If a task is marked PF_SPREAD_PAGE or PF_SPREAD_SLAB (as for
+ * tasks in a memset with is_spread_page or is_spread_slab set),
+ * and if the memory allocation used memset_mem_spread_node()
+ * to determine on which node to start looking, as it will for
+ * certain page cache or slab cache pages such as used for file
+ * system buffers and inode caches, then instead of starting on the
+ * local node to look for a free page, rather spread the starting
+ * node around the tasks mems_allowed nodes.
+ *
+ * We don't have to worry about the returned node being offline
+ * because "it can't happen", and even if it did, it would be ok.
+ *
+ * The routines calling guarantee_online_mems() are careful to
+ * only set nodes in task->mems_allowed that are online.  So it
+ * should not be possible for the following code to return an
```

```
+ * offline node.  But if it did, that would be ok, as this routine
+ * is not returning the node where the allocation must be, only
+ * the node where the search should start.  The zonelist passed to
+ * __alloc_pages() will include all nodes.  If the slab allocator
+ * is passed an offline node, it will fall back to the local node.
+ * See kmem_cache_alloc_node().
+ */
+
+int memset_mem_spread_node(void)
+{
+ int node;
+
+ node = next_node(current->memset_mem_spread_rotor, current->mems_allowed);
+ if (node == MAX_NUMNODES)
+  node = first_node(current->mems_allowed);
+ current->memset_mem_spread_rotor = node;
+ return node;
+}
+EXPORT_SYMBOL_GPL(memset_mem_spread_node);
+
+/**
+ * memset_excl_nodes_overlap - Do we overlap @p's mem_exclusive ancestors?
+ * @p: pointer to task_struct of some other task.
+ *
+ * Description: Return true if the nearest mem_exclusive ancestor
+ * memsets of tasks @p and current overlap.  Used by oom killer to
+ * determine if task @p's memory usage might impact the memory
+ * available to the current task.
+ *
+ * Call while holding callback_mutex.
+ **/
+
+int memset_excl_nodes_overlap(const struct task_struct *p)
+{
+ const struct memset *ms1, *ms2; /* my and p's memset ancestors */
+ int overlap = 1;  /* do memsets overlap? */
+
+ task_lock(current);
+ if (current->flags & PF_EXITING) {
+  task_unlock(current);
+  goto done;
+ }
+ ms1 = nearest_exclusive_ancestor(task_ms(current));
+ task_unlock(current);
+
+ task_lock((struct task_struct *)p);
+ if (p->flags & PF_EXITING) {
+  task_unlock((struct task_struct *)p);
```

```
+  goto done;
+ }
+ ms2 = nearest_exclusive_ancestor(task_ms((struct task_struct *)p));
+ task_unlock((struct task_struct *)p);
+
+ overlap = nodes_intersects(ms1->mems_allowed, ms2->mems_allowed);
+done:
+ return overlap;
+}
+
+/*
+ * Collection of memory_pressure is suppressed unless
+ * this flag is enabled by writing "1" to the special
+ * memset file 'memory_pressure_enabled' in the root memset.
+ */
+
+int memset_memory_pressure_enabled __read_mostly;
+
+/**
+ * memset_memory_pressure_bump - keep stats of per-memset reclaims.
+ *
+ * Keep a running average of the rate of synchronous (direct)
+ * page reclaim efforts initiated by tasks in each memset.
+ *
+ * This represents the rate at which some task in the memset
+ * ran low on memory on all nodes it was allowed to use, and
+ * had to enter the kernels page reclaim code in an effort to
+ * create more free memory by tossing clean pages or swapping
+ * or writing dirty pages.
+ *
+ * Display to user space in the per-memset read-only file
+ * "memory_pressure".  Value displayed is an integer
+ * representing the recent rate of entry into the synchronous
+ * (direct) page reclaim by any task attached to the memset.
+ **/
+
+void __memset_memory_pressure_bump(void)
+{
+ task_lock(current);
+ fmeter_markevent(&task_ms(current)->fmeter);
+ task_unlock(current);
+}
+
+/* Display task mems_allowed in /proc/<pid>/status file. */
+char *memset_task_status_allowed(struct task_struct *task, char *buffer)
+{
+ buffer += sprintf(buffer, "Mems_allowed:\t");
+ buffer += nodemask_scnprintf(buffer, PAGE_SIZE, task->mems_allowed);
```

```
+ buffer += sprintf(buffer, "\n");
+ return buffer;
+}
Index: container-2.6.19-rc6/include/linux/mempolicy.h
===================================================================
--- container-2.6.19-rc6.orig/include/linux/mempolicy.h
+++ container-2.6.19-rc6/include/linux/mempolicy.h
@@ -68,7 +68,7 @@ struct mempolicy {
   nodemask_t  nodes;  /* interleave */
   /* undefined for default */
  } v;
- nodemask_t cpuset_mems_allowed; /* mempolicy relative to these nodes */
+ nodemask_t memset_mems_allowed; /* mempolicy relative to these nodes */
 };

 /*
Index: container-2.6.19-rc6/include/linux/sched.h
===================================================================
--- container-2.6.19-rc6.orig/include/linux/sched.h
+++ container-2.6.19-rc6/include/linux/sched.h
@@ -999,10 +999,10 @@ struct task_struct {
    struct mempolicy *mempolicy;
  short il_next;
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_MEMSETS
  nodemask_t mems_allowed;
- int cpuset_mems_generation;
- int cpuset_mem_spread_rotor;
+ int memset_mems_generation;
+ int memset_mem_spread_rotor;
 #endif
 #ifdef CONFIG_CONTAINERS
  struct container *container[CONFIG_MAX_CONTAINER_HIERARCHIES];
@@ -1112,8 +1112,8 @@ static inline void put_task_struct(struc
 #define PF_BORROWED_MM 0x00200000 /* I am a kthread doing use_mm */
 #define PF_RANDOMIZE 0x00400000 /* randomize virtual address space */
 #define PF_SWAPWRITE 0x00800000 /* Allowed to write to swap */
-#define PF_SPREAD_PAGE 0x01000000 /* Spread page cache over cpuset */
-#define PF_SPREAD_SLAB 0x02000000 /* Spread some slab caches over cpuset */
+#define PF_SPREAD_PAGE 0x01000000 /* Spread page cache over memset */
+#define PF_SPREAD_SLAB 0x02000000 /* Spread some slab caches over memset */
 #define PF_MEMPOLICY 0x10000000 /* Non-default NUMA mempolicy */
 #define PF_MUTEX_TESTER 0x20000000 /* Thread belongs to the rt mutex tester */

Index: container-2.6.19-rc6/kernel/Makefile
===================================================================
--- container-2.6.19-rc6.orig/kernel/Makefile
```

```
+++ container-2.6.19-rc6/kernel/Makefile
@@ -38,6 +38,7 @@ obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
+obj-$(CONFIG_MEMSETS) += memset.o
 obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
Index: container-2.6.19-rc6/mm/mempolicy.c
===================================================================
--- container-2.6.19-rc6.orig/mm/mempolicy.c
+++ container-2.6.19-rc6/mm/mempolicy.c
@@ -74,7 +74,7 @@
 #include <linux/sched.h>
 #include <linux/mm.h>
 #include <linux/nodemask.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/gfp.h>
 #include <linux/slab.h>
 #include <linux/string.h>
@@ -198,7 +198,7 @@ static struct mempolicy *mpol_new(int mo
   break;
  }
  policy->policy = mode;
- policy->cpuset_mems_allowed = cpuset_mems_allowed(current);
+ policy->memset_mems_allowed = memset_mems_allowed(current);
  return policy;
 }

@@ -423,8 +423,8 @@ static int contextualize_policy(int mode
  if (!nodes)
   return 0;

- cpuset_update_task_memory_state();
- if (!cpuset_nodes_subset_current_mems_allowed(*nodes))
+ memset_update_task_memory_state();
+ if (!memset_nodes_subset_current_mems_allowed(*nodes))
   return -EINVAL;
  return mpol_check_policy(mode, nodes);
 }
@@ -529,7 +529,7 @@ long do_get_mempolicy(int *policy, nodem
  struct vm_area_struct *vma = NULL;
  struct mempolicy *pol = current->mempolicy;

- cpuset_update_task_memory_state();
+ memset_update_task_memory_state();
```

```
  if (flags & ~(unsigned long)(MPOL_F_NODE|MPOL_F_ADDR))
   return -EINVAL;
  if (flags & MPOL_F_ADDR) {
@@ -945,7 +945,7 @@ asmlinkage long sys_migrate_pages(pid_t
   goto out;
  }

- task_nodes = cpuset_mems_allowed(task);
+ task_nodes = memset_mems_allowed(task);
  /* Is the user allowed to access the target nodes? */
  if (!nodes_subset(new, task_nodes) && !capable(CAP_SYS_NICE)) {
   err = -EPERM;
@@ -1102,7 +1102,7 @@ static struct zonelist *zonelist_policy(
   /* Lower zones don't get a policy applied */
   /* Careful: current->mems_allowed might have moved */
   if (gfp_zone(gfp) >= policy_zone)
-   if (cpuset_zonelist_valid_mems_allowed(policy->v.zonelist))
+   if (memset_zonelist_valid_mems_allowed(policy->v.zonelist))
     return policy->v.zonelist;
   /*FALL THROUGH*/
  case MPOL_INTERLEAVE: /* should not happen */
@@ -1256,7 +1256,7 @@ alloc_page_vma(gfp_t gfp, struct vm_area
 {
  struct mempolicy *pol = get_vma_policy(current, vma, addr);

- cpuset_update_task_memory_state();
+ memset_update_task_memory_state();

  if (unlikely(pol->policy == MPOL_INTERLEAVE)) {
   unsigned nid;
@@ -1282,8 +1282,8 @@ alloc_page_vma(gfp_t gfp, struct vm_area
  * interrupt context and apply the current process NUMA policy.
  * Returns NULL when no page can be allocated.
  *
- * Don't call cpuset_update_task_memory_state() unless
- * 1) it's ok to take cpuset_sem (can WAIT), and
+ * Don't call memset_update_task_memory_state() unless
+ * 1) it's ok to take memset_sem (can WAIT), and
  * 2) allocating for current task (not interrupt).
  */
 struct page *alloc_pages_current(gfp_t gfp, unsigned order)
@@ -1291,7 +1291,7 @@ struct page *alloc_pages_current(gfp_t g
  struct mempolicy *pol = current->mempolicy;

  if ((gfp & __GFP_WAIT) && !in_interrupt())
-  cpuset_update_task_memory_state();
+  memset_update_task_memory_state();
  if (!pol || in_interrupt() || (gfp & __GFP_THISNODE))
```

```
   pol = &default_policy;
  if (pol->policy == MPOL_INTERLEAVE)
@@ -1301,11 +1301,11 @@ struct page *alloc_pages_current(gfp_t g
 EXPORT_SYMBOL(alloc_pages_current);

 /*
- * If mpol_copy() sees current->cpuset == cpuset_being_rebound, then it
+ * If mpol_copy() sees current->memset == memset_being_rebound, then it
   * rebinds the mempolicy its copying by calling mpol_rebind_policy()
- * with the mems_allowed returned by cpuset_mems_allowed().  This
- * keeps mempolicies cpuset relative after its cpuset moves.  See
- * further kernel/cpuset.c update_nodemask().
+ * with the mems_allowed returned by memset_mems_allowed().  This
+ * keeps mempolicies memset relative after its memset moves.  See
+ * further kernel/memset.c update_nodemask().
   */

 /* Slow path of a mempolicy copy */
@@ -1315,8 +1315,8 @@ struct mempolicy *__mpol_copy(struct mem

   if (!new)
    return ERR_PTR(-ENOMEM);
- if (current_cpuset_is_being_rebound()) {
-  nodemask_t mems = cpuset_mems_allowed(current);
+ if (current_memset_is_being_rebound()) {
+  nodemask_t mems = memset_mems_allowed(current);
    mpol_rebind_policy(old, &mems);
   }
   *new = *old;
@@ -1623,7 +1623,7 @@ void mpol_rebind_policy(struct mempolicy

   if (!pol)
    return;
- mpolmask = &pol->cpuset_mems_allowed;
+ mpolmask = &pol->memset_mems_allowed;
   if (nodes_equal(*mpolmask, *newmask))
    return;

Index: container-2.6.19-rc6/mm/migrate.c
===================================================================
--- container-2.6.19-rc6.orig/mm/migrate.c
+++ container-2.6.19-rc6/mm/migrate.c
@@ -23,7 +23,7 @@
 #include <linux/rmap.h>
 #include <linux/topology.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
```

```
 #include <linux/writeback.h>
 #include <linux/mempolicy.h>
 #include <linux/vmalloc.h>
@@ -911,7 +911,7 @@ asmlinkage long sys_move_pages(pid_t pid
    goto out2;


- task_nodes = cpuset_mems_allowed(task);
+ task_nodes = memset_mems_allowed(task);

  /* Limit nr_pages so that the multiplication may not overflow */
  if (nr_pages >= ULONG_MAX / sizeof(struct page_to_node) - 1) {
Index: container-2.6.19-rc6/mm/page_alloc.c
===================================================================
--- container-2.6.19-rc6.orig/mm/page_alloc.c
+++ container-2.6.19-rc6/mm/page_alloc.c
@@ -31,7 +31,7 @@
 #include <linux/topology.h>
 #include <linux/sysctl.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/memory_hotplug.h>
 #include <linux/nodemask.h>
 #include <linux/vmalloc.h>
@@ -891,7 +891,7 @@ failed:
 #define ALLOC_WMARK_HIGH 0x08 /* use pages_high watermark */
 #define ALLOC_HARDER  0x10 /* try to alloc harder */
 #define ALLOC_HIGH  0x20 /* __GFP_HIGH set */
-#define ALLOC_CPUSET  0x40 /* check for correct cpuset */
+#define ALLOC_MEMSET  0x40 /* check for correct memset */

 /*
  * Return 1 if free pages are above 'mark'. This takes into account the order
@@ -940,15 +940,15 @@ get_page_from_freelist(gfp_t gfp_mask, u

  /*
   * Go through the zonelist once, looking for a zone with enough free.
-  * See also cpuset_zone_allowed() comment in kernel/cpuset.c.
+  * See also memset_zone_allowed() comment in kernel/memset.c.
   */
  do {
   zone = *z;
   if (unlikely(NUMA_BUILD && (gfp_mask & __GFP_THISNODE) &&
    zone->zone_pgdat != zonelist->zones[0]->zone_pgdat))
     break;
-  if ((alloc_flags & ALLOC_CPUSET) &&
-    !cpuset_zone_allowed(zone, gfp_mask))
```

```
+  if ((alloc_flags & ALLOC_MEMSET) &&
+    !memset_zone_allowed(zone, gfp_mask))
     continue;

   if (!(alloc_flags & ALLOC_NO_WATERMARKS)) {
@@ -1001,7 +1001,7 @@ restart:
  }

  page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, order,
-    zonelist, ALLOC_WMARK_LOW|ALLOC_CPUSET);
+    zonelist, ALLOC_WMARK_LOW|ALLOC_MEMSET);
  if (page)
   goto got_pg;

@@ -1025,15 +1025,15 @@ restart:
  if (gfp_mask & __GFP_HIGH)
   alloc_flags |= ALLOC_HIGH;
  if (wait)
-  alloc_flags |= ALLOC_CPUSET;
+  alloc_flags |= ALLOC_MEMSET;

  /*
   * Go through the zonelist again. Let __GFP_HIGH and allocations
   * coming from realtime tasks go deeper into reserves.
   *
   * This is the last chance, in general, before the goto nopage.
-  * Ignore cpuset if GFP_ATOMIC (!wait) rather than fail alloc.
-  * See also cpuset_zone_allowed() comment in kernel/cpuset.c.
+  * Ignore memset if GFP_ATOMIC (!wait) rather than fail alloc.
+  * See also memset_zone_allowed() comment in kernel/memset.c.
   */
  page = get_page_from_freelist(gfp_mask, order, zonelist, alloc_flags);
  if (page)
@@ -1066,7 +1066,7 @@ rebalance:
  cond_resched();

  /* We now go into synchronous reclaim */
- cpuset_memory_pressure_bump();
+ memset_memory_pressure_bump();
  p->flags |= PF_MEMALLOC;
  reclaim_state.reclaimed_slab = 0;
  p->reclaim_state = &reclaim_state;
@@ -1091,7 +1091,7 @@ rebalance:
    * under heavy pressure.
    */
   page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, order,
-    zonelist, ALLOC_WMARK_HIGH|ALLOC_CPUSET);
+    zonelist, ALLOC_WMARK_HIGH|ALLOC_MEMSET);
```

```
   if (page)
    goto got_pg;

@@ -1594,12 +1594,12 @@ void __meminit build_all_zonelists(void)
 {
  if (system_state == SYSTEM_BOOTING) {
   __build_all_zonelists(NULL);
-  cpuset_init_current_mems_allowed();
+  memset_init_current_mems_allowed();
  } else {
   /* we have to stop all cpus to guaranntee there is no user
      of zonelist */
   stop_machine_run(__build_all_zonelists, NULL, NR_CPUS);
-  /* cpuset refresh routine should be here */
+  /* memset refresh routine should be here */
  }
  vm_total_pages = nr_free_pagecache_pages();
  printk("Built %i zonelists.  Total pages: %ld\n",
Index: container-2.6.19-rc6/mm/filemap.c
===================================================================
--- container-2.6.19-rc6.orig/mm/filemap.c
+++ container-2.6.19-rc6/mm/filemap.c
@@ -29,7 +29,7 @@
 #include <linux/blkdev.h>
 #include <linux/security.h>
 #include <linux/syscalls.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include "filemap.h"
 #include "internal.h"

@@ -469,8 +469,8 @@ int add_to_page_cache_lru(struct page *p
 #ifdef CONFIG_NUMA
 struct page *__page_cache_alloc(gfp_t gfp)
 {
- if (cpuset_do_page_mem_spread()) {
-  int n = cpuset_mem_spread_node();
+ if (memset_do_page_mem_spread()) {
+  int n = memset_mem_spread_node();
   return alloc_pages_node(n, gfp, 0);
  }
  return alloc_pages(gfp, 0);
Index: container-2.6.19-rc6/mm/hugetlb.c
===================================================================
--- container-2.6.19-rc6.orig/mm/hugetlb.c
+++ container-2.6.19-rc6/mm/hugetlb.c
@@ -12,7 +12,7 @@
 #include <linux/nodemask.h>
```

```
 #include <linux/pagemap.h>
 #include <linux/mempolicy.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/mutex.h>

 #include <asm/page.h>
@@ -73,7 +73,7 @@ static struct page *dequeue_huge_page(st

  for (z = zonelist->zones; *z; z++) {
   nid = zone_to_nid(*z);
-  if (cpuset_zone_allowed(*z, GFP_HIGHUSER) &&
+  if (memset_zone_allowed(*z, GFP_HIGHUSER) &&
       !list_empty(&hugepage_freelists[nid]))
    break;
  }
Index: container-2.6.19-rc6/mm/memory_hotplug.c
===================================================================
--- container-2.6.19-rc6.orig/mm/memory_hotplug.c
+++ container-2.6.19-rc6/mm/memory_hotplug.c
@@ -22,7 +22,7 @@
 #include <linux/highmem.h>
 #include <linux/vmalloc.h>
 #include <linux/ioport.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>

 #include <asm/tlbflush.h>

@@ -284,7 +284,7 @@ int add_memory(int nid, u64 start, u64 s
  /* we online node here. we can't roll back from here. */
  node_set_online(nid);

- cpuset_track_online_nodes();
+ memset_track_online_nodes();

  if (new_pgdat) {
   ret = register_one_node(nid);
Index: container-2.6.19-rc6/mm/oom_kill.c
===================================================================
--- container-2.6.19-rc6.orig/mm/oom_kill.c
+++ container-2.6.19-rc6/mm/oom_kill.c
@@ -21,7 +21,7 @@
 #include <linux/swap.h>
 #include <linux/timex.h>
 #include <linux/jiffies.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
```

```
 #include <linux/module.h>
 #include <linux/notifier.h>

@@ -140,7 +140,7 @@ unsigned long badness(struct task_struct
   * because p may have allocated or otherwise mapped memory on
   * this node before. However it will be less likely.
   */
- if (!cpuset_excl_nodes_overlap(p))
+ if (!memset_excl_nodes_overlap(p))
   points /= 8;

  /*
@@ -165,7 +165,7 @@ unsigned long badness(struct task_struct
  */
 #define CONSTRAINT_NONE 1
 #define CONSTRAINT_MEMORY_POLICY 2
-#define CONSTRAINT_CPUSET 3
+#define CONSTRAINT_MEMSET 3

 /*
  * Determine the type of allocation constraint.
@@ -177,10 +177,10 @@ static inline int constrained_alloc(stru
  nodemask_t nodes = node_online_map;

  for (z = zonelist->zones; *z; z++)
-  if (cpuset_zone_allowed(*z, gfp_mask))
+  if (memset_zone_allowed(*z, gfp_mask))
    node_clear(zone_to_nid(*z), nodes);
   else
-   return CONSTRAINT_CPUSET;
+   return CONSTRAINT_MEMSET;

  if (!nodes_empty(nodes))
   return CONSTRAINT_MEMORY_POLICY;
@@ -408,9 +408,9 @@ void out_of_memory(struct zonelist *zone
     "No available memory (MPOL_BIND)");
   break;

- case CONSTRAINT_CPUSET:
+ case CONSTRAINT_MEMSET:
   oom_kill_process(current, points,
-    "No available memory in cpuset");
+    "No available memory in memset");
   break;

  case CONSTRAINT_NONE:
Index: container-2.6.19-rc6/mm/slab.c
===================================================================
```

```
--- container-2.6.19-rc6.orig/mm/slab.c
+++ container-2.6.19-rc6/mm/slab.c
@@ -94,7 +94,7 @@
 #include <linux/interrupt.h>
 #include <linux/init.h>
 #include <linux/compiler.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/seq_file.h>
 #include <linux/notifier.h>
 #include <linux/kallsyms.h>
@@ -3120,7 +3120,7 @@ static __always_inline void *__cache_all
 /*
  * Try allocating on another node if PF_SPREAD_SLAB|PF_MEMPOLICY.
  *
- * If we are in_interrupt, then process context, including cpusets and
+ * If we are in_interrupt, then process context, including memsets and
  * mempolicy, may not apply and should not be used for allocation policy.
  */
 static void *alternate_node_alloc(struct kmem_cache *cachep, gfp_t flags)
@@ -3130,8 +3130,8 @@ static void *alternate_node_alloc(struct
  if (in_interrupt() || (flags & __GFP_THISNODE))
   return NULL;
  nid_alloc = nid_here = numa_node_id();
- if (cpuset_do_slab_mem_spread() && (cachep->flags & SLAB_MEM_SPREAD))
-  nid_alloc = cpuset_mem_spread_node();
+ if (memset_do_slab_mem_spread() && (cachep->flags & SLAB_MEM_SPREAD))
+  nid_alloc = memset_mem_spread_node();
  else if (current->mempolicy)
   nid_alloc = slab_node(current->mempolicy);
  if (nid_alloc != nid_here)
@@ -3143,7 +3143,7 @@ static void *alternate_node_alloc(struct
  * Fallback function if there was no memory available and no objects on a
  * certain node and we are allowed to fall back. We mimick the behavior of
  * the page allocator. We fall back according to a zonelist determined by
- * the policy layer while obeying cpuset constraints.
+ * the policy layer while obeying memset constraints.
  */
 void *fallback_alloc(struct kmem_cache *cache, gfp_t flags)
 {
@@ -3156,7 +3156,7 @@ void *fallback_alloc(struct kmem_cache *
   int nid = zone_to_nid(*z);

   if (zone_idx(*z) <= ZONE_NORMAL &&
-      cpuset_zone_allowed(*z, flags) &&
+      memset_zone_allowed(*z, flags) &&
       cache->nodelists[nid])
     obj = __cache_alloc_node(cache,
```

```
      flags | __GFP_THISNODE, nid);
Index: container-2.6.19-rc6/mm/vmscan.c
===================================================================
--- container-2.6.19-rc6.orig/mm/vmscan.c
+++ container-2.6.19-rc6/mm/vmscan.c
@@ -31,7 +31,7 @@
 #include <linux/rmap.h>
 #include <linux/topology.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/notifier.h>
 #include <linux/rwsem.h>
 #include <linux/delay.h>
@@ -983,7 +983,7 @@ static unsigned long shrink_zones(int pr
   if (!populated_zone(zone))
    continue;

-  if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+  if (!memset_zone_allowed(zone, __GFP_HARDWALL))
    continue;

   note_zone_scanning_priority(zone, priority);
@@ -1033,7 +1033,7 @@ unsigned long try_to_free_pages(struct z
  for (i = 0; zones[i] != NULL; i++) {
   struct zone *zone = zones[i];

-  if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+  if (!memset_zone_allowed(zone, __GFP_HARDWALL))
    continue;

   lru_pages += zone->nr_active + zone->nr_inactive;
@@ -1088,7 +1088,7 @@ out:
  for (i = 0; zones[i] != 0; i++) {
   struct zone *zone = zones[i];

-  if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+  if (!memset_zone_allowed(zone, __GFP_HARDWALL))
    continue;

   zone->prev_priority = priority;
@@ -1349,7 +1349,7 @@ void wakeup_kswapd(struct zone *zone, in
   return;
  if (pgdat->kswapd_max_order < order)
   pgdat->kswapd_max_order = order;
- if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+ if (!memset_zone_allowed(zone, __GFP_HARDWALL))
   return;
```

```
  if (!waitqueue_active(&pgdat->kswapd_wait))
   return;
Index: container-2.6.19-rc6/init/main.c
===================================================================
--- container-2.6.19-rc6.orig/init/main.c
+++ container-2.6.19-rc6/init/main.c
@@ -38,6 +38,7 @@
 #include <linux/writeback.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/container.h>
 #include <linux/efi.h>
 #include <linux/taskstats_kern.h>
@@ -571,6 +572,7 @@ asmlinkage void __init start_kernel(void
  vfs_caches_init_early();
  container_init_early();
  cpuset_init_early();
+ memset_init_early();
  mem_init();
  kmem_cache_init();
  setup_per_cpu_pageset();
@@ -602,6 +604,7 @@ asmlinkage void __init start_kernel(void
 #endif
  container_init();
  cpuset_init();
+ memset_init();
  taskstats_init_early();
  delayacct_init();

Index: container-2.6.19-rc6/fs/proc/array.c
===================================================================
--- container-2.6.19-rc6.orig/fs/proc/array.c
+++ container-2.6.19-rc6/fs/proc/array.c
@@ -73,6 +73,7 @@
 #include <linux/file.h>
 #include <linux/times.h>
 #include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/rcupdate.h>
 #include <linux/delayacct.h>

@@ -306,6 +307,7 @@ int proc_pid_status(struct task_struct *
  buffer = task_sig(task, buffer);
  buffer = task_cap(task, buffer);
  buffer = cpuset_task_status_allowed(task, buffer);
+ buffer = memset_task_status_allowed(task, buffer);
 #if defined(CONFIG_S390)
```

```
  buffer = task_show_regs(task, buffer);
 #endif
Index: container-2.6.19-rc6/init/Kconfig
===================================================================
--- container-2.6.19-rc6.orig/init/Kconfig
+++ container-2.6.19-rc6/init/Kconfig
@@ -260,7 +260,7 @@ config CPUSETS
  help
    This option will let you create and manage CPUSETs which
    allow dynamically partitioning a system into sets of CPUs and
-   Memory Nodes and assigning tasks to run only within those sets.
+   assigning tasks to run only within those sets.
    This is primarily useful on large SMP or NUMA systems.

    Say N if unsure.
@@ -269,6 +269,18 @@ config PROC_PID_CPUSET
  bool "Include legacy /proc/<pid>/cpuset file"
  depends on CPUSETS

+config MEMSETS
+ bool "Memset support"
+ depends on SMP
+ select CONTAINERS
+ help
+   This option will let you create and manage Memsets which
+   allow dynamically partitioning a system into sets of
+   Memory Nodes and assigning tasks to run only within those sets.
+   This is primarily useful on large SMP or NUMA systems.
+
+   Say N if unsure.
+
 config CONTAINER_CPUACCT
  bool "Simple CPU accounting container subsystem"
  select CONTAINERS

--
```

Subject: [PATCH 7/7] BeanCounters over generic process containers
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:55 GMT
View Forum Message <> Reply to Message

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

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

---
 fs/file_table.c | 11 +
 include/bc/beancounter.h | 192 ++++++++++++++++++++++
 include/bc/misc.h | 27 +++
 include/linux/fs.h | 3
 init/Kconfig | 4
 init/main.c | 3
 kernel/Makefile | 1
 kernel/bc/Kconfig | 17 ++
 kernel/bc/Makefile | 7
 kernel/bc/beancounter.c | 371 ++
 kernel/bc/misc.c | 56 +++++++
 11 files changed, 691 insertions(+), 1 deletion(-)

Index: container-2.6.19-rc6/init/Kconfig
 === =======
--- container-2.6.19-rc6.orig/init/Kconfig
+++ container-2.6.19-rc6/init/Kconfig
@@ -601,6 +601,10 @@ config STOP_MACHINE
   Need stop_machine() primitive.
 endmenu

+menu "Beancounters"
+source "kernel/bc/Kconfig"
+endmenu
+
 menu "Block layer"
 source "block/Kconfig"
 endmenu
Index: container-2.6.19-rc6/kernel/Makefile
 === =======
--- container-2.6.19-rc6.orig/kernel/Makefile
+++ container-2.6.19-rc6/kernel/Makefile
@@ -12,6 +12,7 @@ obj-y   = sched.o fork.o exec_domain.o

 obj-$(CONFIG_STACKTRACE) += stacktrace.o
 obj-y += time/
+obj-$(CONFIG_BEANCOUNTERS) += bc/
 obj-$(CONFIG_DEBUG_MUTEXES) += mutex-debug.o
 obj-$(CONFIG_LOCKDEP) += lockdep.o
 ifeq ($(CONFIG_PROC_FS),y)
Index: container-2.6.19-rc6/kernel/bc/Kconfig
 === =======

```
--- /dev/null
+++ container-2.6.19-rc6/kernel/bc/Kconfig
@@ -0,0 +1,17 @@
+config BEANCOUNTERS
+ bool "Enable resource accounting/control"
+ default n
+ select CONTAINERS
+ help
+   When Y this option provides accounting and allows configuring
+   limits for user's consumption of exhaustible system resources.
+   The most important resource controlled by this patch is unswappable
+   memory (either mlock'ed or used by internal kernel structures and
+   buffers). The main goal of this patch is to protect processes
+   from running short of important resources because of accidental
+   misbehavior of processes or malicious activity aiming to ``kill''
+   the system. It's worth mentioning that resource limits configured
+   by setrlimit(2) do not give an acceptable level of protection
+   because they cover only a small fraction of resources and work on a
+   per-process basis.  Per-process accounting doesn't prevent malicious
+   users from spawning a lot of resource-consuming processes.
Index: container-2.6.19-rc6/kernel/bc/Makefile
 ===================================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/kernel/bc/Makefile
@@ -0,0 +1,7 @@
+#
+# kernel/bc/Makefile
+#
+# Copyright (C) 2006 OpenVZ SWsoft Inc.
+#
+
+obj-y = beancounter.o misc.o
Index: container-2.6.19-rc6/include/bc/beancounter.h
 ===================================================================== =======
--- /dev/null
+++ container-2.6.19-rc6/include/bc/beancounter.h
@@ -0,0 +1,192 @@
+/*
+ * include/bc/beancounter.h
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#ifndef __BEANCOUNTER_H__
+#define __BEANCOUNTER_H__
+
+#include <linux/container.h>
```

```
+
+enum {
+ BC_KMEMSIZE,
+ BC_PRIVVMPAGES,
+ BC_PHYSPAGES,
+ BC_NUMTASKS,
+ BC_NUMFILES,
+
+ BC_RESOURCES
+};
+
+struct bc_resource_parm {
+ unsigned long barrier;
+ unsigned long limit;
+ unsigned long held;
+ unsigned long minheld;
+ unsigned long maxheld;
+ unsigned long failcnt;
+
+};
+
+#ifdef __KERNEL__
+
+#include <linux/list.h>
+#include <linux/spinlock.h>
+#include <linux/init.h>
+#include <linux/configfs.h>
+#include <asm/atomic.h>
+
+#define BC_MAXVALUE ((unsigned long)LONG_MAX)
+
+enum bc_severity {
+ BC_BARRIER,
+ BC_LIMIT,
+ BC_FORCE,
+};
+
+struct beancounter;
+
+#ifdef CONFIG_BEANCOUNTERS
+
+enum bc_attr_index {
+ BC_RES_HELD,
+ BC_RES_MAXHELD,
+ BC_RES_MINHELD,
+ BC_RES_BARRIER,
+ BC_RES_LIMIT,
+ BC_RES_FAILCNT,
```

```
+
+ BC_ATTRS
+};
+
+struct bc_resource {
+ char *bcr_name;
+ int      res_id;
+
+ int (*bcr_init)(struct beancounter *bc, int res);
+ int (*bcr_change)(struct beancounter *bc,
+   unsigned long new_bar, unsigned long new_lim);
+ void (*bcr_barrier_hit)(struct beancounter *bc);
+ int (*bcr_limit_hit)(struct beancounter *bc, unsigned long val,
+   unsigned long flags);
+ void (*bcr_fini)(struct beancounter *bc);
+
+ /* container file handlers */
+ struct cftype cft_attrs[BC_ATTRS];
+};
+
+extern struct bc_resource *bc_resources[];
+extern struct container_subsys bc_subsys;
+
+struct beancounter {
+ struct container_subsys_state css;
+ spinlock_t  bc_lock;
+
+ struct bc_resource_parm bc_parms[BC_RESOURCES];
+};
+
+/* Update the beancounter for a container */
+static inline void set_container_bc(struct container *cont,
+        struct beancounter *bc)
+{
+ cont->subsys[bc_subsys.subsys_id] = &bc->css;
+}
+
+/* Retrieve the beancounter for a container */
+static inline struct beancounter *container_bc(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, &bc_subsys),
+       struct beancounter, css);
+}
+
+/* Retrieve the beancounter for a task */
+static inline struct beancounter *task_bc(struct task_struct *task)
+{
+ return container_bc(task_container(task, &bc_subsys));
```

```
+}
+
+static inline void bc_adjust_maxheld(struct bc_resource_parm *parm)
+{
+ if (parm->maxheld < parm->held)
+  parm->maxheld = parm->held;
+}
+
+static inline void bc_adjust_minheld(struct bc_resource_parm *parm)
+{
+ if (parm->minheld > parm->held)
+  parm->minheld = parm->held;
+}
+
+static inline void bc_init_resource(struct bc_resource_parm *parm,
+  unsigned long bar, unsigned long lim)
+{
+ parm->barrier = bar;
+ parm->limit = lim;
+ parm->held = 0;
+ parm->minheld = 0;
+ parm->maxheld = 0;
+ parm->failcnt = 0;
+}
+
+int bc_change_param(struct beancounter *bc, int res,
+  unsigned long bar, unsigned long lim);
+
+int __must_check bc_charge_locked(struct beancounter *bc, int res_id,
+  unsigned long val, int strict, unsigned long flags);
+static inline int __must_check bc_charge(struct beancounter *bc, int res_id,
+  unsigned long val, int strict)
+{
+ int ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&bc->bc_lock, flags);
+ ret = bc_charge_locked(bc, res_id, val, strict, flags);
+ spin_unlock_irqrestore(&bc->bc_lock, flags);
+ return ret;
+}
+
+void __must_check bc_uncharge_locked(struct beancounter *bc, int res_id,
+  unsigned long val);
+static inline void bc_uncharge(struct beancounter *bc, int res_id,
+  unsigned long val)
+{
+ unsigned long flags;
```

```
+
+ spin_lock_irqsave(&bc->bc_lock, flags);
+ bc_uncharge_locked(bc, res_id, val);
+ spin_unlock_irqrestore(&bc->bc_lock, flags);
+}
+
+void __init bc_register_resource(int res_id, struct bc_resource *br);
+void __init bc_init_early(void);
+#else /* CONFIG_BEANCOUNTERS */
+static inline int __must_check bc_charge_locked(struct beancounter *bc, int res,
+  unsigned long val, int strict, unsigned long flags)
+{
+ return 0;
+}
+
+static inline int __must_check bc_charge(struct beancounter *bc, int res,
+  unsigned long val, int strict)
+{
+ return 0;
+}
+
+static inline void bc_uncharge_locked(struct beancounter *bc, int res,
+  unsigned long val)
+{
+}
+
+static inline void bc_uncharge(struct beancounter *bc, int res,
+  unsigned long val)
+{
+}
+
+static inline void bc_init_early(void)
+{
+}
+#endif /* CONFIG_BEANCOUNTERS */
+#endif /* __KERNEL__ */
+#endif
Index: container-2.6.19-rc6/init/main.c
 ===================================================================== =======
--- container-2.6.19-rc6.orig/init/main.c
+++ container-2.6.19-rc6/init/main.c
@@ -52,6 +52,8 @@
 #include <linux/debug_locks.h>
 #include <linux/lockdep.h>

+#include <bc/beancounter.h>
+
 #include <asm/io.h>
```

```
 #include <asm/bugs.h>
 #include <asm/setup.h>
@@ -482,6 +484,7 @@ asmlinkage void __init start_kernel(void
  char * command_line;
  extern struct kernel_param __start___param[], __stop___param[];

+ bc_init_early();
  smp_setup_processor_id();

  /*
Index: container-2.6.19-rc6/kernel/bc/beancounter.c
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc6/kernel/bc/beancounter.c
@@ -0,0 +1,371 @@
+/*
+ * kernel/bc/beancounter.c
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#include <linux/sched.h>
+#include <linux/list.h>
+#include <linux/hash.h>
+#include <linux/gfp.h>
+#include <linux/slab.h>
+#include <linux/module.h>
+#include <linux/fs.h>
+#include <linux/uaccess.h>
+
+#include <bc/beancounter.h>
+
+#define BC_HASH_BITS (8)
+#define BC_HASH_SIZE (1 << BC_HASH_BITS)
+
+static int bc_dummy_init(struct beancounter *bc, int i)
+{
+ bc_init_resource(&bc->bc_parms[i], BC_MAXVALUE, BC_MAXVALUE);
+ return 0;
+}
+
+static struct bc_resource bc_dummy_res = {
+ .bcr_name = "dummy",
+ .bcr_init = bc_dummy_init,
+};
+
+struct bc_resource *bc_resources[BC_RESOURCES] = {
```

```
+ [0 ... BC_RESOURCES - 1] = &bc_dummy_res,
+};
+
+struct beancounter init_bc;
+static kmem_cache_t *bc_cache;
+
+static int bc_create(struct container_subsys *ss,
+       struct container *cont)
+{
+ int i;
+ struct beancounter *new_bc;
+
+ if (!cont->parent) {
+  /* Early initialization for top container */
+  set_container_bc(cont, &init_bc);
+  init_bc.css.container = cont;
+  return 0;
+ }
+
+ new_bc = kmem_cache_alloc(bc_cache, GFP_KERNEL);
+ if (!new_bc)
+  return -ENOMEM;
+
+ spin_lock_init(&new_bc->bc_lock);
+
+ for (i = 0; i < BC_RESOURCES; i++) {
+  int err;
+  if ((err = bc_resources[i]->bcr_init(new_bc, i))) {
+   for (i--; i >= 0; i--)
+    if (bc_resources[i]->bcr_fini)
+     bc_resources[i]->bcr_fini(new_bc);
+   kmem_cache_free(bc_cache, new_bc);
+   return err;
+  }
+ }
+
+ new_bc->css.container = cont;
+ set_container_bc(cont, new_bc);
+ return 0;
+}
+
+static void bc_destroy(struct container_subsys *ss,
+         struct container *cont)
+{
+ struct beancounter *bc = container_bc(cont);
+ kmem_cache_free(bc_cache, bc);
+}
+
```

```
+int bc_charge_locked(struct beancounter *bc, int res, unsigned long val,
+  int strict, unsigned long flags)
+{
+ struct bc_resource_parm *parm;
+ unsigned long new_held;
+
+ BUG_ON(val > BC_MAXVALUE);
+
+ parm = &bc->bc_parms[res];
+ new_held = parm->held + val;
+
+ switch (strict) {
+ case BC_LIMIT:
+  if (new_held > parm->limit)
+   break;
+  /* fallthrough */
+ case BC_BARRIER:
+  if (new_held > parm->barrier) {
+   if (strict == BC_BARRIER)
+    break;
+   if (parm->held < parm->barrier &&
+     bc_resources[res]->bcr_barrier_hit)
+    bc_resources[res]->bcr_barrier_hit(bc);
+  }
+  /* fallthrough */
+ case BC_FORCE:
+  parm->held = new_held;
+  bc_adjust_maxheld(parm);
+  return 0;
+ default:
+  BUG();
+ }
+
+ if (bc_resources[res]->bcr_limit_hit)
+  return bc_resources[res]->bcr_limit_hit(bc, val, flags);
+
+ parm->failcnt++;
+ return -ENOMEM;
+}
+
+void bc_uncharge_locked(struct beancounter *bc, int res, unsigned long val)
+{
+ struct bc_resource_parm *parm;
+
+ BUG_ON(val > BC_MAXVALUE);
+
+ parm = &bc->bc_parms[res];
+ if (unlikely(val > parm->held)) {
```

```
+  printk(KERN_ERR "BC: Uncharging too much of %s: %lu vs %lu\n",
+    bc_resources[res]->bcr_name,
+    val, parm->held);
+  val = parm->held;
+ }
+
+ parm->held -= val;
+ bc_adjust_minheld(parm);
+}
+
+int bc_change_param(struct beancounter *bc, int res,
+      unsigned long bar, unsigned long lim)
+{
+ int ret;
+
+ ret = -EINVAL;
+ if (bar > lim)
+  goto out;
+ if (bar > BC_MAXVALUE || lim > BC_MAXVALUE)
+  goto out;
+
+ ret = 0;
+ spin_lock_irq(&bc->bc_lock);
+ if (bc_resources[res]->bcr_change) {
+  ret = bc_resources[res]->bcr_change(bc, bar, lim);
+  if (ret < 0)
+   goto out_unlock;
+ }
+
+ bc->bc_parms[res].barrier = bar;
+ bc->bc_parms[res].limit = lim;
+
+out_unlock:
+ spin_unlock_irq(&bc->bc_lock);
+out:
+ return ret;
+}
+
+static ssize_t bc_resource_show(struct container *cont, struct cftype *cft,
+    struct file *file, char __user *userbuf,
+    size_t nbytes, loff_t *ppos)
+{
+ struct beancounter *bc = container_bc(cont);
+ int idx = cft->private;
+ struct bc_resource *res = container_of(cft, struct bc_resource,
+           cft_attrs[idx]);
+
+ char *page;
```

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

```
+
+ char buffer[64];
+ unsigned long val;
+ char *end;
+ int ret = 0;
+
+ if (nbytes >= sizeof(buffer))
+  return -E2BIG;
+
+ if (copy_from_user(buffer, userbuf, nbytes))
+  return -EFAULT;
+
+ buffer[nbytes] = 0;
+
+ container_manage_lock();
+
+ if (container_is_removed(cont)) {
+  ret = -ENODEV;
+  goto out_unlock;
+ }
+
+ ret = -EINVAL;
+ val = simple_strtoul(buffer, &end, 10);
+ if (*end != '\0')
+  goto out_unlock;
+
+ switch (idx) {
+ case BC_RES_BARRIER:
+  ret = bc_change_param(bc, res->res_id,
+         val, bc->bc_parms[res->res_id].limit);
+  break;
+ case BC_RES_LIMIT:
+  ret = bc_change_param(bc, res->res_id,
+         bc->bc_parms[res->res_id].barrier, val);
+  break;
+ }
+
+ if (ret == 0)
+  ret = nbytes;
+
+ out_unlock:
+ container_manage_unlock();
+
+ return ret;
+}
+
+
+
```

```
+void __init bc_register_resource(int res_id, struct bc_resource *br)
+{
+ int attr;
+ BUG_ON(bc_resources[res_id] != &bc_dummy_res);
+ BUG_ON(res_id >= BC_RESOURCES);
+
+ bc_resources[res_id] = br;
+ br->res_id = res_id;
+
+ for (attr = 0; attr < BC_ATTRS; attr++) {
+
+  /* Construct a file handler for each attribute of this
+   * resource; the name is of the form
+   * "bc.<resname>.<attrname>" */
+
+  struct cftype *cft = &br->cft_attrs[attr];
+  const char *attrname;
+  cft->private = attr;
+  strcpy(cft->name, "bc.");
+  strcat(cft->name, br->bcr_name);
+  strcat(cft->name, ".");
+  switch (attr) {
+  case BC_RES_HELD:
+   attrname = "held"; break;
+  case BC_RES_BARRIER:
+   attrname = "barrier"; break;
+  case BC_RES_LIMIT:
+   attrname = "limit"; break;
+  case BC_RES_MAXHELD:
+   attrname = "maxheld"; break;
+  case BC_RES_MINHELD:
+   attrname = "minheld"; break;
+  case BC_RES_FAILCNT:
+   attrname = "failcnt"; break;
+  default:
+   BUG();
+  }
+  strcat(cft->name, attrname);
+  cft->read = bc_resource_show;
+  cft->write = bc_resource_store;
+ }
+}
+
+void __init bc_init_early(void)
+{
+ int i;
+
+ spin_lock_init(&init_bc.bc_lock);
```

```
+
+ for (i = 0; i < BC_RESOURCES; i++) {
+  init_bc.bc_parms[i].barrier = BC_MAXVALUE;
+  init_bc.bc_parms[i].limit = BC_MAXVALUE;
+
+ }
+
+ if (container_register_subsys(&bc_subsys) < 0)
+  panic("Couldn't register beancounter subsystem");
+
+}
+
+int __init bc_init_late(void)
+{
+ bc_cache = kmem_cache_create("beancounters",
+   sizeof(struct beancounter), 0,
+   SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL, NULL);
+ return 0;
+}
+
+__initcall(bc_init_late);
+
+static int bc_populate(struct container_subsys *ss, struct container *cont)
+{
+ int err;
+ int attr, res;
+ for (res = 0; res < BC_RESOURCES; res++) {
+  struct bc_resource *bcr = bc_resources[res];
+
+  for (attr = 0; attr < BC_ATTRS; attr++) {
+   struct cftype *cft = &bcr->cft_attrs[attr];
+   if (!cft->name[0]) continue;
+   err = container_add_file(cont, cft);
+   if (err < 0) return err;
+  }
+ }
+ return 0;
+}
+
+struct container_subsys bc_subsys = {
+ .name = "bc",
+ .create = bc_create,
+ .destroy = bc_destroy,
+ .populate = bc_populate,
+ .subsys_id = -1,
+};
+
+EXPORT_SYMBOL(bc_resources);
```

```
+EXPORT_SYMBOL(init_bc);
+EXPORT_SYMBOL(bc_change_param);
Index: container-2.6.19-rc6/include/bc/misc.h
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc6/include/bc/misc.h
@@ -0,0 +1,27 @@
+/*
+ * include/bc/misc.h
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#ifndef __BC_MISC_H__
+#define __BC_MISC_H__
+
+struct file;
+
+#ifdef CONFIG_BEANCOUNTERS
+int __must_check bc_file_charge(struct file *);
+void bc_file_uncharge(struct file *);
+#else
+static inline int __must_check bc_file_charge(struct file *f)
+{
+ return 0;
+}
+
+static inline void bc_file_uncharge(struct file *f)
+{
+}
+#endif
+
+#endif
Index: container-2.6.19-rc6/kernel/bc/misc.c
 ===================================================================
--- /dev/null
+++ container-2.6.19-rc6/kernel/bc/misc.c
@@ -0,0 +1,56 @@
+
+#include <linux/fs.h>
+#include <bc/beancounter.h>
+
+int bc_file_charge(struct file *file)
+{
+ int sev;
+ struct beancounter *bc;
+
```

```
+ task_lock(current);
+ bc = task_bc(current);
+ css_get_current(&bc->css);
+ task_unlock(current);
+
+ sev = (capable(CAP_SYS_ADMIN) ? BC_LIMIT : BC_BARRIER);
+
+ if (bc_charge(bc, BC_NUMFILES, 1, sev)) {
+  css_put(&bc->css);
+  return -EMFILE;
+ }
+
+ file->f_bc = bc;
+ return 0;
+}
+
+void bc_file_uncharge(struct file *file)
+{
+ struct beancounter *bc;
+
+ bc = file->f_bc;
+ bc_uncharge(bc, BC_NUMFILES, 1);
+ css_put(&bc->css);
+}
+
+#define BC_NUMFILES_BARRIER 256
+#define BC_NUMFILES_LIMIT 512
+
+static int bc_files_init(struct beancounter *bc, int i)
+{
+ bc_init_resource(&bc->bc_parms[BC_NUMFILES],
+   BC_NUMFILES_BARRIER, BC_NUMFILES_LIMIT);
+ return 0;
+}
+
+static struct bc_resource bc_files_resource = {
+ .bcr_name = "numfiles",
+ .bcr_init = bc_files_init,
+};
+
+static int __init bc_misc_init_resource(void)
+{
+ bc_register_resource(BC_NUMFILES, &bc_files_resource);
+  return 0;
+}
+
+__initcall(bc_misc_init_resource);
Index: container-2.6.19-rc6/fs/file_table.c
```

```
 ===================================================== =======
--- container-2.6.19-rc6.orig/fs/file_table.c
+++ container-2.6.19-rc6/fs/file_table.c
@@ -22,6 +22,8 @@
 #include <linux/sysctl.h>
 #include <linux/percpu_counter.h>

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

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

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

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

@@ -135,6 +140,10 @@ fail_sec:
  file_free(f);
 fail:
  return NULL;
+
+ fail_charge:
+ kmem_cache_free(filp_cachep, f);
+ return NULL;
 }

 EXPORT_SYMBOL(get_empty_filp);
Index: container-2.6.19-rc6/include/linux/fs.h
 ===================================================== =======
--- container-2.6.19-rc6.orig/include/linux/fs.h
+++ container-2.6.19-rc6/include/linux/fs.h
@@ -750,6 +750,9 @@ struct file {
```

```
  spinlock_t  f_ep_lock;
 #endif /* #ifdef CONFIG_EPOLL */
  struct address_space *f_mapping;
+#ifdef CONFIG_BEANCOUNTERS
+ struct beancounter      *f_bc;
+#endif
 };
 extern spinlock_t files_lock;
 #define file_list_lock() spin_lock(&files_lock);

--
```

Subject: Re: [PATCH 0/7] Generic Process Containers (+ ResGroups/BeanCounters)
Posted by Paul Jackson on Thu, 30 Nov 2006 07:32:29 GMT
View Forum Message <> Reply to Message

I got a chance to build and test this patch set, to see if it behaved like I expected cpusets to behave, on an ia64 SN2 Altix system.

Two details - otherwise looked good. I continue to like this approach.

The two details are (1) /proc/<pid>/cpuset not configured by default if CPUSETS configured, and (2) a locking bug wedging tasks trying to rmdir a cpuset off the notify_on_release hook.

1) I had to enable CONFIG_PROC_PID_CPUSET. I used the following one line change to do this. I am willing to consider, in due time, phasing out such legacy cpuset support. But so long as it is small stuff that is not getting in anyone's way, I think we should take our sweet time about doing so -- as in a year or two after marking it deprecated or some such. No sense deciding that matter now; keep the current cpuset API working throughout any transitition to container based cpusets, then revisit the question of whether to deprecate and eventually remove these kernel API details, later on, after the major reconstruction dust settles.
   In general, we try to avoid removing kernel API's, especially if they are happily being used and working and not causing anyone grief.

```
============================ begin ============================
--- 2.6.19-rc5.orig/init/Kconfig 2006-11-29 21:14:48.071114833 -0800
+++ 2.6.19-rc5/init/Kconfig 2006-11-29 22:19:02.015166048 -0800
@@ -268,6 +268,7 @@ config CPUSETS
 config PROC_PID_CPUSET
```

```
  bool "Include legacy /proc/<pid>/cpuset file"
  depends on CPUSETS
+ default y if CPUSETS

 config CONTAINER_CPUACCT
  bool "Simple CPU accounting container subsystem"
============================== end ==============================
```

2) I wedged the kernel on the container_lock, doing a removal of a cpuset
   using notify_on_release.

   Right now, that test system has the following two tasks, wedged:

```
============================== begin ==============================
F S UID   PID PPID C PRI NI ADDR SZ  WCHAN  STIME TTY  TIME     CMD
0 S root 4992   34 0  71 -5 -   380   wait   22:51 ?   00:00:00 /bin/sh /sbin/cpuset_release_agent
/cpuset_test_tree
0 D root 4994 4992 0  72 -5 -   200 contai   22:51 ?   00:00:00 rmdir /dev/cpuset//cpuset_test_tree
============================== end ==============================
```

   I had a cpuset called /cpuset_test_tree, and some sub-cpusets
   below it. I marked it 'notify_on_release' and then removed all
   tasks from it, and then removed the child cpusets that it had.
   Removing that last child cpuset presumably triggered the above
   callout to /sbin/cpuset_release_agent, which called rmdir.

   That wait address (from /proc/4994/stat) in hex is a0000001000f1060,
   and my System.map has the two lines:

```
 a0000001000f1040 T container_lock
 a0000001000f1360 T container_manage_unlock
```

   So it is wedged in container_lock.

   I have subsequently also wedged an 'ls' command trying to scan this
   /dev/cpuset directory, waiting in the kernel routine vfs_readdir
   (not surprising, given that I'm in the middle of doing a rmdir on
   that directory.)

   If you don't immediately see the problem, I can go back and get a
   kernel stack trace or whatever else you need.

   This lockup occurred the first, and thus far only, time that I tried
   to use notify_on_release to rmdir a cpuset. So I presume it is an
   easy failure for me to reproduce.

--

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

Subject: Re: [PATCH 0/7] Generic Process Containers (+ ResGroups/BeanCounters)
Posted by Paul Menage on Thu, 30 Nov 2006 08:01:06 GMT
View Forum Message <> Reply to Message

On 11/29/06, Paul Jackson <pj@sgi.com> wrote:
> config PROC_PID_CPUSET
> bool "Include legacy /proc/<pid>/cpuset file"
> depends on CPUSETS
> + default y if CPUSETS
>

Sounds very reasonable.

> 2) I wedged the kernel on the container_lock, doing a removal of a cpuset
> using notify_on_release.
>

I guess I've not really tested doing interesting things from the notify_on_release code, just checked that it successfully executed a simple command. I'll look into it.

Thanks for the feedback.

Paul

Subject: Re: [PATCH 0/7] Generic Process Containers (+ ResGroups/BeanCounters)
Posted by Paul Menage on Thu, 30 Nov 2006 08:51:26 GMT
View Forum Message <> Reply to Message

On 11/29/06, Paul Jackson <pj@sgi.com> wrote:
>
> 2) I wedged the kernel on the container_lock, doing a removal of a cpuset
> using notify_on_release.

I couldn't reproduce this, with a /sbin/cpuset_release_agent that does:

#!/bin/bash
logger cpuset_release_agent $1
rmdir /dev/cpuset/$1

and running the commands:

```
while true; do
  mkdir -p /dev/cpuset/bar/foo
  echo 1 > /dev/cpuset/bar/notify_on_release
  rmdir /dev/cpuset/bar/foo
  usleep 1000
done
```

Is it actually reproducible for you? If so, could you get a fuller backtrace?

Thanks,

Paul

Subject: [Patch 1/3] Miscellaneous container fixes
Posted by Srivatsa Vaddagiri on Fri, 01 Dec 2006 16:46:32 GMT
View Forum Message <> Reply to Message

This patches fixes various bugs I hit in the recently posted container patches.

1. If a subsystem registers with fork/exit hook during bootup (much before rcu is initialized), then the resulting synchronize_rcu() in container_register_subsys() hangs. Avoid this by not calling synchronize_rcu() if we arent fully booted yet.

2. If cpuset_create fails() for some reason, then the resulting call to cpuset_destroy can trip. Avoid this by initializing container->...->cpuset pointer to NULL in cpuset_create().

3. container_rmdir->cpuset_destroy->update_flag can deadlock on container_lock(). Avoid this by introducing __update_flag, which doesnt take container_lock().

(I have also hit some lockdep warnings. Will post them after some review, to make sure that they are not introduced by my patches).

Signed-off-by : Srivatsa Vaddagiri <vatsa@in.ibm.com>

```
 linux-2.6.19-rc6-vatsa/kernel/container.c |    4 ++-
 linux-2.6.19-rc6-vatsa/kernel/cpuset.c    |   35 ++++++++++++++++++++++++------
 2 files changed, 31 insertions(+), 8 deletions(-)
```

```
diff -puN kernel/container.c~container_fixes kernel/container.c
--- linux-2.6.19-rc6/kernel/container.c~container_fixes 2006-12-01 16:19:41.000000000 +0530
+++ linux-2.6.19-rc6-vatsa/kernel/container.c 2006-12-01 16:20:20.000000000 +0530
@@ -1344,6 +1344,7 @@ static long container_create(struct cont
  cont->parent = parent;
  cont->root = parent->root;
  cont->hierarchy = parent->hierarchy;
+ cont->top_container = parent->top_container;

  for_each_subsys(cont->hierarchy, ss) {
   err = ss->create(ss, cont);
@@ -1580,7 +1581,8 @@ int container_register_subsys(struct con
   if (!need_forkexit_callback &&
       (new_subsys->fork || new_subsys->exit)) {
    need_forkexit_callback = 1;
-   synchronize_rcu();
+  if (system_state == SYSTEM_RUNNING)
+    synchronize_rcu();
   }

   /* If this subsystem requested that it be notified with fork
diff -puN kernel/cpuset.c~container_fixes kernel/cpuset.c
--- linux-2.6.19-rc6/kernel/cpuset.c~container_fixes 2006-12-01 19:02:35.000000000 +0530
+++ linux-2.6.19-rc6-vatsa/kernel/cpuset.c 2006-12-01 20:43:52.000000000 +0530
@@ -97,14 +97,22 @@ struct cpuset {
 /* Update the cpuset for a container */
 static inline void set_container_cs(struct container *cont, struct cpuset *cs)
 {
- cont->subsys[cpuset_subsys.subsys_id] = &cs->css;
+ if (cs)
+  cont->subsys[cpuset_subsys.subsys_id] = &cs->css;
+ else
+  cont->subsys[cpuset_subsys.subsys_id] = NULL;
 }

 /* Retrieve the cpuset for a container */
 static inline struct cpuset *container_cs(struct container *cont)
 {
- return container_of(container_subsys_state(cont, &cpuset_subsys),
-       struct cpuset, css);
+ struct container_subsys_state *css;
+
+ css = container_subsys_state(cont, &cpuset_subsys);
+ if (css)
+  return container_of(css, struct cpuset, css);
+ else
+  return NULL;
```

```
 }

 /* Retrieve the cpuset for a task */
@@ -698,7 +706,7 @@ static int update_memory_pressure_enable
  * Call with manage_mutex held.
  */

-static int update_flag(cpuset_flagbits_t bit, struct cpuset *cs, char *buf)
+static int __update_flag(cpuset_flagbits_t bit, struct cpuset *cs, char *buf)
 {
  int turning_on;
  struct cpuset trialcs;
@@ -717,18 +725,27 @@ static int update_flag(cpuset_flagbits_t
   return err;
  cpu_exclusive_changed =
   (is_cpu_exclusive(cs) != is_cpu_exclusive(&trialcs));
- container_lock();
  if (turning_on)
   set_bit(bit, &cs->flags);
  else
   clear_bit(bit, &cs->flags);
- container_unlock();

  if (cpu_exclusive_changed)
   update_cpu_domains(cs);
  return 0;
 }

+static int update_flag(cpuset_flagbits_t bit, struct cpuset *cs, char *buf)
+{
+ int err;
+
+ container_lock();
+ err = __update_flag(bit, cs, buf);
+ container_unlock();
+
+ return err;
+}
+
 /*
  * Frequency meter - How fast is some event occurring?
  *
@@ -1165,6 +1182,7 @@ int cpuset_create(struct container_subsy
   return 0;
  }
  parent = container_cs(cont->parent);
+ set_container_cs(cont, NULL);
  cs = kmalloc(sizeof(*cs), GFP_KERNEL);
```

```
  if (!cs)
   return -ENOMEM;
@@ -1202,9 +1220,12 @@ void cpuset_destroy(struct container_sub
 {
  struct cpuset *cs = container_cs(cont);

+ if (!cs)
+  return;
+
  cpuset_update_task_memory_state();
  if (is_cpu_exclusive(cs)) {
-  int retval = update_flag(CS_CPU_EXCLUSIVE, cs, "0");
+  int retval = __update_flag(CS_CPU_EXCLUSIVE, cs, "0");
   BUG_ON(retval);
  }
  number_of_cpusets--;
_


--
Regards,
vatsa
```

Subject: [Patch 2/3] cpu controller (v3) - based on RTLIMIT_RT_CPU patch
Posted by Srivatsa Vaddagiri on Fri, 01 Dec 2006 16:51:25 GMT
View Forum Message <> Reply to Message

Nick/Ingo,

 Here's another approach for a minimal cpu controller. Would greatly
appreciate any feedback as before.

This version is inspired by Ingo's RTLIMIT_RT_CPU patches found here:

 http://kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2
.6.11-rc2/2.6.11-rc2-mm2/broken-out/rlimit_rt_cpu.patch

This patch is also about 80% smaller than the patches I had posted earlier:

 http://lkml.org/lkml/2006/9/28/236

Primary differences between Ingo's RT_LIMIT_CPU patch and this one:

- This patch handles starvation of lower priority tasks in a group.
- This patch uses tokens for accounting cpu consumption. I didnt
  get good results with decaying avg approach used in the rtlimit patches.
- Task grouping based on cpuset/containers and not on rtlimit.

Other features:

- Retains one-runqueue-per-cpu, as is prevalent today
- scheduling no longer of O(1) complexity, similar to rtlimit patches.
  This can be avoided if we use separate runqueues for different groups
  (as was done in http://lkml.org/lkml/2006/9/28/236)
- Only limit supported (no guarantee)

Unsupported feature:

- SMP load balance aware of group limits. This can be handled using
  smpnice later if required (http://lkml.org/lkml/2006/9/28/244)

Signed-off-by : Srivatsa Vaddagiri <vatsa@in.ibm.com>

```
 linux-2.6.19-rc6-vatsa/include/linux/sched.h |    3
 linux-2.6.19-rc6-vatsa/kernel/sched.c        |  195 ++++++++++++++++++++++++++-
 2 files changed, 195 insertions(+), 3 deletions(-)

diff -puN include/linux/sched.h~cpu_ctlr include/linux/sched.h
--- linux-2.6.19-rc6/include/linux/sched.h~cpu_ctlr 2006-12-01 20:45:08.000000000 +0530
+++ linux-2.6.19-rc6-vatsa/include/linux/sched.h 2006-12-01 20:45:08.000000000 +0530
@@ -1095,6 +1095,9 @@ static inline void put_task_struct(struc
      /* Not implemented yet, only for 486*/
 #define PF_STARTING 0x00000002 /* being created */
 #define PF_EXITING 0x00000004 /* getting shut down */
+#ifdef CONFIG_CPUMETER
+#define PF_STARVING 0x00000010      /* Task starving for CPU */
+#endif
 #define PF_FORKNOEXEC 0x00000040 /* forked but didn't exec */
 #define PF_SUPERPRIV 0x00000100 /* used super-user privileges */
 #define PF_DUMPCORE 0x00000200 /* dumped core */
diff -puN kernel/sched.c~cpu_ctlr kernel/sched.c
--- linux-2.6.19-rc6/kernel/sched.c~cpu_ctlr 2006-12-01 20:45:08.000000000 +0530
+++ linux-2.6.19-rc6-vatsa/kernel/sched.c 2006-12-01 20:45:08.000000000 +0530
@@ -264,10 +264,25 @@ struct rq {
  unsigned long ttwu_local;
 #endif
  struct lock_class_key rq_lock_key;
+#ifdef CONFIG_CPUMETER
+ unsigned long last_update;
+ int need_recheck;
+#endif
 };
```

```
 static DEFINE_PER_CPU(struct rq, runqueues);

+struct cpu_usage {
+ long tokens;
+ unsigned long last_update;
+ int starve_count;
+};
+
+struct task_grp {
+ unsigned long limit;
+ struct cpu_usage *cpu_usage; /* per-cpu ptr */
+};
+
 static inline int cpu_of(struct rq *rq)
 {
 #ifdef CONFIG_SMP
@@ -705,6 +720,137 @@ enqueue_task_head(struct task_struct *p,
  p->array = array;
 }

+#ifdef CONFIG_CPUMETER
+
+#define task_starving(p) (p->flags & PF_STARVING)
+
+static inline struct task_grp *task_grp(struct task_struct *p)
+{
+ return NULL;
+}
+
+/* Mark a task starving - either we shortcircuited its timeslice or we didnt
+ * pick it to run (because its group ran out of bandwidth limit).
+ */
+static inline void set_tsk_starving(struct task_struct *p, struct task_grp *grp)
+{
+ struct cpu_usage *grp_usage;
+
+ if (task_starving(p))
+  return;
+
+ BUG_ON(!grp);
+ grp_usage = per_cpu_ptr(grp->cpu_usage, task_cpu(p));
+ grp_usage->starve_count++;
+ p->flags |= PF_STARVING;
+}
+
+/* Clear a task's starving flag */
+static inline void clear_tsk_starving(struct task_struct *p,
```

```
+       struct task_grp *grp)
+{
+ struct cpu_usage *grp_usage;
+
+ if (!task_starving(p))
+  return;
+
+ BUG_ON(!grp);
+ grp_usage = per_cpu_ptr(grp->cpu_usage, task_cpu(p));
+ grp_usage->starve_count--;
+ p->flags &= ~PF_STARVING;
+}
+
+/* Does the task's group have starving tasks? */
+static inline int is_grp_starving(struct task_struct *p)
+{
+ struct task_grp *grp = task_grp(p);
+ struct cpu_usage *grp_usage;
+
+ if (!grp)
+  return 0;
+
+ grp_usage = per_cpu_ptr(grp->cpu_usage, task_cpu(p));
+ if (grp_usage->starve_count)
+  return 1;
+
+ return 0;
+}
+
+/* Are we past the 1-sec control window? If so, all groups get to renew their
+ * expired tokens.
+ */
+static inline void adjust_control_window(struct task_struct *p)
+{
+ struct rq *rq = task_rq(p);
+ unsigned long delta;
+
+ delta = jiffies - rq->last_update;
+ if (delta >= HZ) {
+  rq->last_update += (delta/HZ) * HZ;
+  rq->need_recheck = 1;
+ }
+}
+
+/* Account group's cpu usage */
+static inline void inc_cpu_usage(struct task_struct *p)
+{
+ struct task_grp *grp = task_grp(p);
```

```
+ struct cpu_usage *grp_usage;
+
+ adjust_control_window(p);
+
+ if (!grp || !grp->limit || rt_task(p))
+  return;
+
+ grp_usage = per_cpu_ptr(grp->cpu_usage, task_cpu(p));
+ grp_usage->tokens--;
+}
+
+static inline int task_over_cpu_limit(struct task_struct *p)
+{
+ struct rq *rq = task_rq(p);
+ struct task_grp *grp = task_grp(p);
+ struct cpu_usage *grp_usage;
+
+ adjust_control_window(p);
+
+ if (!grp || !grp->limit || !rq->need_recheck)
+   return 0;
+
+ grp_usage = per_cpu_ptr(grp->cpu_usage, task_cpu(p));
+ if (grp_usage->last_update != rq->last_update) {
+  /* Replenish tokens */
+  grp_usage->tokens = grp->limit * HZ / 100;
+  grp_usage->last_update = rq->last_update;
+ }
+
+ if (grp_usage->tokens <= 0)
+  return 1;
+
+ return 0;
+}
+
+static inline void rq_set_recheck(struct rq *rq, int check)
+{
+ rq->need_recheck = check;
+}
+
+#else
+
+#define task_starving(p) 0
+
+struct task_grp;
+
+static struct task_grp *task_grp(struct task_struct *p) { return NULL;}
+static void inc_cpu_usage(struct task_struct *p) { }
```

```
+static int task_over_cpu_limit(struct task_struct *p) { return 0; }
+static void set_tsk_starving(struct task_struct *p, struct task_grp *grp) { }
+static void clear_tsk_starving(struct task_struct *p, struct task_grp *grp) { }
+static int is_grp_starving(struct task_struct *p) { return 0;}
+static inline void rq_set_recheck(struct rq *rq, int check) { }
+
+#endif  /* CONFIG_CPUMETER */
+
 /*
  * __normal_prio - return the priority that is based on the static
  * priority but is modified by bonuses/penalties.
@@ -847,6 +993,7 @@ static void __activate_task(struct task_
   target = rq->expired;
  enqueue_task(p, target);
  inc_nr_running(p, rq);
+ rq_set_recheck(rq, 1);
 }

 /*
@@ -1586,6 +1733,9 @@ void fastcall sched_fork(struct task_str
  /* Want to start with kernel preemption disabled. */
  task_thread_info(p)->preempt_count = 1;
 #endif
+#ifdef CONFIG_CPUMETER
+ p->flags &= ~PF_STARVING;
+#endif
  /*
   * Share the timeslice between parent and child, thus the
   * total amount of pending timeslices in the system doesn't change,
@@ -2047,6 +2197,8 @@ static void pull_task(struct rq *src_rq,
 {
  dequeue_task(p, src_array);
  dec_nr_running(p, src_rq);
+ clear_tsk_starving(p, task_grp(p));
+ rq_set_recheck(this_rq, 1);
  set_task_cpu(p, this_cpu);
  inc_nr_running(p, this_rq);
  enqueue_task(p, this_array);
@@ -3068,6 +3220,9 @@ void scheduler_tick(void)
   goto out;
  }
  spin_lock(&rq->lock);
+
+ inc_cpu_usage(p);
+
  /*
   * The task was running during this tick - update the
   * time slice counter. Note: we do not update a thread's
```

```
@@ -3094,17 +3249,18 @@ void scheduler_tick(void)
   dequeue_task(p, rq->active);
   set_tsk_need_resched(p);
   p->prio = effective_prio(p);
-  p->time_slice = task_timeslice(p);
   p->first_time_slice = 0;

   if (!rq->expired_timestamp)
    rq->expired_timestamp = jiffies;
-  if (!TASK_INTERACTIVE(p) || expired_starving(rq)) {
+  if (!TASK_INTERACTIVE(p) || expired_starving(rq)
+       || task_over_cpu_limit(p)) {
    enqueue_task(p, rq->expired);
    if (p->static_prio < rq->best_expired_prio)
     rq->best_expired_prio = p->static_prio;
   } else
    enqueue_task(p, rq->active);
+  goto out_unlock;
  } else {
   /*
    * Prevent a too long timeslice allowing a task to monopolize
@@ -3131,6 +3287,14 @@ void scheduler_tick(void)
   }
  }
+
+ if (task_over_cpu_limit(p)) {
+  dequeue_task(p, rq->active);
+  set_tsk_need_resched(p);
+  enqueue_task(p, rq->expired);
+  set_tsk_starving(p, task_grp(p));
+ }
+
 out_unlock:
  spin_unlock(&rq->lock);
 out:
@@ -3320,7 +3484,7 @@ asmlinkage void __sched schedule(void)
  struct list_head *queue;
  unsigned long long now;
  unsigned long run_time;
- int cpu, idx, new_prio;
+ int cpu, idx, new_prio, array_switch;
  long *switch_count;
  struct rq *rq;

@@ -3379,6 +3543,7 @@ need_resched_nonpreemptible:
   else {
    if (prev->state == TASK_UNINTERRUPTIBLE)
```

```
     rq->nr_uninterruptible++;
+    clear_tsk_starving(prev, task_grp(prev));
     deactivate_task(prev, rq);
    }
   }
@@ -3394,11 +3559,15 @@ need_resched_nonpreemptible:
    }
   }

+ array_switch = 0;
+
+pick_next_task:
   array = rq->active;
   if (unlikely(!array->nr_active)) {
    /*
     * Switch the active and expired arrays.
     */
+   array_switch++;
    schedstat_inc(rq, sched_switch);
    rq->active = rq->expired;
    rq->expired = array;
@@ -3411,6 +3580,25 @@ need_resched_nonpreemptible:
   queue = array->queue + idx;
   next = list_entry(queue->next, struct task_struct, run_list);

+ /* If we have done an array switch twice, it means we cant find any
+  * task which isn't above its limit and hence we just run the
+  * first task on the active array.
+  */
+ if (array_switch < 2 && (task_over_cpu_limit(next) ||
+    (!task_starving(next) && is_grp_starving(next)))) {
+   dequeue_task(next, rq->active);
+   enqueue_task(next, rq->expired);
+   if (next->time_slice)
+    set_tsk_starving(next, task_grp(next));
+   goto pick_next_task;
+ }
+
+ if (task_over_cpu_limit(next))
+   rq_set_recheck(rq, 0);
+ if (!next->time_slice)
+   next->time_slice = task_timeslice(next);
+ clear_tsk_starving(next, task_grp(next));
+
   if (!rt_task(next) && interactive_sleep(next->sleep_type)) {
    unsigned long long delta = now - next->timestamp;
    if (unlikely((long long)(now - next->timestamp) < 0))
@@ -4965,6 +5153,7 @@ static int __migrate_task(struct task_st
```

```
  if (!cpu_isset(dest_cpu, p->cpus_allowed))
   goto out;

+ clear_tsk_starving(p, task_grp(p));
  set_task_cpu(p, dest_cpu);
  if (p->array) {
   /*
```

_
--
Regards,
vatsa