Subject: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Menage on Tue, 29 May 2007 13:01:04 GMT
View Forum Message <> Reply to Message

This is an update to my multi-hierarchy multi-subsystem generic
process containers patch. Changes since V9 (April 27th) include:

- The patchset has been rebased over 2.6.22-rc2-mm1

- A lattice of lists linking tasks to their css_groups and css_groups
to their containers has been added to support more efficient iteration
across the member tasks of a container.

- Support for the cpusets "release agent" functionality has been added
back in; this is based on a workqueue concept similar to the changes
that Cliff Wickman has been pushing for supporting CPU hot-unplug.

- Several uses of tasklist_lock replaced by reliance on RCU

- Misc cleanups

- Tested with a tweaked version of PaulJ's cpuset_test script

Still TODO:

- decide whether "Containers" is an acceptable name for the system
given its usage by some other development groups, or whether something
else (ProcessSets? ResourceGroups? TaskGroups?) would be better. I'm
inclined to leave this political decision to Andrew/Linus once they're
happy with the technical aspects of the patches.

- add a hash-table based lookup for css_group objects.

- use seq_file properly in container tasks files to avoid having to
allocate a big array for all the container's task pointers.

- lots more testing

- define standards for container file names

--

Generic Process Containers

There have recently been various proposals floating around for
resource management/accounting and other task grouping subsystems in
the kernel, including ResGroups, User BeanCounters, NSProxy

containers, 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, or control other behaviour of the 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, along with a couple of simple example subsystems.

The patch set is structured as follows:

1) Basic container framework - filesystem and tracking structures

2) Simple CPU Accounting example subsystem

3) Support for the "tasks" control file

4) Hooks for fork() and exit()

5) Support for the container_clone() operation

6) Add /proc reporting interface

7) Make cpusets a container subsystem

8) Share container subsystem pointer arrays between tasks with the
   same assignments

9) Simple container debugging subsystem

10) Support for a userspace "release agent", similar to the cpusets
    release agent functionality

The intention is that the various resource management and virtualization 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, or use either of them as the resource-control portion of a virtual server system.

- 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 07/10] Containers(V10): Make cpusets a client of containers
Posted by Paul Menage on Tue, 29 May 2007 13:01:11 GMT
View Forum Message <> Reply to Message

This patch removes the filesystem support logic from the cpusets
system and makes cpusets a container subsystem

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

 Documentation/cpusets.txt | 91 +--
 fs/proc/base.c | 4
 include/linux/container_subsys.h | 6
 include/linux/cpuset.h | 12
 include/linux/mempolicy.h | 12
 include/linux/sched.h | 3
 init/Kconfig | 6
 kernel/cpuset.c | 1151 +++++++---------------------------------
 kernel/exit.c | 2
 kernel/fork.c | 3
 mm/mempolicy.c | 2
 11 files changed, 241 insertions(+), 1051 deletions(-)

Index: container-2.6.22-rc2-mm1/Documentation/cpusets.txt
 === =======
--- container-2.6.22-rc2-mm1.orig/Documentation/cpusets.txt
+++ container-2.6.22-rc2-mm1/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
@@ -114,7 +115,7 @@ 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
@@ -144,15 +145,10 @@ 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)
@@ -162,16 +158,15 @@ in the format seen in the following exam
   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:
@@ -236,21 +231,7 @@ 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
@@ -307,7 +288,7 @@ 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
@@ -378,7 +359,7 @@ 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
@@ -468,7 +449,7 @@ 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 -ocpuset cpuset /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.
@@ -480,7 +461,7 @@ 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 -ocpuset cpuset /dev/cpuset
   cd /dev/cpuset
   mkdir Charlie
   cd Charlie
@@ -512,7 +493,7 @@ Creating, modifying, using the cpusets c
 virtual filesystem.

 To mount it, type:
-# mount -t cpuset none /dev/cpuset
+# mount -t container -o cpuset cpuset /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
@@ -555,6 +536,18 @@ To remove a cpuset, just use rmdir:
 This will fail if the cpuset is in use (has cpusets inside, or has
 processes attached).

+Note that for legacy reasons, the "cpuset" filesystem exists as a
+wrapper around the container filesystem.
+
+The command
+
+mount -t cpuset X /dev/cpuset
+
+is equivalent to
+
+mount -t container -ocpuset X /dev/cpuset
+echo "/sbin/cpuset_release_agent" > /dev/cpuset/release_agent
+
 2.2 Adding/removing cpus
 ------------------------

Index: container-2.6.22-rc2-mm1/include/linux/cpuset.h
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/include/linux/cpuset.h
+++ container-2.6.22-rc2-mm1/include/linux/cpuset.h
@@ -11,6 +11,7 @@
 #include <linux/sched.h>
 #include <linux/cpumask.h>
 #include <linux/nodemask.h>
+#include <linux/container.h>

 #ifdef CONFIG_CPUSETS

@@ -19,8 +20,6 @@ extern int number_of_cpusets;	/* How man
 extern int cpuset_init_early(void);
 extern int cpuset_init(void);
 extern void cpuset_init_smp(void);
```

```
-extern void cpuset_fork(struct task_struct *p);
-extern void cpuset_exit(struct task_struct *p);
 extern cpumask_t cpuset_cpus_allowed(struct task_struct *p);
 extern nodemask_t cpuset_mems_allowed(struct task_struct *p);
 #define cpuset_current_mems_allowed (current->mems_allowed)
@@ -75,13 +74,13 @@ static inline int cpuset_do_slab_mem_spr

 extern void cpuset_track_online_nodes(void);

+extern int current_cpuset_is_being_rebound(void);
+
 #else /* !CONFIG_CPUSETS */

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

 static inline cpumask_t cpuset_cpus_allowed(struct task_struct *p)
 {
@@ -146,6 +145,11 @@ 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-2.6.22-rc2-mm1/include/linux/mempolicy.h
 ===================================================================== =======
--- container-2.6.22-rc2-mm1.orig/include/linux/mempolicy.h
+++ container-2.6.22-rc2-mm1/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->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, gfp_t gfp_flags)
 {
Index: container-2.6.22-rc2-mm1/include/linux/sched.h
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/include/linux/sched.h
+++ container-2.6.22-rc2-mm1/include/linux/sched.h
@@ -782,8 +782,6 @@ static inline int above_background_load(
 }

 struct io_context;   /* See blkdev.h */
-struct cpuset;
-
 #define NGROUPS_SMALL  32
 #define NGROUPS_PER_BLOCK ((int)(PAGE_SIZE / sizeof(gid_t)))
 struct group_info {
@@ -1130,7 +1128,6 @@ 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;
Index: container-2.6.22-rc2-mm1/init/Kconfig
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/init/Kconfig
+++ container-2.6.22-rc2-mm1/init/Kconfig
@@ -309,6 +309,7 @@ config CONTAINERS
```

```
 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
@@ -344,6 +345,11 @@ config CONTAINER_CPUACCT
    Provides a simple Resource Controller for monitoring the
    total CPU consumed by the tasks in a container

+config PROC_PID_CPUSET
+ bool "Include legacy /proc/<pid>/cpuset file"
+ depends on CPUSETS
+ default y
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
Index: container-2.6.22-rc2-mm1/kernel/cpuset.c
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/kernel/cpuset.c
+++ container-2.6.22-rc2-mm1/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
@@ -53,8 +55,6 @@
 #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
```

```
@@ -62,6 +62,10 @@
  */
 int number_of_cpusets __read_mostly;

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

 struct fmeter {
@@ -72,24 +76,13 @@ 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 */

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

  /*
   * Copy of global cpuset_mems_generation as of the most
@@ -100,13 +93,32 @@ 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_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_id),
+       struct cpuset, css);
+}
+
+/* Retrieve the cpuset for a task */
+static inline struct cpuset *task_cs(struct task_struct *task)
+{
+ return container_of(task_subsys_state(task, cpuset_subsys_id),
+       struct cpuset, css);
+}
+
+
 /* bits in struct cpuset flags field */
 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;
@@ -122,16 +134,6 @@ 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);
@@ -172,14 +174,8 @@ 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
@@ -263,297 +259,31 @@ static struct super_block *cpuset_sb;
  * 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;
+ if (container_fs) {
+  ret = container_fs->get_sb(container_fs, flags,
+        unused_dev_name,
+        "cpuset", 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_path.dentry->d_parent->d_fsdata
- * - the 'cftype' of the file is file->f_path.dentry->d_fsdata
- */
-
-struct cftype {
- char *name;
- int private;
- int (*open) (struct inode *inode, struct file *file);
- ssize_t (*read) (struct file *file, char __user *buf, size_t nbytes,
-       loff_t *ppos);
- int (*write) (struct file *file, const char __user *buf, size_t nbytes,
-       loff_t *ppos);
- int (*release) (struct inode *inode, struct file *file);
-};
-
-static inline struct cpuset *__d_cs(struct dentry *dentry)
-{
- return dentry->d_fsdata;
```

```
-}
-
-static inline struct cftype *__d_cft(struct dentry *dentry)
-{
- return dentry->d_fsdata;
-}
-
-/*
- * Call with manage_mutex held.  Writes path of cpuset into buf.
- * Returns 0 on success, -errno on error.
- */
-
-static int cpuset_path(const struct cpuset *cs, char *buf, int buflen)
-{
- char *start;
-
- start = buf + buflen;
-
- *--start = '\0';
- for (;;) {
-  int len = cs->dentry->d_name.len;
-  if ((start -= len) < buf)
-   return -ENAMETOOLONG;
-  memcpy(start, cs->dentry->d_name.name, len);
-  cs = cs->parent;
-  if (!cs)
-   break;
-  if (!cs->parent)
-   continue;
-  if (--start < buf)
-   return -ENAMETOOLONG;
-  *start = '/';
- }
- memmove(buf, start, buf + buflen - start);
- return 0;
-}
-
-/*
- * Notify userspace when a cpuset is released, by running
- * /sbin/cpuset_release_agent with the name of the cpuset (path
- * relative to the root of cpuset file system) as the argument.
- *
- * Most likely, this user command will try to rmdir this cpuset.
- *
- * This races with the possibility that some other task will be
- * attached to this cpuset before it is removed, or that some other
- * user task will 'mkdir' a child cpuset of this cpuset.  That's ok.
- * The presumed 'rmdir' will fail quietly if this cpuset is no longer
```

```
- * unused, and this cpuset will be reprieved from its death sentence,
- * to continue to serve a useful existence.  Next time it's released,
- * we will get notified again, if it still has 'notify_on_release' set.
- *
- * The final arg to call_usermodehelper() is 0, which means don't
- * wait.  The separate /sbin/cpuset_release_agent task is forked by
- * call_usermodehelper(), then control in this thread returns here,
- * without waiting for the release agent task.  We don't bother to
- * wait because the caller of this routine has no use for the exit
- * status of the /sbin/cpuset_release_agent task, so no sense holding
- * our caller up for that.
- *
- * When we had only one cpuset mutex, we had to call this
- * without holding it, to avoid deadlock when call_usermodehelper()
- * allocated memory.  With two locks, we could now call this while
- * holding manage_mutex, but we still don't, so as to minimize
- * the time manage_mutex is held.
- */
-
-static void cpuset_release_agent(const char *pathbuf)
-{
- char *argv[3], *envp[3];
- int i;
-
- if (!pathbuf)
-  return;
-
- i = 0;
- argv[i++] = "/sbin/cpuset_release_agent";
- argv[i++] = (char *)pathbuf;
- argv[i] = NULL;
-
- i = 0;
- /* minimal command environment */
- envp[i++] = "HOME=/";
- envp[i++] = "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
- envp[i] = NULL;
-
- call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
- 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
@@ -651,20 +381,19 @@ void cpuset_update_task_memory_state(voi
  struct task_struct *tsk = current;
  struct cpuset *cs;

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

  if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
```

```
   mutex_lock(&callback_mutex);
   task_lock(tsk);
-  cs = tsk->cpuset; /* Maybe changed when task not locked */
+  cs = task_cs(tsk); /* Maybe changed when task not locked */
   guarantee_online_mems(cs, &tsk->mems_allowed);
   tsk->cpuset_mems_generation = cs->mems_generation;
   if (is_spread_page(cs))
@@ -719,11 +448,12 @@ 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) {
-  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;
  }

@@ -738,7 +468,8 @@ 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->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))
@@ -781,7 +512,8 @@ static int update_cpumask(struct cpuset
  }
  cpus_and(trialcs.cpus_allowed, trialcs.cpus_allowed, cpu_online_map);
  /* cpus_allowed cannot be empty for a cpuset with attached tasks. */
- if (atomic_read(&cs->count) && cpus_empty(trialcs.cpus_allowed))
+ if (container_task_count(cs->css.container) &&
+           cpus_empty(trialcs.cpus_allowed))
   return -ENOSPC;
  retval = validate_change(cs, &trialcs);
  if (retval < 0)
@@ -837,7 +569,7 @@ static void cpuset_migrate_mm(struct mm_
  do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);

  mutex_lock(&callback_mutex);
- guarantee_online_mems(tsk->cpuset, &tsk->mems_allowed);
+ guarantee_online_mems(task_cs(tsk),&tsk->mems_allowed);
```

```
  mutex_unlock(&callback_mutex);
 }

@@ -855,6 +587,8 @@ 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;
@@ -891,7 +625,8 @@ static int update_nodemask(struct cpuset
   goto done;
  }
  /* mems_allowed cannot be empty for a cpuset with attached tasks. */
- if (atomic_read(&cs->count) && nodes_empty(trialcs.mems_allowed)) {
+ if (container_task_count(cs->css.container) &&
+     nodes_empty(trialcs.mems_allowed)) {
   retval = -ENOSPC;
   goto done;
  }
@@ -904,7 +639,7 @@ static int update_nodemask(struct cpuset
  cs->mems_generation = cpuset_mems_generation++;
  mutex_unlock(&callback_mutex);

- set_cpuset_being_rebound(cs);  /* causes mpol_copy() rebind */
+ cpuset_being_rebound = cs;  /* causes mpol_copy() rebind */

  fudge = 10;    /* spare mmarray[] slots */
  fudge += cpus_weight(cs->cpus_allowed); /* imagine one fork-bomb/cpu */
@@ -918,15 +653,15 @@ static int update_nodemask(struct cpuset
   * enough mmarray[] w/o using GFP_ATOMIC.
   */
  while (1) {
-  ntasks = atomic_read(&cs->count); /* guess */
+  ntasks = container_task_count(cs->css.container);  /* 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)
+  read_lock(&tasklist_lock);  /* block fork */
+  if (__container_task_count(cs->css.container) <= ntasks)
    break;    /* got enough */
-  write_unlock_irq(&tasklist_lock); /* try again */
+  read_unlock(&tasklist_lock);  /* try again */
```

```
   kfree(mmarray);
  }

@@ -941,14 +676,14 @@ static int update_nodemask(struct cpuset
      "Cpuset mempolicy rebind incomplete.\n");
    continue;
   }
-  if (p->cpuset != cs)
+  if (task_cs(p) != cs)
    continue;
   mm = get_task_mm(p);
   if (!mm)
    continue;
   mmarray[n++] = mm;
  } while_each_thread(g, p);
- write_unlock_irq(&tasklist_lock);
+ read_unlock(&tasklist_lock);

  /*
   * Now that we've dropped the tasklist spinlock, we can
@@ -975,12 +710,17 @@ 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.
  */
@@ -1127,85 +867,34 @@ static int fmeter_getrate(struct fmeter
  return val;
 }

-/*
- * Attack task specified by pid in 'pidbuf' to cpuset 'cs', possibly
- * writing the path of the old cpuset in 'ppathbuf' if it needs to be
- * notified on release.
- *
```

```
- * Call holding manage_mutex.  May take callback_mutex and task_lock of
- * the task 'pid' during call.
- */
-
-static int attach_task(struct cpuset *cs, char *pidbuf, char **ppathbuf)
+int cpuset_can_attach(struct container_subsys *ss,
+        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 = container_cs(cont);

- 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);
- }
+ return security_task_setscheduler(tsk, 0, NULL);
+}

- retval = security_task_setscheduler(tsk, 0, NULL);
- if (retval) {
```

```
-  put_task_struct(tsk);
-  return retval;
- }
+void cpuset_attach(struct container_subsys *ss,
+      struct container *cont, struct container *oldcont,
+      struct task_struct *tsk)
+{
+ cpumask_t cpus;
+ nodemask_t from, to;
+ struct mm_struct *mm;
+ struct cpuset *cs = container_cs(cont);
+ struct cpuset *oldcs = container_cs(oldcont);

  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);
-
  guarantee_online_cpus(cs, &cpus);
  set_cpus_allowed(tsk, cpus);
+ mutex_unlock(&callback_mutex);

  from = oldcs->mems_allowed;
  to = cs->mems_allowed;
-
- mutex_unlock(&callback_mutex);
-
  mm = get_task_mm(tsk);
  if (mm) {
   mpol_rebind_mm(mm, &to);
@@ -1214,40 +903,31 @@ 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,
+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_path.dentry->d_parent);
- struct cftype *cft = __d_cft(file->f_path.dentry);
+ struct cpuset *cs = container_cs(cont);
  cpuset_filetype_t type = cft->private;
  char *buffer;
- char *pathbuf = NULL;
  int retval = 0;

  /* Crude upper limit on largest legitimate cpulist user might write. */
@@ -1264,9 +944,9 @@ static ssize_t cpuset_common_file_write(
  }
  buffer[nbytes] = 0; /* nul-terminate */

- mutex_lock(&manage_mutex);
+ container_lock();
```

```
- if (is_removed(cs)) {
+ if (container_is_removed(cont)) {
   retval = -ENODEV;
   goto out2;
  }
@@ -1284,9 +964,6 @@ 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;
@@ -1304,9 +981,6 @@ 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;
@@ -1315,30 +989,12 @@ static ssize_t cpuset_common_file_write(
  if (retval == 0)
   retval = nbytes;
 out2:
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
+ container_unlock();
 out1:
  kfree(buffer);
  return retval;
 }

-static ssize_t cpuset_file_write(struct file *file, const char __user *buf,
-      size_t nbytes, loff_t *ppos)
-{
- ssize_t retval = 0;
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (!cft)
-  return -ENODEV;
-
- /* special function ? */
- if (cft->write)
-  retval = cft->write(file, buf, nbytes, ppos);
```

```
- else
-  retval = cpuset_common_file_write(file, buf, nbytes, ppos);
-
- return retval;
-}
-
 /*
  * These ascii lists should be read in a single call, by using a user
  * buffer large enough to hold the entire map.  If read in smaller
@@ -1373,11 +1029,13 @@ static int cpuset_sprintf_memlist(char *
  return nodelist_scnprintf(page, PAGE_SIZE, mask);
 }

-static ssize_t cpuset_common_file_read(struct file *file, char __user *buf,
-    size_t nbytes, loff_t *ppos)
+static ssize_t cpuset_common_file_read(struct container *cont,
+           struct cftype *cft,
+           struct file *file,
+           char __user *buf,
+           size_t nbytes, loff_t *ppos)
 {
- struct cftype *cft = __d_cft(file->f_path.dentry);
- struct cpuset *cs = __d_cs(file->f_path.dentry->d_parent);
+ struct cpuset *cs = container_cs(cont);
  cpuset_filetype_t type = cft->private;
  char *page;
  ssize_t retval = 0;
@@ -1401,9 +1059,6 @@ 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;
@@ -1431,386 +1086,100 @@ out:
  return retval;
 }

-static ssize_t cpuset_file_read(struct file *file, char __user *buf, size_t nbytes,
-       loff_t *ppos)
-{
- ssize_t retval = 0;
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (!cft)
-  return -ENODEV;
```

```
-
- /* special function ? */
- if (cft->read)
-  retval = cft->read(file, buf, nbytes, ppos);
- else
-  retval = cpuset_common_file_read(file, buf, nbytes, ppos);

- return retval;
-}

-static int cpuset_file_open(struct inode *inode, struct file *file)
-{
- int err;
- struct cftype *cft;

- err = generic_file_open(inode, file);
- if (err)
-  return err;
-
- cft = __d_cft(file->f_path.dentry);
- if (!cft)
-  return -ENODEV;
- if (cft->open)
-  err = cft->open(inode, file);
- else
-  err = 0;
-
- return err;
-}
-
-static int cpuset_file_release(struct inode *inode, struct file *file)
-{
- struct cftype *cft = __d_cft(file->f_path.dentry);
- if (cft->release)
-  return cft->release(inode, file);
- return 0;
-}
-
-/*
- * cpuset_rename - Only allow simple rename of directories in place.
- */
-static int cpuset_rename(struct inode *old_dir, struct dentry *old_dentry,
-                         struct inode *new_dir, struct dentry *new_dentry)
-{
- if (!S_ISDIR(old_dentry->d_inode->i_mode))
-  return -ENOTDIR;
- if (new_dentry->d_inode)
-  return -EEXIST;
```

```
- if (old_dir != new_dir)
-  return -EIO;
- return simple_rename(old_dir, old_dentry, new_dir, new_dentry);
-}
-
-static const struct file_operations cpuset_file_operations = {
- .read = cpuset_file_read,
- .write = cpuset_file_write,
- .llseek = generic_file_llseek,
- .open = cpuset_file_open,
- .release = cpuset_file_release,
-};
-
-static const struct inode_operations cpuset_dir_inode_operations = {
- .lookup = simple_lookup,
- .mkdir = cpuset_mkdir,
- .rmdir = cpuset_rmdir,
- .rename = cpuset_rename,
-};
-
-static int cpuset_create_file(struct dentry *dentry, int mode)
-{
- struct inode *inode;
-
- if (!dentry)
-  return -ENOENT;
- if (dentry->d_inode)
-  return -EEXIST;
-
- inode = cpuset_new_inode(mode);
- if (!inode)
-  return -ENOMEM;
-
- if (S_ISDIR(mode)) {
-  inode->i_op = &cpuset_dir_inode_operations;
-  inode->i_fop = &simple_dir_operations;
-
-  /* start off with i_nlink == 2 (for "." entry) */
-  inc_nlink(inode);
- } else if (S_ISREG(mode)) {
-  inode->i_size = 0;
-  inode->i_fop = &cpuset_file_operations;
- }
-
- d_instantiate(dentry, inode);
- dget(dentry); /* Extra count - pin the dentry in core */
- return 0;
-}
```

```
-
-/*
- * cpuset_create_dir - create a directory for an object.
- * cs: the cpuset we create the directory for.
- *  It must have a valid ->parent field
- *  And we are going to fill its ->dentry field.
- * name: The name to give to the cpuset directory. Will be copied.
- * mode: mode to set on new directory.
- */
-
-static int cpuset_create_dir(struct cpuset *cs, const char *name, int mode)
-{
- struct dentry *dentry = NULL;
- struct dentry *parent;
- int error = 0;
-
- parent = cs->parent->dentry;
- dentry = cpuset_get_dentry(parent, name);
- if (IS_ERR(dentry))
-  return PTR_ERR(dentry);
- error = cpuset_create_file(dentry, S_IFDIR | mode);
- if (!error) {
-  dentry->d_fsdata = cs;
-  inc_nlink(parent->d_inode);
-  cs->dentry = dentry;
- }
- dput(dentry);
-
- return error;
-}
-
-static int cpuset_add_file(struct dentry *dir, const struct cftype *cft)
-{
- struct dentry *dentry;
- int error;
-
- mutex_lock(&dir->d_inode->i_mutex);
- dentry = cpuset_get_dentry(dir, cft->name);
- if (!IS_ERR(dentry)) {
-  error = cpuset_create_file(dentry, 0644 | S_IFREG);
-  if (!error)
-   dentry->d_fsdata = (void *)cft;
-  dput(dentry);
- } else
-  error = PTR_ERR(dentry);
- mutex_unlock(&dir->d_inode->i_mutex);
- return error;
-}
```

```
-
-/*
- * Stuff for reading the 'tasks' file.
- *
- * Reading this file can return large amounts of data if a cpuset has
- * *lots* of attached tasks. So it may need several calls to read(),
- * but we cannot guarantee that the information we produce is correct
- * unless we produce it entirely atomically.
- *
- * Upon tasks file open(), a struct ctr_struct is allocated, that
- * will have a pointer to an array (also allocated here).  The struct
- * ctr_struct * is stored in file->private_data.  Its resources will
- * be freed by release() when the file is closed.  The array is used
- * to sprintf the PIDs and then used by read().
- */
-
-/* cpusets_tasks_read array */
-
-struct ctr_struct {
- char *buf;
- int bufsz;
-};
-
-/*
- * Load into 'pidarray' up to 'npids' of the tasks using cpuset 'cs'.
- * Return actual number of pids loaded.  No need to task_lock(p)
- * when reading out p->cpuset, as we don't really care if it changes
- * on the next cycle, and we are not going to try to dereference it.
- */
-static int pid_array_load(pid_t *pidarray, int npids, struct cpuset *cs)
-{
- int n = 0;
- struct task_struct *g, *p;
-
- read_lock(&tasklist_lock);
-
- do_each_thread(g, p) {
-  if (p->cpuset == cs) {
-   pidarray[n++] = p->pid;
-   if (unlikely(n == npids))
-    goto array_full;
-  }
- } while_each_thread(g, p);
-
-array_full:
- read_unlock(&tasklist_lock);
- return n;
-}
```

```
-
-static int cmppid(const void *a, const void *b)
-{
- return *(pid_t *)a - *(pid_t *)b;
-}
-
-/*
- * Convert array 'a' of 'npids' pid_t's to a string of newline separated
- * decimal pids in 'buf'.  Don't write more than 'sz' chars, but return
- * count 'cnt' of how many chars would be written if buf were large enough.
- */
-static int pid_array_to_buf(char *buf, int sz, pid_t *a, int npids)
-{
- int cnt = 0;
- int i;
-
- for (i = 0; i < npids; i++)
-  cnt += snprintf(buf + cnt, max(sz - cnt, 0), "%d\n", a[i]);
- return cnt;
-}
-
-/*
- * Handle an open on 'tasks' file.  Prepare a buffer listing the
- * process id's of tasks currently attached to the cpuset being opened.
- *
- * Does not require any specific cpuset mutexes, and does not take any.
- */
-static int cpuset_tasks_open(struct inode *unused, struct file *file)
-{
- struct cpuset *cs = __d_cs(file->f_path.dentry->d_parent);
- struct ctr_struct *ctr;
- pid_t *pidarray;
- int npids;
- char c;
-
- if (!(file->f_mode & FMODE_READ))
-  return 0;
-
- ctr = kmalloc(sizeof(*ctr), GFP_KERNEL);
- if (!ctr)
-  goto err0;
-
- /*
-  * If cpuset gets more users after we read count, we won't have
-  * enough space - tough.  This race is indistinguishable to the
-  * caller from the case that the additional cpuset users didn't
-  * show up until sometime later on.
-  */
```

```
- npids = atomic_read(&cs->count);
- pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
- if (!pidarray)
-  goto err1;
-
- npids = pid_array_load(pidarray, npids, cs);
- sort(pidarray, npids, sizeof(pid_t), cmppid, NULL);
-
- /* Call pid_array_to_buf() twice, first just to get bufsz */
- ctr->bufsz = pid_array_to_buf(&c, sizeof(c), pidarray, npids) + 1;
- ctr->buf = kmalloc(ctr->bufsz, GFP_KERNEL);
- if (!ctr->buf)
-  goto err2;
- ctr->bufsz = pid_array_to_buf(ctr->buf, ctr->bufsz, pidarray, npids);
-
- kfree(pidarray);
- file->private_data = ctr;
- return 0;
-
-err2:
- kfree(pidarray);
-err1:
- kfree(ctr);
-err0:
- return -ENOMEM;
-}
-
-static ssize_t cpuset_tasks_read(struct file *file, char __user *buf,
-      size_t nbytes, loff_t *ppos)
-{
- struct ctr_struct *ctr = file->private_data;
-
- return simple_read_from_buffer(buf, nbytes, ppos, ctr->buf, ctr->bufsz);
-}
-
-static int cpuset_tasks_release(struct inode *unused_inode, struct file *file)
-{
- struct ctr_struct *ctr;
-
- if (file->f_mode & FMODE_READ) {
-  ctr = file->private_data;
-  kfree(ctr->buf);
-  kfree(ctr);
- }
- return 0;
-}

 /*
```

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

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

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

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

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

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

```
  .private = FILE_MEMORY_MIGRATE,
 };

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

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

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

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

-static int cpuset_populate_dir(struct dentry *cs_dentry)
+int cpuset_populate(struct container_subsys *ss, 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)
-  return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_cpu_exclusive)) < 0)
+ if ((err = container_add_file(cont, &cft_cpus)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_mem_exclusive)) < 0)
+ if ((err = container_add_file(cont, &cft_mems)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_notify_on_release)) < 0)
+ if ((err = container_add_file(cont, &cft_cpu_exclusive)) < 0)
```

```
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_memory_migrate)) < 0)
+ if ((err = container_add_file(cont, &cft_mem_exclusive)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_memory_pressure)) < 0)
+ if ((err = container_add_file(cont, &cft_memory_migrate)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_spread_page)) < 0)
+ if ((err = container_add_file(cont, &cft_memory_pressure)) < 0)
   return err;
- if ((err = cpuset_add_file(cs_dentry, &cft_spread_slab)) < 0)
+ if ((err = container_add_file(cont, &cft_spread_page)) < 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;
 }

@@ -1823,106 +1192,61 @@ 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_subsys *ss, struct container *cont)
 {
  struct cpuset *cs;
- int err;
+ 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;

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

-static int cpuset_rmdir(struct inode *unused_dir, struct dentry *dentry)
-{
- struct cpuset *cs = dentry->d_fsdata;
- struct dentry *d;
- struct cpuset *parent;
- char *pathbuf = NULL;
-
- /* the vfs holds both inode->i_mutex already */
-
- mutex_lock(&manage_mutex);
  cpuset_update_task_memory_state();
- if (atomic_read(&cs->count) > 0) {
-  mutex_unlock(&manage_mutex);
-  return -EBUSY;
- }
- if (!list_empty(&cs->children)) {
-  mutex_unlock(&manage_mutex);
-  return -EBUSY;
- }
- parent = cs->parent;
- mutex_lock(&callback_mutex);
- set_bit(CS_REMOVED, &cs->flags);
- list_del(&cs->sibling); /* delete my sibling from parent->children */
- spin_lock(&cs->dentry->d_lock);
- d = dget(cs->dentry);
- cs->dentry = NULL;
- spin_unlock(&d->d_lock);
- cpuset_d_remove_dir(d);
- dput(d);
  number_of_cpusets--;
- mutex_unlock(&callback_mutex);
- if (list_empty(&parent->children))
-  check_for_release(parent, &pathbuf);
- mutex_unlock(&manage_mutex);
- cpuset_release_agent(pathbuf);
- return 0;
+ kfree(cs);
 }

+struct container_subsys cpuset_subsys = {
+ .name = "cpuset",
+ .create = cpuset_create,
```

```
+ .destroy  = cpuset_destroy,
+ .can_attach = cpuset_can_attach,
+ .attach = cpuset_attach,
+ .populate = cpuset_populate,
+ .subsys_id = cpuset_subsys_id,
+ .early_init = 1,
+};
+
 /*
  * cpuset_init_early - just enough so that the calls to
  * cpuset_update_task_memory_state() in early init code
@@ -1931,13 +1255,11 @@ 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++;
+ top_cpuset.mems_generation = cpuset_mems_generation++;
  return 0;
 }

+
 /**
  * cpuset_init - initialize cpusets at system boot
  *
@@ -1946,8 +1268,7 @@ 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;
@@ -1955,30 +1276,12 @@ int __init cpuset_init(void)
  fmeter_init(&top_cpuset.fmeter);
  top_cpuset.mems_generation = cpuset_mems_generation++;

- init_task.cpuset = &top_cpuset;
-
  err = register_filesystem(&cpuset_fs_type);
  if (err < 0)
-  goto out;
- cpuset_mount = kern_mount(&cpuset_fs_type);
- if (IS_ERR(cpuset_mount)) {
```

```
-  printk(KERN_ERR "cpuset: could not mount!\n");
-  err = PTR_ERR(cpuset_mount);
-  cpuset_mount = NULL;
-  goto out;
- }
- root = cpuset_mount->mnt_sb->s_root;
- root->d_fsdata = &top_cpuset;
- inc_nlink(root->d_inode);
- top_cpuset.dentry = root;
- root->d_inode->i_op = &cpuset_dir_inode_operations;
+  return err;
+
  number_of_cpusets = 1;
- err = cpuset_populate_dir(root);
- /* memory_pressure_enabled is in root cpuset only */
- if (err == 0)
-  err = cpuset_add_file(root, &cft_memory_pressure_enabled);
-out:
- return err;
+ return 0;
 }

 /*
@@ -2004,10 +1307,12 @@ 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(cont, &cur->css.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);
@@ -2034,7 +1339,7 @@ static void guarantee_online_cpus_mems_i

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

  guarantee_online_cpus_mems_in_subtree(&top_cpuset);
@@ -2042,7 +1347,7 @@ static void common_cpu_mem_hotplug_unplu
  top_cpuset.mems_allowed = node_online_map;
```

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

 /*
@@ -2090,109 +1395,7 @@ 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().
- *
- * 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;
-
- task_lock(current);
```

```
- cs = tsk->cpuset;
- tsk->cpuset = &top_cpuset; /* the_top_cpuset_hack - see above */
- task_unlock(current);
-
- 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.
  *
@@ -2208,7 +1411,7 @@ cpumask_t cpuset_cpus_allowed(struct tas

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

@@ -2236,7 +1439,7 @@ nodemask_t cpuset_mems_allowed(struct ta

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

@@ -2367,7 +1570,7 @@ int __cpuset_zone_allowed_softwall(struc
  mutex_lock(&callback_mutex);

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

  allowed = node_isset(node, cs->mems_allowed);
```

```
@@ -2504,7 +1707,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock(current);
   goto done;
  }
- cs1 = nearest_exclusive_ancestor(current->cpuset);
+ cs1 = nearest_exclusive_ancestor(task_cs(current));
  task_unlock(current);

  task_lock((struct task_struct *)p);
@@ -2512,7 +1715,7 @@ int cpuset_excl_nodes_overlap(const stru
   task_unlock((struct task_struct *)p);
   goto done;
  }
- cs2 = nearest_exclusive_ancestor(p->cpuset);
+ cs2 = nearest_exclusive_ancestor(task_cs((struct task_struct *)p));
  task_unlock((struct task_struct *)p);

  overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
@@ -2548,14 +1751,12 @@ int cpuset_memory_pressure_enabled __rea

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

+#ifdef CONFIG_PROC_PID_CPUSET
 /*
  * proc_cpuset_show()
  *  - Print tasks cpuset path into seq_file.
@@ -2572,6 +1773,7 @@ static int proc_cpuset_show(struct seq_f
  struct pid *pid;
  struct task_struct *tsk;
  char *buf;
+ struct container_subsys_state *css;
  int retval;

  retval = -ENOMEM;
@@ -2586,15 +1788,15 @@ static int proc_cpuset_show(struct seq_f
   goto out_free;

  retval = -EINVAL;
- mutex_lock(&manage_mutex);
```

```
-
- retval = cpuset_path(tsk->cpuset, buf, PAGE_SIZE);
+ container_lock();
+ css = task_subsys_state(tsk, cpuset_subsys_id);
+ retval = container_path(css->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_unlock();
  put_task_struct(tsk);
 out_free:
  kfree(buf);
@@ -2614,6 +1816,7 @@ const struct file_operations proc_cpuset
  .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-2.6.22-rc2-mm1/kernel/exit.c
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/kernel/exit.c
+++ container-2.6.22-rc2-mm1/kernel/exit.c
@@ -31,7 +31,6 @@
 #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>
@@ -935,7 +934,6 @@ fastcall void do_exit(long code)
  __exit_files(tsk);
  __exit_fs(tsk);
  exit_thread();
- cpuset_exit(tsk);
  container_exit(tsk, 1);
  exit_keys(tsk);

Index: container-2.6.22-rc2-mm1/kernel/fork.c
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/kernel/fork.c
+++ container-2.6.22-rc2-mm1/kernel/fork.c
@@ -29,7 +29,6 @@
```

```
 #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>
@@ -1064,7 +1063,6 @@ static struct task_struct *copy_process(
  p->io_context = NULL;
  p->io_wait = NULL;
  p->audit_context = NULL;
- cpuset_fork(p);
  container_fork(p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
@@ -1311,7 +1309,6 @@ bad_fork_cleanup_policy:
  mpol_free(p->mempolicy);
 bad_fork_cleanup_container:
 #endif
- cpuset_exit(p);
  container_exit(p, container_callbacks_done);
  delayacct_tsk_free(p);
  if (p->binfmt)
Index: container-2.6.22-rc2-mm1/mm/mempolicy.c
 ===================================================================== =======
--- container-2.6.22-rc2-mm1.orig/mm/mempolicy.c
+++ container-2.6.22-rc2-mm1/mm/mempolicy.c
@@ -1311,7 +1311,6 @@ 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)
@@ -1910,4 +1909,3 @@ out:
   m->version = (vma != priv->tail_vma) ? vma->vm_start : 0;
  return 0;
 }
-
Index: container-2.6.22-rc2-mm1/fs/proc/base.c
 ===================================================================== =======
--- container-2.6.22-rc2-mm1.orig/fs/proc/base.c
+++ container-2.6.22-rc2-mm1/fs/proc/base.c
@@ -2026,7 +2026,7 @@ static const struct pid_entry tgid_base_
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat",  S_IRUGO, pid_schedstat),
 #endif
```

```
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",     S_IRUGO, cpuset),
 #endif
 #ifdef CONFIG_CONTAINERS
@@ -2320,7 +2320,7 @@ static const struct pid_entry tid_base_s
 #ifdef CONFIG_SCHEDSTATS
  INF("schedstat", S_IRUGO, pid_schedstat),
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_PROC_PID_CPUSET
  REG("cpuset",    S_IRUGO, cpuset),
 #endif
 #ifdef CONFIG_CONTAINERS
Index: container-2.6.22-rc2-mm1/include/linux/container_subsys.h
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/include/linux/container_subsys .h
+++ container-2.6.22-rc2-mm1/include/linux/container_subsys.h
@@ -13,4 +13,10 @@ SUBSYS(cpuacct)

 /* */

+#ifdef CONFIG_CPUSETS
+SUBSYS(cpuset)
+#endif
+
+/* */
+
 /* */

--
```

Subject: [PATCH 09/10] Containers(V10): Simple debug info subsystem
Posted by Paul Menage on Tue, 29 May 2007 13:01:13 GMT
View Forum Message <> Reply to Message

This example subsystem exports debugging information as an aid to
diagnosing refcount leaks, etc, in the container framework.

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

```
---
 include/linux/container_subsys.h |    4 +
 init/Kconfig                     |   10 ++++
 kernel/Makefile                  |    1
 kernel/container_debug.c         |   89 +++++++++++++++++++++++++++++++++++++++
 4 files changed, 104 insertions(+)
```

```
Index: container-2.6.22-rc2-mm1/include/linux/container_subsys.h
 ===================================================== =======
---  container-2.6.22-rc2-mm1.orig/include/linux/container_subsys .h
+++ container-2.6.22-rc2-mm1/include/linux/container_subsys.h
@@ -19,4 +19,8 @@ SUBSYS(cpuset)

 /* */

+#ifdef CONFIG_CONTAINER_DEBUG
+SUBSYS(debug)
+#endif
+
 /* */
Index: container-2.6.22-rc2-mm1/init/Kconfig
 ===================================================== =======
--- container-2.6.22-rc2-mm1.orig/init/Kconfig
+++ container-2.6.22-rc2-mm1/init/Kconfig
@@ -306,6 +306,16 @@ config LOG_BUF_SHIFT
 config CONTAINERS
  bool

+config CONTAINER_DEBUG
+ bool "Example debug container subsystem"
+ select CONTAINERS
+ help
+   This option enables a simple container subsystem that
+   exports useful debugging information about the containers
+   framework
+
+   Say N if unsure
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
Index: container-2.6.22-rc2-mm1/kernel/container_debug.c
 ===================================================== =======
--- /dev/null
+++ container-2.6.22-rc2-mm1/kernel/container_debug.c
@@ -0,0 +1,89 @@
+/*
+ * kernel/ccontainer_debug.c - Example container subsystem that
+ * exposes debug info
+ *
+ * Copyright (C) Google Inc, 2007
+ *
+ * Developed by Paul Menage (menage@google.com)
+ *
```

```
+ */
+
+#include <linux/container.h>
+#include <linux/fs.h>
+
+static int debug_create(struct container_subsys *ss, struct container *cont)
+{
+ struct container_subsys_state *css = kzalloc(sizeof(*css), GFP_KERNEL);
+ if (!css)
+  return -ENOMEM;
+ cont->subsys[debug_subsys_id] = css;
+ return 0;
+}
+
+static void debug_destroy(struct container_subsys *ss, struct container *cont)
+{
+ kfree(cont->subsys[debug_subsys_id]);
+}
+
+static u64 container_refcount_read(struct container *cont, struct cftype *cft)
+{
+ return atomic_read(&cont->count);
+}
+
+static u64 taskcount_read(struct container *cont, struct cftype *cft)
+{
+ u64 count;
+ container_lock();
+ count = container_task_count(cont);
+ container_unlock();
+ return count;
+}
+
+static u64 current_css_group_read(struct container *cont, struct cftype *cft)
+{
+ return (u64) current->containers;
+}
+
+static u64 current_css_group_refcount_read(struct container *cont,
+        struct cftype *cft)
+{
+ u64 count;
+ rcu_read_lock();
+ count = atomic_read(&current->containers->ref.refcount);
+ rcu_read_unlock();
+ return count;
+}
+
```

```
+static struct cftype files[] =  {
+ {
+  .name = "debug.container_refcount",
+  .read_uint = container_refcount_read,
+ },
+ {
+  .name = "debug.taskcount",
+  .read_uint = taskcount_read,
+ },
+
+ {
+  .name = "debug.current_css_group",
+  .read_uint = current_css_group_read,
+ },
+
+ {
+  .name = "debug.current_css_group_refcount",
+  .read_uint = current_css_group_refcount_read,
+ },
+};
+
+static int debug_populate(struct container_subsys *ss, struct container *cont)
+{
+ return container_add_files(cont, files, ARRAY_SIZE(files));
+}
+
+struct container_subsys debug_subsys = {
+ .name = "debug",
+ .create = debug_create,
+ .destroy = debug_destroy,
+ .populate = debug_populate,
+ .subsys_id = debug_subsys_id,
+};
Index: container-2.6.22-rc2-mm1/kernel/Makefile
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/kernel/Makefile
+++ container-2.6.22-rc2-mm1/kernel/Makefile
@@ -37,6 +37,7 @@ obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CONTAINERS) += container.o
+obj-$(CONFIG_CONTAINER_DEBUG) += container_debug.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o

--
```

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Andrew Morton on Wed, 30 May 2007 07:14:55 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2007 06:01:04 -0700 menage@google.com wrote:

> This is an update to my multi-hierarchy multi-subsystem generic
> process containers patch.
>
> ...
>
> Still TODO:
>
> ...
>
> - lots more testing
>

So how do we do this?

Is there any sneaky way in which we can modify the kernel so that this new code gets exercised more? Obviously, tossing init into some default system-wide container would be a start. But I wonder if we can be sneakier - for example, create a new container on each setuid(), toss the task into that. Or something along those lines?

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by William Lee Irwin III on Wed, 30 May 2007 07:39:59 GMT
View Forum Message <> Reply to Message

On Wed, May 30, 2007 at 12:14:55AM -0700, Andrew Morton wrote:
> So how do we do this?
> Is there any sneaky way in which we can modify the kernel so that this new
> code gets exercised more? Obviously, tossing init into some default
> system-wide container would be a start. But I wonder if we can be
> sneakier - for example, create a new container on each setuid(), toss the
> task into that. Or something along those lines?

How about a container for each thread group, pgrp, session, and user?

-- wli

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Balbir Singh on Wed, 30 May 2007 08:09:37 GMT

View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Tue, 29 May 2007 06:01:04 -0700 menage@google.com wrote:
>
>> This is an update to my multi-hierarchy multi-subsystem generic
>> process containers patch.
>>
>> ...
>>
>> Still TODO:
>>
>> ...
>>
>> - lots more testing
>>
>
> So how do we do this?
>
> Is there any sneaky way in which we can modify the kernel so that this new
> code gets exercised more? Obviously, tossing init into some default
> system-wide container would be a start. But I wonder if we can be
> sneakier - for example, create a new container on each setuid(), toss the
> task into that. Or something along those lines?

Please, lets get the RSS controller in. It's ready, been tested
and commented on widely.

I'll also start porting my containerstats patches on top of -v10.

--
 Thanks,
 Balbir Singh
 Linux Technology Center
 IBM, ISTL

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Pavel Emelianov on Wed, 30 May 2007 08:58:29 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:
> Andrew Morton wrote:
>> On Tue, 29 May 2007 06:01:04 -0700 menage@google.com wrote:
>>
>>> This is an update to my multi-hierarchy multi-subsystem generic
>>> process containers patch.
>>>

>>> ...
>>>
>>> Still TODO:
>>>
>>> ...
>>>
>>> - lots more testing
>>>
>> So how do we do this?
>>
>> Is there any sneaky way in which we can modify the kernel so that this new
>> code gets exercised more? Obviously, tossing init into some default
>> system-wide container would be a start. But I wonder if we can be
>> sneakier - for example, create a new container on each setuid(), toss the
>> task into that. Or something along those lines?
>
> Please, lets get the RSS controller in. It's ready, been tested

It is not 100% ready yet actually :) I am working on it right now
and hope to get ready till tomorrow.

> and commented on widely.

Yup :) Balbir, thanks for testing, your patches are already in.

> I'll also start porting my containerstats patches on top of -v10.
>

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Balbir Singh on Wed, 30 May 2007 09:02:47 GMT
View Forum Message <> Reply to Message

Pavel Emelianov wrote:
>>> Is there any sneaky way in which we can modify the kernel so that this new
>>> code gets exercised more? Obviously, tossing init into some default
>>> system-wide container would be a start. But I wonder if we can be
>>> sneakier - for example, create a new container on each setuid(), toss the
>>> task into that. Or something along those lines?
>> Please, lets get the RSS controller in. It's ready, been tested
>
> It is not 100% ready yet actually :) I am working on it right now
> and hope to get ready till tomorrow.
>

By ready, I meant ready for inclusion as a concept/approach.

>> and commented on widely.

>
> Yup :) Balbir, thanks for testing, your patches are already in.
>

Thanks for including them.

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

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Pavel Emelianov on Wed, 30 May 2007 10:44:24 GMT
View Forum Message <> Reply to Message

Hi Paul.

I have faced a warning during testing your patches.
The testcase is simple:

```
  # ssh to the node
  mount -t container none /cnt/rss/ -o rss
  mkdir /cnt/rss/0
  /bin/echo $$ > /cnt/rss/0/tasks
  # exit with ^d and ssh again
  rmdir /cnt/rss/0
  dmesg
```

```
BUG: at mm/slab.c:777 __find_general_cachep()
 [<c04656c8>] __kmalloc+0x3f/0xa5
 [<c0440e3a>] container_tasks_open+0x56/0x11f
 [<c0440bcc>] container_file_open+0x0/0x36
 [<c0440bfb>] container_file_open+0x2f/0x36
 [<c0467a12>] __dentry_open+0xc1/0x178
 [<c0467b43>] nameidata_to_filp+0x24/0x33
 [<c0467b84>] do_filp_open+0x32/0x39
 [<c04678eb>] get_unused_fd+0x50/0xb6
 [<c0467bcd>] do_sys_open+0x42/0xbe
 [<c0467c82>] sys_open+0x1c/0x1e
 [<c0404c12>] sysenter_past_esp+0x5f/0x85
 [<c05b0000>] __xfrm_policy_check+0x11a/0x4f6
```

The bug seems to be here:

```
static int container_tasks_open(struct inode *unused, struct file *file)
{
        ...
        npids = container_task_count(cont);
```

```
        pidarray = kmalloc(npids * sizeof(pid_t), GFP_KERNEL);
        if (!pidarray)
                goto err1;
        ...
}
```
The npids happened to be 0 and kmalloc warns that size is zero.

Thanks,
Pavel.

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Mon, 04 Jun 2007 19:14:12 GMT
View Forum Message <> Reply to Message

Hi Paul,

I've got two problems working with this patchset:

1. A task can't join a cpuset unless 'cpus' and 'mems' are set. These
don't seem to automatically inherit the parent's values. So when I do

```
 mount -t container -o ns,cpuset nsproxy /containers
 (unshare a namespace)
```

the unshare fails because container_clone() created a new cpuset
container but the task couldn't automatically enter that new cpuset.

2. I can't delete containers because of the files they contain, and
am not allowed to delete those files by hand.

thanks,
-serge

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Mon, 04 Jun 2007 19:31:51 GMT
View Forum Message <> Reply to Message

What you describe, Serge, sounds like semantics carried over from cpusets.

Serge wrote:
> A task can't join a cpuset unless 'cpus' and 'mems' are set.

Yup - don't want to run a task in a cpuset that lacks cpu, or lacks
memory. Hard to run without those.

> These don't seem to automatically inherit the parent's values

Yup - early in the life of cpusets, a created cpuset inherited the cpus
and mems of its parent. But that broke the exclusive property big
time. You will recall that a cpu_exclusive or mem_exclusive cpuset
cannot overlap the cpus or memory, respectively, of any of its sibling
cpusets.

So we changed it to creating new cpusets empty of cpus or memory.

--

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

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Menage on Mon, 04 Jun 2007 20:30:41 GMT
View Forum Message <> Reply to Message

On 6/4/07, Paul Jackson <pj@sgi.com> wrote:
>
> Yup - early in the life of cpusets, a created cpuset inherited the cpus
> and mems of its parent. But that broke the exclusive property big
> time. You will recall that a cpu_exclusive or mem_exclusive cpuset
> cannot overlap the cpus or memory, respectively, of any of its sibling
> cpusets.
>

Maybe we could make it a per-cpuset option whether children should
inherit mems/cpus or not?

Paul

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Menage on Mon, 04 Jun 2007 20:32:06 GMT
View Forum Message <> Reply to Message

On 6/4/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
>
> 2. I can't delete containers because of the files they contain, and
> am not allowed to delete those files by hand.
>

You should be able to delete a container with rmdir as long as it's
not in use - its control files will get cleaned up automatically.

If you're getting an EBUSY error that means that either there are
still tasks running in the container (look in the "tasks" file) or
else there's a reference counting bug somewhere.

Can you post an example to reproduce the problem?

Paul

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Mon, 04 Jun 2007 20:37:29 GMT
View Forum Message <> Reply to Message

Paul M wrote:
> Maybe we could make it a per-cpuset option whether children should
> inherit mems/cpus or not?

I suppose, if those needing inherited mems/cpus need it bad enough.

--

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

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Mon, 04 Jun 2007 20:41:31 GMT
View Forum Message <> Reply to Message

Quoting Paul Menage (menage@google.com):
> On 6/4/07, Paul Jackson <pj@sgi.com> wrote:
> >
> >Yup - early in the life of cpusets, a created cpuset inherited the cpus
> >and mems of its parent. But that broke the exclusive property big
> >time. You will recall that a cpu_exclusive or mem_exclusive cpuset
> >cannot overlap the cpus or memory, respectively, of any of its sibling
> >cpusets.
> >
>
> Maybe we could make it a per-cpuset option whether children should
> inherit mems/cpus or not?

The values can be changed after the cpuset is populated, right? So
really these are just defaults? Would it then make sense to just
default to (parent_set - sibling_exclusive_set) for a new sibling's
value?

An option is fine with me, but without such an option at all, cpusets
could not be applied to namespaces...

thanks,
-serge

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Mon, 04 Jun 2007 20:51:06 GMT
View Forum Message <> Reply to Message

Quoting Paul Menage (menage@google.com):
> On 6/4/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
> >
> >2. I can't delete containers because of the files they contain, and
> >am not allowed to delete those files by hand.
> >
>
> You should be able to delete a container with rmdir as long as it's
> not in use - its control files will get cleaned up automatically.
>
> If you're getting an EBUSY error that means that either there are
> still tasks running in the container (look in the "tasks" file) or
> else there's a reference counting bug somewhere.
>
> Can you post an example to reproduce the problem?

here is an excerpt:

```
[root@linuz11 root]# mount -t container -ocpuset cpuset /containers/
[root@linuz11 root]# ls /containers/
cpu_exclusive   memory_pressure          mems              tasks
cpus            memory_pressure_enabled  notify_on_release
mem_exclusive   memory_spread_page       releasable
memory_migrate  memory_spread_slab       release_agent
[root@linuz11 root]# mkdir /containers/1
[root@linuz11 root]# echo 0 > /containers/1/mems
[root@linuz11 root]# echo 0 > /containers/1/cpus
[root@linuz11 root]# sh
sh-2.05b# echo $$ > /containers/1/tasks
sh-2.05b# cat /containers/1/tasks
4325
4326
sh-2.05b# exit
[root@linuz11 root]# ls /containers/1/tasks
/containers/1/tasks
[root@linuz11 root]# rm -rf /containers/1
```

rm: cannot remove `/containers/1/memory_spread_slab': Operation not
permitted
rm: cannot remove `/containers/1/memory_spread_page': Operation not
permitted
rm: cannot remove `/containers/1/memory_pressure': Operation not
permitted
rm: cannot remove `/containers/1/memory_migrate': Operation not
permitted
rm: cannot remove `/containers/1/mem_exclusive': Operation not permitted
rm: cannot remove `/containers/1/cpu_exclusive': Operation not permitted
rm: cannot remove `/containers/1/mems': Operation not permitted
rm: cannot remove `/containers/1/cpus': Operation not permitted
rm: cannot remove `/containers/1/releasable': Operation not permitted
rm: cannot remove `/containers/1/notify_on_release': Operation not
permitted
rm: cannot remove `/containers/1/tasks': Operation not permitted

Ah, I see the second time I typed 'ls /containers/1/tasks' instead of
cat. When I then used cat, the file was empty, and I got an oops just
like Pavel reported. I bet if I solve the problem he reported, then I
solve my problem :)

thanks,
-serge

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Menage on Mon, 04 Jun 2007 20:56:12 GMT
View Forum Message <> Reply to Message

On 6/4/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
> root@linuz11 root]# rm -rf /containers/1

Just use "rmdir /containers/1" here.

>
> Ah, I see the second time I typed 'ls /containers/1/tasks' instead of
> cat. When I then used cat, the file was empty, and I got an oops just
> like Pavel reported. I bet if I solve the problem he reported, then I
> solve my problem :)
>

As far as I could see, Pavel's problem wasn't actually an Oops, it was
a WARN_ON() when allocating a zero length chunk of memory. There's
ongoing discussion as to whether this counts as a problem with the
allocators or the kmalloc() code, since it used to be OK to allocate a
zero-length chunk.

Paul

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Mon, 04 Jun 2007 21:05:33 GMT
View Forum Message <> Reply to Message

> Would it then make sense to just
> default to (parent_set - sibling_exclusive_set) for a new sibling's
> value?

Which could well be empty, which in turn puts one back in the position
of dealing with a newborn cpuset that is empty (of cpus or of memory),
or else it introduces a new and odd constraint on when cpusets can be
created (only when there are non-exclusive cpus and mems available.)

> An option is fine with me, but without such an option at all, cpusets
> could not be applied to namespaces...

I wasn't paying close enough attention to understand why you couldn't
do it in two steps - make the container, and then populate it with
resources.

But if indeed that's not possible, then I guess we need some sort of
option specifying whether to create kids empty, or inheriting.

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

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Mon, 04 Jun 2007 21:11:03 GMT
View Forum Message <> Reply to Message

Quoting Paul Menage (menage@google.com):
> On 6/4/07, Serge E. Hallyn <serue@us.ibm.com> wrote:
> >root@linuz11 root]# rm -rf /containers/1
>
> Just use "rmdir /containers/1" here.

Hmm. Ok, that works... Odd, I thought rm -rf used to work in the past,
but i'm likely wrong.

thanks,
-serge

> >Ah, I see the second time I typed 'ls /containers/1/tasks' instead of
> >cat. When I then used cat, the file was empty, and I got an oops just
> >like Pavel reported. I bet if I solve the problem he reported, then I
> >solve my problem :)
> >
>
> As far as I could see, Pavel's problem wasn't actually an Oops, it was
> a WARN_ON() when allocating a zero length chunk of memory. There's
> ongoing discussion as to whether this counts as a problem with the
> allocators or the kmalloc() code, since it used to be OK to allocate a
> zero-length chunk.
>
> Paul

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Mon, 04 Jun 2007 21:16:17 GMT
View Forum Message <> Reply to Message

Serge wrote:
> Odd, I thought rm -rf used to work in the past,
> but i'm likely wrong.

I'm pretty sure it never worked.

And I've probably tested it myself, every few months,
since the birth of cpusets, when I forget and type it
again, and then stare dumbly at the screen wondering
what all the complaining is about.

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

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Wed, 06 Jun 2007 22:39:52 GMT
View Forum Message <> Reply to Message

Quoting Paul Jackson (pj@sgi.com):
> > Would it then make sense to just
> > default to (parent_set - sibling_exclusive_set) for a new sibling's
> > value?
>
> Which could well be empty, which in turn puts one back in the position

> of dealing with a newborn cpuset that is empty (of cpus or of memory),
> or else it introduces a new and odd constraint on when cpusets can be
> created (only when there are non-exclusive cpus and mems available.)
>
> > An option is fine with me, but without such an option at all, cpusets
> > could not be applied to namespaces...
>
> I wasn't paying close enough attention to understand why you couldn't
> do it in two steps - make the container, and then populate it with
> resources.

Sorry, please clarify - are you saying that now you do understand, or
that I should explain?

> But if indeed that's not possible, then I guess we need some sort of
> option specifying whether to create kids empty, or inheriting.

Paul (uh, Menage :) should I do a patch for this or have you got it
already?

thanks,
-serge

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Wed, 06 Jun 2007 22:43:47 GMT
View Forum Message <> Reply to Message

> > I wasn't paying close enough attention to understand why you couldn't
> > do it in two steps - make the container, and then populate it with
> > resources.
>
> Sorry, please clarify - are you saying that now you do understand, or
> that I should explain?

Could you explain -- I still don't understand why you need this option.
I still don't understand why you can't do it in two steps - make the
container, then add cpu/mem separately.

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

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Thu, 07 Jun 2007 00:05:59 GMT

Quoting Paul Jackson (pj@sgi.com):
> > > I wasn't paying close enough attention to understand why you couldn't
> > > do it in two steps - make the container, and then populate it with
> > > resources.
> >
> > Sorry, please clarify - are you saying that now you do understand, or
> > that I should explain?
>
> Could you explain -- I still don't understand why you need this option.
> I still don't understand why you can't do it in two steps - make the
> container, then add cpu/mem separately.

Sure - the key is that the ns subsystem uses container_clone() to
automatically create a new container (on sys_unshare() or clone(2)
with certain flags) and move the current task into it. Let's say
we have done

mount -t container -o ns,cpuset nsproxy /containers

and we, as task 875, happen to be in the topmost container:

/containers/

Now we fork task 999 which does an unshare(CLONE_NEWNS), or we just
clone(CLONE_NEWNS). This will create

/containers/node_999

and move task 999 into that container. Except that when it tries
attach_task() it is refused by cpuset. So the container_clone() fails,
and in turn the sys_unshare() or clone() fails. A login making use
of the pam_namespace.so library would fail this way with the
ns and cpuset subsystems composed.

We could special case this by having
kernel/container.c:container_clone() check whether one of the subsystems
is cpusets and, if so, setting the defaults for mems and cpus, but
that is kind of ugly. I suppose as a cleaner alternative we could
add a container_subsys->inherit_defaults() handler, to be called at
container_clone(), and for cpusets this would set cpus and mems to
the parent values - sibling exclusive values. If that comes to nothing,
then the attach_task() is still refused, and the unshare() or clone()
fails, but this time with good reason.

thanks,
-serge

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Thu, 07 Jun 2007 00:46:09 GMT
View Forum Message <> Reply to Message

> I suppose as a cleaner alternative we could
> add a container_subsys->inherit_defaults() handler, to be called at
> container_clone(), and for cpusets this would set cpus and mems to
> the parent values - sibling exclusive values. If that comes to nothing,
> then the attach_task() is still refused, and the unshare() or clone()
> fails, but this time with good reason.

Unfortunately, I haven't spent the time I should thinking about
container cloning, namespaces and such.

I don't know, for the workloads that matter to me, when, how or
if this container cloning will be used.

I'm tempted to suggest the following.

First, I am assuming that the classic method of creating cpuset
children will still work, such as the following (which can fail
for certain combinations of exclusive cpus or mems):
 cd /dev/cpuset/foobar
 mkdir foochild
 cp cpus foochild
 cp mems foochild
 echo $$ > foochild/tasks

Second, given that, how about you fail the unshare() or clone()
anytime that the instance to be cloned has any sibling cpusets
with any exclusive flags set.

The exclusive property is not really on friendly terms with cloning.

Now if the above classic code must be encoded using cloning under
the covers, then we've got problems, probably more problems than
just this.

--

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

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Thu, 07 Jun 2007 18:01:58 GMT

Quoting Paul Jackson (pj@sgi.com):
> > I suppose as a cleaner alternative we could
> > add a container_subsys->inherit_defaults() handler, to be called at
> > container_clone(), and for cpusets this would set cpus and mems to
> > the parent values - sibling exclusive values. If that comes to nothing,
> > then the attach_task() is still refused, and the unshare() or clone()
> > fails, but this time with good reason.
>
> Unfortunately, I haven't spent the time I should thinking about
> container cloning, namespaces and such.
>
> I don't know, for the workloads that matter to me, when, how or
> if this container cloning will be used.
>
> I'm tempted to suggest the following.
>
> First, I am assuming that the classic method of creating cpuset
> children will still work, such as the following (which can fail
> for certain combinations of exclusive cpus or mems):
> cd /dev/cpuset/foobar
> mkdir foochild
> cp cpus foochild
> cp mems foochild
> echo $$ > foochild/tasks
>
> Second, given that, how about you fail the unshare() or clone()
> anytime that the instance to be cloned has any sibling cpusets
> with any exclusive flags set.

The below patch (on top of my previous patch) does basically that. But
I wasn't able to test it, bc i wasn't able to set cpus_exclusive...

For /cpusets/set0/set1 to have cpu 1 exclusively, does /cpusets/set0
also have to have it exclusively?

If so, then clearly this approach won't work, since if any container has
exclusive cpus, then every container will have siblings with exclusive
cpus, and unshare still isn't possible on the system.

> The exclusive property is not really on friendly terms with cloning.
>
> Now if the above classic code must be encoded using cloning under
> the covers, then we've got problems, probably more problems than
> just this.
>
> --
>   I won't rest till it's the best ...

> Programmer, Linux Scalability
> Paul Jackson <pj@sgi.com> 1.925.600.0401

thanks,
-serge

>From 821de58b6ba446e50225606e907baac00130586c Mon Sep 17 00:00:00 2001
From: Serge E. Hallyn <serue@us.ibm.com>
Date: Thu, 7 Jun 2007 13:53:43 -0400
Subject: [PATCH 1/1] containers: implement subsys->auto_setup

container_clone() in one step creates a new container and moves
the current task into it. Since cpusets do not automatically
fill in the allowed cpus and mems, and do not allow a task to
be attached without these filled in, composing the ns subsystem,
which uses container_clone(), and the cpuset subsystem, results
in sys_unshare() (and clone(CLONE_NEWNS)) always being denied.

To allow the two subsystems to be meaningfully composed, implement
subsystem->auto_setup, called from container_clone() after
creating the new container.

Only the cpuset_auto_setup() is currently implemented. If any
sibling containers have exclusive cpus or mems, then the cpus
and mems are not filled in for the new container, meaning that
unshare/clone(CLONE_NEWNS) will be denied. However so long as
no siblings have exclusive cpus or mems, the new container's
cpus and mems are inherited from the parent container.

Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>

 Documentation/containers.txt | 7 +++++++
 include/linux/container.h | 1 +
 kernel/container.c | 7 +++++++
 kernel/cpuset.c | 21 +++++++++++++++++++++
 4 files changed, 36 insertions(+), 0 deletions(-)

diff --git a/Documentation/containers.txt b/Documentation/containers.txt
index ae159b9..28c9e10 100644
--- a/Documentation/containers.txt
+++ b/Documentation/containers.txt
@@ -514,6 +514,13 @@ include/linux/container.h for details). Note that although this
 method can return an error code, the error code is currently not
 always handled well.

+void auto_setup(struct container_subsys *ss, struct container *cont)
+
+Called at container_clone() to do any paramater initialization

```
+which might be required before a task could attach.  For example
+in cpusets, no task may attach before 'cpus' and 'mems' are
+set up.
+
 void bind(struct container_subsys *ss, struct container *root)
 LL=callback_mutex

diff --git a/include/linux/container.h b/include/linux/container.h
index 37c0bdf..d809b41 100644
--- a/include/linux/container.h
+++ b/include/linux/container.h
@@ -213,6 +213,7 @@ struct container_subsys {
  void (*exit)(struct container_subsys *ss, struct task_struct *task);
  int (*populate)(struct container_subsys *ss,
    struct container *cont);
+ void (*auto_setup)(struct container_subsys *ss, struct container *cont);
  void (*bind)(struct container_subsys *ss, struct container *root);
  int subsys_id;
  int active;
diff --git a/kernel/container.c b/kernel/container.c
index 988cd8b..e0793f4 100644
--- a/kernel/container.c
+++ b/kernel/container.c
@@ -2316,6 +2316,7 @@ int container_clone(struct task_struct *tsk, struct container_subsys
*subsys)
  struct inode *inode;
  struct css_group *cg;
  struct containerfs_root *root;
+ struct container_subsys *ss;

  /* We shouldn't be called by an unregistered subsystem */
  BUG_ON(!subsys->active);
@@ -2397,6 +2398,12 @@ int container_clone(struct task_struct *tsk, struct container_subsys
*subsys)
   goto again;
  }

+ /* do any required auto-setup */
+ for_each_subsys(root, ss) {
+  if (ss->auto_setup)
+   ss->auto_setup(ss, child);
+ }
+
  /* All seems fine. Finish by moving the task into the new container */
  ret = attach_task(child, tsk);
  mutex_unlock(&container_mutex);
diff --git a/kernel/cpuset.c b/kernel/cpuset.c
index 0f9ce7d..ff01aaa 100644
```

```
--- a/kernel/cpuset.c
+++ b/kernel/cpuset.c
@@ -1189,6 +1189,26 @@ int cpuset_populate(struct container_subsys *ss, struct container
*cont)
  return 0;
 }

+void cpuset_auto_setup(struct container_subsys *ss,
+  struct container *container)
+{
+ struct container *parent, *child;
+ struct cpuset *cs, *parent_cs;
+
+ parent = container->parent;
+ list_for_each_entry(child, &parent->children, sibling) {
+  cs = container_cs(child);
+  if (is_mem_exclusive(cs) || is_cpu_exclusive(cs))
+   return;
+ }
+ cs = container_cs(container);
+ parent_cs = container_cs(parent);
+
+ cs->mems_allowed = parent_cs->mems_allowed;
+ cs->cpus_allowed = parent_cs->cpus_allowed;
+ return;
+}
+
 /*
  * cpuset_create - create a cpuset
  * parent: cpuset that will be parent of the new cpuset.
@@ -1249,6 +1269,7 @@ struct container_subsys cpuset_subsys = {
  .can_attach = cpuset_can_attach,
  .attach = cpuset_attach,
  .populate = cpuset_populate,
+ .auto_setup = cpuset_auto_setup,
  .subsys_id = cpuset_subsys_id,
  .early_init = 1,
 };
--
1.5.1.1.GIT
```

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers

Posted by Paul Jackson on Thu, 07 Jun 2007 19:21:21 GMT

View Forum Message <> Reply to Message

> For /cpusets/set0/set1 to have cpu 1 exclusively, does /cpusets/set0

> also have to have it exclusively?

Yes.

> If so, then clearly this approach won't work, since if any container has
> exclusive cpus, then every container will have siblings with exclusive
> cpus, and unshare still isn't possible on the system.

Well, if I'm following you, not exactly.

If we have some exclusive flags set, then every top level container
will have exclusive siblings, but further down the hierarchy, some
subtree might be entirely free of any exclusive settings. Then nodes
below the top of that subtree would not have exclusive set, and would
not have any exclusive siblings.

But, overall, yeah, exclusive is no friend of container cloning.

I just wish I had been thinking harder about how container cloning
will impact my life, and the lives of the customers in my cpuset
intensive corner of the world.

There are certainly a whole bunch of people who will never have any
need for exclusive cpusets.

Perhaps (speculating wildly from great ignorance) there are a whole
bunch of people who will never have need for container cloning.

And perhaps, hoping to get lucky here, the set of people who need both
at the same time on the same system is sufficiently close to empty
that we can just tell them tough toenails - you cannot do both at once.

How wide spread will be the use of container cloning, if it proceeds
as envisioned?

The set of people using exclusive cpusets is roughly some subset of
those running multiple, cpuset isolated, non-cooperating jobs on big
iron, usually with the aid of a batch scheduler. Well, that's what
I am aware of anyway. If there are any other friends of exclusive
cpusets lurking here, you might want to speak up, before I sell your
interests down the river.

--

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

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serue on Thu, 07 Jun 2007 20:17:23 GMT
View Forum Message <> Reply to Message

Quoting Paul Jackson (pj@sgi.com):
> > For /cpusets/set0/set1 to have cpu 1 exclusively, does /cpusets/set0
> > also have to have it exclusively?
>
> Yes.
>
> > If so, then clearly this approach won't work, since if any container has
> > exclusive cpus, then every container will have siblings with exclusive
> > cpus, and unshare still isn't possible on the system.
>
> Well, if I'm following you, not exactly.
>
> If we have some exclusive flags set, then every top level container
> will have exclusive siblings, but further down the hierarchy, some
> subtree might be entirely free of any exclusive settings. Then nodes
> below the top of that subtree would not have exclusive set, and would
> not have any exclusive siblings.
>
> But, overall, yeah, exclusive is no friend of container cloning.
>
> I just wish I had been thinking harder about how container cloning
> will impact my life, and the lives of the customers in my cpuset
> intensive corner of the world.
>
> There are certainly a whole bunch of people who will never have any
> need for exclusive cpusets.
>
> Perhaps (speculating wildly from great ignorance) there are a whole
> bunch of people who will never have need for container cloning.
>
> And perhaps, hoping to get lucky here, the set of people who need both
> at the same time on the same system is sufficiently close to empty
> that we can just tell them tough toenails - you cannot do both at once.
>
> How wide spread will be the use of container cloning, if it proceeds
> as envisioned?

It's not just container cloning, but all namespace unsharing. So uses include (1) providing 'polyinstantiated directory' functionality, i.e. private per-user /tmp's or per-security-level /tmp and /home's. (2) any virtual server usage (3) hpc checkpoint/restart users.

> The set of people using exclusive cpusets is roughly some subset of
> those running multiple, cpuset isolated, non-cooperating jobs on big

> iron, usually with the aid of a batch scheduler.

Unfortunately I would imagine these users to be very intereseted in
providing checkpoint/restart/migrate functionality.

> Well, that's what
> I am aware of anyway. If there are any other friends of exclusive
> cpusets lurking here, you might want to speak up, before I sell your
> interests down the river.
>
> --
> I won't rest till it's the best ...
> Programmer, Linux Scalability
> Paul Jackson <pj@sgi.com> 1.925.600.0401

Can you explain to me, though, why it should be that if /cpusets/set0
has access to cpus 0-8, and /cpusets/set0/set1 has exclusive access to
cpus 0-2, and /cpusets/set0/set2 has exclusive access to cpus 3-4,
why i a process in /cpusets/set0 creates /cpusets/set0/set3 through
container_clone, it would be unsafe to have it automatically get cpus 5-8?

Surely if the admin wants to give cpus 5-6 exclusively to /cpusets/set0/set4
later, those cpus can just be taken away from set3?

thanks,
-serge

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Thu, 07 Jun 2007 22:01:13 GMT
View Forum Message <> Reply to Message

> > The set of people using exclusive cpusets is roughly some subset of
> > those running multiple, cpuset isolated, non-cooperating jobs on big
> > iron, usually with the aid of a batch scheduler.
>
> Unfortunately I would imagine these users to be very intereseted in
> providing checkpoint/restart/migrate functionality.

Yup - such customers are very interested in checkpoint, restart, and
migrate functionality.

> Surely if the admin wants to give cpus 5-6 exclusively to /cpusets/set0/set4
> later, those cpus can just be taken away from set3?

Yeah - that works, so far as I know (which isn't all that far ..')

But both:
 1) that (using whatever cpus are still available) and
 2) my other idea, of not allowing any cloning of cpusets with
    exclusive siblings at all,

looked a little ugly to me.

For example, such customers as above would not appreciate having their checkpoint/restart/migrate fail in any case where there weren't spare non-exclusive cpus, which for users of the exclusive flag, is often the more common case.

My rule of thumb when doing ugly stuff is to constrain it as best I can -- minimize what it allows. This led me to prefer (2) above over (1) above.

Perhaps there's a better way to think of this ... When we clone like this for checkpoint/restart/migrate functionality, perhaps we are not really starting up a new, separate, competing job that should have its resources isolated and separated from the original.

Perhaps instead we are firing up a convenient alter-ego of the original job, which will be co-operatively using the same resources by default. If that's the normal case, then it seems wrong to force the clone onto disjoint CPUs, or fail for lack thereof.

So perhaps we should refine the meaning of 'exclusive', from:
 - no overlapping siblings
to:
 - no overlapping siblings other than clones of ones self.

Then default to cloning right on the same CPU resources as the original, possibly with both original and clone marked exclusive.

--

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

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serge on Fri, 08 Jun 2007 14:32:50 GMT
View Forum Message <> Reply to Message

Quoting Paul Jackson (pj@sgi.com):
> > > The set of people using exclusive cpusets is roughly some subset of
> > > those running multiple, cpuset isolated, non-cooperating jobs on big

> > > iron, usually with the aid of a batch scheduler.
> >
> > Unfortunately I would imagine these users to be very intereseted in
> > providing checkpoint/restart/migrate functionality.
>
> Yup - such customers are very interested in checkpoint, restart, and
> migrate functionality.
>
> > Surely if the admin wants to give cpus 5-6 exclusively to /cpusets/set0/set4
> > later, those cpus can just be taken away from set3?
>
> Yeah - that works, so far as I know (which isn't all that far ..')
>
> But both:
> 1) that (using whatever cpus are still available) and
> 2) my other idea, of not allowing any cloning of cpusets with
>   exclusive siblings at all,
>
> looked a little ugly to me.
>
> For example, such customers as above would not appreciate having their
> checkpoint/restart/migrate fail in any case where there weren't spare
> non-exclusive cpus, which for users of the exclusive flag, is often the
> more common case.
>
> My rule of thumb when doing ugly stuff is to constrain it as best
> I can -- minimize what it allows. This led me to prefer (2) above
> over (1) above.
>
> Perhaps there's a better way to think of this ... When we clone
> like this for checkpoint/restart/migrate functionality, perhaps
> we are not really starting up a new, separate, competing job that
> should have its resources isolated and separated from the original.

Depends on whether the cpus are allocated to a customer or to a job.

For the most part I would expect any job to be restart either on a
different machine, or at a different time, but of course it doesn't have
to be that way.

> Perhaps instead we are firing up a convenient alter-ego of the
> original job, which will be co-operatively using the same resources
> by default. If that's the normal case, then it seems wrong to
> force the clone onto disjoint CPUs, or fail for lack thereof.
>
> So perhaps we should refine the meaning of 'exclusive', from:
> - no overlapping siblings
> to:

> - no overlapping siblings other than clones of ones self.

I'm not sure that clones of self will happen often enough to make a
special case for them :)

Anyway the patch I sent is simple enough, and if users end up demanding
the ability to better deal with exclusive cpusets, the patch will be
simple enough to extend by changing cpuset_auto_setup(), so let's
stick with that patch since it's your preference (IIUC).

> Then default to cloning right on the same CPU resources as the
> original, possibly with both original and clone marked exclusive.

Thanks,
-serge

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Menage on Fri, 08 Jun 2007 15:55:47 GMT
View Forum Message <> Reply to Message

On 6/8/07, Serge E. Hallyn <serge@hallyn.com> wrote:
>
> Anyway the patch I sent is simple enough, and if users end up demanding
> the ability to better deal with exclusive cpusets, the patch will be
> simple enough to extend by changing cpuset_auto_setup(), so let's
> stick with that patch since it's your preference (IIUC).
>

Sounds good to me, although I think my preference would be to extend
the "create()" subsystem callback with a "struct task_struct
*clone_task" parameter that indicates that clone_task is cloning its
own container; a subsystem like cpusets that needs to do additional
setup at that point could inherit settings either from the parent or
from clone_task's container (or something else) as desired. (It could
also do permission checking based on properties of clone_task, etc).

Paul

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by serge on Fri, 08 Jun 2007 16:08:40 GMT
View Forum Message <> Reply to Message

Quoting Paul Menage (menage@google.com):

> On 6/8/07, Serge E. Hallyn <serge@hallyn.com> wrote:
> >
> >Anyway the patch I sent is simple enough, and if users end up demanding
> >the ability to better deal with exclusive cpusets, the patch will be
> >simple enough to extend by changing cpuset_auto_setup(), so let's
> >stick with that patch since it's your preference (IIUC).
> >
>
> Sounds good to me, although I think my preference would be to extend
> the "create()" subsystem callback with a "struct task_struct
> *clone_task" parameter that indicates that clone_task is cloning its
> own container; a subsystem like cpusets that needs to do additional
> setup at that point could inherit settings either from the parent or
> from clone_task's container (or something else) as desired. (It could
> also do permission checking based on properties of clone_task, etc).

The problem is container_clone() doesn't call ->create explicitly, it
does vfs_mkdir. So we have no real way of passing in clone_task.

-serge

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Jackson on Fri, 08 Jun 2007 17:37:56 GMT
View Forum Message <> Reply to Message

> Anyway the patch I sent is simple enough, and if users end up demanding
> the ability to better deal with exclusive cpusets, the patch will be
> simple enough to extend by changing cpuset_auto_setup(), so let's
> stick with that patch since it's your preference (IIUC).

Yeah - probably so.

> When someone gets serious about things like checkpoint, restart, and
> migrate functionality, based on this container cloning, working with
> cpusets, they will probably have to revisit this interaction with
> exclusive cpusets.

Perhaps a comment could be put in the code, saying something like the
above, so whomever does this will realize they are traveling in
unchartered territory.

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

Subject: Re: [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Paul Menage on Thu, 28 Jun 2007 21:27:25 GMT
View Forum Message <> Reply to Message

On 5/30/07, William Lee Irwin III <wli@holomorphy.com> wrote:
> On Wed, May 30, 2007 at 12:14:55AM -0700, Andrew Morton wrote:
> > So how do we do this?
> > Is there any sneaky way in which we can modify the kernel so that this new
> > code gets exercised more? Obviously, tossing init into some default
> > system-wide container would be a start. But I wonder if we can be
> > sneakier - for example, create a new container on each setuid(), toss the
> > task into that. Or something along those lines?
>
> How about a container for each thread group, pgrp, session, and user?
>

I've been thinking about this, and figured that it could be quite useful to be able to mount a container tree that groups tasks by userid or thread group - for doing per-user resource controls, for example, without having to write a controller that specifically handles the per-user case.

One option would be to add a mount option, something like

mount -t container -ogroupkey=<X>

where X could be one of: uid, gid, pgrp, sid, tgid

And put hooks in the various places where these ids could change, in order to move tasks between contaners as appropriate. But after some thought it seems to me that this is putting complexity in the kernel that probably doesn't belong there, and additionally is probably not sufficently flexible for some real-life situations. (E.g. the user wants all users in the "student" group to be lumped into the same container, but each user in the "professor" group gets their own container).

So maybe this would be better handled in userspace? Have a daemon listing on a process connector socket, and move processes between containers based on notifications from the connector and user-defined rules.

We'd probably also want to add some new connector events, such as PROC_EVENT_PGRP, and PROC_EVENT_SID

A simple daemon that handles the case where we're classifying based on a single key with no complex rules shouldn't be too hard to write.

It also sounds rather like the classification engine that

ResourceGroups had originally in the kernel and moved to userspace, so
I'll take a look at that and see if it's adaptable for this.

Paul

Subject: Re: [ckrm-tech] [PATCH 00/10] Containers(V10): Generic Process Containers
Posted by Srivatsa Vaddagiri on Thu, 28 Jun 2007 22:13:22 GMT
View Forum Message <> Reply to Message

On Thu, Jun 28, 2007 at 05:27:25PM -0400, Paul Menage wrote:
> So maybe this would be better handled in userspace? Have a daemon
> listing on a process connector socket, and move processes between
> containers based on notifications from the connector and user-defined
> rules.
>
> We'd probably also want to add some new connector events, such as
> PROC_EVENT_PGRP, and PROC_EVENT_SID

Yep, this is what I did to test fair-user scheduling on top of my
patches.

Dhaval has a working program, which listens for UID change events and
moves the task to approp. container. I will review that and have it
posted soon.

--
Regards,
vatsa
