Subject: [PATCH 01/33] task containersv11 basic task container framework
Posted by Paul Menage on Mon, 17 Sep 2007 21:03:08 GMT
View Forum Message <> Reply to Message

Generic Process Control Groups

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
cgroups, 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.

This patchset provides a framework for tracking and grouping processes
into arbitrary "cgroups" and assigning arbitrary state to those
groupings, in order to control the behaviour of the cgroup as an
aggregate.

The intention is that the various resource management and
virtualization/cgroup efforts can also become task cgroup
clients, with the result that:

- the userspace APIs are (somewhat) normalised

- it's easier to test 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

This patch:

Add the main task cgroups framework - the cgroup filesystem, and the
basic structures for tracking membership and associating subsystem state
objects to tasks.

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

Documentation/cgroups.txt | 526 ++++++++++++++++

```
 include/linux/cgroup.h        |  214 +++++
 include/linux/cgroup_subsys.h |   10
 include/linux/magic.h         |    1
 include/linux/sched.h         |   34
 init/Kconfig                  |    8
 init/main.c                   |    3
 kernel/Makefile               |    1
 kernel/cgroup.c               | 1199 ++++++++++++++++++++++++++++++++++++++++++++
 9 files changed, 1995 insertions(+), 1 deletion(-)

diff -puN /dev/null Documentation/cgroups.txt
--- /dev/null
+++ a/Documentation/cgroups.txt
@@ -0,0 +1,526 @@
+    CGROUPS
+    -------
+
+Written by Paul Menage <menage@google.com> based on Documentation/cpusets.txt
+
+Original copyright statements from cpusets.txt:
+Portions Copyright (C) 2004 BULL SA.
+Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
+Modified by Paul Jackson <pj@sgi.com>
+Modified by Christoph Lameter <clameter@sgi.com>
+
+CONTENTS:
+=========
+
+1. Control Groups
+  1.1 What are cgroups ?
+  1.2 Why are cgroups needed ?
+  1.3 How are cgroups implemented ?
+  1.4 What does notify_on_release do ?
+  1.5 How do I use cgroups ?
+2. Usage Examples and Syntax
+  2.1 Basic Usage
+  2.2 Attaching processes
+3. Kernel API
+  3.1 Overview
+  3.2 Synchronization
+  3.3 Subsystem API
+4. Questions
+
+1. Control Groups
+==========
+
+1.1 What are cgroups ?
+----------------------
```

+
+Control Groups provide a mechanism for aggregating/partitioning sets of
+tasks, and all their future children, into hierarchical groups with
+specialized behaviour.
+
+Definitions:
+
+A *cgroup* associates a set of tasks with a set of parameters for one
+or more subsystems.
+
+A *subsystem* is a module that makes use of the task grouping
+facilities provided by cgroups to treat groups of tasks in
+particular ways. A subsystem is typically a "resource controller" that
+schedules a resource or applies per-cgroup limits, but it may be
+anything that wants to act on a group of processes, e.g. a
+virtualization subsystem.
+
+A *hierarchy* is a set of cgroups arranged in a tree, such that
+every task in the system is in exactly one of the cgroups in the
+hierarchy, and a set of subsystems; each subsystem has system-specific
+state attached to each cgroup in the hierarchy. Each hierarchy has
+an instance of the cgroup virtual filesystem associated with it.
+
+At any one time there may be multiple active hierachies of task
+cgroups. Each hierarchy is a partition of all tasks in the system.
+
+User level code may create and destroy cgroups by name in an
+instance of the cgroup virtual file system, specify and query to
+which cgroup a task is assigned, and list the task pids assigned to
+a cgroup. Those creations and assignments only affect the hierarchy
+associated with that instance of the cgroup file system.
+
+On their own, the only use for cgroups is for simple job
+tracking. The intention is that other subsystems hook into the generic
+cgroup support to provide new attributes for cgroups, such as
+accounting/limiting the resources which processes in a cgroup can
+access. For example, cpusets (see Documentation/cpusets.txt) allows
+you to associate a set of CPUs and a set of memory nodes with the
+tasks in each cgroup.
+
+1.2 Why are cgroups needed ?
+----------------------------
+
+There are multiple efforts to provide process aggregations in the
+Linux kernel, mainly for resource tracking purposes. Such efforts
+include cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server
+namespaces. These all require the basic notion of a
+grouping/partitioning of processes, with newly forked processes ending

+in the same group (cgroup) as their parent process.
+
+The kernel cgroup patch provides the minimum essential kernel
+mechanisms required to efficiently implement such groups. It has
+minimal impact on the system fast paths, and provides hooks for
+specific subsystems such as cpusets to provide additional behaviour as
+desired.
+
+Multiple hierarchy support is provided to allow for situations where
+the division of tasks into cgroups is distinctly different for
+different subsystems - having parallel hierarchies allows each
+hierarchy to be a natural division of tasks, without having to handle
+complex combinations of tasks that would be present if several
+unrelated subsystems needed to be forced into the same tree of
+cgroups.
+
+At one extreme, each resource controller or subsystem could be in a
+separate hierarchy; at the other extreme, all subsystems
+would be attached to the same hierarchy.
+
+As an example of a scenario (originally proposed by vatsa@in.ibm.com)
+that can benefit from multiple hierarchies, consider a large
+university server with various users - students, professors, system
+tasks etc. The resource planning for this server could be along the
+following lines:
+
+   CPU :   Top cpuset
+   /   \
+   CPUSet1   CPUSet2
+ | |
+   (Profs)   (Students)
+
+   In addition (system tasks) are attached to topcpuset (so
+   that they can run anywhere) with a limit of 20%
+
+   Memory : Professors (50%), students (30%), system (20%)
+
+   Disk : Prof (50%), students (30%), system (20%)
+
+   Network : WWW browsing (20%), Network File System (60%), others (20%)
+   / \
+   Prof (15%) students (5%)
+
+Browsers like firefox/lynx go into the WWW network class, while (k)nfsd go
+into NFS network class.
+
+At the same time firefox/lynx will share an appropriate CPU/Memory class
+depending on who launched it (prof/student).

+
+With the ability to classify tasks differently for different resources
+(by putting those resource subsystems in different hierarchies) then
+the admin can easily set up a script which receives exec notifications
+and depending on who is launching the browser he can
+
+	# echo browser_pid > /mnt/<restype>/<userclass>/tasks
+
+With only a single hierarchy, he now would potentially have to create
+a separate cgroup for every browser launched and associate it with
+approp network and other resource class. This may lead to
+proliferation of such cgroups.
+
+Also lets say that the administrator would like to give enhanced network
+access temporarily to a student's browser (since it is night and the user
+wants to do online gaming :) OR give one of the students simulation
+apps enhanced CPU power,
+
+With ability to write pids directly to resource classes, its just a
+matter of :
+
+	# echo pid > /mnt/network/<new_class>/tasks
+	(after some time)
+	# echo pid > /mnt/network/<orig_class>/tasks
+
+Without this ability, he would have to split the cgroup into
+multiple separate ones and then associate the new cgroups with the
+new resource classes.
+
+
+
+1.3 How are cgroups implemented ?
+---------------------------------
+
+Control Groups extends the kernel as follows:
+
+ - Each task in the system has a reference-counted pointer to a
+   css_set.
+
+ - A css_set contains a set of reference-counted pointers to
+   cgroup_subsys_state objects, one for each cgroup subsystem
+   registered in the system. There is no direct link from a task to
+   the cgroup of which it's a member in each hierarchy, but this
+   can be determined by following pointers through the
+   cgroup_subsys_state objects. This is because accessing the
+   subsystem state is something that's expected to happen frequently
+   and in performance-critical code, whereas operations that require a
+   task's actual cgroup assignments (in particular, moving between

+   cgroups) are less common.
+
+ - A cgroup hierarchy filesystem can be mounted for browsing and
+   manipulation from user space.
+
+ - You can list all the tasks (by pid) attached to any cgroup.
+
+The implementation of cgroups requires a few, simple hooks
+into the rest of the kernel, none in performance critical paths:
+
+ - in init/main.c, to initialize the root cgroups and initial
+   css_set at system boot.
+
+ - in fork and exit, to attach and detach a task from its css_set.
+
+In addition a new file system, of type "cgroup" may be mounted, to
+enable browsing and modifying the cgroups presently known to the
+kernel. When mounting a cgroup hierarchy, you may specify a
+comma-separated list of subsystems to mount as the filesystem mount
+options. By default, mounting the cgroup filesystem attempts to
+mount a hierarchy containing all registered subsystems.
+
+If an active hierarchy with exactly the same set of subsystems already
+exists, it will be reused for the new mount. If no existing hierarchy
+matches, and any of the requested subsystems are in use in an existing
+hierarchy, the mount will fail with -EBUSY. Otherwise, a new hierarchy
+is activated, associated with the requested subsystems.
+
+It's not currently possible to bind a new subsystem to an active
+cgroup hierarchy, or to unbind a subsystem from an active cgroup
+hierarchy. This may be possible in future, but is fraught with nasty
+error-recovery issues.
+
+When a cgroup filesystem is unmounted, if there are any
+child cgroups created below the top-level cgroup, that hierarchy
+will remain active even though unmounted; if there are no
+child cgroups then the hierarchy will be deactivated.
+
+No new system calls are added for cgroups - all support for
+querying and modifying cgroups is via this cgroup file system.
+
+Each task under /proc has an added file named 'cgroup' displaying,
+for each active hierarchy, the subsystem names and the cgroup name
+as the path relative to the root of the cgroup file system.
+
+Each cgroup is represented by a directory in the cgroup file system
+containing the following files describing that cgroup:
+

+ - tasks: list of tasks (by pid) attached to that cgroup
+ - notify_on_release flag: run /sbin/cgroup_release_agent on exit?
+
+Other subsystems such as cpusets may add additional files in each
+cgroup dir
+
+New cgroups are created using the mkdir system call or shell
+command. The properties of a cgroup, such as its flags, are
+modified by writing to the appropriate file in that cgroups
+directory, as listed above.
+
+The named hierarchical structure of nested cgroups allows partitioning
+a large system into nested, dynamically changeable, "soft-partitions".
+
+The attachment of each task, automatically inherited at fork by any
+children of that task, to a cgroup allows organizing the work load
+on a system into related sets of tasks. A task may be re-attached to
+any other cgroup, if allowed by the permissions on the necessary
+cgroup file system directories.
+
+When a task is moved from one cgroup to another, it gets a new
+css_set pointer - if there's an already existing css_set with the
+desired collection of cgroups then that group is reused, else a new
+css_set is allocated. Note that the current implementation uses a
+linear search to locate an appropriate existing css_set, so isn't
+very efficient. A future version will use a hash table for better
+performance.
+
+The use of a Linux virtual file system (vfs) to represent the
+cgroup hierarchy provides for a familiar permission and name space
+for cgroups, with a minimum of additional kernel code.
+
+1.4 What does notify_on_release do ?
+----------------------------------
+
+*** notify_on_release is disabled in the current patch set. It will be
+*** reactivated in a future patch in a less-intrusive manner
+
+If the notify_on_release flag is enabled (1) in a cgroup, then
+whenever the last task in the cgroup leaves (exits or attaches to
+some other cgroup) and the last child cgroup of that cgroup
+is removed, then the kernel runs the command specified by the contents
+of the "release_agent" file in that hierarchy's root directory,
+supplying the pathname (relative to the mount point of the cgroup
+file system) of the abandoned cgroup. This enables automatic
+removal of abandoned cgroups. The default value of
+notify_on_release in the root cgroup at system boot is disabled
+(0). The default value of other cgroups at creation is the current

+value of their parents notify_on_release setting. The default value of
+a cgroup hierarchy's release_agent path is empty.
+
+1.5 How do I use cgroups ?
+--------------------------
+
+To start a new job that is to be contained within a cgroup, using
+the "cpuset" cgroup subsystem, the steps are something like:
+
+ 1) mkdir /dev/cgroup
+ 2) mount -t cgroup -ocpuset cpuset /dev/cgroup
+ 3) Create the new cgroup by doing mkdir's and write's (or echo's) in
+   the /dev/cgroup virtual file system.
+ 4) Start a task that will be the "founding father" of the new job.
+ 5) Attach that task to the new cgroup by writing its pid to the
+   /dev/cgroup tasks file for that cgroup.
+ 6) fork, exec or clone the job tasks from this founding father task.
+
+For example, the following sequence of commands will setup a cgroup
+named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
+and then start a subshell 'sh' in that cgroup:
+
+ mount -t cgroup cpuset -ocpuset /dev/cgroup
+ cd /dev/cgroup
+ mkdir Charlie
+ cd Charlie
+ /bin/echo 2-3 > cpus
+ /bin/echo 1 > mems
+ /bin/echo $$ > tasks
+ sh
+ # The subshell 'sh' is now running in cgroup Charlie
+ # The next line should display '/Charlie'
+ cat /proc/self/cgroup
+
+2. Usage Examples and Syntax
+============================
+
+2.1 Basic Usage
+---------------
+
+Creating, modifying, using the cgroups can be done through the cgroup
+virtual filesystem.
+
+To mount a cgroup hierarchy will all available subsystems, type:
+# mount -t cgroup xxx /dev/cgroup
+
+The "xxx" is not interpreted by the cgroup code, but will appear in
+/proc/mounts so may be any useful identifying string that you like.

+
+To mount a cgroup hierarchy with just the cpuset and numtasks
+subsystems, type:
+# mount -t cgroup -o cpuset,numtasks hier1 /dev/cgroup
+
+To change the set of subsystems bound to a mounted hierarchy, just
+remount with different options:
+
+# mount -o remount,cpuset,ns /dev/cgroup
+
+Note that changing the set of subsystems is currently only supported
+when the hierarchy consists of a single (root) cgroup. Supporting
+the ability to arbitrarily bind/unbind subsystems from an existing
+cgroup hierarchy is intended to be implemented in the future.
+
+Then under /dev/cgroup you can find a tree that corresponds to the
+tree of the cgroups in the system. For instance, /dev/cgroup
+is the cgroup that holds the whole system.
+
+If you want to create a new cgroup under /dev/cgroup:
+# cd /dev/cgroup
+# mkdir my_cgroup
+
+Now you want to do something with this cgroup.
+# cd my_cgroup
+
+In this directory you can find several files:
+# ls
+notify_on_release release_agent tasks
+(plus whatever files are added by the attached subsystems)
+
+Now attach your shell to this cgroup:
+# /bin/echo $$ > tasks
+
+You can also create cgroups inside your cgroup by using mkdir in this
+directory.
+# mkdir my_sub_cs
+
+To remove a cgroup, just use rmdir:
+# rmdir my_sub_cs
+
+This will fail if the cgroup is in use (has cgroups inside, or
+has processes attached, or is held alive by other subsystem-specific
+reference).
+
+2.2 Attaching processes
+----------------------
+

+# /bin/echo PID > tasks
+
+Note that it is PID, not PIDs. You can only attach ONE task at a time.
+If you have several tasks to attach, you have to do it one after another:
+
+# /bin/echo PID1 > tasks
+# /bin/echo PID2 > tasks
+ ...
+# /bin/echo PIDn > tasks
+
+3. Kernel API
+=============
+
+3.1 Overview
+------------
+
+Each kernel subsystem that wants to hook into the generic cgroup
+system needs to create a cgroup_subsys object. This contains
+various methods, which are callbacks from the cgroup system, along
+with a subsystem id which will be assigned by the cgroup system.
+
+Other fields in the cgroup_subsys object include:
+
+- subsys_id: a unique array index for the subsystem, indicating which
+ entry in cgroup->subsys[] this subsystem should be
+ managing. Initialized by cgroup_register_subsys(); prior to this
+ it should be initialized to -1
+
+- hierarchy: an index indicating which hierarchy, if any, this
+ subsystem is currently attached to. If this is -1, then the
+ subsystem is not attached to any hierarchy, and all tasks should be
+ considered to be members of the subsystem's top_cgroup. It should
+ be initialized to -1.
+
+- name: should be initialized to a unique subsystem name prior to
+ calling cgroup_register_subsystem. Should be no longer than
+ MAX_CGROUP_TYPE_NAMELEN
+
+Each cgroup object created by the system has an array of pointers,
+indexed by subsystem id; this pointer is entirely managed by the
+subsystem; the generic cgroup code will never touch this pointer.
+
+3.2 Synchronization
+-------------------
+
+There is a global mutex, cgroup_mutex, used by the cgroup
+system. This should be taken by anything that wants to modify a
+cgroup. It may also be taken to prevent cgroups from being

+modified, but more specific locks may be more appropriate in that
+situation.
+
+See kernel/cgroup.c for more details.
+
+Subsystems can take/release the cgroup_mutex via the functions
+cgroup_lock()/cgroup_unlock(), and can
+take/release the callback_mutex via the functions
+cgroup_lock()/cgroup_unlock().
+
+Accessing a task's cgroup pointer may be done in the following ways:
+- while holding cgroup_mutex
+- while holding the task's alloc_lock (via task_lock())
+- inside an rcu_read_lock() section via rcu_dereference()
+
+3.3 Subsystem API
+--------------------------
+
+Each subsystem should:
+
+- add an entry in linux/cgroup_subsys.h
+- define a cgroup_subsys object called <name>_subsys
+
+Each subsystem may export the following methods. The only mandatory
+methods are create/destroy. Any others that are null are presumed to
+be successful no-ops.
+
+struct cgroup_subsys_state *create(struct cgroup *cont)
+LL=cgroup_mutex
+
+Called to create a subsystem state object for a cgroup. The
+subsystem should allocate its subsystem state object for the passed
+cgroup, returning a pointer to the new object on success or a
+negative error code. On success, the subsystem pointer should point to
+a structure of type cgroup_subsys_state (typically embedded in a
+larger subsystem-specific object), which will be initialized by the
+cgroup system. Note that this will be called at initialization to
+create the root subsystem state for this subsystem; this case can be
+identified by the passed cgroup object having a NULL parent (since
+it's the root of the hierarchy) and may be an appropriate place for
+initialization code.
+
+void destroy(struct cgroup *cont)
+LL=cgroup_mutex
+
+The cgroup system is about to destroy the passed cgroup; the
+subsystem should do any necessary cleanup
+

+int can_attach(struct cgroup_subsys *ss, struct cgroup *cont,
+	struct task_struct *task)
+LL=cgroup_mutex
+
+Called prior to moving a task into a cgroup; if the subsystem
+returns an error, this will abort the attach operation. If a NULL
+task is passed, then a successful result indicates that *any*
+unspecified task can be moved into the cgroup. Note that this isn't
+called on a fork. If this method returns 0 (success) then this should
+remain valid while the caller holds cgroup_mutex.
+
+void attach(struct cgroup_subsys *ss, struct cgroup *cont,
+	struct cgroup *old_cont, struct task_struct *task)
+LL=cgroup_mutex
+
+
+Called after the task has been attached to the cgroup, to allow any
+post-attachment activity that requires memory allocations or blocking.
+
+void fork(struct cgroup_subsy *ss, struct task_struct *task)
+LL=callback_mutex, maybe read_lock(tasklist_lock)
+
+Called when a task is forked into a cgroup. Also called during
+registration for all existing tasks.
+
+void exit(struct cgroup_subsys *ss, struct task_struct *task)
+LL=callback_mutex
+
+Called during task exit
+
+int populate(struct cgroup_subsys *ss, struct cgroup *cont)
+LL=none
+
+Called after creation of a cgroup to allow a subsystem to populate
+the cgroup directory with file entries. The subsystem should make
+calls to cgroup_add_file() with objects of type cftype (see
+include/linux/cgroup.h for details). Note that although this
+method can return an error code, the error code is currently not
+always handled well.
+
+void bind(struct cgroup_subsys *ss, struct cgroup *root)
+LL=callback_mutex
+
+Called when a cgroup subsystem is rebound to a different hierarchy
+and root cgroup. Currently this will only involve movement between
+the default hierarchy (which never has sub-cgroups) and a hierarchy
+that is being created/destroyed (and hence has no sub-cgroups).
+

```
+4. Questions
+============
+
+Q: what's up with this '/bin/echo' ?
+A: bash's builtin 'echo' command does not check calls to write() against
+   errors. If you use it in the cgroup file system, you won't be
+   able to tell whether a command succeeded or failed.
+
+Q: When I attach processes, only the first of the line gets really attached !
+A: We can only return one error code per call to write(). So you should also
+   put only ONE pid.
+
diff -puN /dev/null include/linux/cgroup.h
--- /dev/null
+++ a/include/linux/cgroup.h
@@ -0,0 +1,214 @@
+#ifndef _LINUX_CGROUP_H
+#define _LINUX_CGROUP_H
+/*
+ *  cgroup interface
+ *
+ *  Copyright (C) 2003 BULL SA
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ */
+
+#include <linux/sched.h>
+#include <linux/kref.h>
+#include <linux/cpumask.h>
+#include <linux/nodemask.h>
+#include <linux/rcupdate.h>
+
+#ifdef CONFIG_CGROUPS
+
+struct cgroupfs_root;
+struct cgroup_subsys;
+struct inode;
+
+extern int cgroup_init_early(void);
+extern int cgroup_init(void);
+extern void cgroup_init_smp(void);
+extern void cgroup_lock(void);
+extern void cgroup_unlock(void);
+
+/* Per-subsystem/per-cgroup state maintained by the system. */
+struct cgroup_subsys_state {
+ /* The cgroup that this subsystem is attached to. Useful
+  * for subsystems that want to know about the cgroup
```

```
+  * hierarchy structure */
+ struct cgroup *cgroup;
+
+ /* State maintained by the cgroup system to allow
+  * subsystems to be "busy". Should be accessed via css_get()
+  * and css_put() */
+
+ atomic_t refcnt;
+
+ unsigned long flags;
+};
+
+/* bits in struct cgroup_subsys_state flags field */
+enum {
+ CSS_ROOT, /* This CSS is the root of the subsystem */
+};
+
+/*
+ * Call css_get() to hold a reference on the cgroup;
+ *
+ */
+
+static inline void css_get(struct cgroup_subsys_state *css)
+{
+ /* We don't need to reference count the root state */
+ if (!test_bit(CSS_ROOT, &css->flags))
+  atomic_inc(&css->refcnt);
+}
+/*
+ * css_put() should be called to release a reference taken by
+ * css_get()
+ */
+
+static inline void css_put(struct cgroup_subsys_state *css)
+{
+ if (!test_bit(CSS_ROOT, &css->flags))
+  atomic_dec(&css->refcnt);
+}
+
+struct cgroup {
+ unsigned long flags;  /* "unsigned long" so bitops work */
+
+ /* count users of this cgroup. >0 means busy, but doesn't
+  * necessarily indicate the number of tasks in the
+  * cgroup */
+ atomic_t count;
+
+ /*
```

```
+  * We link our 'sibling' struct into our parent's 'children'.
+  * Our children link their 'sibling' into our 'children'.
+  */
+ struct list_head sibling; /* my parent's children */
+ struct list_head children; /* my children */
+
+ struct cgroup *parent; /* my parent */
+ struct dentry *dentry;    /* cgroup fs entry */
+
+ /* Private pointers for each registered subsystem */
+ struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
+
+ struct cgroupfs_root *root;
+ struct cgroup *top_cgroup;
+};
+
+/* struct cftype:
+ *
+ * The files in the cgroup 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 cgroup to use in file->f_dentry->d_parent->d_fsdata
+ * - the 'cftype' of the file is file->f_dentry->d_fsdata
+ */
+
+#define MAX_CFTYPE_NAME 64
+struct cftype {
+ /* By convention, the name should begin with the name of the
+  * subsystem, followed by a period */
+ char name[MAX_CFTYPE_NAME];
+ int private;
+ int (*open) (struct inode *inode, struct file *file);
+ ssize_t (*read) (struct cgroup *cont, struct cftype *cft,
+    struct file *file,
+    char __user *buf, size_t nbytes, loff_t *ppos);
+ /*
+  * read_uint() is a shortcut for the common case of returning a
+  * single integer. Use it in place of read()
+  */
+ u64 (*read_uint) (struct cgroup *cont, struct cftype *cft);
+ ssize_t (*write) (struct cgroup *cont, struct cftype *cft,
+     struct file *file,
+     const char __user *buf, size_t nbytes, loff_t *ppos);
+ int (*release) (struct inode *inode, struct file *file);
```

```
+};
+
+/* Add a new file to the given cgroup directory. Should only be
+ * called by subsystems from within a populate() method */
+int cgroup_add_file(struct cgroup *cont, struct cgroup_subsys *subsys,
+         const struct cftype *cft);
+
+/* Add a set of new files to the given cgroup directory. Should
+ * only be called by subsystems from within a populate() method */
+int cgroup_add_files(struct cgroup *cont,
+   struct cgroup_subsys *subsys,
+   const struct cftype cft[],
+   int count);
+
+int cgroup_is_removed(const struct cgroup *cont);
+
+int cgroup_path(const struct cgroup *cont, char *buf, int buflen);
+
+/* Return true if the cgroup is a descendant of the current cgroup */
+int cgroup_is_descendant(const struct cgroup *cont);
+
+/* Control Group subsystem type. See Documentation/cgroups.txt for details */
+
+struct cgroup_subsys {
+ struct cgroup_subsys_state *(*create)(struct cgroup_subsys *ss,
+        struct cgroup *cont);
+ void (*destroy)(struct cgroup_subsys *ss, struct cgroup *cont);
+ int (*can_attach)(struct cgroup_subsys *ss,
+     struct cgroup *cont, struct task_struct *tsk);
+ void (*attach)(struct cgroup_subsys *ss, struct cgroup *cont,
+   struct cgroup *old_cont, struct task_struct *tsk);
+ void (*fork)(struct cgroup_subsys *ss, struct task_struct *task);
+ void (*exit)(struct cgroup_subsys *ss, struct task_struct *task);
+ int (*populate)(struct cgroup_subsys *ss,
+   struct cgroup *cont);
+ void (*bind)(struct cgroup_subsys *ss, struct cgroup *root);
+ int subsys_id;
+ int active;
+ int early_init;
+#define MAX_CGROUP_TYPE_NAMELEN 32
+ const char *name;
+
+ /* Protected by RCU */
+ struct cgroupfs_root *root;
+
+ struct list_head sibling;
+
+ void *private;
```

```
+};
+
+#define SUBSYS(_x) extern struct cgroup_subsys _x ## _subsys;
+#include <linux/cgroup_subsys.h>
+#undef SUBSYS
+
+static inline struct cgroup_subsys_state *cgroup_subsys_state(
+ struct cgroup *cont, int subsys_id)
+{
+ return cont->subsys[subsys_id];
+}
+
+static inline struct cgroup_subsys_state *task_subsys_state(
+ struct task_struct *task, int subsys_id)
+{
+ return rcu_dereference(task->cgroups.subsys[subsys_id]);
+}
+
+static inline struct cgroup* task_cgroup(struct task_struct *task,
+            int subsys_id)
+{
+ return task_subsys_state(task, subsys_id)->cgroup;
+}
+
+int cgroup_path(const struct cgroup *cont, char *buf, int buflen);
+
+#else /* !CONFIG_CGROUPS */
+
+static inline int cgroup_init_early(void) { return 0; }
+static inline int cgroup_init(void) { return 0; }
+static inline void cgroup_init_smp(void) {}
+
+static inline void cgroup_lock(void) {}
+static inline void cgroup_unlock(void) {}
+
+#endif /* !CONFIG_CGROUPS */
+
+#endif /* _LINUX_CGROUP_H */
diff -puN /dev/null include/linux/cgroup_subsys.h
--- /dev/null
+++ a/include/linux/cgroup_subsys.h
@@ -0,0 +1,10 @@
+/* Add subsystem definitions of the form SUBSYS(<name>) in this
+ * file. Surround each one by a line of comment markers so that
+ * patches don't collide
+ */
+
+/* */
```

```
+
+/* */
+
+/* */
diff -puN include/linux/magic.h~task-cgroupsv11-basic-task-cgroup-framework
include/linux/magic.h
--- a/include/linux/magic.h~task-cgroupsv11-basic-task-cgroup-framework
+++ a/include/linux/magic.h
@@ -40,5 +40,6 @@

 #define SMB_SUPER_MAGIC  0x517B
 #define USBDEVICE_SUPER_MAGIC 0x9fa2
+#define CGROUP_SUPER_MAGIC 0x27e0eb

 #endif /* __LINUX_MAGIC_H__ */
diff -puN include/linux/sched.h~task-cgroupsv11-basic-task-cgroup-framework
include/linux/sched.h
--- a/include/linux/sched.h~task-cgroupsv11-basic-task-cgroup-framework
+++ a/include/linux/sched.h
@@ -861,6 +861,34 @@ struct sched_entity {
 #endif
 };

+#ifdef CONFIG_CGROUPS
+
+#define SUBSYS(_x) _x ## _subsys_id,
+enum cgroup_subsys_id {
+#include <linux/cgroup_subsys.h>
+ CGROUP_SUBSYS_COUNT
+};
+#undef SUBSYS
+
+/* A css_set is a structure holding pointers to a set of
+ * cgroup_subsys_state objects.
+ */
+
+struct css_set {
+
+ /* Set of subsystem states, one for each subsystem. NULL for
+  * subsystems that aren't part of this hierarchy. These
+  * pointers reduce the number of dereferences required to get
+  * from a task to its state for a given cgroup, but result
+  * in increased space usage if tasks are in wildly different
+  * groupings across different hierarchies. This array is
+  * immutable after creation */
+ struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
+
+};
```

```
+
+#endif /* CONFIG_CGROUPS */
+
 struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
@@ -1096,6 +1124,9 @@ struct task_struct {
  int cpuset_mems_generation;
  int cpuset_mem_spread_rotor;
 #endif
+#ifdef CONFIG_CGROUPS
+ struct css_set cgroups;
+#endif
 #ifdef CONFIG_FUTEX
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
@@ -1585,7 +1616,8 @@ static inline int thread_group_empty(str
 /*
  * Protects ->fs, ->files, ->mm, ->group_info, ->comm, keyring
  * subscriptions and synchronises with wait4().  Also used in procfs.  Also
- * pins the final release of task.io_context.  Also protects ->cpuset.
+ * pins the final release of task.io_context.  Also protects ->cpuset and
+ * ->cgroup.subsys[].
  *
  * Nests both inside and outside of read_lock(&tasklist_lock).
  * It must not be nested with write_lock_irq(&tasklist_lock),
diff -puN init/Kconfig~task-cgroupsv11-basic-task-cgroup-framework init/Kconfig
--- a/init/Kconfig~task-cgroupsv11-basic-task-cgroup-framework
+++ a/init/Kconfig
@@ -274,6 +274,14 @@ config LOG_BUF_SHIFT
       13 =>  8 KB
       12 =>  4 KB

+config CGROUPS
+ bool "Control Group support"
+ help
+   This option will let you use process cgroup subsystems
+   such as Cpusets
+
+   Say N if unsure.
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP
diff -puN init/main.c~task-cgroupsv11-basic-task-cgroup-framework init/main.c
--- a/init/main.c~task-cgroupsv11-basic-task-cgroup-framework
+++ a/init/main.c
@@ -39,6 +39,7 @@
```

```
 #include <linux/writeback.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/cgroup.h>
 #include <linux/efi.h>
 #include <linux/tick.h>
 #include <linux/interrupt.h>
@@ -523,6 +524,7 @@ asmlinkage void __init start_kernel(void
   */
  unwind_init();
  lockdep_init();
+ cgroup_init_early();

  local_irq_disable();
  early_boot_irqs_off();
@@ -640,6 +642,7 @@ asmlinkage void __init start_kernel(void
 #ifdef CONFIG_PROC_FS
  proc_root_init();
 #endif
+ cgroup_init();
  cpuset_init();
  taskstats_init_early();
  delayacct_init();
diff -puN kernel/Makefile~task-cgroupsv11-basic-task-cgroup-framework kernel/Makefile
--- a/kernel/Makefile~task-cgroupsv11-basic-task-cgroup-framework
+++ a/kernel/Makefile
@@ -38,6 +38,7 @@ obj-$(CONFIG_PM) += power/
 obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
+obj-$(CONFIG_CGROUPS) += cgroup.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
diff -puN /dev/null kernel/cgroup.c
--- /dev/null
+++ a/kernel/cgroup.c
@@ -0,0 +1,1199 @@
+/*
+ *  kernel/cgroup.c
+ *
+ *  Generic process-grouping system.
+ *
+ *  Based originally on the cpuset system, extracted by Paul Menage
+ *  Copyright (C) 2006 Google, Inc
+ *
+ *  Copyright notices from the original cpuset code:
+ *  --------------------------------------------------
```

```
+ *  Copyright (C) 2003 BULL SA.
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ *  Portions derived from Patrick Mochel's sysfs code.
+ *  sysfs is Copyright (c) 2001-3 Patrick Mochel
+ *
+ *  2003-10-10 Written by Simon Derr.
+ *  2003-10-22 Updates by Stephen Hemminger.
+ *  2004 May-July Rework by Paul Jackson.
+ *  ---------------------------------------------------
+ *
+ *  This file is subject to the terms and conditions of the GNU General Public
+ *  License.  See the file COPYING in the main directory of the Linux
+ *  distribution for more details.
+ */
+
+#include <linux/cgroup.h>
+#include <linux/errno.h>
+#include <linux/fs.h>
+#include <linux/kernel.h>
+#include <linux/list.h>
+#include <linux/mm.h>
+#include <linux/mutex.h>
+#include <linux/mount.h>
+#include <linux/pagemap.h>
+#include <linux/rcupdate.h>
+#include <linux/sched.h>
+#include <linux/seq_file.h>
+#include <linux/slab.h>
+#include <linux/magic.h>
+#include <linux/spinlock.h>
+#include <linux/string.h>
+
+#include <asm/atomic.h>
+
+/* Generate an array of cgroup subsystem pointers */
+#define SUBSYS(_x) &_x ## _subsys,
+
+static struct cgroup_subsys *subsys[] = {
+#include <linux/cgroup_subsys.h>
+};
+
+/*
+ * A cgroupfs_root represents the root of a cgroup hierarchy,
+ * and may be associated with a superblock to form an active
+ * hierarchy
+ */
+struct cgroupfs_root {
```

```
+ struct super_block *sb;
+
+ /*
+  * The bitmask of subsystems intended to be attached to this
+  * hierarchy
+  */
+ unsigned long subsys_bits;
+
+ /* The bitmask of subsystems currently attached to this hierarchy */
+ unsigned long actual_subsys_bits;
+
+ /* A list running through the attached subsystems */
+ struct list_head subsys_list;
+
+ /* The root cgroup for this hierarchy */
+ struct cgroup top_cgroup;
+
+ /* Tracks how many cgroups are currently defined in hierarchy.*/
+ int number_of_cgroups;
+
+ /* A list running through the mounted hierarchies */
+ struct list_head root_list;
+
+ /* Hierarchy-specific flags */
+ unsigned long flags;
+};
+
+
+/*
+ * The "rootnode" hierarchy is the "dummy hierarchy", reserved for the
+ * subsystems that are otherwise unattached - it never has more than a
+ * single cgroup, and all tasks are part of that cgroup.
+ */
+static struct cgroupfs_root rootnode;
+
+/* The list of hierarchy roots */
+
+static LIST_HEAD(roots);
+
+/* dummytop is a shorthand for the dummy hierarchy's top cgroup */
+#define dummytop (&rootnode.top_cgroup)
+
+/* This flag indicates whether tasks in the fork and exit paths should
+ * take callback_mutex and check for fork/exit handlers to call. This
+ * avoids us having to do extra work in the fork/exit path if none of the
+ * subsystems need to be called.
+ */
+static int need_forkexit_callback;
```

```
+
+/* bits in struct cgroup flags field */
+enum {
+ CONT_REMOVED,
+};
+
+/* convenient tests for these bits */
+inline int cgroup_is_removed(const struct cgroup *cont)
+{
+ return test_bit(CONT_REMOVED, &cont->flags);
+}
+
+/* bits in struct cgroupfs_root flags field */
+enum {
+ ROOT_NOPREFIX, /* mounted subsystems have no named prefix */
+};
+
+/*
+ * for_each_subsys() allows you to iterate on each subsystem attached to
+ * an active hierarchy
+ */
+#define for_each_subsys(_root, _ss) \
+list_for_each_entry(_ss, &_root->subsys_list, sibling)
+
+/* for_each_root() allows you to iterate across the active hierarchies */
+#define for_each_root(_root) \
+list_for_each_entry(_root, &roots, root_list)
+
+/*
+ * There is one global cgroup mutex. We also require taking
+ * task_lock() when dereferencing a task's cgroup subsys pointers.
+ * See "The task_lock() exception", at the end of this comment.
+ *
+ * A task must hold cgroup_mutex to modify cgroups.
+ *
+ * Any task can increment and decrement the count field without lock.
+ * So in general, code holding cgroup_mutex can't rely on the count
+ * field not changing.  However, if the count goes to zero, then only
+ * attach_task() can increment it again.  Because a count of zero
+ * means that no tasks are currently attached, therefore there is no
+ * way a task attached to that cgroup can fork (the other way to
+ * increment the count).  So code holding cgroup_mutex can safely
+ * assume that if the count is zero, it will stay zero. Similarly, if
+ * a task holds cgroup_mutex on a cgroup with zero count, it
+ * knows that the cgroup won't be removed, as cgroup_rmdir()
+ * needs that mutex.
+ *
+ * The cgroup_common_file_write handler for operations that modify
```

```
+ * the cgroup hierarchy holds cgroup_mutex across the entire operation,
+ * single threading all such cgroup modifications across the system.
+ *
+ * The fork and exit callbacks cgroup_fork() and cgroup_exit(), don't
+ * (usually) take cgroup_mutex.  These are the two most performance
+ * critical pieces of code here.  The exception occurs on cgroup_exit(),
+ * when a task in a notify_on_release cgroup exits.  Then cgroup_mutex
+ * is taken, and if the cgroup count is zero, a usermode call made
+ * to /sbin/cgroup_release_agent with the name of the cgroup (path
+ * relative to the root of cgroup file system) as the argument.
+ *
+ * A cgroup can only be deleted if both its 'count' of using tasks
+ * is zero, and its list of 'children' cgroups is empty.  Since all
+ * tasks in the system use _some_ cgroup, and since there is always at
+ * least one task in the system (init, pid == 1), therefore, top_cgroup
+ * always has either children cgroups and/or using tasks.  So we don't
+ * need a special hack to ensure that top_cgroup cannot be deleted.
+ *
+ * The task_lock() exception
+ *
+ * The need for this exception arises from the action of
+ * attach_task(), which overwrites one tasks cgroup pointer with
+ * another.  It does so using cgroup_mutexe, however there are
+ * several performance critical places that need to reference
+ * task->cgroup without the expense of grabbing a system global
+ * mutex.  Therefore except as noted below, when dereferencing or, as
+ * in attach_task(), modifying a task'ss cgroup pointer we use
+ * task_lock(), which acts on a spinlock (task->alloc_lock) already in
+ * the task_struct routinely used for such matters.
+ *
+ * P.S.  One more locking exception.  RCU is used to guard the
+ * update of a tasks cgroup pointer by attach_task()
+ */
+
+static DEFINE_MUTEX(cgroup_mutex);
+
+/**
+ * cgroup_lock - lock out any changes to cgroup structures
+ *
+ */
+
+void cgroup_lock(void)
+{
+ mutex_lock(&cgroup_mutex);
+}
+
+/**
+ * cgroup_unlock - release lock on cgroup changes
```

```
+ *
+ * Undo the lock taken in a previous cgroup_lock() call.
+ */
+
+void cgroup_unlock(void)
+{
+ mutex_unlock(&cgroup_mutex);
+}
+
+/*
+ * A couple of forward declarations required, due to cyclic reference loop:
+ * cgroup_mkdir -> cgroup_create -> cgroup_populate_dir ->
+ * cgroup_add_file -> cgroup_create_file -> cgroup_dir_inode_operations
+ * -> cgroup_mkdir.
+ */
+
+static int cgroup_mkdir(struct inode *dir, struct dentry *dentry, int mode);
+static int cgroup_rmdir(struct inode *unused_dir, struct dentry *dentry);
+static int cgroup_populate_dir(struct cgroup *cont);
+static struct inode_operations cgroup_dir_inode_operations;
+
+static struct inode *cgroup_new_inode(mode_t mode, struct super_block *sb)
+{
+ struct inode *inode = new_inode(sb);
+ static struct backing_dev_info cgroup_backing_dev_info = {
+  .capabilities = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK,
+ };
+
+ 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 = &cgroup_backing_dev_info;
+ }
+ return inode;
+}
+
+static void cgroup_diput(struct dentry *dentry, struct inode *inode)
+{
+ /* is dentry a directory ? if so, kfree() associated cgroup */
+ if (S_ISDIR(inode->i_mode)) {
+  struct cgroup *cont = dentry->d_fsdata;
+  BUG_ON(!(cgroup_is_removed(cont)));
+  kfree(cont);
+ }
+ iput(inode);
```

```
+}
+
+static struct dentry *cgroup_get_dentry(struct dentry *parent,
+        const char *name)
+{
+ struct dentry *d = lookup_one_len(name, parent, strlen(name));
+ static struct dentry_operations cgroup_dops = {
+  .d_iput = cgroup_diput,
+ };
+
+ if (!IS_ERR(d))
+  d->d_op = &cgroup_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);
+}
+
+static void cgroup_clear_directory(struct dentry *dentry)
+{
+ struct list_head *node;
+
+ BUG_ON(!mutex_is_locked(&dentry->d_inode->i_mutex));
+ spin_lock(&dcache_lock);
+ node = dentry->d_subdirs.next;
+ while (node != &dentry->d_subdirs) {
+  struct dentry *d = list_entry(node, struct dentry, d_u.d_child);
+  list_del_init(node);
+  if (d->d_inode) {
+   /* This should never be called on a cgroup
+    * directory with child cgroups */
+   BUG_ON(d->d_inode->i_mode & S_IFDIR);
+   d = dget_locked(d);
+   spin_unlock(&dcache_lock);
+   d_delete(d);
+   simple_unlink(dentry->d_inode, d);
+   dput(d);
+   spin_lock(&dcache_lock);
+  }
+  node = dentry->d_subdirs.next;
+ }
+ spin_unlock(&dcache_lock);
```

```
+}
+
+/*
+ * NOTE : the dentry must have been dget()'ed
+ */
+static void cgroup_d_remove_dir(struct dentry *dentry)
+{
+ cgroup_clear_directory(dentry);
+
+ spin_lock(&dcache_lock);
+ list_del_init(&dentry->d_u.d_child);
+ spin_unlock(&dcache_lock);
+ remove_dir(dentry);
+}
+
+static int rebind_subsystems(struct cgroupfs_root *root,
+         unsigned long final_bits)
+{
+ unsigned long added_bits, removed_bits;
+ struct cgroup *cont = &root->top_cgroup;
+ int i;
+
+ removed_bits = root->actual_subsys_bits & ~final_bits;
+ added_bits = final_bits & ~root->actual_subsys_bits;
+ /* Check that any added subsystems are currently free */
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+  unsigned long long bit = 1ull << i;
+  struct cgroup_subsys *ss = subsys[i];
+  if (!(bit & added_bits))
+   continue;
+  if (ss->root != &rootnode) {
+   /* Subsystem isn't free */
+   return -EBUSY;
+  }
+ }
+
+ /* Currently we don't handle adding/removing subsystems when
+  * any child cgroups exist. This is theoretically supportable
+  * but involves complex error handling, so it's being left until
+  * later */
+ if (!list_empty(&cont->children))
+  return -EBUSY;
+
+ /* Process each subsystem */
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+  struct cgroup_subsys *ss = subsys[i];
+  unsigned long bit = 1UL << i;
+  if (bit & added_bits) {
```

```
+   /* We're binding this subsystem to this hierarchy */
+   BUG_ON(cont->subsys[i]);
+   BUG_ON(!dummytop->subsys[i]);
+   BUG_ON(dummytop->subsys[i]->cgroup != dummytop);
+   cont->subsys[i] = dummytop->subsys[i];
+   cont->subsys[i]->cgroup = cont;
+   list_add(&ss->sibling, &root->subsys_list);
+   rcu_assign_pointer(ss->root, root);
+   if (ss->bind)
+    ss->bind(ss, cont);
+
+  } else if (bit & removed_bits) {
+   /* We're removing this subsystem */
+   BUG_ON(cont->subsys[i] != dummytop->subsys[i]);
+   BUG_ON(cont->subsys[i]->cgroup != cont);
+   if (ss->bind)
+    ss->bind(ss, dummytop);
+   dummytop->subsys[i]->cgroup = dummytop;
+   cont->subsys[i] = NULL;
+   rcu_assign_pointer(subsys[i]->root, &rootnode);
+   list_del(&ss->sibling);
+  } else if (bit & final_bits) {
+   /* Subsystem state should already exist */
+   BUG_ON(!cont->subsys[i]);
+  } else {
+   /* Subsystem state shouldn't exist */
+   BUG_ON(cont->subsys[i]);
+  }
+ }
+ root->subsys_bits = root->actual_subsys_bits = final_bits;
+ synchronize_rcu();
+
+ return 0;
+}
+
+static int cgroup_show_options(struct seq_file *seq, struct vfsmount *vfs)
+{
+ struct cgroupfs_root *root = vfs->mnt_sb->s_fs_info;
+ struct cgroup_subsys *ss;
+
+ mutex_lock(&cgroup_mutex);
+ for_each_subsys(root, ss)
+  seq_printf(seq, ",%s", ss->name);
+ if (test_bit(ROOT_NOPREFIX, &root->flags))
+  seq_puts(seq, ",noprefix");
+ mutex_unlock(&cgroup_mutex);
+ return 0;
+}
```

```
+
+struct cgroup_sb_opts {
+ unsigned long subsys_bits;
+ unsigned long flags;
+};
+
+/* Convert a hierarchy specifier into a bitmask of subsystems and
+ * flags. */
+static int parse_cgroupfs_options(char *data,
+         struct cgroup_sb_opts *opts)
+{
+ char *token, *o = data ?: "all";
+
+ opts->subsys_bits = 0;
+ opts->flags = 0;
+
+ while ((token = strsep(&o, ",")) != NULL) {
+  if (!*token)
+   return -EINVAL;
+  if (!strcmp(token, "all")) {
+   opts->subsys_bits = (1 << CGROUP_SUBSYS_COUNT) - 1;
+  } else if (!strcmp(token, "noprefix")) {
+   set_bit(ROOT_NOPREFIX, &opts->flags);
+  } else {
+   struct cgroup_subsys *ss;
+   int i;
+   for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+    ss = subsys[i];
+    if (!strcmp(token, ss->name)) {
+     set_bit(i, &opts->subsys_bits);
+     break;
+    }
+   }
+   if (i == CGROUP_SUBSYS_COUNT)
+    return -ENOENT;
+  }
+ }
+
+ /* We can't have an empty hierarchy */
+ if (!opts->subsys_bits)
+  return -EINVAL;
+
+ return 0;
+}
+
+static int cgroup_remount(struct super_block *sb, int *flags, char *data)
+{
+ int ret = 0;
```

```
+ struct cgroupfs_root *root = sb->s_fs_info;
+ struct cgroup *cont = &root->top_cgroup;
+ struct cgroup_sb_opts opts;
+
+ mutex_lock(&cont->dentry->d_inode->i_mutex);
+ mutex_lock(&cgroup_mutex);
+
+ /* See what subsystems are wanted */
+ ret = parse_cgroupfs_options(data, &opts);
+ if (ret)
+  goto out_unlock;
+
+ /* Don't allow flags to change at remount */
+ if (opts.flags != root->flags) {
+  ret = -EINVAL;
+  goto out_unlock;
+ }
+
+ ret = rebind_subsystems(root, opts.subsys_bits);
+
+ /* (re)populate subsystem files */
+ if (!ret)
+  cgroup_populate_dir(cont);
+
+ out_unlock:
+ mutex_unlock(&cgroup_mutex);
+ mutex_unlock(&cont->dentry->d_inode->i_mutex);
+ return ret;
+}
+
+static struct super_operations cgroup_ops = {
+ .statfs = simple_statfs,
+ .drop_inode = generic_delete_inode,
+ .show_options = cgroup_show_options,
+ .remount_fs = cgroup_remount,
+};
+
+static void init_cgroup_root(struct cgroupfs_root *root)
+{
+ struct cgroup *cont = &root->top_cgroup;
+ INIT_LIST_HEAD(&root->subsys_list);
+ INIT_LIST_HEAD(&root->root_list);
+ root->number_of_cgroups = 1;
+ cont->root = root;
+ cont->top_cgroup = cont;
+ INIT_LIST_HEAD(&cont->sibling);
+ INIT_LIST_HEAD(&cont->children);
+}
```

```
+
+static int cgroup_test_super(struct super_block *sb, void *data)
+{
+ struct cgroupfs_root *new = data;
+ struct cgroupfs_root *root = sb->s_fs_info;
+
+ /* First check subsystems */
+ if (new->subsys_bits != root->subsys_bits)
+     return 0;
+
+ /* Next check flags */
+ if (new->flags != root->flags)
+  return 0;
+
+ return 1;
+}
+
+static int cgroup_set_super(struct super_block *sb, void *data)
+{
+ int ret;
+ struct cgroupfs_root *root = data;
+
+ ret = set_anon_super(sb, NULL);
+ if (ret)
+  return ret;
+
+ sb->s_fs_info = root;
+ root->sb = sb;
+
+ sb->s_blocksize = PAGE_CACHE_SIZE;
+ sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
+ sb->s_magic = CGROUP_SUPER_MAGIC;
+ sb->s_op = &cgroup_ops;
+
+ return 0;
+}
+
+static int cgroup_get_rootdir(struct super_block *sb)
+{
+ struct inode *inode =
+  cgroup_new_inode(S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR, sb);
+ struct dentry *dentry;
+
+ if (!inode)
+  return -ENOMEM;
+
+ inode->i_op = &simple_dir_inode_operations;
+ inode->i_fop = &simple_dir_operations;
```

```
+ inode->i_op = &cgroup_dir_inode_operations;
+ /* directories start off with i_nlink == 2 (for "." entry) */
+ inc_nlink(inode);
+ dentry = d_alloc_root(inode);
+ if (!dentry) {
+  iput(inode);
+  return -ENOMEM;
+ }
+ sb->s_root = dentry;
+ return 0;
+}
+
+static int cgroup_get_sb(struct file_system_type *fs_type,
+    int flags, const char *unused_dev_name,
+    void *data, struct vfsmount *mnt)
+{
+ struct cgroup_sb_opts opts;
+ int ret = 0;
+ struct super_block *sb;
+ struct cgroupfs_root *root;
+
+ /* First find the desired set of subsystems */
+ ret = parse_cgroupfs_options(data, &opts);
+ if (ret)
+  return ret;
+
+ root = kzalloc(sizeof(*root), GFP_KERNEL);
+ if (!root)
+  return -ENOMEM;
+
+ init_cgroup_root(root);
+ root->subsys_bits = opts.subsys_bits;
+ root->flags = opts.flags;
+
+ sb = sget(fs_type, cgroup_test_super, cgroup_set_super, root);
+
+ if (IS_ERR(sb)) {
+  kfree(root);
+  return PTR_ERR(sb);
+ }
+
+ if (sb->s_fs_info != root) {
+  /* Reusing an existing superblock */
+  BUG_ON(sb->s_root == NULL);
+  kfree(root);
+  root = NULL;
+ } else {
+  /* New superblock */
```

```
+  struct cgroup *cont = &root->top_cgroup;
+
+  BUG_ON(sb->s_root != NULL);
+
+  ret = cgroup_get_rootdir(sb);
+  if (ret)
+   goto drop_new_super;
+
+  mutex_lock(&cgroup_mutex);
+
+  ret = rebind_subsystems(root, root->subsys_bits);
+  if (ret == -EBUSY) {
+   mutex_unlock(&cgroup_mutex);
+   goto drop_new_super;
+  }
+
+  /* EBUSY should be the only error here */
+  BUG_ON(ret);
+
+  list_add(&root->root_list, &roots);
+
+  sb->s_root->d_fsdata = &root->top_cgroup;
+  root->top_cgroup.dentry = sb->s_root;
+
+  BUG_ON(!list_empty(&cont->sibling));
+  BUG_ON(!list_empty(&cont->children));
+  BUG_ON(root->number_of_cgroups != 1);
+
+  /*
+   * I believe that it's safe to nest i_mutex inside
+   * cgroup_mutex in this case, since no-one else can
+   * be accessing this directory yet. But we still need
+   * to teach lockdep that this is the case - currently
+   * a cgroupfs remount triggers a lockdep warning
+   */
+  mutex_lock(&cont->dentry->d_inode->i_mutex);
+  cgroup_populate_dir(cont);
+  mutex_unlock(&cont->dentry->d_inode->i_mutex);
+  mutex_unlock(&cgroup_mutex);
+ }
+
+ return simple_set_mnt(mnt, sb);
+
+ drop_new_super:
+ up_write(&sb->s_umount);
+ deactivate_super(sb);
+ return ret;
+}
```

```
+
+static void cgroup_kill_sb(struct super_block *sb) {
+ struct cgroupfs_root *root = sb->s_fs_info;
+ struct cgroup *cont = &root->top_cgroup;
+ int ret;
+
+ BUG_ON(!root);
+
+ BUG_ON(root->number_of_cgroups != 1);
+ BUG_ON(!list_empty(&cont->children));
+ BUG_ON(!list_empty(&cont->sibling));
+
+ mutex_lock(&cgroup_mutex);
+
+ /* Rebind all subsystems back to the default hierarchy */
+ ret = rebind_subsystems(root, 0);
+ /* Shouldn't be able to fail ... */
+ BUG_ON(ret);
+
+ if (!list_empty(&root->root_list))
+  list_del(&root->root_list);
+ mutex_unlock(&cgroup_mutex);
+
+ kfree(root);
+ kill_litter_super(sb);
+}
+
+static struct file_system_type cgroup_fs_type = {
+ .name = "cgroup",
+ .get_sb = cgroup_get_sb,
+ .kill_sb = cgroup_kill_sb,
+};
+
+static inline struct cgroup *__d_cont(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+static inline struct cftype *__d_cft(struct dentry *dentry)
+{
+ return dentry->d_fsdata;
+}
+
+/*
+ * Called with cgroup_mutex held.  Writes path of cgroup into buf.
+ * Returns 0 on success, -errno on error.
+ */
+int cgroup_path(const struct cgroup *cont, char *buf, int buflen)
```

```
+{
+ char *start;
+
+ start = buf + buflen;
+
+ *--start = '\0';
+ for (;;) {
+  int len = cont->dentry->d_name.len;
+  if ((start -= len) < buf)
+   return -ENAMETOOLONG;
+  memcpy(start, cont->dentry->d_name.name, len);
+  cont = cont->parent;
+  if (!cont)
+   break;
+  if (!cont->parent)
+   continue;
+  if (--start < buf)
+   return -ENAMETOOLONG;
+  *start = '/';
+ }
+ memmove(buf, start, buf + buflen - start);
+ return 0;
+}
+
+/* The various types of files and directories in a cgroup file system */
+
+enum cgroup_filetype {
+ FILE_ROOT,
+ FILE_DIR,
+ FILE_TASKLIST,
+};
+
+static ssize_t cgroup_file_write(struct file *file, const char __user *buf,
+      size_t nbytes, loff_t *ppos)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct cgroup *cont = __d_cont(file->f_dentry->d_parent);
+
+ if (!cft)
+  return -ENODEV;
+ if (!cft->write)
+  return -EINVAL;
+
+ return cft->write(cont, cft, file, buf, nbytes, ppos);
+}
+
+static ssize_t cgroup_read_uint(struct cgroup *cont, struct cftype *cft,
+       struct file *file,
```

```
+       char __user *buf, size_t nbytes,
+       loff_t *ppos)
+{
+ char tmp[64];
+ u64 val = cft->read_uint(cont, cft);
+ int len = sprintf(tmp, "%llu\n", (unsigned long long) val);
+
+ return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
+}
+
+static ssize_t cgroup_file_read(struct file *file, char __user *buf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct cftype *cft = __d_cft(file->f_dentry);
+ struct cgroup *cont = __d_cont(file->f_dentry->d_parent);
+
+ if (!cft)
+  return -ENODEV;
+
+ if (cft->read)
+  return cft->read(cont, cft, file, buf, nbytes, ppos);
+ if (cft->read_uint)
+  return cgroup_read_uint(cont, cft, file, buf, nbytes, ppos);
+ return -EINVAL;
+}
+
+static int cgroup_file_open(struct inode *inode, struct file *file)
+{
+ int err;
+ struct cftype *cft;
+
+ err = generic_file_open(inode, file);
+ if (err)
+  return err;
+
+ cft = __d_cft(file->f_dentry);
+ if (!cft)
+  return -ENODEV;
+ if (cft->open)
+  err = cft->open(inode, file);
+ else
+  err = 0;
+
+ return err;
+}
+
+static int cgroup_file_release(struct inode *inode, struct file *file)
+{
```

```
+ struct cftype *cft = __d_cft(file->f_dentry);
+ if (cft->release)
+  return cft->release(inode, file);
+ return 0;
+}
+
+/*
+ * cgroup_rename - Only allow simple rename of directories in place.
+ */
+static int cgroup_rename(struct inode *old_dir, struct dentry *old_dentry,
+       struct inode *new_dir, struct dentry *new_dentry)
+{
+ if (!S_ISDIR(old_dentry->d_inode->i_mode))
+  return -ENOTDIR;
+ if (new_dentry->d_inode)
+  return -EEXIST;
+ if (old_dir != new_dir)
+  return -EIO;
+ return simple_rename(old_dir, old_dentry, new_dir, new_dentry);
+}
+
+static struct file_operations cgroup_file_operations = {
+ .read = cgroup_file_read,
+ .write = cgroup_file_write,
+ .llseek = generic_file_llseek,
+ .open = cgroup_file_open,
+ .release = cgroup_file_release,
+};
+
+static struct inode_operations cgroup_dir_inode_operations = {
+ .lookup = simple_lookup,
+ .mkdir = cgroup_mkdir,
+ .rmdir = cgroup_rmdir,
+ .rename = cgroup_rename,
+};
+
+static int cgroup_create_file(struct dentry *dentry, int mode,
+    struct super_block *sb)
+{
+ struct inode *inode;
+
+ if (!dentry)
+  return -ENOENT;
+ if (dentry->d_inode)
+  return -EEXIST;
+
+ inode = cgroup_new_inode(mode, sb);
+ if (!inode)
```

```
+  return -ENOMEM;
+
+ if (S_ISDIR(mode)) {
+  inode->i_op = &cgroup_dir_inode_operations;
+  inode->i_fop = &simple_dir_operations;
+
+  /* start off with i_nlink == 2 (for "." entry) */
+  inc_nlink(inode);
+
+  /* start with the directory inode held, so that we can
+   * populate it without racing with another mkdir */
+  mutex_lock(&inode->i_mutex);
+ } else if (S_ISREG(mode)) {
+  inode->i_size = 0;
+  inode->i_fop = &cgroup_file_operations;
+ }
+
+ d_instantiate(dentry, inode);
+ dget(dentry); /* Extra count - pin the dentry in core */
+ return 0;
+}
+
+/*
+ * cgroup_create_dir - create a directory for an object.
+ * cont: the cgroup 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 cgroup directory. Will be copied.
+ * mode: mode to set on new directory.
+ */
+static int cgroup_create_dir(struct cgroup *cont, struct dentry *dentry,
+    int mode)
+{
+ struct dentry *parent;
+ int error = 0;
+
+ parent = cont->parent->dentry;
+ if (IS_ERR(dentry))
+  return PTR_ERR(dentry);
+ error = cgroup_create_file(dentry, S_IFDIR | mode, cont->root->sb);
+ if (!error) {
+  dentry->d_fsdata = cont;
+  inc_nlink(parent->d_inode);
+  cont->dentry = dentry;
+ }
+ dput(dentry);
+
+ return error;
```

```
+}
+
+int cgroup_add_file(struct cgroup *cont,
+         struct cgroup_subsys *subsys,
+         const struct cftype *cft)
+{
+ struct dentry *dir = cont->dentry;
+ struct dentry *dentry;
+ int error;
+
+ char name[MAX_CGROUP_TYPE_NAMELEN + MAX_CFTYPE_NAME + 2] = { 0 };
+ if (subsys && !test_bit(ROOT_NOPREFIX, &cont->root->flags)) {
+  strcpy(name, subsys->name);
+  strcat(name, ".");
+ }
+ strcat(name, cft->name);
+ BUG_ON(!mutex_is_locked(&dir->d_inode->i_mutex));
+ dentry = cgroup_get_dentry(dir, name);
+ if (!IS_ERR(dentry)) {
+  error = cgroup_create_file(dentry, 0644 | S_IFREG,
+       cont->root->sb);
+  if (!error)
+   dentry->d_fsdata = (void *)cft;
+  dput(dentry);
+ } else
+  error = PTR_ERR(dentry);
+ return error;
+}
+
+int cgroup_add_files(struct cgroup *cont,
+    struct cgroup_subsys *subsys,
+    const struct cftype cft[],
+    int count)
+{
+ int i, err;
+ for (i = 0; i < count; i++) {
+  err = cgroup_add_file(cont, subsys, &cft[i]);
+  if (err)
+   return err;
+ }
+ return 0;
+}
+
+static int cgroup_populate_dir(struct cgroup *cont)
+{
+ int err;
+ struct cgroup_subsys *ss;
+
```

```
+ /* First clear out any existing files */
+ cgroup_clear_directory(cont->dentry);
+
+ for_each_subsys(cont->root, ss) {
+  if (ss->populate && (err = ss->populate(ss, cont)) < 0)
+   return err;
+ }
+
+ return 0;
+}
+
+static void init_cgroup_css(struct cgroup_subsys_state *css,
+          struct cgroup_subsys *ss,
+          struct cgroup *cont)
+{
+ css->cgroup = cont;
+ atomic_set(&css->refcnt, 0);
+ css->flags = 0;
+ if (cont == dummytop)
+  set_bit(CSS_ROOT, &css->flags);
+ BUG_ON(cont->subsys[ss->subsys_id]);
+ cont->subsys[ss->subsys_id] = css;
+}
+
+/*
+ * cgroup_create - create a cgroup
+ * parent: cgroup that will be parent of the new cgroup.
+ * name:  name of the new cgroup. Will be strcpy'ed.
+ * mode:  mode to set on new inode
+ *
+ * Must be called with the mutex on the parent inode held
+ */
+
+static long cgroup_create(struct cgroup *parent, struct dentry *dentry,
+        int mode)
+{
+ struct cgroup *cont;
+ struct cgroupfs_root *root = parent->root;
+ int err = 0;
+ struct cgroup_subsys *ss;
+ struct super_block *sb = root->sb;
+
+ cont = kzalloc(sizeof(*cont), GFP_KERNEL);
+ if (!cont)
+  return -ENOMEM;
+
+ /* Grab a reference on the superblock so the hierarchy doesn't
+  * get deleted on unmount if there are child cgroups.  This
```

```
+  * can be done outside cgroup_mutex, since the sb can't
+  * disappear while someone has an open control file on the
+  * fs */
+ atomic_inc(&sb->s_active);
+
+ mutex_lock(&cgroup_mutex);
+
+ cont->flags = 0;
+ INIT_LIST_HEAD(&cont->sibling);
+ INIT_LIST_HEAD(&cont->children);
+
+ cont->parent = parent;
+ cont->root = parent->root;
+ cont->top_cgroup = parent->top_cgroup;
+
+ for_each_subsys(root, ss) {
+  struct cgroup_subsys_state *css = ss->create(ss, cont);
+  if (IS_ERR(css)) {
+   err = PTR_ERR(css);
+   goto err_destroy;
+  }
+  init_cgroup_css(css, ss, cont);
+ }
+
+ list_add(&cont->sibling, &cont->parent->children);
+ root->number_of_cgroups++;
+
+ err = cgroup_create_dir(cont, dentry, mode);
+ if (err < 0)
+  goto err_remove;
+
+ /* The cgroup directory was pre-locked for us */
+ BUG_ON(!mutex_is_locked(&cont->dentry->d_inode->i_mutex));
+
+ err = cgroup_populate_dir(cont);
+ /* If err < 0, we have a half-filled directory - oh well ;) */
+
+ mutex_unlock(&cgroup_mutex);
+ mutex_unlock(&cont->dentry->d_inode->i_mutex);
+
+ return 0;
+
+ err_remove:
+
+ list_del(&cont->sibling);
+ root->number_of_cgroups--;
+
+ err_destroy:
```

```
+
+ for_each_subsys(root, ss) {
+  if (cont->subsys[ss->subsys_id])
+   ss->destroy(ss, cont);
+ }
+
+ mutex_unlock(&cgroup_mutex);
+
+ /* Release the reference count that we took on the superblock */
+ deactivate_super(sb);
+
+ kfree(cont);
+ return err;
+}
+
+static int cgroup_mkdir(struct inode *dir, struct dentry *dentry, int mode)
+{
+ struct cgroup *c_parent = dentry->d_parent->d_fsdata;
+
+ /* the vfs holds inode->i_mutex already */
+ return cgroup_create(c_parent, dentry, mode | S_IFDIR);
+}
+
+static int cgroup_rmdir(struct inode *unused_dir, struct dentry *dentry)
+{
+ struct cgroup *cont = dentry->d_fsdata;
+ struct dentry *d;
+ struct cgroup *parent;
+ struct cgroup_subsys *ss;
+ struct super_block *sb;
+ struct cgroupfs_root *root;
+ int css_busy = 0;
+
+ /* the vfs holds both inode->i_mutex already */
+
+ mutex_lock(&cgroup_mutex);
+ if (atomic_read(&cont->count) != 0) {
+  mutex_unlock(&cgroup_mutex);
+  return -EBUSY;
+ }
+ if (!list_empty(&cont->children)) {
+  mutex_unlock(&cgroup_mutex);
+  return -EBUSY;
+ }
+
+ parent = cont->parent;
+ root = cont->root;
+ sb = root->sb;
```

```
+
+ /* Check the reference count on each subsystem. Since we
+  * already established that there are no tasks in the
+  * cgroup, if the css refcount is also 0, then there should
+  * be no outstanding references, so the subsystem is safe to
+  * destroy */
+ for_each_subsys(root, ss) {
+  struct cgroup_subsys_state *css;
+  css = cont->subsys[ss->subsys_id];
+  if (atomic_read(&css->refcnt)) {
+   css_busy = 1;
+   break;
+  }
+ }
+ if (css_busy) {
+  mutex_unlock(&cgroup_mutex);
+  return -EBUSY;
+ }
+
+ for_each_subsys(root, ss) {
+  if (cont->subsys[ss->subsys_id])
+   ss->destroy(ss, cont);
+ }
+
+ set_bit(CONT_REMOVED, &cont->flags);
+ /* delete my sibling from parent->children */
+ list_del(&cont->sibling);
+ spin_lock(&cont->dentry->d_lock);
+ d = dget(cont->dentry);
+ cont->dentry = NULL;
+ spin_unlock(&d->d_lock);
+
+ cgroup_d_remove_dir(d);
+ dput(d);
+ root->number_of_cgroups--;
+
+ mutex_unlock(&cgroup_mutex);
+ /* Drop the active superblock reference that we took when we
+  * created the cgroup */
+ deactivate_super(sb);
+ return 0;
+}
+
+static void cgroup_init_subsys(struct cgroup_subsys *ss)
+{
+ struct task_struct *g, *p;
+ struct cgroup_subsys_state *css;
+ printk(KERN_ERR "Initializing cgroup subsys %s\n", ss->name);
```

```
+
+ /* Create the top cgroup state for this subsystem */
+ ss->root = &rootnode;
+ css = ss->create(ss, dummytop);
+ /* We don't handle early failures gracefully */
+ BUG_ON(IS_ERR(css));
+ init_cgroup_css(css, ss, dummytop);
+
+ /* Update all tasks to contain a subsys pointer to this state
+  * - since the subsystem is newly registered, all tasks are in
+  * the subsystem's top cgroup. */
+
+  /* If this subsystem requested that it be notified with fork
+   * events, we should send it one now for every process in the
+   * system */
+
+ read_lock(&tasklist_lock);
+ init_task.cgroups.subsys[ss->subsys_id] = css;
+ if (ss->fork)
+  ss->fork(ss, &init_task);
+
+ do_each_thread(g, p) {
+  printk(KERN_INFO "Setting task %p css to %p (%d)\n", css, p, p->pid);
+  p->cgroups.subsys[ss->subsys_id] = css;
+  if (ss->fork)
+   ss->fork(ss, p);
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+
+ need_forkexit_callback |= ss->fork || ss->exit;
+
+ ss->active = 1;
+}
+
+/**
+ * cgroup_init_early - initialize cgroups at system boot, and
+ * initialize any subsystems that request early init.
+ */
+int __init cgroup_init_early(void)
+{
+ int i;
+ init_cgroup_root(&rootnode);
+ list_add(&rootnode.root_list, &roots);
+
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+  struct cgroup_subsys *ss = subsys[i];
+
+  BUG_ON(!ss->name);
```

```
+  BUG_ON(strlen(ss->name) > MAX_CGROUP_TYPE_NAMELEN);
+  BUG_ON(!ss->create);
+  BUG_ON(!ss->destroy);
+  if (ss->subsys_id != i) {
+   printk(KERN_ERR "Subsys %s id == %d\n",
+          ss->name, ss->subsys_id);
+   BUG();
+  }
+
+  if (ss->early_init)
+   cgroup_init_subsys(ss);
+ }
+ return 0;
+}
+
+/**
+ * cgroup_init - register cgroup filesystem and /proc file, and
+ * initialize any subsystems that didn't request early init.
+ */
+int __init cgroup_init(void)
+{
+ int err;
+ int i;
+
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+  struct cgroup_subsys *ss = subsys[i];
+  if (!ss->early_init)
+   cgroup_init_subsys(ss);
+ }
+
+ err = register_filesystem(&cgroup_fs_type);
+ if (err < 0)
+  goto out;
+
+out:
+ return err;
+}
_
```

--

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers