Subject: [PATCH 5/7] containers (V7): Resource Groups over generic containers
Posted by Paul Menage on Mon, 12 Feb 2007 08:15:26 GMT
View Forum Message <> Reply to Message

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

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

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

Index: container-2.6.20/include/linux/moduleparam.h
 ============================================================ =======
--- container-2.6.20.orig/include/linux/moduleparam.h
+++ container-2.6.20/include/linux/moduleparam.h
@@ -78,11 +78,17 @@ struct kparam_array
 /* Helper functions: type is byte, short, ushort, int, uint, long,
    ulong, charp, bool or invbool, or XXX if you define param_get_XXX,
    param_set_XXX and param_check_XXX. */
-#define module_param_named(name, value, type, perm)      \
- param_check_##type(name, &(value));       \
- module_param_call(name, param_set_##type, param_get_##type, &value, perm); \
+#define module_param_named_call(name, value, type, set, perm)  \
+ param_check_##type(name, &(value));    \
+ module_param_call(name, set, param_get_##type, &(value), perm); \
  __MODULE_PARM_TYPE(name, #type)
```

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

Index: container-2.6.20/include/linux/numtasks.h
 ===================================================================
--- /dev/null
+++ container-2.6.20/include/linux/numtasks.h
@@ -0,0 +1,28 @@
+/* numtasks.h - No. of tasks resource controller for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *
+ * Provides No. of tasks resource controller for Resource Groups
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+#ifndef _LINUX_NUMTASKS_H
+#define _LINUX_NUMTASKS_H
+
+#ifdef CONFIG_RES_GROUPS_NUMTASKS
+#include <linux/res_group_rc.h>
+
+extern int numtasks_allow_fork(struct task_struct *);
+
+#else /* CONFIG_RES_GROUPS_NUMTASKS */
+
+#define numtasks_allow_fork(task) (0)
+
+#endif /* CONFIG_RES_GROUPS_NUMTASKS */
+#endif /* _LINUX_NUMTASKS_H */
Index: container-2.6.20/include/linux/res_group.h
 ===================================================================
--- /dev/null
+++ container-2.6.20/include/linux/res_group.h
@@ -0,0 +1,87 @@
```

```
+/*
+ *  res_group.h - Header file to be used by Resource Groups
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003, 2004
+ *  (C) Shailabh Nagar,  IBM Corp. 2003, 2004
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *
+ * Provides data structures, macros and kernel APIs
+ *
+ * More details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+#ifndef _LINUX_RES_GROUP_H
+#define _LINUX_RES_GROUP_H
+
+#ifdef CONFIG_RES_GROUPS
+#include <linux/spinlock.h>
+#include <linux/list.h>
+#include <linux/kref.h>
+#include <linux/container.h>
+
+#define SHARE_UNCHANGED (-1) /* implicitly specified by userspace,
+      * never stored in a resource group'
+      * shares struct; never displayed */
+#define SHARE_UNSUPPORTED (-2) /* If the resource controller doesn't
+      * support user changing a shares value
+      * it sets the corresponding share
+      * value to UNSUPPORTED when it returns
+      * the newly allocated shares data
+      * structure */
+#define SHARE_DONT_CARE (-3)
+
+#define SHARE_DEFAULT_DIVISOR  (100)
+
+#define MAX_RES_CTLRS CONFIG_MAX_CONTAINER_SUBSYS /* max # of resource
controllers */
+#define MAX_DEPTH 5 /* max depth of hierarchy supported */
+
+#define NO_RES_GROUP  NULL
+#define NO_SHARE  NULL
+#define NO_RES_ID  MAX_RES_CTLRS /* Invalid ID */
+
```

```
+/*
+ * Share quantities are a child's fraction of the parent's resource
+ * specified by a divisor in the parent and a dividend in the child.
+ *
+ * Shares are represented as a relative quantity between parent and child
+ * to simplify locking when propagating modifications to the shares of a
+ * resource group. Only the parent and the children of the modified
+ * resource group need to be locked.
+*/
+struct res_shares {
+ /* shares only set by userspace */
+ int min_shares; /* minimun fraction of parent's resources allowed */
+ int max_shares; /* maximum fraction of parent's resources allowed */
+ int child_shares_divisor; /* >= 1, may not be DONT_CARE */
+
+ /*
+  * share values invisible to userspace.  adjusted when userspace
+  * sets shares
+  */
+ int unused_min_shares;
+  /* 0 <= unused_min_shares <= (child_shares_divisor -
+   *    Sum of min_shares of children)
+   */
+ int cur_max_shares; /* max(children's max_shares). need better name */
+
+ /* State maintained by container system - only relevant when
+  * this shares struct is the actual shares struct for a
+  * container */
+ struct container_subsys_state css;
+};
+
+/*
+ * Class is the grouping of tasks with shares of each resource that has
+ * registered a resource controller (see include/linux/res_group_rc.h).
+ */
+
+#define resource_group container
+
+#endif /* CONFIG_RES_GROUPS */
+#endif /* _LINUX_RES_GROUP_H */
Index: container-2.6.20/include/linux/res_group_rc.h
 ===================================================================
--- /dev/null
+++ container-2.6.20/include/linux/res_group_rc.h
@@ -0,0 +1,97 @@
+/*
+ *  res_group_rc.h - Header file to be used by Resource controllers of
+ *       Resource Groups
```

```
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003
+ *  (C) Shailabh Nagar,  IBM Corp. 2003
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *  (C) Vivek Kashyap , IBM Corp. 2004
+ *
+ * Provides data structures, macros and kernel API of Resource Groups for
+ * resource controllers.
+ *
+ * More details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+#ifndef _LINUX_RES_GROUP_RC_H
+#define _LINUX_RES_GROUP_RC_H
+
+#include <linux/res_group.h>
+#include <linux/container.h>
+
+struct res_group_cft {
+ struct cftype cft;
+ struct res_controller *ctlr;
+};
+
+struct res_controller {
+ struct container_subsys subsys;
+ struct res_group_cft shares_cft;
+ struct res_group_cft stats_cft;
+
+ const char *name;
+ unsigned int ctlr_id;
+
+ /*
+  * Keeps number of references to this controller structure. kref
+  * does not work as we want to be able to allow removal of a
+  * controller even when some resource group are still defined.
+  */
+ atomic_t count;
+
+ /*
+  * Allocate a new shares struct for this resource controller.
+  * Called when registering a resource controller with pre-existing
+  * resource groups and when new resource group is created by the user.
```

```
+  */
+ struct res_shares *(*alloc_shares_struct)(struct container *);
+ /* Corresponding free of shares struct for this resource controller */
+ void (*free_shares_struct)(struct res_shares *);
+
+ /* Notifies the controller when the shares are changed */
+ void (*shares_changed)(struct res_shares *);
+
+ /* resource statistics */
+ ssize_t (*show_stats)(struct res_shares *, char *, size_t);
+ int (*reset_stats)(struct res_shares *, const char *);
+
+ /*
+  * move_task is called when a task moves from one resource group to
+  * another. First parameter is the task that is moving, the second
+  * is the resource specific shares of the resource group the task
+  * was in, and the third is the shares of the resource group the
+  * task has moved to.
+  */
+ void (*move_task)(struct task_struct *, struct res_shares *,
+    struct res_shares *);
+};
+
+extern int register_controller(struct res_controller *);
+extern int unregister_controller(struct res_controller *);
+extern struct resource_group default_res_group;
+static inline int is_res_group_root(const struct resource_group *rgroup)
+{
+ return (rgroup->parent == NULL);
+}
+
+#define for_each_child(child, parent) \
+ list_for_each_entry(child, &parent->children, sibling)
+
+/* Get controller specific shares structure for the given resource group */
+static inline struct res_shares *get_controller_shares(
+ struct container *rgroup, struct res_controller *ctlr)
+{
+ if (rgroup && ctlr)
+  return container_of(rgroup->subsys[ctlr->subsys.subsys_id],
+        struct res_shares, css);
+ else
+  return NO_SHARE;
+}
+
+#endif /* _LINUX_RES_GROUP_RC_H */
Index: container-2.6.20/init/Kconfig
 ========================================================== =======
```

```
--- container-2.6.20.orig/init/Kconfig
+++ container-2.6.20/init/Kconfig
@@ -341,6 +341,28 @@ config TASK_IO_ACCOUNTING

    Say N if unsure.

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

Index: container-2.6.20/kernel/Makefile
 ===================================================== =======
--- container-2.6.20.orig/kernel/Makefile
+++ container-2.6.20/kernel/Makefile
@@ -52,6 +52,7 @@ obj-$(CONFIG_RELAY) += relay.o
 obj-$(CONFIG_UTS_NS) += utsname.o
 obj-$(CONFIG_TASK_DELAY_ACCT) += delayacct.o
 obj-$(CONFIG_TASKSTATS) += taskstats.o tsacct.o
+obj-$(CONFIG_RES_GROUPS) += res_group/

 ifneq ($(CONFIG_SCHED_NO_NO_OMIT_FRAME_POINTER),y)
 # According to Alan Modra <alan@linuxcare.com.au>, the -fno-omit-frame-pointer is
Index: container-2.6.20/kernel/fork.c
 ===================================================== =======
--- container-2.6.20.orig/kernel/fork.c
+++ container-2.6.20/kernel/fork.c
@@ -49,6 +49,7 @@
```

```
 #include <linux/delayacct.h>
 #include <linux/taskstats_kern.h>
 #include <linux/random.h>
+#include <linux/numtasks.h>

 #include <asm/pgtable.h>
 #include <asm/pgalloc.h>
@@ -1355,7 +1356,7 @@ long do_fork(unsigned long clone_flags,
       int __user *child_tidptr)
 {
  struct task_struct *p;
- int trace = 0;
+ int trace = 0, rc;
  struct pid *pid = alloc_pid();
  long nr;

@@ -1368,6 +1369,10 @@ long do_fork(unsigned long clone_flags,
    clone_flags |= CLONE_PTRACE;
  }

+ rc = numtasks_allow_fork(current);
+ if (rc)
+  return rc;
+
  p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr, child_tidptr, nr);
  /*
   * Do this prior waking up the new thread - the thread pointer
Index: container-2.6.20/kernel/res_group/Makefile
 ===================================================================== =======
--- /dev/null
+++ container-2.6.20/kernel/res_group/Makefile
@@ -0,0 +1,2 @@
+obj-y = res_group.o shares.o rgcs.o
+obj-$(CONFIG_RES_GROUPS_NUMTASKS) += numtasks.o
Index: container-2.6.20/kernel/res_group/local.h
 ===================================================================== =======
--- /dev/null
+++ container-2.6.20/kernel/res_group/local.h
@@ -0,0 +1,38 @@
+/*
+ * Contains function definitions that are local to the Resource Groups.
+ * NOT to be included by controllers.
+ */
+
+#include <linux/res_group_rc.h>
+
+extern struct res_controller *get_controller_by_name(const char *);
+extern struct res_controller *get_controller_by_id(unsigned int);
```

```
+extern void put_controller(struct res_controller *);
+extern struct resource_group *alloc_res_group(struct resource_group *,
+       const char *);
+extern int free_res_group(struct resource_group *);
+extern void release_res_group(struct kref *);
+extern int set_controller_shares(struct resource_group *,
+   struct res_controller *, const struct res_shares *);
+/* Set shares for the given resource group and resource to default values */
+extern void set_shares_to_default(struct resource_group *,
+      struct res_controller *);
+extern void res_group_teardown(void);
+extern int set_res_group(pid_t, struct resource_group *);
+extern void move_tasks_to_parent(struct resource_group *);
+
+ssize_t res_group_file_read(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       char __user *buf,
+       size_t nbytes, loff_t *ppos);
+ssize_t res_group_file_write(struct container *cont,
+        struct cftype *cft,
+        struct file *file,
+        const char __user *userbuf,
+        size_t nbytes, loff_t *ppos);
+
+enum {
+ RG_FILE_SHARES,
+ RG_FILE_STATS,
+};
Index: container-2.6.20/kernel/res_group/numtasks.c
 ===================================================================== =======
--- /dev/null
+++ container-2.6.20/kernel/res_group/numtasks.c
@@ -0,0 +1,467 @@
+/* numtasks.c - "Number of tasks" resource controller for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman,  IBM Corp. 2003-2006
+ *       (C) Matt Helsley, IBM Corp. 2006
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
```

```
+/*
+ * Resource controller for tracking number of tasks in a resource group.
+ */
+#include <linux/module.h>
+#include <linux/res_group_rc.h>
+#include <linux/numtasks.h>
+
+static const char res_ctlr_name[] = "numtasks";
+
+#define UNLIMITED INT_MAX
+#define DEF_TOTAL_NUM_TASKS UNLIMITED
+static int total_numtasks __read_mostly = DEF_TOTAL_NUM_TASKS;
+
+static struct resource_group *root_rgroup;
+static int total_cnt_alloc = 0;
+
+#define DEF_FORKRATE UNLIMITED
+#define DEF_FORKRATE_INTERVAL (1)
+static int forkrate __read_mostly = DEF_FORKRATE;
+static int forkrate_interval __read_mostly = DEF_FORKRATE_INTERVAL;
+
+struct numtasks {
+ struct res_shares shares;
+ int cnt_min_shares; /* num_tasks min_shares in local units */
+ int cnt_unused;  /* has to borrow if more than this is needed */
+ int cnt_max_shares; /* no tasks over this limit. */
+    /* Three above cnt_* fields are protected
+     * by resource group's group_lock */
+ atomic_t cnt_cur_alloc; /* current alloc from self */
+ atomic_t cnt_borrowed; /* borrowed from the parent */
+
+ /* stats */
+ int successes;
+ int failures;
+ int forkrate_failures;
+
+ /* Fork rate fields */
+ int forks_in_period;
+ unsigned long period_start;
+};
+
+struct res_controller numtasks_ctlr;
+
+static struct numtasks *get_shares_numtasks(struct res_shares *shares)
+{
+ if (shares)
+  return container_of(shares, struct numtasks, shares);
+ return NULL;
```

```
+}
+
+static struct numtasks *get_numtasks(struct resource_group *rgroup)
+{
+ return get_shares_numtasks(get_controller_shares(rgroup,
+       &numtasks_ctlr));
+}
+
+static struct resource_group *numtasks_rgroup(struct numtasks *nt)
+{
+ return nt->shares.css.container;
+}
+
+static inline int check_forkrate(struct numtasks *res)
+{
+ if (time_after(jiffies, res->period_start + forkrate_interval * HZ)) {
+  res->period_start = jiffies;
+  res->forks_in_period = 0;
+ }
+
+ if (res->forks_in_period >= forkrate) {
+    res->forkrate_failures++;
+  return -ENOSPC;
+ }
+ res->forks_in_period++;
+ return 0;
+}
+
+int numtasks_allow_fork(struct task_struct *task)
+{
+ int rc = 0;
+ struct numtasks *res;
+
+ /* task->container won't be deleted during an RCU critical section */
+ rcu_read_lock();
+
+ /* controller is not registered; no resource group is given */
+ if (numtasks_ctlr.ctlr_id == NO_RES_ID)
+  goto out;
+ res = get_numtasks(task_container(task, &numtasks_ctlr.subsys));
+
+ /* numtasks not available for this resource group */
+ if (!res)
+  goto out;
+
+ /* Check forkrate before checking resource group's usage */
+ rc = check_forkrate(res);
+ if (rc)
```

```
+  goto out;
+
+ if (res->cnt_max_shares == SHARE_DONT_CARE)
+  goto out;
+
+ /* Over the limit ? */
+ if (atomic_read(&res->cnt_cur_alloc) >= res->cnt_max_shares) {
+  res->failures++;
+  rc = -ENOSPC;
+  goto out;
+ }
+ out:
+ rcu_read_unlock();
+ return rc;
+}
+
+static void inc_usage_count(struct numtasks *res)
+{
+ struct resource_group *rgroup = numtasks_rgroup(res);
+ atomic_inc(&res->cnt_cur_alloc);
+
+ if (is_res_group_root(rgroup)) {
+  total_cnt_alloc++;
+  res->successes++;
+  return;
+ }
+ /* Do we need to borrow from our parent ? */
+ if ((res->cnt_unused == SHARE_DONT_CARE) ||
+   (atomic_read(&res->cnt_cur_alloc) > res->cnt_unused)) {
+  inc_usage_count(get_numtasks(rgroup->parent));
+  atomic_inc(&res->cnt_borrowed);
+ } else {
+  total_cnt_alloc++;
+    res->successes++;
+ }
+}
+
+static void dec_usage_count(struct numtasks *res)
+{
+ if (atomic_read(&res->cnt_cur_alloc) == 0)
+  return;
+ atomic_dec(&res->cnt_cur_alloc);
+ if (atomic_read(&res->cnt_borrowed) > 0) {
+  atomic_dec(&res->cnt_borrowed);
+  dec_usage_count(get_numtasks(numtasks_rgroup(res)->parent));
+ } else
+  total_cnt_alloc--;
+
```

```
+}
+
+static void numtasks_move_task(struct task_struct *task,
+  struct res_shares *old, struct res_shares *new)
+{
+ struct numtasks *oldres, *newres;
+
+ if (old == new)
+  return;
+
+ /* Decrement usage count of old resource group */
+ oldres = get_shares_numtasks(old);
+ if (oldres)
+  dec_usage_count(oldres);
+
+ /* Increment usage count of new resource group */
+ newres = get_shares_numtasks(new);
+ if (newres)
+  inc_usage_count(newres);
+}
+
+/* Initialize share struct values */
+static void numtasks_res_init_one(struct numtasks *numtasks_res)
+{
+ numtasks_res->shares.min_shares = SHARE_DONT_CARE;
+ numtasks_res->shares.max_shares = SHARE_DONT_CARE;
+ numtasks_res->shares.child_shares_divisor = SHARE_DEFAULT_DIVISOR;
+ numtasks_res->shares.unused_min_shares = SHARE_DEFAULT_DIVISOR;
+
+ numtasks_res->cnt_min_shares = SHARE_DONT_CARE;
+ numtasks_res->cnt_unused = SHARE_DONT_CARE;
+ numtasks_res->cnt_max_shares = SHARE_DONT_CARE;
+ numtasks_res->period_start = jiffies;
+}
+
+static struct res_shares *numtasks_alloc_shares_struct(
+      struct resource_group *rgroup)
+{
+ struct numtasks *res;
+
+ res = kzalloc(sizeof(struct numtasks), GFP_KERNEL);
+ if (!res)
+  return NULL;
+ numtasks_res_init_one(res);
+ if (is_res_group_root(rgroup))
+  root_rgroup = rgroup; /* store root's resource group. */
+ return &res->shares;
+}
```

```
+
+/*
+ * No locking of this resource group object necessary as we are not
+ * supposed to be assigned (or used) when/after this function is called.
+ */
+static void numtasks_free_shares_struct(struct res_shares *my_res)
+{
+ struct numtasks *res, *parres;
+ int i, borrowed;
+ struct resource_group *rgroup;
+
+ res = get_shares_numtasks(my_res);
+ rgroup = numtasks_rgroup(res);
+ if (!is_res_group_root(rgroup)) {
+  parres = get_numtasks(rgroup->parent);
+  borrowed = atomic_read(&res->cnt_borrowed);
+  for (i = 0; i < borrowed; i++)
+   dec_usage_count(parres);
+ }
+ kfree(res);
+}
+
+static int recalc_shares(int self_shares, int parent_shares, int parent_divisor)
+{
+ u64 numerator;
+
+ if ((self_shares == SHARE_DONT_CARE) ||
+    (parent_shares == SHARE_DONT_CARE))
+  return SHARE_DONT_CARE;
+ if (parent_divisor == 0)
+  return 0;
+ numerator = (u64) self_shares * parent_shares;
+ do_div(numerator, parent_divisor);
+ return numerator;
+}
+
+static int recalc_unused_shares(int self_cnt_min_shares,
+    int self_unused_min_shares, int self_divisor)
+{
+ u64 numerator;
+
+ if (self_cnt_min_shares == SHARE_DONT_CARE)
+  return SHARE_DONT_CARE;
+ if (self_divisor == 0)
+  return 0;
+ numerator = (u64) self_unused_min_shares * self_cnt_min_shares;
+ do_div(numerator, self_divisor);
+ return numerator;
```

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

```
+{
+ struct numtasks *parres, *res;
+ struct res_shares *cur, *par;
+ struct resource_group *rgroup;
+
+ res = get_shares_numtasks(my_res);
+ if (!res)
+  return;
+ rgroup = numtasks_rgroup(res);
+ cur = &res->shares;
+
+ if (!is_res_group_root(rgroup)) {
+  parres = get_numtasks(rgroup->parent);
+  par = &parres->shares;
+ } else {
+  parres = NULL;
+  par = NULL;
+ }
+ if (parres)
+  parres->cnt_unused = recalc_unused_shares(
+      parres->cnt_min_shares,
+      par->unused_min_shares,
+      par->child_shares_divisor);
+ recalc_and_propagate(res, parres);
+}
+
+static ssize_t numtasks_show_stats(struct res_shares *my_res,
+     char *buf, size_t buf_size)
+{
+ ssize_t i, j = 0;
+ struct numtasks *res;
+
+ res = get_shares_numtasks(my_res);
+ if (!res)
+  return -EINVAL;
+
+ i = snprintf(buf, buf_size, "%s: Current usage %d\n",
+     res_ctlr_name,
+     atomic_read(&res->cnt_cur_alloc));
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of successes %d\n",
+     res_ctlr_name, res->successes);
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of failures %d\n",
+     res_ctlr_name, res->failures);
+ buf += i; j += i; buf_size -= i;
+ i = snprintf(buf, buf_size, "%s: Number of forkrate failures %d\n",
+     res_ctlr_name, res->forkrate_failures);
```

```
+ j += i;
+ return j;
+}
+
+struct res_controller numtasks_ctlr = {
+ .name = res_ctlr_name,
+ .ctlr_id = NO_RES_ID,
+ .alloc_shares_struct = numtasks_alloc_shares_struct,
+ .free_shares_struct = numtasks_free_shares_struct,
+ .move_task = numtasks_move_task,
+ .shares_changed = numtasks_shares_changed,
+ .show_stats = numtasks_show_stats,
+};
+
+/*
+ * Writeable module parameters use these set_<parameter> functions to respond
+ * to changes. Otherwise the values can be read and used any time.
+ */
+static int set_numtasks_config_val(int *var, int old_value, const char *val,
+    struct kernel_param *kp)
+{
+ int rc = param_set_int(val, kp);
+
+ if (rc < 0)
+  return rc;
+ if (*var < 1) {
+  *var = old_value;
+  return -EINVAL;
+ }
+ return 0;
+}
+
+static int set_total_numtasks(const char *val, struct kernel_param *kp)
+{
+ int prev = total_numtasks;
+ int rc = set_numtasks_config_val(&total_numtasks, prev, val, kp);
+ struct numtasks *res = NULL;
+
+ if (!root_rgroup)
+  return 0;
+ if (rc < 0)
+  return rc;
+ if (total_numtasks <= total_cnt_alloc) {
+  total_numtasks = prev;
+  return -EINVAL;
+ }
+ container_lock();
+ res = get_numtasks(root_rgroup);
```

```
+ res->cnt_min_shares = total_numtasks;
+ res->cnt_unused = total_numtasks;
+ res->cnt_max_shares = total_numtasks;
+ recalc_and_propagate(res, NULL);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(total_numtasks, int, set_total_numtasks,
+   S_IRUGO | S_IWUSR);
+
+static void reset_forkrates(struct resource_group *rgroup, unsigned long now)
+{
+ struct numtasks *res;
+ struct resource_group *child = NULL;
+
+ res = get_numtasks(rgroup);
+ if (!res)
+  return;
+ res->forks_in_period = 0;
+ res->period_start = now;
+
+ for_each_child(child, rgroup)
+  reset_forkrates(child, now);
+}
+
+static int set_forkrate(const char *val, struct kernel_param *kp)
+{
+ int prev = forkrate;
+ int rc = set_numtasks_config_val(&forkrate, prev, val, kp);
+ if (rc < 0)
+  return rc;
+ container_lock();
+ reset_forkrates(root_rgroup, jiffies);
+ container_unlock();
+ return 0;
+}
+module_param_set_call(forkrate, int, set_forkrate, S_IRUGO | S_IWUSR);
+
+static int set_forkrate_interval(const char *val, struct kernel_param *kp)
+{
+ int prev = forkrate_interval;
+ int rc = set_numtasks_config_val(&forkrate_interval, prev, val, kp);
+ if (rc < 0)
+  return rc;
+ container_lock();
+ reset_forkrates(root_rgroup, jiffies);
+ container_unlock();
+ return 0;
```

```
+}
+module_param_set_call(forkrate_interval, int, set_forkrate_interval,
+   S_IRUGO | S_IWUSR);
+
+int __init init_numtasks_res(void)
+{
+ if (numtasks_ctlr.ctlr_id != NO_RES_ID)
+  return -EBUSY; /* already registered */
+ return register_controller(&numtasks_ctlr);
+}
+
+void __exit exit_numtasks_res(void)
+{
+ int rc;
+ do {
+  rc = unregister_controller(&numtasks_ctlr);
+ } while (rc == -EBUSY);
+ BUG_ON(rc != 0);
+}
+module_init(init_numtasks_res)
+module_exit(exit_numtasks_res)
Index: container-2.6.20/kernel/res_group/res_group.c
 ===================================================================== =======
--- /dev/null
+++ container-2.6.20/kernel/res_group/res_group.c
@@ -0,0 +1,160 @@
+/* res_group.c - Resource Groups: Resource management through grouping of
+ *    unrelated tasks.
+ *
+ * Copyright (C) Hubertus Franke, IBM Corp. 2003, 2004
+ *  (C) Shailabh Nagar, IBM Corp. 2003, 2004
+ *  (C) Chandra Seetharaman, IBM Corp. 2003, 2004, 2005
+ *  (C) Vivek Kashyap, IBM Corp. 2004
+ *  (C) Matt Helsley, IBM Corp. 2006
+ *
+ * Provides kernel API of Resource Groups for in-kernel,per-resource
+ * controllers (one each for cpu, memory and io).
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ */
+
+#include <linux/module.h>
```

```
+#include <asm/uaccess.h>
+#include <linux/fs.h>
+#include "local.h"
+
+
+static int res_group_create(struct container_subsys *ss,
+       struct container *cont)
+{
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ struct res_shares *shares = ctlr->alloc_shares_struct(cont);
+ cont->subsys[ss->subsys_id] = &shares->css;
+ return 0;
+}
+
+static void res_group_destroy(struct container_subsys *ss,
+        struct container *cont)
+{
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ struct res_shares *shares = get_controller_shares(cont, ctlr);
+ ctlr->free_shares_struct(shares);
+}
+
+static int res_group_populate(struct container_subsys *ss,
+         struct container *cont) {
+ int err;
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ if ((err = container_add_file(cont, &ctlr->shares_cft.cft)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &ctlr->stats_cft.cft)) < 0)
+  return err;
+
+ return 0;
+}
+
+static void res_group_attach(struct container_subsys *ss,
+        struct container *cont,
+        struct container *old_cont,
+        struct task_struct *tsk) {
+ struct res_controller *ctlr = container_of(ss, struct res_controller, subsys);
+ struct res_shares *oldshares = get_controller_shares(old_cont, ctlr);
+ struct res_shares *newshares = get_controller_shares(cont, ctlr);
+
+ if (ctlr->move_task) {
+  ctlr->move_task(tsk, oldshares, newshares);
+ }
+}
+
+static void res_group_fork(struct container_subsys *ss,
```

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

```
+
+ ctlr->shares_cft.ctlr = ctlr;
+ strcpy(ctlr->shares_cft.cft.name, ctlr->name);
+ strcat(ctlr->shares_cft.cft.name, ".shares");
+ ctlr->shares_cft.cft.private = RG_FILE_SHARES;
+ ctlr->shares_cft.cft.read = res_group_file_read;
+ ctlr->shares_cft.cft.write = res_group_file_write;
+
+ ctlr->stats_cft.ctlr = ctlr;
+ strcpy(ctlr->stats_cft.cft.name, ctlr->name);
+ strcat(ctlr->stats_cft.cft.name, ".stats");
+ ctlr->stats_cft.cft.private = RG_FILE_STATS;
+ ctlr->stats_cft.cft.read = res_group_file_read;
+ ctlr->stats_cft.cft.write = res_group_file_write;
+
+ ss->name = ctlr->name;
+
+ ret = container_register_subsys(ss);
+
+ if (ret < 0)
+  return ret;
+
+ ctlr->ctlr_id = ss->subsys_id;
+
+ return 0;
+}
+
+/*
+ * Unregistering resource controller.
+ *
+ * Returns 0 on success -errno for failure.
+ */
+int unregister_controller(struct res_controller *ctlr)
+{
+ BUG();
+ return 0;
+}
+
+
+EXPORT_SYMBOL_GPL(register_controller);
+EXPORT_SYMBOL_GPL(unregister_controller);
+EXPORT_SYMBOL_GPL(set_controller_shares);
Index: container-2.6.20/kernel/res_group/rgcs.c
 ===================================================================
--- /dev/null
+++ container-2.6.20/kernel/res_group/rgcs.c
@@ -0,0 +1,302 @@
+/*
```

```
+ * kernel/res_group/rgcs.c
+ *
+ * Copyright (C) Shailabh Nagar,  IBM Corp. 2005
+ *          Chandra Seetharaman,   IBM Corp. 2005, 2006
+ *
+ * Resource Group Configfs Subsystem (rgcs) provides the user interface
+ * for Resource groups.
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of version 2 of the  GNU General Public License
+ * as published by the Free Software Foundation.
+ *
+ */
+#include <linux/ctype.h>
+#include <linux/module.h>
+#include <linux/configfs.h>
+#include <linux/parser.h>
+#include <linux/fs.h>
+#include <asm/uaccess.h>
+
+#include "local.h"
+
+#define RES_STRING "res"
+#define MIN_SHARES_STRING "min_shares"
+#define MAX_SHARES_STRING "max_shares"
+#define CHILD_SHARES_DIVISOR_STRING "child_shares_divisor"
+
+static ssize_t show_stats(struct resource_group *rgroup,
+     struct res_controller *ctlr,
+     char *buf)
+{
+ int j = 0, rc = 0;
+ size_t buf_size = PAGE_SIZE-1; /* allow only PAGE_SIZE # of bytes */
+ struct res_shares *shares;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares && ctlr->show_stats)
+  j = ctlr->show_stats(shares, buf, buf_size);
+ rc += j;
+ buf += j;
+ buf_size -= j;
+ return rc;
+}
+
+enum parse_token_t {
+ parse_res_type, parse_err
```

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

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

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

```
+  rc = set_controller_shares(rgroup, ctlr, &shares);
+  kfree(resname);
+ }
+ return rc;
+}
+
+static ssize_t show_shares(struct resource_group *rgroup,
+       struct res_controller *ctlr,
+       char *buf)
+{
+ ssize_t j, rc = 0, bufsize = PAGE_SIZE;
+ struct res_shares *shares;
+
+ shares = get_controller_shares(rgroup, ctlr);
+ if (shares) {
+  j = snprintf(buf, bufsize, "%s=%s,%s=%d,%s=%d,%s=%d\n",
+         RES_STRING, ctlr->name,
+         MIN_SHARES_STRING, shares->min_shares,
+         MAX_SHARES_STRING, shares->max_shares,
+         CHILD_SHARES_DIVISOR_STRING,
+         shares->child_shares_divisor);
+  rc += j; buf += j; bufsize -= j;
+ }
+ return rc;
+}
+
+ssize_t res_group_file_write(struct container *cont,
+       struct cftype *cft,
+       struct file *file,
+       const char __user *userbuf,
+       size_t nbytes, loff_t *ppos)
+{
+ struct res_group_cft *rgcft = container_of(cft, struct res_group_cft, cft);
+ struct res_controller *ctlr = rgcft->ctlr;
+
+ char *buf;
+ ssize_t retval;
+ int filetype = cft->private;
+
+ if (nbytes >= PAGE_SIZE)
+  return -E2BIG;
+
+ buf = kmalloc(nbytes + 1, GFP_USER);
+ if (!buf) return -ENOMEM;
+ if (copy_from_user(buf, userbuf, nbytes)) {
+  retval = -EFAULT;
+  goto out1;
+ }
```

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

```
+  retval = show_stats(cont, ctlr, page);
+  break;
+ default:
+  retval = -EINVAL;
+ }
+
+ if (retval >= 0) {
+  retval = simple_read_from_buffer(buf, nbytes,
+        ppos, page, retval);
+ }
+ kfree(page);
+ return retval;
+}
Index: container-2.6.20/kernel/res_group/shares.c
 ===================================================================
--- /dev/null
+++ container-2.6.20/kernel/res_group/shares.c
@@ -0,0 +1,228 @@
+/*
+ * shares.c - Share management functions for Resource Groups
+ *
+ * Copyright (C) Chandra Seetharaman,  IBM Corp. 2003, 2004, 2005, 2006
+ *  (C) Hubertus Franke,  IBM Corp. 2004
+ *  (C) Matt Helsley,  IBM Corp. 2006
+ *
+ * Latest version, more details at http://ckrm.sf.net
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License version 2 as
+ * published by the Free Software Foundation.
+ */
+
+#include <linux/errno.h>
+#include <linux/res_group_rc.h>
+#include <linux/container.h>
+
+/*
+ * Share values can be quantitative (quantity of memory for instance) or
+ * symbolic. The symbolic value DONT_CARE allows for any quantity of a resource
+ * to be substituted in its place. The symbolic value UNCHANGED is only used
+ * when setting share values and means that the old value should be used.
+ */
+
+/* Is the share a quantity (as opposed to "symbols" DONT_CARE or UNCHANGED) */
+static inline int is_share_quantitative(int share)
+{
+ return (share >= 0);
+}
```

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

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

```
+   (current_usage > child->child_shares_divisor))
+  return -EINVAL;
+
+ /*
+  * CHILD validation: is the new child_shares_divisor large
+  * enough to accomodate largest max_shares of any of my child
+  */
+ if (child->child_shares_divisor < child->cur_max_shares)
+  return -EINVAL;
+
+ /* CHILD validation: min <= max */
+ if (is_share_quantitative(child->min_shares) &&
+   is_share_quantitative(child->max_shares) &&
+   (child->min_shares > child->max_shares))
+  return -EINVAL;
+
+ return 0;
+}
+
+/*
+ * Set the resource shares of a child resource group given the new shares
+ * specified by userspace, the child's current shares, and the parent
+ * resource group's shares.
+ *
+ * Caller is responsible for holding group_lock of child and parent
+ * resource groups to protect the shares structures passed to this function.
+ */
+static int set_shares(const struct res_shares *new,
+       struct res_shares *child_shares,
+           struct res_shares *parent_shares)
+{
+ int rc, current_usage, min_shares_increase;
+ struct res_shares final_shares;
+
+ BUG_ON(!new || !child_shares);
+
+ final_shares = *child_shares;
+ if (did_share_change(new->child_shares_divisor) &&
+   change_supported(child_shares->child_shares_divisor))
+  final_shares.child_shares_divisor = new->child_shares_divisor;
+ if (did_share_change(new->min_shares) &&
+   change_supported(child_shares->min_shares))
+  final_shares.min_shares = new->min_shares;
+ if (did_share_change(new->max_shares) &&
+   change_supported(child_shares->max_shares))
+  final_shares.max_shares = new->max_shares;
+
+ current_usage = child_shares->child_shares_divisor -
```

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

```
+done:
+ container_unlock(); /* XXX */
+ if (!rc)
+  ctlr->shares_changed(shares);
+ return rc;
+}

--
```