Subject: [RFC][PATCH 6/6] Hook up to container infrastructure
Posted by Srivatsa Vaddagiri on Mon, 11 Jun 2007 15:58:21 GMT
View Forum Message <> Reply to Message

This patch hooks up cpu scheduler with Paul Menage's container
infrastructure.

The container patches allows administrator to create arbitrary groups of tasks
and define resource allocation for each group. By registering with container
infrastructure, cpu scheduler is made aware of group membership information for
each task, creation/deletion of groups etc and can use that information to
provide fairness between groups.

This mechanism can indirectly be used to provide fairness between users
also. All that is needed is a user-space program (which I am working on
and will post later) which monitors for PROC_EVENT_UID events (using
process event connector) and moves the task to appropriate user-directory in
container filesystem.

As an example for "HOWTO use this feature", follow these steps:

1. Define CONFIG_FAIR_GROUP_SCHED (General Setup->Fair Group Scheduler)
   and compile the kernel
2. After booting:

```
# cd /dev
# mkdir cpuctl
# mount -t container -ocpuctl none /dev/cpuctl
# cd cpuctl
# mkdir grpA
# mkdir grpB

# echo some_pid1 > grpA/tasks
# echo some_pid2 > grpA/tasks
# echo some_pid3 > grpA/tasks
# echo some_pid4 > grpA/tasks

...

# echo another_pidX > grpB/tasks
# echo another_pidY > grpB/tasks
```

All tasks in grpA/tasks should cumulatively share same CPU as all tasks
in grpB/tasks.

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

```
---
 include/linux/container_subsys.h |    6 +
 include/linux/sched.h            |    1
 init/Kconfig                     |    8 +
 kernel/sched.c                   |  234 ++++++++++++++++++++++++++++++++++++++-
 kernel/sched_fair.c              |   36 ++++--
 5 files changed, 274 insertions(+), 11 deletions(-)

Index: current/kernel/sched.c
===================================================================
--- current.orig/kernel/sched.c 2007-06-09 15:07:37.000000000 +0530
+++ current/kernel/sched.c 2007-06-09 15:07:38.000000000 +0530
@@ -116,6 +116,39 @@
  struct list_head queue[MAX_RT_PRIO];
 };

+#ifdef CONFIG_FAIR_GROUP_SCHED
+
+#include <linux/container.h>
+
+struct lrq;
+
+struct task_grp {
+ struct container_subsys_state css;
+ /* schedulable entities of this group on each cpu */
+ struct sched_entity **se;
+ /* runqueue "owned" by this group on each cpu */
+ struct lrq **lrq;
+};
+
+static DEFINE_PER_CPU_SHARED_ALIGNED(struct sched_entity, init_sched_entity);
+static DEFINE_PER_CPU_SHARED_ALIGNED(struct lrq, init_lrq);
+
+static struct sched_entity *init_sched_entity_p[CONFIG_NR_CPUS];
+static struct lrq *init_lrq_p[CONFIG_NR_CPUS];
+
+static struct task_grp init_task_grp =  {
+     .se = init_sched_entity_p,
+     .lrq = init_lrq_p,
+     };
+
+static inline struct task_grp *task_grp(struct task_struct *p)
+{
+ return container_of(task_subsys_state(p, cpuctlr_subsys_id),
+    struct task_grp, css);
+}
+
+#endif
```

```
+
 /* CFS-related fields in a runqueue */
 struct lrq {
  long nr_running;
@@ -146,6 +179,7 @@
   * is used during load balance.
   */
  struct list_head leaf_lrq_list;
+ struct task_grp *tg; /* group that "owns" this runqueue */
 #endif
 };

@@ -785,7 +819,8 @@
 {
  task_thread_info(p)->cpu = cpu;
 #ifdef CONFIG_FAIR_GROUP_SCHED
- p->se.lrq = &cpu_rq(cpu)->lrq;
+ p->se.lrq = task_grp(p)->lrq[cpu];
+ p->se.parent = task_grp(p)->se[cpu];
 #endif
 }

@@ -812,7 +847,8 @@
  task_thread_info(p)->cpu = new_cpu;

 #ifdef CONFIG_FAIR_GROUP_SCHED
- p->se.lrq = &new_rq->lrq;
+ p->se.lrq = task_grp(p)->lrq[new_cpu];
+ p->se.parent = task_grp(p)->se[new_cpu];
 #endif
 }

@@ -4505,7 +4541,8 @@
  task_thread_info(idle)->preempt_count = 0;
 #endif
 #ifdef CONFIG_FAIR_GROUP_SCHED
- idle->se.lrq = &rq->lrq;
+ idle->se.lrq = init_task_grp.lrq[cpu];
+ idle->se.parent = init_task_grp.se[cpu];
 #endif
 }

@@ -6119,7 +6156,22 @@
   init_lrq(&rq->lrq, rq);
 #ifdef CONFIG_FAIR_GROUP_SCHED
   INIT_LIST_HEAD(&rq->leaf_lrq_list);
-  list_add(&rq->lrq.leaf_lrq_list, &rq->leaf_lrq_list);
+  {
```

```
+   struct lrq *lrq = &per_cpu(init_lrq, i);
+   struct sched_entity *se =
+      &per_cpu(init_sched_entity, i);
+
+   init_lrq_p[i] = lrq;
+   init_lrq(lrq, rq);
+   lrq->tg = &init_task_grp;
+   list_add(&lrq->leaf_lrq_list, &rq->leaf_lrq_list);
+
+   init_sched_entity_p[i] = se;
+   se->lrq = &rq->lrq;
+   se->my_q = lrq;
+   se->load_weight = NICE_0_LOAD;
+   se->parent = NULL;
+  }
 #endif

 #ifdef CONFIG_SMP
@@ -6300,3 +6352,177 @@
 }

 #endif
+
+#ifdef CONFIG_FAIR_GROUP_SCHED
+
+/* return corresponding task_grp object of a container */
+static inline struct task_grp *container_tg(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, cpuctlr_subsys_id),
+     struct task_grp, css);
+}
+
+/* allocate runqueue etc for a new task group */
+static int sched_create_group(struct container_subsys *ss,
+     struct container *cont)
+{
+ struct task_grp *tg;
+ struct lrq *lrq;
+ struct sched_entity *se;
+ int i;
+
+ if (!cont->parent) {
+   /* This is early initialization for the top container */
+   cont->subsys[cpuctlr_subsys_id] = &init_task_grp.css;
+   init_task_grp.css.container = cont;
+   return 0;
+ }
+
```

```
+ /* we support only 1-level deep hierarchical scheduler atm */
+ if (cont->parent->parent)
+  return -EINVAL;
+
+ tg = kzalloc(sizeof(*tg), GFP_KERNEL);
+ if (!tg)
+  return -ENOMEM;
+
+ tg->lrq = kzalloc(sizeof(lrq) * num_possible_cpus(), GFP_KERNEL);
+ if (!tg->lrq)
+  goto err;
+ tg->se = kzalloc(sizeof(se) * num_possible_cpus(), GFP_KERNEL);
+ if (!tg->se)
+  goto err;
+
+ for_each_possible_cpu(i) {
+  struct rq *rq = cpu_rq(i);
+
+  lrq = kmalloc_node(sizeof(struct lrq), GFP_KERNEL,
+         cpu_to_node(i));
+  if (!lrq)
+   goto err;
+
+  se = kmalloc_node(sizeof(struct sched_entity), GFP_KERNEL,
+        cpu_to_node(i));
+  if (!se)
+   goto err;
+
+  memset(lrq, 0, sizeof(struct lrq));
+  memset(se, 0, sizeof(struct sched_entity));
+
+  tg->lrq[i] = lrq;
+  init_lrq(lrq, rq);
+  lrq->tg = tg;
+  list_add_rcu(&lrq->leaf_lrq_list, &rq->leaf_lrq_list);
+
+  tg->se[i] = se;
+  se->lrq = &rq->lrq;
+  se->my_q = lrq;
+  se->load_weight = NICE_0_LOAD;
+  se->parent = NULL;
+ }
+
+ /* Bind the container to task_grp object we just created */
+ cont->subsys[cpuctlr_subsys_id] = &tg->css;
+ tg->css.container = cont;
+
+ return 0;
```

```
+
+err:
+ for_each_possible_cpu(i) {
+  if (tg->lrq && tg->lrq[i])
+   kfree(tg->lrq[i]);
+  if (tg->se && tg->se[i])
+   kfree(tg->se[i]);
+ }
+ if (tg->lrq)
+  kfree(tg->lrq);
+ if (tg->se)
+  kfree(tg->se);
+ if (tg)
+  kfree(tg);
+
+ return -ENOMEM;
+}
+
+
+/* destroy runqueue etc associated with a task group */
+static void sched_destroy_group(struct container_subsys *ss,
+     struct container *cont)
+{
+ struct task_grp *tg = container_tg(cont);
+ struct lrq *lrq;
+ struct sched_entity *se;
+ int i;
+
+ for_each_possible_cpu(i) {
+  lrq = tg->lrq[i];
+  list_del_rcu(&lrq->leaf_lrq_list);
+ }
+
+ /* wait for possible concurrent references to lrqs complete */
+ synchronize_sched();
+
+ /* now it should be safe to free those lrqs */
+ for_each_possible_cpu(i) {
+  lrq = tg->lrq[i];
+  kfree(lrq);
+
+  se = tg->se[i];
+  kfree(se);
+ }
+
+ kfree(tg);
+}
+
```

```
+/* change task's runqueue when it moves between groups */
+static void sched_move_task(struct container_subsys *ss, struct container *cont,
+   struct container *old_cont, struct task_struct *tsk)
+{
+ int on_rq;
+ unsigned long flags;
+ struct rq *rq;
+ struct task_grp *tg_new = container_tg(cont);
+
+ rq = task_rq_lock(tsk, &flags);
+
+ on_rq = tsk->se.on_rq;
+ if (on_rq)
+  deactivate_task(rq, tsk, 0);
+
+ if (unlikely(rq->curr == tsk) && tsk->sched_class == &fair_sched_class)
+  tsk->sched_class->put_prev_task(rq, tsk, rq_clock(rq));
+
+ tsk->se.parent = tg_new->se[task_cpu(tsk)];
+ tsk->se.lrq = tg_new->lrq[task_cpu(tsk)];
+
+ /* todo: change task's load_weight to reflect its new group */
+
+ if (on_rq)
+   activate_task(rq, tsk, 0);
+
+ if (unlikely(rq->curr == tsk) && tsk->sched_class == &fair_sched_class)
+  tsk->sched_class->set_curr_task(rq);
+
+ task_rq_unlock(rq, &flags);
+}
+
+
+static int sched_populate(struct container_subsys *ss, struct container *cont)
+{
+ /* todo: create a cpu_shares file to modify group weight */
+
+ return 0;
+}
+
+struct container_subsys cpuctlr_subsys = {
+ .name = "cpuctl",
+ .create = sched_create_group,
+ .destroy  = sched_destroy_group,
+ .attach = sched_move_task,
+ .populate = sched_populate,
+ .subsys_id = cpuctlr_subsys_id,
+ .early_init = 1,
```

```
+};
+
+#endif
+
Index: current/include/linux/container_subsys.h
===================================================================
--- current.orig/include/linux/container_subsys.h 2007-06-09 14:56:43.000000000 +0530
+++ current/include/linux/container_subsys.h 2007-06-09 15:07:38.000000000 +0530
@@ -24,3 +24,9 @@
 #endif

 /* */
+
+#ifdef CONFIG_FAIR_GROUP_SCHED
+SUBSYS(cpuctlr)
+#endif
+
+/* */
Index: current/kernel/sched_fair.c
===================================================================
--- current.orig/kernel/sched_fair.c 2007-06-09 15:07:37.000000000 +0530
+++ current/kernel/sched_fair.c 2007-06-09 15:07:38.000000000 +0530
@@ -558,10 +558,9 @@
   resched_task(lrq_rq(lrq)->curr);
 }

-static struct sched_entity * pick_next_entity(struct lrq *lrq, u64 now)
+static inline void
+set_next_entity(struct lrq *lrq, struct sched_entity *se, u64 now)
 {
- struct sched_entity *p = __pick_next_entity(lrq);
-
  /*
   * Any task has to be enqueued before it get to execute on
   * a CPU. So account for the time it spent waiting on the
@@ -569,9 +568,16 @@
   * done a put_prev_task_fair() shortly before this, which
   * updated rq->fair_clock - used by update_stats_wait_end())
   */
- update_stats_wait_end(lrq, p, now);
- update_stats_curr_start(lrq, p, now);
- set_lrq_curr(lrq, p);
+ update_stats_wait_end(lrq, se, now);
+ update_stats_curr_start(lrq, se, now);
+ set_lrq_curr(lrq, se);
+}
+
+static struct sched_entity * pick_next_entity(struct lrq *lrq, u64 now)
```

```
+{
+ struct sched_entity *p = __pick_next_entity(lrq);
+
+ set_next_entity(lrq, p, now);

  return p;
 }
@@ -723,7 +729,7 @@

 static inline struct lrq *cpu_lrq(struct lrq *lrq, int this_cpu)
 {
- return &cpu_rq(this_cpu)->lrq;
+ return lrq->tg->lrq[this_cpu];
 }

 #define for_each_leaf_lrq(a, b) \
@@ -1085,6 +1091,20 @@
  inc_nr_running(p, rq);
 }

+/* Account for a task changing its policy or group */
+static void set_curr_task_fair(struct rq *rq)
+{
+ struct task_struct *curr = rq->curr;
+ struct sched_entity *se = &curr->se;
+ struct lrq *lrq;
+ u64 now = rq_clock(rq);
+
+ for_each_sched_entity(se) {
+  lrq = sched_entity_lrq(se);
+  set_next_entity(lrq, se, now);
+ }
+}
+
 /*
  * All the scheduling class methods:
  */
@@ -1098,6 +1118,8 @@
  .pick_next_task  = pick_next_task_fair,
  .put_prev_task  = put_prev_task_fair,

+ .set_curr_task  = set_curr_task_fair,
+
 #ifdef CONFIG_SMP
  .load_balance  = load_balance_fair,
 #endif
Index: current/include/linux/sched.h
===================================================================
```

```
--- current.orig/include/linux/sched.h 2007-06-09 15:07:37.000000000 +0530
+++ current/include/linux/sched.h 2007-06-09 15:07:38.000000000 +0530
@@ -865,6 +865,7 @@

  struct task_struct * (*pick_next_task) (struct rq *rq, u64 now);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p, u64 now);
+ void (*set_curr_task) (struct rq *rq);

 #ifdef CONFIG_SMP
  int (*load_balance) (struct rq *this_rq, int this_cpu,
Index: current/init/Kconfig
===================================================================
--- current.orig/init/Kconfig 2007-06-09 14:56:43.000000000 +0530
+++ current/init/Kconfig 2007-06-09 15:07:38.000000000 +0530
@@ -328,6 +328,14 @@

    Say N if unsure.

+config FAIR_GROUP_SCHED
+ select CONTAINERS
+ help
+   This option enables you to group tasks and control CPU resource
+   allocation to such groups.
+
+   Say N if unsure.
+
 config SYSFS_DEPRECATED
  bool "Create deprecated sysfs files"
  default y

--
Regards,
vatsa
```