Subject: [RFC | PATCH 0/9] CPU controller over process container
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:51:11 GMT
View Forum Message <> Reply to Message

Here's a respin of my earlier CPU controller to work on top of Paul Menage's process container patches.

Problem:

Current CPU scheduler is very task centric, which makes it difficult to manage cpu resource consumption of a group of (related) tasks.

For ex: with the current O(1) scheduler, it is possible for a user to monopolize CPU simply by spawning more and more threads, causing DoS to other users.

Requirements:

A few of them are:

- Provide means to group tasks from user-land and specify limits of CPU bandwidth consumption of each group. CPU bandwidth limit is enforced over some suitable time period. For ex: a 40% limit could mean the task group's usage is limited to 4 sec every 10 sec or 24 sec every minute.

- Time period over which bandwidth is controlled to each group to be configurable (?)

- Work conserving - Do not let the CPU be idle if there are runnable tasks (even if that means running task-groups that are above their allowed limit)

- SMP behavior - Limit to be enforced on all CPUs put together

- Real-time tasks - Should be left alone as they are today? i.e real time tasks across groups should be scheduled as if they are in same group

- Should cater to requirements of variety of workload characteristics, including bursty ones (?)

Salient points about this patch:

- Each task-group gets its own runqueue on every cpu.

- In addition, there is an active and expired array of
  task-groups themselves. Task-groups that have expired their
  quota are put into expired array.

- Task-groups have priorities. Priority of a task-group is the
  same as the priority of the highest-priority runnable task it
  has. This I feel will retain interactiveness of the system
  as it is today.

- Scheduling the next task involves picking highest priority task-group
  from active array first and then picking highest-priority task
  within it. Both steps are O(1).

- Token are assigned to task-groups based on their assigned
  quota. Once they run out of tokens, the task-group is put
  in an expired array. Array switch happens when active array
  is empty.

- SMP load-balancing is accomplished on the lines of smpnice.

Results of the patch
====================

Machine : 2way x86_64 Intel Xeon (3.6 GHz) box

Note: All test were forced to run on only one CPU using cpusets

1. Volanomark [1]

  Group A [50% limit] Group B [50% limit]

Elapsed time   35.83 sec   36.6002
Avg throughput 11179.3 msg/sec   10944.3 msg/sec

  Group A [80% limit] Group B [20% limit]

Elapsed time   23.4466 sec   36.1857
Avg throughput 17072 msg/sec   11080 msg/sec

---

2. Kernel compilation

Group A [50% limit] Group B [50% limit]
time -p make -j4 bzImage time -p make -j8 bzImage

real 771.00 sec   769.08 sec

Group A [80% limit] Group B [20% limit]
time -p make -j4 bzImage time -p make -j8 bzImage

real 484.12 sec   769.70 sec

--
Regards,
vatsa

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

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

---

Subject: [PATCH 1/9] split runqueue
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:52:47 GMT
View Forum Message <> Reply to Message

This patch introduces a new runqueue structure (struct task_grp_rq) which is
used to hold "related" tasks together. The structure is quite similar to how
the runqueue structure looks today.

The scheme is every task group will have its own per-cpu runqueues (which is
of type 'struct task_grp_rq'), to hold its tasks together on each CPU.

This is in addition to a per-cpu runqueue which holds runnable task-groups

themselves.

The patch also converts over several references of 'struct rq' to
'struct task_grp_rq' (since tasks now reside in task_grp_rq).

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

```
diff -puN init/Kconfig~cpu-controller-v2-split-runqueue init/Kconfig
--- linux-2.6.20/init/Kconfig~cpu-controller-v2-split-runqueue 2007-04-11 11:48:33.000000000
+0530
+++ linux-2.6.20-vatsa/init/Kconfig 2007-04-11 11:48:33.000000000 +0530
@@ -285,6 +285,14 @@ config CONTAINER_CPUACCT
    Provides a simple Resource Controller for monitoring the
    total CPU consumed by the tasks in a container

+config CPUMETER
+ bool "CPU resource control"
+ depends on CPUSETS && EXPERIMENTAL
+ help
+   This options lets you create cpu resource partitions within
+   cpusets. Each resource partition can be given a different quota
+   of CPU usage.
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
diff -puN kernel/sched.c~cpu-controller-v2-split-runqueue kernel/sched.c
--- linux-2.6.20/kernel/sched.c~cpu-controller-v2-split-runqueue 2007-04-11 11:48:33.000000000
+0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:19.000000000 +0530
@@ -192,11 +192,48 @@ static inline unsigned int task_timeslic

 struct prio_array {
  unsigned int nr_active;
+ int best_static_prio;
  DECLARE_BITMAP(bitmap, MAX_PRIO+1); /* include 1 bit for delimiter */
  struct list_head queue[MAX_PRIO];
 };

 /*
+ * Each task belong to some task-group. Task-groups and what tasks they contain
+ * are defined by the administrator using some suitable interface.
+ * Administrator can also define the CPU bandwidth provided to each task-group.
+ *
+ * Task-groups are given a certain priority to run on every CPU. Currently
```

```
+ * task-group priority on a CPU is defined to be the same as that of
+ * highest-priority runnable task it has on that CPU. Task-groups also
+ * get their own runqueue on every CPU. The main runqueue on each CPU is
+ * used to hold task-groups, rather than tasks.
+ *
+ * Scheduling decision on a CPU is now two-step : first pick highest priority
+ * task-group from the main runqueue and next pick highest priority task from
+ * the runqueue of that group. Both decisions are of O(1) complexity.
+ */
+
+/* runqueue used for every task-group */
+struct task_grp_rq {
+ struct prio_array arrays[2];
+ struct prio_array *active, *expired, *rq_array;
+ unsigned long expired_timestamp;
+ int prio;   /* Priority of the task-group */
+ struct list_head list;
+ struct task_grp *tg;
+};
+
+static DEFINE_PER_CPU(struct task_grp_rq, init_tg_rq);
+
+/* task-group object - maintains information about each task-group */
+struct task_grp {
+ struct task_grp_rq *rq[NR_CPUS]; /* runqueue pointer for every cpu */
+};
+
+/* The "default" task-group */
+struct task_grp init_task_grp;
+
+/*
  * This is the main, per-CPU runqueue data structure.
  *
  * Locking rule: those places that want to lock multiple runqueues
@@ -225,14 +262,15 @@ struct rq {
   */
  unsigned long nr_uninterruptible;

- unsigned long expired_timestamp;
  /* Cached timestamp set by update_cpu_clock() */
  unsigned long long most_recent_timestamp;
  struct task_struct *curr, *idle;
  unsigned long next_balance;
  struct mm_struct *prev_mm;
+ /* these arrays hold task-groups.
+  * xxx: Avoid this for !CONFIG_CPUMETER?
+  */
  struct prio_array *active, *expired, arrays[2];
```

```
- int best_expired_prio;
  atomic_t nr_iowait;

 #ifdef CONFIG_SMP
@@ -248,6 +286,7 @@ struct rq {
 #endif

 #ifdef CONFIG_SCHEDSTATS
+ /* xxx: move these to task-group runqueue where necessary */
  /* latency stats */
  struct sched_info rq_sched_info;

@@ -280,6 +319,13 @@ static inline int cpu_of(struct rq *rq)
 #endif
 }

+#define switch_array(array1, array2) \
+  { \
+    struct prio_array *tmp = array2; \
+    array2 = array1; \
+    array1 = tmp; \
+  }
+
 /*
  * The domain tree (rq->sd) is protected by RCU's quiescent state transition.
  * See detach_destroy_domains: synchronize_sched for details.
@@ -293,6 +339,7 @@ static inline int cpu_of(struct rq *rq)
 #define cpu_rq(cpu)  (&per_cpu(runqueues, (cpu)))
 #define this_rq()  (&__get_cpu_var(runqueues))
 #define task_rq(p)  cpu_rq(task_cpu(p))
+#define task_grp_rq(p)          (task_grp(p)->rq[task_cpu(p)])
 #define cpu_curr(cpu)  (cpu_rq(cpu)->curr)

 #ifndef prepare_arch_switch
@@ -372,6 +419,15 @@ static inline void finish_lock_switch(st
 }
 #endif /* __ARCH_WANT_UNLOCKED_CTXSW */

+/* return the task-group to which a task belongs */
+static inline struct task_grp *task_grp(struct task_struct *p)
+{
+ /* Simply return the default group for now. A later patch modifies
+  * this function.
+  */
+ return &init_task_grp;
+}
+
 /*
```

```
 * __task_rq_lock - lock the runqueue a given task resides on.
 * Must be called interrupts disabled.
@@ -847,10 +903,11 @@ static int effective_prio(struct task_st
 */
static void __activate_task(struct task_struct *p, struct rq *rq)
{
- struct prio_array *target = rq->active;
+ struct task_grp_rq *tgrq = task_grp_rq(p);
+ struct prio_array *target = tgrq->active;

 if (batch_task(p))
-  target = rq->expired;
+  target = tgrq->expired;
 enqueue_task(p, target);
 inc_nr_running(p, rq);
}
@@ -860,7 +917,10 @@ static void __activate_task(struct task_
 */
static inline void __activate_idle_task(struct task_struct *p, struct rq *rq)
{
- enqueue_task_head(p, rq->active);
+ struct task_grp_rq *tgrq = task_grp_rq(p);
+ struct prio_array *target = tgrq->active;
+
+ enqueue_task_head(p, target);
 inc_nr_running(p, rq);
}

@@ -2128,7 +2188,7 @@ int can_migrate_task(struct task_struct
 return 1;
}

-#define rq_best_prio(rq) min((rq)->curr->prio, (rq)->best_expired_prio)
+#define rq_best_prio(rq) min((rq)->curr->prio, (rq)->expired->best_static_prio)

/*
 * move_tasks tries to move up to max_nr_move tasks and max_load_move weighted
@@ -3036,6 +3096,8 @@ unsigned long long current_sched_time(co
 return ns;
}

+#define nr_tasks(tgrq) (tgrq->active->nr_active + tgrq->expired->nr_active)
+
/*
 * We place interactive tasks back into the active array, if possible.
 *
@@ -3046,13 +3108,13 @@ unsigned long long current_sched_time(co
 * increasing number of running tasks. We also ignore the interactivity
```

```
  * if a better static_prio task has expired:
  */
-static inline int expired_starving(struct rq *rq)
+static inline int expired_starving(struct task_grp_rq *rq)
 {
- if (rq->curr->static_prio > rq->best_expired_prio)
+ if (current->static_prio > rq->expired->best_static_prio)
    return 1;
   if (!STARVATION_LIMIT || !rq->expired_timestamp)
    return 0;
- if (jiffies - rq->expired_timestamp > STARVATION_LIMIT * rq->nr_running)
+ if (jiffies - rq->expired_timestamp > STARVATION_LIMIT * nr_tasks(rq))
    return 1;
   return 0;
 }
@@ -3139,7 +3201,9 @@ void account_steal_time(struct task_stru

 static void task_running_tick(struct rq *rq, struct task_struct *p)
 {
- if (p->array != rq->active) {
+ struct task_grp_rq *tgrq = task_grp_rq(p);
+
+ if (p->array != tgrq->active) {
    /* Task has expired but was not scheduled yet */
    set_tsk_need_resched(p);
    return;
@@ -3163,25 +3227,25 @@ static void task_running_tick(struct rq
     set_tsk_need_resched(p);

     /* put it at the end of the queue: */
-    requeue_task(p, rq->active);
+    requeue_task(p, tgrq->active);
    }
    goto out_unlock;
   }
   if (!--p->time_slice) {
-   dequeue_task(p, rq->active);
+   dequeue_task(p, tgrq->active);
    set_tsk_need_resched(p);
    p->prio = effective_prio(p);
    p->time_slice = task_timeslice(p);
    p->first_time_slice = 0;

-   if (!rq->expired_timestamp)
-    rq->expired_timestamp = jiffies;
-   if (!TASK_INTERACTIVE(p) || expired_starving(rq)) {
-    enqueue_task(p, rq->expired);
-    if (p->static_prio < rq->best_expired_prio)
```

```
-    rq->best_expired_prio = p->static_prio;
+  if (!tgrq->expired_timestamp)
+    tgrq->expired_timestamp = jiffies;
+  if (!TASK_INTERACTIVE(p) || expired_starving(tgrq)) {
+    enqueue_task(p, tgrq->expired);
+    if (p->static_prio < tgrq->expired->best_static_prio)
+      rq->expired->best_static_prio = p->static_prio;
   } else
-    enqueue_task(p, rq->active);
+    enqueue_task(p, tgrq->active);
  } else {
   /*
    * Prevent a too long timeslice allowing a task to monopolize
@@ -3202,9 +3266,9 @@ static void task_running_tick(struct rq
   if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
     p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
     (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
-    (p->array == rq->active)) {
+    (p->array == tgrq->active)) {

-    requeue_task(p, rq->active);
+    requeue_task(p, tgrq->active);
     set_tsk_need_resched(p);
   }
  }
@@ -3427,6 +3491,7 @@ asmlinkage void __sched schedule(void)
  int cpu, idx, new_prio;
  long *switch_count;
  struct rq *rq;
+ struct task_grp_rq *next_grp;

  /*
   * Test if we are atomic.  Since do_exit() needs to call into
@@ -3495,23 +3560,36 @@ need_resched_nonpreemptible:
   idle_balance(cpu, rq);
   if (!rq->nr_running) {
    next = rq->idle;
-    rq->expired_timestamp = 0;
    wake_sleeping_dependent(cpu);
    goto switch_tasks;
   }
  }

+ /* Pick a task group first */
+#ifdef CONFIG_CPUMETER
  array = rq->active;
  if (unlikely(!array->nr_active)) {
+  switch_array(rq->active, rq->expired);
```

```
+  array = rq->active;
+ }
+ idx = sched_find_first_bit(array->bitmap);
+ queue = array->queue + idx;
+ next_grp = list_entry(queue->next, struct task_grp_rq, list);
+#else
+ next_grp = init_task_grp.rq[cpu];
+#endif
+
+ /* Pick a task within that group next */
+ array = next_grp->active;
+ if (unlikely(!array->nr_active)) {
   /*
    * Switch the active and expired arrays.
    */
   schedstat_inc(rq, sched_switch);
-  rq->active = rq->expired;
-  rq->expired = array;
-  array = rq->active;
-  rq->expired_timestamp = 0;
-  rq->best_expired_prio = MAX_PRIO;
+  switch_array(next_grp->active, next_grp->expired);
+  array = next_grp->active;
+  next_grp->expired_timestamp = 0;
+  next_grp->expired->best_static_prio = MAX_PRIO;
  }

  idx = sched_find_first_bit(array->bitmap);
@@ -3989,11 +4067,13 @@ void rt_mutex_setprio(struct task_struct
  struct prio_array *array;
  unsigned long flags;
  struct rq *rq;
+ struct task_grp_rq *tgrq;
  int oldprio;

  BUG_ON(prio < 0 || prio > MAX_PRIO);

  rq = task_rq_lock(p, &flags);
+ tgrq = task_grp_rq(p);

  oldprio = p->prio;
  array = p->array;
@@ -4007,7 +4087,7 @@ void rt_mutex_setprio(struct task_struct
    * in the active array!
    */
   if (rt_task(p))
-   array = rq->active;
+   array = tgrq->active;
```

```
   enqueue_task(p, array);
   /*
    * Reschedule if we are currently running on this runqueue and
@@ -4582,7 +4662,8 @@ asmlinkage long sys_sched_getaffinity(pi
 asmlinkage long sys_sched_yield(void)
 {
  struct rq *rq = this_rq_lock();
- struct prio_array *array = current->array, *target = rq->expired;
+ struct task_grp_rq *tgrq = task_grp_rq(current);
+ struct prio_array *array = current->array, *target = tgrq->expired;

  schedstat_inc(rq, yld_cnt);
  /*
@@ -4593,13 +4674,13 @@ asmlinkage long sys_sched_yield(void)
   *  array.)
   */
  if (rt_task(current))
-  target = rq->active;
+  target = tgrq->active;

  if (array->nr_active == 1) {
   schedstat_inc(rq, yld_act_empty);
-  if (!rq->expired->nr_active)
+  if (!tgrq->expired->nr_active)
    schedstat_inc(rq, yld_both_empty);
- } else if (!rq->expired->nr_active)
+ } else if (!tgrq->expired->nr_active)
   schedstat_inc(rq, yld_exp_empty);

  if (array != target) {
@@ -5304,9 +5385,9 @@ static void migrate_dead(unsigned int de
 }

 /* release_task() removes task from tasklist, so we won't find dead tasks. */
-static void migrate_dead_tasks(unsigned int dead_cpu)
+static void migrate_dead_tasks_from_grp(struct task_grp_rq *rq,
+      unsigned int dead_cpu)
 {
- struct rq *rq = cpu_rq(dead_cpu);
  unsigned int arr, i;

  for (arr = 0; arr < 2; arr++) {
@@ -5319,6 +5400,27 @@ static void migrate_dead_tasks(unsigned
   }
  }
 }
+
+static void migrate_dead_tasks(unsigned int dead_cpu)
```

```
+{
+ struct rq *rq = cpu_rq(dead_cpu);
+ unsigned int arr, i;
+
+ for (arr = 0; arr < 2; arr++) {
+  for (i = 0; i < MAX_PRIO; i++) {
+   struct list_head *list = &rq->arrays[arr].queue[i];
+
+   while (!list_empty(list)) {
+    struct task_grp_rq *tgrq =
+     list_entry(list->next,
+       struct task_grp_rq, list);
+
+    migrate_dead_tasks_from_grp(tgrq, dead_cpu);
+    list_del(&tgrq->list);
+   }
+  }
+ }
+}
 #endif /* CONFIG_HOTPLUG_CPU */

 /*
@@ -6900,21 +7002,48 @@ int in_sched_functions(unsigned long add
   && addr < (unsigned long)__sched_text_end);
 }

+static void task_grp_rq_init(struct task_grp_rq *tgrq, struct task_grp *tg)
+{
+ int j, k;
+
+ tgrq->active = tgrq->arrays;
+ tgrq->expired = tgrq->arrays + 1;
+ tgrq->rq_array = NULL;
+ tgrq->expired->best_static_prio = MAX_PRIO;
+ tgrq->active->best_static_prio = MAX_PRIO;
+ tgrq->prio = MAX_PRIO;
+ tgrq->tg = tg;
+ INIT_LIST_HEAD(&tgrq->list);
+
+ for (j = 0; j < 2; j++) {
+  struct prio_array *array;
+
+  array = tgrq->arrays + j;
+  for (k = 0; k < MAX_PRIO; k++) {
+   INIT_LIST_HEAD(array->queue + k);
+   __clear_bit(k, array->bitmap);
+  }
+  // delimiter for bitsearch
```

```
+  __set_bit(MAX_PRIO, array->bitmap);
+ }
+}
+
 void __init sched_init(void)
 {
- int i, j, k;
+ int i, j;

  for_each_possible_cpu(i) {
-  struct prio_array *array;
   struct rq *rq;
+  struct task_grp_rq *tgrq;

   rq = cpu_rq(i);
+  tgrq = init_task_grp.rq[i] = &per_cpu(init_tg_rq, i);
   spin_lock_init(&rq->lock);
+  task_grp_rq_init(tgrq, &init_task_grp);
   lockdep_set_class(&rq->lock, &rq->rq_lock_key);
   rq->nr_running = 0;
   rq->active = rq->arrays;
   rq->expired = rq->arrays + 1;
-  rq->best_expired_prio = MAX_PRIO;

 #ifdef CONFIG_SMP
   rq->sd = NULL;
@@ -6929,6 +7058,9 @@ void __init sched_init(void)
   atomic_set(&rq->nr_iowait, 0);

   for (j = 0; j < 2; j++) {
+   struct prio_array *array;
+   int k;
+
    array = rq->arrays + j;
    for (k = 0; k < MAX_PRIO; k++) {
     INIT_LIST_HEAD(array->queue + k);
```

_
--
Regards,
vatsa

Subject: [PATCH 2/9] task group priority handling
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:53:58 GMT
View Forum Message <> Reply to Message

To retain good interactivity, priority of a task-group is defined to be the
same as the priority of the highest-priority runnable task it has in its
runqueue.

This means as tasks come and go, priority of a task-group can change.

This patch deals with such changes to task-group priority.

P.S : Is this a good idea? Don't know. Other option is to to have a static
task-group priority which is decided by its quota.

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

1 file changed, 78 insertions(+), 2 deletions(-)

```
diff -puN kernel/sched.c~rfc kernel/sched.c
--- linux-2.6.20/kernel/sched.c~rfc 2007-04-11 12:00:48.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:18.000000000 +0530
@@ -192,7 +192,7 @@ static inline unsigned int task_timeslic

 struct prio_array {
  unsigned int nr_active;
- int best_static_prio;
+ int best_static_prio, best_dyn_prio;
  DECLARE_BITMAP(bitmap, MAX_PRIO+1); /* include 1 bit for delimiter */
  struct list_head queue[MAX_PRIO];
 };
@@ -729,6 +729,55 @@ sched_info_switch(struct task_struct *pr
 #define sched_info_switch(t, next) do { } while (0)
 #endif /* CONFIG_SCHEDSTATS || CONFIG_TASK_DELAY_ACCT */

+/* Dequeue task-group object from the main per-cpu runqueue */
+static void dequeue_task_grp(struct task_grp_rq *tgrq)
+{
+ struct prio_array *array = tgrq->rq_array;
+
```

```
+ array->nr_active--;
+ list_del(&tgrq->list);
+ if (list_empty(array->queue + tgrq->prio))
+  __clear_bit(tgrq->prio, array->bitmap);
+ tgrq->rq_array = NULL;
+}
+
+/* Enqueue task-group object on the main per-cpu runqueue */
+static void enqueue_task_grp(struct task_grp_rq *tgrq, struct prio_array *array,
+    int head)
+{
+ if (!head)
+  list_add_tail(&tgrq->list, array->queue + tgrq->prio);
+ else
+         list_add(&tgrq->list, array->queue + tgrq->prio);
+        __set_bit(tgrq->prio, array->bitmap);
+        array->nr_active++;
+        tgrq->rq_array = array;
+}
+
+/* Priority of a task-group is defined to be the priority of the highest
+ * priority runnable task it has. As tasks belonging to a task-group come in
+ * and go, the task-group priority can change on a particular CPU.
+ */
+static inline void update_task_grp_prio(struct task_grp_rq *tgrq, struct rq *rq,
+         int head)
+{
+ int new_prio;
+ struct prio_array *array = tgrq->rq_array;
+
+ new_prio = tgrq->active->best_dyn_prio;
+ if (new_prio == MAX_PRIO)
+  new_prio = tgrq->expired->best_dyn_prio;
+
+ if (array)
+  dequeue_task_grp(tgrq);
+ tgrq->prio = new_prio;
+ if (new_prio != MAX_PRIO) {
+  if (!array)
+   array = rq->active;
+  enqueue_task_grp(tgrq, array, head);
+ }
+}
+
 /*
  * Adding/removing a task to/from a priority array:
  */
@@ -736,8 +785,17 @@ static void dequeue_task(struct task_str
```

```
 {
  array->nr_active--;
  list_del(&p->run_list);
- if (list_empty(array->queue + p->prio))
+ if (list_empty(array->queue + p->prio)) {
    __clear_bit(p->prio, array->bitmap);
+   if (p->prio == array->best_dyn_prio) {
+    struct task_grp_rq *tgrq = task_grp_rq(p);
+
+    array->best_dyn_prio =
+      sched_find_first_bit(array->bitmap);
+    if (array == tgrq->active || !tgrq->active->nr_active)
+     update_task_grp_prio(tgrq, task_rq(p), 0);
+   }
+ }
 }

 static void enqueue_task(struct task_struct *p, struct prio_array *array)
@@ -747,6 +805,14 @@ static void enqueue_task(struct task_str
  __set_bit(p->prio, array->bitmap);
  array->nr_active++;
  p->array = array;
+
+ if (p->prio < array->best_dyn_prio) {
+   struct task_grp_rq *tgrq = task_grp_rq(p);
+
+   array->best_dyn_prio = p->prio;
+   if (array == tgrq->active || !tgrq->active->nr_active)
+    update_task_grp_prio(tgrq, task_rq(p), 0);
+ }
 }

 /*
@@ -765,6 +831,14 @@ enqueue_task_head(struct task_struct *p,
  __set_bit(p->prio, array->bitmap);
  array->nr_active++;
  p->array = array;
+
+ if (p->prio < array->best_dyn_prio) {
+   struct task_grp_rq *tgrq = task_grp_rq(p);
+
+   array->best_dyn_prio = p->prio;
+   if (array == tgrq->active || !tgrq->active->nr_active)
+    update_task_grp_prio(tgrq, task_rq(p), 1);
+ }
 }

 /*
```

```
@@ -7010,7 +7084,9 @@ static void task_grp_rq_init(struct task
  tgrq->expired = tgrq->arrays + 1;
  tgrq->rq_array = NULL;
  tgrq->expired->best_static_prio = MAX_PRIO;
+ tgrq->expired->best_dyn_prio = MAX_PRIO;
  tgrq->active->best_static_prio = MAX_PRIO;
+ tgrq->active->best_dyn_prio = MAX_PRIO;
  tgrq->prio = MAX_PRIO;
  tgrq->tg = tg;
  INIT_LIST_HEAD(&tgrq->list);
```

_
--
Regards,
vatsa

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

---

Subject: [PATCH 3/9] time slice management
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:54:52 GMT
View Forum Message <> Reply to Message

This patch does the actual job of dividing up CPU time amonmg various
task-groups as per their quota.

High-level overview:

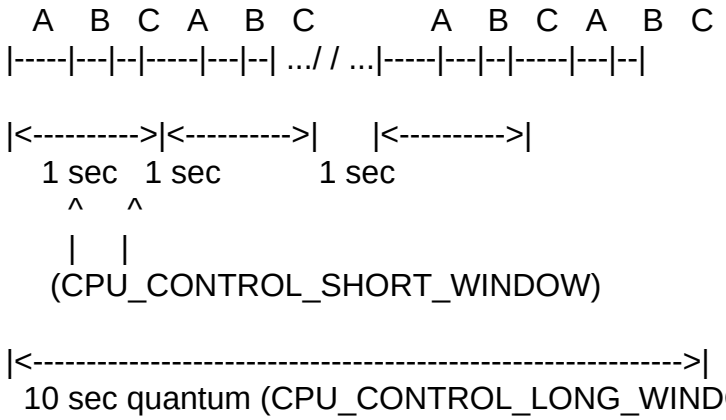
Lets say that we have three task-groups/classes A, B and C
whose quota is 50%, 30% and 20%. The quota specifies how
much of execution time each group gets per some quantum.
If 10 seconds is chosen as this quantum, then group A's tasks
get 5 seconds worth of execution time -every- 10 seconds,
grp B gets 3 seconds and so on, as shown below:

```
 A (5 sec)       B (3 sec)     C (2 sec)
|-----------------------|---------------|-----------|

T0<---------------- 10 sec quantum ----------------->T1
```

In this scheme, grp C's tasks could starve potentially,
waiting for their turn.

This patch avoids the above problem by breaking the 10 sec
quantum into several smaller quantums. A, B and C get their
due share in each smaller quantum and hence each group
progress more quickly than in the above scheme.



The latter scheme is implemented by maintaining two counters (tokens)
for each task-group (on each CPU).

 o ticks -> which is exhausted over the short quantum
 o long_ticks -> which is exhausted over the longer quantum

To begin with:

 grp->ticks = grp->quota * CPU_CONTROL_SHORT_WINDOW * HZ
 grp->long_ticks = CPU_CONTROL_LONG_WINDOW/CPU_CONTROL_SHORT_WINDOW

grp->ticks is decremented every timer ticks. When it hits zero,
grp->long_ticks is decremented. If resulting grp->long_ticks is
non-zero, grp->ticks wraps back to the initial value and the group
is put in an expired bucket. Else if resulting grp->long_ticks is
zero, the group is put in a greedy bucket.

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

---

 1 file changed, 115 insertions(+), 9 deletions(-)

```
diff -puN kernel/sched.c~grp_timeslice kernel/sched.c
--- linux-2.6.20/kernel/sched.c~grp_timeslice 2007-04-11 12:00:49.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:18.000000000 +0530
@@ -158,9 +158,6 @@
  (JIFFIES_TO_NS(MAX_SLEEP_AVG * \
   (MAX_BONUS / 2 + DELTA((p)) + 1) / MAX_BONUS - 1))

-#define TASK_PREEMPTS_CURR(p, rq) \
- ((p)->prio < (rq)->curr->prio)
-
 #define SCALE_PRIO(x, prio) \
  max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO / 2), MIN_TIMESLICE)

@@ -218,7 +215,9 @@ struct task_grp_rq {
  struct prio_array arrays[2];
  struct prio_array *active, *expired, *rq_array;
  unsigned long expired_timestamp;
- int prio;   /* Priority of the task-group */
+ unsigned short ticks, long_ticks;
+    int prio;   /* Priority of the task-group */
+ unsigned long last_update;
  struct list_head list;
  struct task_grp *tg;
 };
@@ -227,6 +226,7 @@ static DEFINE_PER_CPU(struct task_grp_rq

 /* task-group object - maintains information about each task-group */
 struct task_grp {
+ unsigned short ticks, long_ticks; /* bandwidth given to task-group */
  struct task_grp_rq *rq[NR_CPUS]; /* runqueue pointer for every cpu */
 };

@@ -270,7 +270,9 @@ struct rq {
   * xxx: Avoid this for !CONFIG_CPUMETER?
   */
- struct prio_array *active, *expired, arrays[2];
+ struct prio_array *active, *expired, *greedy_active, *greedy_expired;
+ struct prio_array arrays[4];
+ unsigned long last_update;
  atomic_t nr_iowait;

 #ifdef CONFIG_SMP
@@ -729,6 +731,14 @@ sched_info_switch(struct task_struct *pr
 #define sched_info_switch(t, next) do { } while (0)
 #endif /* CONFIG_SCHEDSTATS || CONFIG_TASK_DELAY_ACCT */

+#define CPU_CONTROL_SHORT_WINDOW        (HZ)
```

```
+#define CPU_CONTROL_LONG_WINDOW         (5 * HZ)
+#define NUM_LONG_TICKS  (unsigned short) (CPU_CONTROL_LONG_WINDOW / \
+       CPU_CONTROL_SHORT_WINDOW)
+
+#define greedy_grp(grp) (!grp->long_ticks)
+#define greedy_task(p) (greedy_grp(task_grp_rq(p)))
+
 /* Dequeue task-group object from the main per-cpu runqueue */
 static void dequeue_task_grp(struct task_grp_rq *tgrq)
 {
@@ -772,12 +782,29 @@ static inline void update_task_grp_prio(
   dequeue_task_grp(tgrq);
  tgrq->prio = new_prio;
  if (new_prio != MAX_PRIO) {
-  if (!array)
+  if (!greedy_grp(tgrq))
    array = rq->active;
+  else
+   array = rq->greedy_active;
   enqueue_task_grp(tgrq, array, head);
  }
 }

+#ifdef CONFIG_CPUMETER
+
+#define TASK_PREEMPTS_CURR(p, rq) \
+ (idle_cpu(task_cpu(p)) || \
+  (greedy_task((rq)->curr) && !greedy_task(p)) || \
+  ((task_grp_rq(p)->rq_array == task_grp_rq((rq)->curr)->rq_array) && \
+                                        (((p)->prio < (rq)->curr->prio))))
+
+#else
+
+#define TASK_PREEMPTS_CURR(p, rq) \
+         ((p)->prio < (rq)->curr->prio)
+
+#endif
+
 /*
  * Adding/removing a task to/from a priority array:
  */
@@ -3273,6 +3300,29 @@ void account_steal_time(struct task_stru
  }
 }

+#ifdef CONFIG_CPUMETER
+static inline void move_task_grp_list(struct prio_array *src,
+     struct prio_array *dst)
```

```
+{
+ unsigned int i;
+
+ if (!src->nr_active)
+  return;
+
+ for (i = 0; i < MAX_PRIO; i++) {
+  struct list_head *list = src->queue + i;
+
+  while (!list_empty(list)) {
+   struct task_grp_rq *tgrq;
+
+   tgrq = list_entry(list->next, struct task_grp_rq, list);
+   dequeue_task_grp(tgrq);
+   enqueue_task_grp(tgrq, dst, 0);
+  }
+ }
+}
+#endif
+
 static void task_running_tick(struct rq *rq, struct task_struct *p)
 {
  struct task_grp_rq *tgrq = task_grp_rq(p);
@@ -3347,6 +3397,39 @@ static void task_running_tick(struct rq
   }
  }
 out_unlock:
+#ifdef CONFIG_CPUMETER
+ if (!--tgrq->ticks) {
+                struct prio_array *target;
+
+  if (tgrq->long_ticks) {
+   --tgrq->long_ticks;
+   if (tgrq->long_ticks)
+    target = rq->expired;
+   else
+    target = rq->greedy_active;
+  } else /* should be in rq->greedy_active */
+   target = rq->greedy_expired;
+
+  /* Move the task group to expired list */
+  dequeue_task_grp(tgrq);
+  tgrq->ticks = task_grp(p)->ticks;
+  enqueue_task_grp(tgrq, target, 0);
+  set_tsk_need_resched(p);
+ }
+
+ if (unlikely(jiffies - rq->last_update > CPU_CONTROL_LONG_WINDOW)) {
```

```
+  if (rq->active->nr_active || rq->expired->nr_active) {
+   move_task_grp_list(rq->greedy_active, rq->active);
+   move_task_grp_list(rq->greedy_expired, rq->active);
+  } else {
+   switch_array(rq->active, rq->greedy_active);
+   switch_array(rq->expired, rq->greedy_expired);
+  }
+  rq->last_update = jiffies;
+  set_tsk_need_resched(p);
+ }
+#endif
+
  spin_unlock(&rq->lock);
 }

@@ -3643,12 +3726,27 @@ need_resched_nonpreemptible:
 #ifdef CONFIG_CPUMETER
  array = rq->active;
  if (unlikely(!array->nr_active)) {
-  switch_array(rq->active, rq->expired);
-  array = rq->active;
+  if (rq->expired->nr_active) {
+   switch_array(rq->active, rq->expired);
+   array = rq->active;
+  } else {
+   array = rq->greedy_active;
+   if (!array->nr_active) {
+    switch_array(rq->greedy_active,
+      rq->greedy_expired);
+    array = rq->greedy_active;
+   }
+  }
  }
  idx = sched_find_first_bit(array->bitmap);
  queue = array->queue + idx;
  next_grp = list_entry(queue->next, struct task_grp_rq, list);
+
+ if (unlikely(next_grp->last_update != rq->last_update)) {
+  next_grp->ticks = next_grp->tg->ticks;
+  next_grp->long_ticks = next_grp->tg->long_ticks;
+  next_grp->last_update = rq->last_update;
+ }
 #else
  next_grp = init_task_grp.rq[cpu];
 #endif
@@ -7088,6 +7186,8 @@ static void task_grp_rq_init(struct task
  tgrq->active->best_static_prio = MAX_PRIO;
  tgrq->active->best_dyn_prio = MAX_PRIO;
```

```
  tgrq->prio = MAX_PRIO;
+ tgrq->ticks = tg->ticks;
+ tgrq->long_ticks = tg->long_ticks;
  tgrq->tg = tg;
  INIT_LIST_HEAD(&tgrq->list);

@@ -7108,6 +7208,9 @@ void __init sched_init(void)
 {
  int i, j;

+ init_task_grp.ticks = CPU_CONTROL_SHORT_WINDOW;   /* 100% bandwidth */
+ init_task_grp.long_ticks = NUM_LONG_TICKS;
+
  for_each_possible_cpu(i) {
   struct rq *rq;
   struct task_grp_rq *tgrq;
@@ -7120,6 +7223,9 @@ void __init sched_init(void)
   rq->nr_running = 0;
   rq->active = rq->arrays;
   rq->expired = rq->arrays + 1;
+  rq->greedy_active = rq->arrays + 2;
+  rq->greedy_expired = rq->arrays + 3;
+  rq->last_update = tgrq->last_update = jiffies;

 #ifdef CONFIG_SMP
   rq->sd = NULL;
@@ -7133,7 +7239,7 @@ void __init sched_init(void)
 #endif
   atomic_set(&rq->nr_iowait, 0);

-  for (j = 0; j < 2; j++) {
+  for (j = 0; j < 4; j++) {
    struct prio_array *array;
    int k;


_
--
Regards,
vatsa
```

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

---

Subject: [PATCH 4/9] define group operations
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:56:08 GMT

Define these operations for a task-group:

- create new group
- destroy existing group
- set bandwidth (quota) for a group
- get bandwidth (quota) of a group

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

1 file changed, 71 insertions(+)

```
diff -puN kernel/sched.c~grp_ops kernel/sched.c
--- linux-2.6.20/kernel/sched.c~grp_ops 2007-04-11 12:00:50.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:17.000000000 +0530
@@ -7379,3 +7379,74 @@ void set_curr_task(int cpu, struct task_
 }

 #endif
+
+#ifdef CONFIG_CPUMETER
+
+/* Allocate runqueue structures for the new task-group */
+static int sched_create_group(void)
+{
+ struct task_grp *tg;
+ struct task_grp_rq *tgrq;
+ int i;
+
+ tg = kzalloc(sizeof(*tg), GFP_KERNEL);
+ if (!tg)
+  return -ENOMEM;
+
+ tg->ticks = CPU_CONTROL_SHORT_WINDOW;
+ tg->long_ticks = NUM_LONG_TICKS;
+
+ for_each_possible_cpu(i) {
+  tgrq = kzalloc(sizeof(*tgrq), GFP_KERNEL);
+  if (!tgrq)
+   goto oom;
+  tg->rq[i] = tgrq;
+  task_grp_rq_init(tgrq, tg);
+ }
```

```
+
+ /* A later patch will make 'tg' accessible beyond this function */
+ return 0;
+oom:
+ while (i--)
+  kfree(tg->rq[i]);
+
+ kfree(tg);
+ return -ENOMEM;
+}
+
+/* Deallocate runqueue structures */
+static void sched_destroy_group(struct task_grp *tg)
+{
+ int i;
+
+ for_each_possible_cpu(i)
+  kfree(tg->rq[i]);
+
+ kfree(tg);
+}
+
+/* Assign quota to this group */
+static int sched_set_quota(struct task_grp *tg, int quota)
+{
+ int i;
+
+ tg->ticks = (quota * CPU_CONTROL_SHORT_WINDOW) / 100;
+
+ for_each_possible_cpu(i)
+  tg->rq[i]->ticks = tg->ticks;
+
+ return 0;
+}
+
+static inline int cpu_quota(struct task_grp *tg)
+{
+ return (tg->ticks * 100) / CPU_CONTROL_SHORT_WINDOW;
+}
+
+/* Return assigned quota for this group */
+static int sched_get_quota(struct task_grp *tg)
+{
+ return cpu_quota(tg);
+}
+
+#endif /* CONFIG_CPUMETER */

_
```

--
Regards,
vatsa

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

---

Subject: [PATCH 5/9] deal with movement of tasks across groups
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:57:11 GMT
View Forum Message <> Reply to Message

When a task moves between groups (as initiated by an administrator), it has to
be removed from the runqueue of its old group and added to the runqueue of its
new group. This patch defines this move operation.

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

1 file changed, 28 insertions(+)

```
diff -puN kernel/sched.c~task_move kernel/sched.c
--- linux-2.6.20/kernel/sched.c~task_move 2007-04-11 12:00:51.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:16.000000000 +0530
@@ -7449,4 +7449,32 @@ static int sched_get_quota(struct task_g
  return cpu_quota(tg);
 }

+/*
+ * Move a task from one group to another. If the task is already on a
+ * runqueue, this involves removing the task from its old group's runqueue
+ * and adding to its new group's runqueue.
+ */
+static void sched_move_task(struct task_grp *tg_new, struct task_grp *tg_old,
+     struct task_struct *tsk)
+{
+ struct rq *rq;
+ unsigned long flags;
+
+ if (tg_new == tg_old)
+  return;
+
+ rq = task_rq_lock(tsk, &flags);
```

```
+
+ if (tsk->array) {
+  /* Set tsk->group to tg_old here */
+  deactivate_task(tsk, rq);
+  /* Set tsk->group to tg_new here */
+  __activate_task(tsk, rq);
+ }
+
+ task_rq_unlock(rq, &flags);
+
+ return;
+}
+
 #endif /* CONFIG_CPUMETER */

_
--
Regards,
vatsa
```

Subject: [PATCH 6/9] Handle dont care groups
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:57:54 GMT
View Forum Message <> Reply to Message

Deal with task-groups whose bandwidth hasnt been explicitly set by the
administrator. Unallocated CPU bandwidth is equally distributed among such
"don't care" groups.

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

```
 linux-2.6.18-root/include/linux/sched.h |    2
```

```
 1 file changed, 91 insertions(+), 5 deletions(-)
```

```
diff -puN kernel/sched.c~dont_care kernel/sched.c
--- linux-2.6.20/kernel/sched.c~dont_care 2007-04-12 09:09:48.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:15.000000000 +0530
```

```
@@ -227,6 +227,12 @@ static DEFINE_PER_CPU(struct task_grp_rq
 /* task-group object - maintains information about each task-group */
 struct task_grp {
  unsigned short ticks, long_ticks; /* bandwidth given to task-group */
+  int left_over_pct;
+  int total_dont_care_grps;
+  int dont_care;          /* Does this group care for its bandwidth ? */
+  struct task_grp *parent;
+  struct list_head dont_care_list;
+  struct list_head list;
  struct task_grp_rq *rq[NR_CPUS]; /* runqueue pointer for every cpu */
 };

@@ -7210,6 +7216,12 @@ void __init sched_init(void)

  init_task_grp.ticks = CPU_CONTROL_SHORT_WINDOW;   /* 100% bandwidth */
  init_task_grp.long_ticks = NUM_LONG_TICKS;
+  init_task_grp.left_over_pct = 100;  /* 100% unallocated bandwidth */
+  init_task_grp.parent = NULL;
+  init_task_grp.total_dont_care_grps = 1; /* init_task_grp itself */
+  init_task_grp.dont_care = 1;
+  INIT_LIST_HEAD(&init_task_grp.dont_care_list);
+ list_add_tail(&init_task_grp.list, &init_task_grp.dont_care_list);

  for_each_possible_cpu(i) {
   struct rq *rq;
@@ -7382,19 +7394,50 @@ void set_curr_task(int cpu, struct task_

 #ifdef CONFIG_CPUMETER

+/* Distribute left over bandwidth equally to all "dont care" task groups */
+static void recalc_dontcare(struct task_grp *tg_root)
+{
+ int ticks;
+ struct list_head *entry;
+
+ if (!tg_root->total_dont_care_grps)
+  return;
+
+ ticks = ((tg_root->left_over_pct / tg_root->total_dont_care_grps) *
+           CPU_CONTROL_SHORT_WINDOW) / 100;
+
+ list_for_each(entry, &tg_root->dont_care_list) {
+  struct task_grp *tg;
+  int i;
+
+  tg = list_entry(entry, struct task_grp, list);
+  tg->ticks = ticks;
```

```
+  for_each_possible_cpu(i)
+   tg->rq[i]->ticks = tg->ticks;
+ }
+}
+
 /* Allocate runqueue structures for the new task-group */
-static int sched_create_group(void)
+static int sched_create_group(struct task_grp *tg_parent)
 {
  struct task_grp *tg;
  struct task_grp_rq *tgrq;
  int i;

+ if (tg_parent->parent)
+  /* We don't support hierarchical CPU res mgmt (yet) */
+  return -EINVAL;
+
  tg = kzalloc(sizeof(*tg), GFP_KERNEL);
  if (!tg)
   return -ENOMEM;

  tg->ticks = CPU_CONTROL_SHORT_WINDOW;
  tg->long_ticks = NUM_LONG_TICKS;
+ tg->parent = tg_parent;
+ tg->dont_care = 1;
+ tg->left_over_pct = 100;
+ INIT_LIST_HEAD(&tg->dont_care_list);

  for_each_possible_cpu(i) {
   tgrq = kzalloc(sizeof(*tgrq), GFP_KERNEL);
@@ -7404,6 +7447,15 @@ static int sched_create_group(void)
   task_grp_rq_init(tgrq, tg);
  }

+ if (tg->parent) {
+  tg->parent->total_dont_care_grps++;
+  list_add_tail(&tg->list, &tg->parent->dont_care_list);
+  recalc_dontcare(tg->parent);
+ } else {
+  tg->total_dont_care_grps = 1;
+  list_add_tail(&tg->list, &tg->dont_care_list);
+ }
+
  /* A later patch will make 'tg' accessible beyond this function */
  return 0;
 oom:
@@ -7418,6 +7470,16 @@ oom:
 static void sched_destroy_group(struct task_grp *tg)
```

```
 {
  int i;
+ struct task_grp *tg_root = tg->parent;
+
+ if (!tg_root)
+  tg_root = tg;
+
+ if (tg->dont_care) {
+  tg_root->total_dont_care_grps--;
+  list_del(&tg->list);
+  recalc_dontcare(tg_root);
+ }

  for_each_possible_cpu(i)
   kfree(tg->rq[i]);
@@ -7428,12 +7490,33 @@ static void sched_destroy_group(struct t
 /* Assign quota to this group */
 static int sched_set_quota(struct task_grp *tg, int quota)
 {
- int i;
+ int i, old_quota = 0;
+ struct task_grp *tg_root = tg->parent;
+
+ if (!tg_root)
+  tg_root = tg;
+
+ if (!tg->dont_care)
+  old_quota = (tg->ticks * 100) / CPU_CONTROL_SHORT_WINDOW;
+
+ if ((quota - old_quota) > tg_root->left_over_pct)
+  return -EINVAL;
+
+ if (tg->dont_care) {
+  tg->dont_care = 0;
+  tg_root->total_dont_care_grps--;
+  list_del(&tg->list);
+ }

  tg->ticks = (quota * CPU_CONTROL_SHORT_WINDOW) / 100;
+ for_each_possible_cpu(i) {
+    tg->rq[i]->ticks = tg->ticks;
+  tg->rq[i]->long_ticks = tg->long_ticks;
+ }

- for_each_possible_cpu(i)
-  tg->rq[i]->ticks = tg->ticks;
+ /* xxx: needs some locking */
+ tg_root->left_over_pct -= (quota - old_quota);
```

```
+ recalc_dontcare(tg_root);

  return 0;
 }
@@ -7446,7 +7529,10 @@ static inline int cpu_quota(struct task_
 /* Return assigned quota for this group */
 static int sched_get_quota(struct task_grp *tg)
 {
- return cpu_quota(tg);
+ if (tg->dont_care)
+  return 0;
+ else
+  return cpu_quota(tg);
 }

 /*

_
--
Regards,
vatsa
```

Subject: [PATCH 7/9] SMP Load balance changes

Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:58:44 GMT

View Forum Message <> Reply to Message

When balancing tasks across CPU runqueues, we need to take into account the quota of task-groups (for example: we dont want to pull all tasks of a group with 90% limit on the same CPU).

This is easily accomplished by piggy backing on the smpnice mechanism.

This patch also modifies move_tasks() to look at all the arrays of runqueue for pulling tasks.

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

---

 1 file changed, 119 insertions(+), 36 deletions(-)

```
diff -puN kernel/sched.c~smp kernel/sched.c
--- linux-2.6.20/kernel/sched.c~smp 2007-04-12 09:10:38.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:15.000000000 +0530
@@ -217,7 +217,7 @@ struct task_grp_rq {
  unsigned long expired_timestamp;
  unsigned short ticks, long_ticks;
    int prio;   /* Priority of the task-group */
- unsigned long last_update;
+ unsigned long last_update, raw_weighted_load;
  struct list_head list;
  struct task_grp *tg;
 };
@@ -925,6 +925,11 @@ static inline int __normal_prio(struct t
 #define RTPRIO_TO_LOAD_WEIGHT(rp) \
  (PRIO_TO_LOAD_WEIGHT(MAX_RT_PRIO) + LOAD_WEIGHT(rp))

+static inline int cpu_quota(struct task_grp *tg)
+{
+ return (tg->ticks * 100) / CPU_CONTROL_SHORT_WINDOW;
+}
+
 static void set_load_weight(struct task_struct *p)
 {
  if (has_rt_policy(p)) {
@@ -938,21 +943,25 @@ static void set_load_weight(struct task_
    p->load_weight = 0;
   else
 #endif
-   p->load_weight = RTPRIO_TO_LOAD_WEIGHT(p->rt_priority);
+   p->load_weight = (RTPRIO_TO_LOAD_WEIGHT(p->rt_priority)
+       * cpu_quota(task_grp(p))) / 100;
  } else
-  p->load_weight = PRIO_TO_LOAD_WEIGHT(p->static_prio);
+  p->load_weight = (PRIO_TO_LOAD_WEIGHT(p->static_prio)
+       * cpu_quota(task_grp(p))) / 100;
 }

 static inline void
-inc_raw_weighted_load(struct rq *rq, const struct task_struct *p)
+inc_raw_weighted_load(struct rq *rq, struct task_struct *p)
 {
  rq->raw_weighted_load += p->load_weight;
+ task_grp_rq(p)->raw_weighted_load += p->load_weight;
 }
```

```
 static inline void
-dec_raw_weighted_load(struct rq *rq, const struct task_struct *p)
+dec_raw_weighted_load(struct rq *rq, struct task_struct *p)
 {
  rq->raw_weighted_load -= p->load_weight;
+ task_grp_rq(p)->raw_weighted_load -= p->load_weight;
 }

 static inline void inc_nr_running(struct task_struct *p, struct rq *rq)
@@ -2295,42 +2304,30 @@ int can_migrate_task(struct task_struct
  return 1;
 }

-#define rq_best_prio(rq) min((rq)->curr->prio, (rq)->expired->best_static_prio)
+#define rq_best_prio(rq) min((rq)->active->best_dyn_prio, \
+     (rq)->expired->best_static_prio)

-/*
- * move_tasks tries to move up to max_nr_move tasks and max_load_move weighted
- * load from busiest to this_rq, as part of a balancing operation within
- * "domain". Returns the number of tasks moved.
- *
- * Called with both runqueues locked.
- */
-static int move_tasks(struct rq *this_rq, int this_cpu, struct rq *busiest,
-       unsigned long max_nr_move, unsigned long max_load_move,
-       struct sched_domain *sd, enum idle_type idle,
-       int *all_pinned)
+static int __move_tasks(struct task_grp_rq *this_rq, int this_cpu,
+   struct task_grp_rq *busiest, unsigned long max_nr_move,
+   unsigned long max_load_move, struct sched_domain *sd,
+   enum idle_type idle, int *all_pinned, long *load_moved)
 {
  int idx, pulled = 0, pinned = 0, this_best_prio, best_prio,
-     best_prio_seen, skip_for_load;
+     best_prio_seen = 0, skip_for_load;
  struct prio_array *array, *dst_array;
  struct list_head *head, *curr;
  struct task_struct *tmp;
- long rem_load_move;
+ long rem_load_move, grp_load_diff;
+
+ grp_load_diff = busiest->raw_weighted_load - this_rq->raw_weighted_load;
+ rem_load_move = grp_load_diff/2;

- if (max_nr_move == 0 || max_load_move == 0)
+ if (grp_load_diff < 0 || max_nr_move == 0 || max_load_move == 0)
   goto out;
```

```
- rem_load_move = max_load_move;
  pinned = 1;
  this_best_prio = rq_best_prio(this_rq);
  best_prio = rq_best_prio(busiest);
- /*
-  * Enable handling of the case where there is more than one task
-  * with the best priority.   If the current running task is one
-  * of those with prio==best_prio we know it won't be moved
-  * and therefore it's safe to override the skip (based on load) of
-  * any task we find with that prio.
-  */
- best_prio_seen = best_prio == busiest->curr->prio;

   /*
    * We first consider expired tasks. Those will likely not be
@@ -2379,7 +2376,7 @@ skip_queue:
   if (skip_for_load && idx < this_best_prio)
    skip_for_load = !best_prio_seen && idx == best_prio;
   if (skip_for_load ||
-      !can_migrate_task(tmp, busiest, this_cpu, sd, idle, &pinned)) {
+      !can_migrate_task(tmp, task_rq(tmp), this_cpu, sd, idle, &pinned)) {

    best_prio_seen |= idx == best_prio;
    if (curr != head)
@@ -2388,7 +2385,8 @@ skip_queue:
    goto skip_bitmap;
   }

- pull_task(busiest, array, tmp, this_rq, dst_array, this_cpu);
+ pull_task(task_rq(tmp), array, tmp, cpu_rq(this_cpu), dst_array,
+         this_cpu);
   pulled++;
   rem_load_move -= tmp->load_weight;

@@ -2414,9 +2412,98 @@ out:

   if (all_pinned)
    *all_pinned = pinned;
+ *load_moved = grp_load_diff/2 - rem_load_move;
   return pulled;
  }

+static inline int choose_next_array(struct prio_array *arr[], int start_idx)
+{
+ int i;
+
+ for (i = start_idx; arr[i]; i++)
```

```
+  if (arr[i]->nr_active)
+   break;
+
+ return i;
+}
+
+/*
+ * move_tasks tries to move up to max_nr_move tasks and max_load_move weighted
+ * load from busiest to this_rq, as part of a balancing operation within
+ * "domain". Returns the number of tasks moved.
+ *
+ * Called with both runqueues locked.
+ */
+static int move_tasks(struct rq *this_rq, int this_cpu, struct rq *busiest,
+       unsigned long max_nr_move, unsigned long max_load_move,
+       struct sched_domain *sd, enum idle_type idle,
+       int *all_pinned)
+{
+ struct task_grp_rq *busy_grp, *this_grp;
+ long load_moved;
+ unsigned long total_nr_moved = 0, nr_moved;
+ int idx;
+ int arr_idx = 0;
+ struct list_head *head, *curr;
+ struct prio_array *array, *array_prefs[5];
+
+
+ /* order in which tasks are picked up */
+ array_prefs[0] = busiest->greedy_expired;
+ array_prefs[1] = busiest->greedy_active;
+ array_prefs[2] = busiest->expired;
+ array_prefs[3] = busiest->active;
+ array_prefs[4] = NULL;
+
+ arr_idx = choose_next_array(array_prefs, arr_idx);
+ array = array_prefs[arr_idx++];
+
+ if (!array)
+  goto out;
+
+new_array:
+ /* Start searching at priority 0: */
+ idx = 0;
+skip_bitmap:
+ if (!idx)
+  idx = sched_find_first_bit(array->bitmap);
+ else
+  idx = find_next_bit(array->bitmap, MAX_PRIO, idx);
```

```
+ if (idx >= MAX_PRIO) {
+  arr_idx = choose_next_array(array_prefs, arr_idx);
+  array = array_prefs[arr_idx++];
+  if (!array)
+   goto out;
+  goto new_array;
+ }
+
+ head = array->queue + idx;
+ curr = head->prev;
+skip_queue:
+ busy_grp = list_entry(curr, struct task_grp_rq, list);
+ this_grp = busy_grp->tg->rq[this_cpu];
+
+ curr = curr->prev;
+
+ nr_moved = __move_tasks(this_grp, this_cpu, busy_grp, max_nr_move,
+   max_load_move, sd, idle, all_pinned, &load_moved);
+
+ total_nr_moved += nr_moved;
+ max_nr_move -= nr_moved;
+ max_load_move -= load_moved;
+
+ if (!max_nr_move || (long)max_load_move <= 0)
+  goto out;
+
+ if (curr != head)
+  goto skip_queue;
+ idx++;
+ goto skip_bitmap;
+
+out:
+ return total_nr_moved;
+}
+
 /*
  * find_busiest_group finds and returns the busiest CPU group within the
  * domain. It calculates and returns the amount of weighted load which
@@ -7521,11 +7608,6 @@ static int sched_set_quota(struct task_g
  return 0;
 }

-static inline int cpu_quota(struct task_grp *tg)
-{
- return (tg->ticks * 100) / CPU_CONTROL_SHORT_WINDOW;
-}
-
 /* Return assigned quota for this group */
```

```
 static int sched_get_quota(struct task_grp *tg)
 {
@@ -7555,6 +7637,7 @@ static void sched_move_task(struct task_
   /* Set tsk->group to tg_old here */
   deactivate_task(tsk, rq);
   /* Set tsk->group to tg_new here */
+  set_load_weight(tsk);
   __activate_task(tsk, rq);
  }
```

_
--
Regards,
vatsa

Subject: [PATCH 8/9] task_cpu(p) needs to be correct always
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 17:59:37 GMT
View Forum Message <> Reply to Message

We rely very much on task_cpu(p) to be correct at all times, so that we can
correctly find the task_grp_rq from which the task has to be removed or added
to.

There is however one place in the scheduler where this assumption of
task_cpu(p) being correct is broken. This patch fixes that piece of code.

(Thanks to Balbir Singh for pointing this out to me)

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

---

 1 file changed, 8 insertions(+), 2 deletions(-)

```
diff -puN kernel/sched.c~task_cpu kernel/sched.c
--- linux-2.6.20/kernel/sched.c~task_cpu 2007-04-12 09:10:39.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:14.000000000 +0530
```

```
@@ -5400,6 +5400,7 @@ static int __migrate_task(struct task_st
 {
  struct rq *rq_dest, *rq_src;
  int ret = 0;
+ struct prio_array *array;

  if (unlikely(cpu_is_offline(dest_cpu)))
   return ret;
@@ -5415,8 +5416,8 @@ static int __migrate_task(struct task_st
  if (!cpu_isset(dest_cpu, p->cpus_allowed))
   goto out;

- set_task_cpu(p, dest_cpu);
- if (p->array) {
+ array = p->array;
+ if (array) {
   /*
    * Sync timestamp with rq_dest's before activating.
    * The same thing could be achieved by doing this step
@@ -5426,6 +5427,11 @@ static int __migrate_task(struct task_st
   p->timestamp = p->timestamp - rq_src->most_recent_timestamp
     + rq_dest->most_recent_timestamp;
   deactivate_task(p, rq_src);
+ }
+
+ set_task_cpu(p, dest_cpu);
+
+ if (array) {
   __activate_task(p, rq_dest);
   if (TASK_PREEMPTS_CURR(p, rq_dest))
    resched_task(rq_dest->curr);
```

_
--
Regards,
vatsa

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

---

Subject: [PATCH 9/9] Interface with process container patchset
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 18:01:27 GMT
View Forum Message <> Reply to Message

This patch registers cpu controller as a subsystem in process container
patches.

How to use the controller:

1. Apply these patches first on top of 2.6.20 kernel
   http://lkml.org/lkml/2007/4/6/301
   http://lkml.org/lkml/2007/4/6/300
   http://marc.info/?l=ckrm-tech&m=117590464104300&w=2

2. Apply all patches in this thread (from 1 - 9)

3. Select CONFIG_CPUMETER and build kernel

After bootup:

```
# mkdir /dev/cpuctl
# mount -t container -o cpuctl none /dev/cpuctl
# cd /dev/cpuctl
# mkdir a       # Create group A
# mkdir b       # Create group B

# echo 80 > a/quota     # Give 80% quota to A
# echo 20 > b/quota     # Give 20% quota to B

# echo some_pid > a/tasks       # Move some_pid to Group A

# echo another_pid > b/tasks    # move another task to Group B
```

some_pid and another_pid should share CPU in the ratio of 80:20 now

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

```
 linux-2.6.20-vatsa/include/linux/container_subsys.h |    6
```

```
diff -puN include/linux/container_subsys.h~container_if include/linux/container_subsys.h
--- linux-2.6.20/include/linux/container_subsys.h~container_if 2007-04-12 09:10:40.000000000
+0530
+++ linux-2.6.20-vatsa/include/linux/container_subsys.h 2007-04-12 09:10:40.000000000 +0530
@@ -17,4 +17,10 @@ SUBSYS(cpuacct)

 /* */
```

```
+#ifdef CONFIG_CPUMETER
+SUBSYS(cpuctlr)
+#endif
+
+/* */
+
 /* */
diff -puN kernel/sched.c~container_if kernel/sched.c
--- linux-2.6.20/kernel/sched.c~container_if 2007-04-12 09:10:40.000000000 +0530
+++ linux-2.6.20-vatsa/kernel/sched.c 2007-04-12 11:07:10.000000000 +0530
@@ -53,6 +53,8 @@
 #include <linux/kprobes.h>
 #include <linux/delayacct.h>
 #include <linux/cpu_acct.h>
+#include <linux/container.h>
+#include <linux/fs.h>
 #include <asm/tlb.h>

 #include <asm/unistd.h>
@@ -226,6 +228,7 @@ static DEFINE_PER_CPU(struct task_grp_rq

 /* task-group object - maintains information about each task-group */
 struct task_grp {
+ struct container_subsys_state css;
  unsigned short ticks, long_ticks; /* bandwidth given to task-group */
   int left_over_pct;
   int total_dont_care_grps;
@@ -430,10 +433,8 @@ static inline void finish_lock_switch(st
 /* return the task-group to which a task belongs */
 static inline struct task_grp *task_grp(struct task_struct *p)
 {
- /* Simply return the default group for now. A later patch modifies
-  * this function.
-  */
- return &init_task_grp;
+ return container_of(task_subsys_state(p, cpuctlr_subsys_id),
+  struct task_grp, css);
 }

 /*
@@ -7487,6 +7488,12 @@ void set_curr_task(int cpu, struct task_

 #ifdef CONFIG_CPUMETER

+static struct task_grp *container_tg(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, cpuctlr_subsys_id),
```

```
+    struct task_grp, css);
+}
+
 /* Distribute left over bandwidth equally to all "dont care" task groups */
 static void recalc_dontcare(struct task_grp *tg_root)
 {
@@ -7511,16 +7518,26 @@ static void recalc_dontcare(struct task_
 }

 /* Allocate runqueue structures for the new task-group */
-static int sched_create_group(struct task_grp *tg_parent)
+static int sched_create_group(struct container_subsys *ss,
+    struct container *cont)
 {
- struct task_grp *tg;
+ struct task_grp *tg, *tg_parent;
  struct task_grp_rq *tgrq;
  int i;

- if (tg_parent->parent)
+ 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;
+ }
+
+ if (cont->parent->parent)
   /* We don't support hierarchical CPU res mgmt (yet) */
   return -EINVAL;

+ tg_parent = container_tg(cont->parent);
+
  tg = kzalloc(sizeof(*tg), GFP_KERNEL);
  if (!tg)
   return -ENOMEM;
@@ -7549,7 +7566,9 @@ static int sched_create_group(struct tas
   list_add_tail(&tg->list, &tg->dont_care_list);
  }

- /* A later patch will make 'tg' accessible beyond this function */
+ cont->subsys[cpuctlr_subsys_id] = &tg->css;
+ tg->css.container = cont;
+
  return 0;
 oom:
  while (i--)
@@ -7560,9 +7579,11 @@ oom:
```

```
 }

 /* Deallocate runqueue structures */
-static void sched_destroy_group(struct task_grp *tg)
+static void sched_destroy_group(struct container_subsys *ss,
+      struct container *cont)
 {
  int i;
+ struct task_grp *tg = container_tg(cont);
  struct task_grp *tg_root = tg->parent;

  if (!tg_root)
@@ -7581,10 +7602,22 @@ static void sched_destroy_group(struct t
 }

 /* Assign quota to this group */
-static int sched_set_quota(struct task_grp *tg, int quota)
+static ssize_t sched_set_quota(struct container *cont, struct cftype *cft,
+    struct file *file, const char __user *userbuf,
+    size_t nbytes, loff_t *ppos)
 {
+ struct task_grp *tg = container_tg(cont);
  int i, old_quota = 0;
  struct task_grp *tg_root = tg->parent;
+ int quota;
+ char buffer[64];
+
+ if (copy_from_user(buffer, userbuf, sizeof(quota)))
+                return -EFAULT;
+
+ buffer[sizeof(quota)] = 0;
+
+ quota = simple_strtoul(buffer, NULL, 10);

  if (!tg_root)
   tg_root = tg;
@@ -7611,16 +7644,29 @@ static int sched_set_quota(struct task_g
  tg_root->left_over_pct -= (quota - old_quota);
  recalc_dontcare(tg_root);

- return 0;
+ return nbytes;
 }

 /* Return assigned quota for this group */
-static int sched_get_quota(struct task_grp *tg)
-{
+static ssize_t sched_get_quota(struct container *cont, struct cftype *cft,
```

```
+   struct file *file, char __user *buf, size_t nbytes,
+   loff_t *ppos)
+{
+ struct task_grp *tg = container_tg(cont);
+ char quotabuf[64];
+ char *s = quotabuf;
+ int quota;
+
  if (tg->dont_care)
-  return 0;
+  quota = 0;
  else
-  return cpu_quota(tg);
+  quota = cpu_quota(tg);
+
+ s += sprintf(s, "%d\n", quota);
+
+ return simple_read_from_buffer(buf, nbytes, ppos, quotabuf,
+       s - quotabuf);
+
 }

 /*
@@ -7628,23 +7674,35 @@ static int sched_get_quota(struct task_g
  * runqueue, this involves removing the task from its old group's runqueue
  * and adding to its new group's runqueue.
  */
-static void sched_move_task(struct task_grp *tg_new, struct task_grp *tg_old,
-     struct task_struct *tsk)
+static void sched_move_task(struct container_subsys *ss, struct container *cont,
+   struct container *old_cont, struct task_struct *tsk)
 {
  struct rq *rq;
  unsigned long flags;

- if (tg_new == tg_old)
-  return;
-
  rq = task_rq_lock(tsk, &flags);

  if (tsk->array) {
-  /* Set tsk->group to tg_old here */
+  struct task_grp *tg_old, *tg_new;
+
+  task_lock(tsk);
+
+  tg_old = container_tg(old_cont);
+  tg_new = container_tg(cont);
```

```
+
+  /* deactivate_task and activate_task rely on
+   * tsk->containers->subsys[cpuctlr_subsys_id] to know the
+   * appropritate group from which the task has to be dequeued
+   * and queued. Modify that appropriately before invoking them
+   */
+  tsk->containers->subsys[cpuctlr_subsys_id] = &tg_old->css;
   deactivate_task(tsk, rq);
-  /* Set tsk->group to tg_new here */
+
+  tsk->containers->subsys[cpuctlr_subsys_id] = &tg_new->css;
   set_load_weight(tsk);
   __activate_task(tsk, rq);
+
+  task_unlock(tsk);
  }

  task_rq_unlock(rq, &flags);
@@ -7652,4 +7710,49 @@ static void sched_move_task(struct task_
  return;
 }

+static struct cftype cft_quota = {
+ .name = "quota",
+ .read = sched_get_quota,
+ .write = sched_set_quota,
+};
+
+static int sched_populate(struct container_subsys *ss, struct container *cont)
+{
+ int err;
+
+ if ((err = container_add_file(cont, &cft_quota)))
+  return err;
+
+ return err;
+}
+
+static void sched_exit_task(struct container_subsys *ss, struct task_struct *p)
+{
+ struct rq *rq;
+ unsigned long flags;
+
+ rq = task_rq_lock(p, &flags);
+
+ if (p->array) {
+  struct task_grp_rq *init_tgrq = init_task_grp.rq[task_cpu(p)];
+
```

```
+  dequeue_task(p, p->array);
+  enqueue_task(p, init_tgrq->active);
+ }
+
+ task_rq_unlock(rq, &flags);
+}
+
+
+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,
+ .exit = sched_exit_task,
+ .early_init = 1,
+};
+
 #endif /* CONFIG_CPUMETER */
_
```

--
Regards,
vatsa

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

---

Subject: Re: [ckrm-tech] [RFC | PATCH 0/9] CPU controller over process container
Posted by Srivatsa Vaddagiri on Thu, 12 Apr 2007 18:04:40 GMT
View Forum Message <> Reply to Message

On Thu, Apr 12, 2007 at 11:21:11PM +0530, Srivatsa Vaddagiri wrote:
> - Real-time tasks - Should be left alone as they are today?
> i.e real time tasks across groups should be scheduled as if
> they are in same group

This patchset doesnt handle this requirement (yet). Will handle in
future revisions based on feedback.

--
Regards,
vatsa

_____
Containers mailing list