Subject: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Srivatsa Vaddagiri on Wed, 23 May 2007 16:48:59 GMT
View Forum Message <> Reply to Message

Here's an attempt to extend CFS (v13) to be fair at a group level, rather than just at task level. The patch is in a very premature state (passes simple tests, smp load balance not supported yet) at this point. I am sending it out early to know if this is a good direction to proceed.

Salient points which needs discussion:

1. This patch reuses CFS core to achieve fairness at group level also.

   To make this possible, CFS core has been abstracted to deal with generic schedulable "entities" (tasks, users etc).

2. The per-cpu rb-tree has been split to be per-group per-cpu.

   schedule() now becomes two step on every cpu : pick a group first (from group rb-tree) and a task within that group next (from that group's task rb-tree)

3. Grouping mechanism - I have used 'uid' as the basis of grouping for timebeing (since that grouping concept is already in mainline today). The patch can be adapted to a more generic process grouping mechanism (like http://lkml.org/lkml/2007/4/27/146) later.

Some results below, obtained on a 4way (with HT) Intel Xeon box. All number are reflective of single CPU performance (tests were forced to run on single cpu since load balance is not yet supported).

	uid "vatsa" (make -s -j4 bzImage)	uid "guest" (make -s -j20 bzImage)
2.6.22-rc1	772.02 sec	497.42 sec (real)
2.6.22-rc1+cfs-v13	780.62 sec	478.35 sec (real)
2.6.22-rc1+cfs-v13+this patch	776.36 sec	776.68 sec (real)

[An exclusive cpuset containing only one CPU was created and the compilation jobs of both users were run simultaneously in this cpuset]

I also disabled CONFIG_FAIR_USER_SCHED and compared the results with cfs-v13:

uid "vatsa"
make -s -j4 bzImage

2.6.22-rc1+cfs-v13   395.57 sec (real)
2.6.22-rc1+cfs-v13+this_patch 388.54 sec (real)

There is no regression I can see (rather some improvement, which I can't understand atm). I will run more tests later to check this regression aspect.

Request your comments on the future direction to proceed!

--
Regards,
vatsa

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

---

Subject: [RFC] [PATCH 1/3] task_cpu(p) needs to be correct always
Posted by Srivatsa Vaddagiri on Wed, 23 May 2007 16:51:26 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>

---
 kernel/sched.c | 8 +++++---
 1 file changed, 5 insertions(+), 3 deletions(-)

Index: linux-2.6.21-rc7/kernel/sched.c
===
--- linux-2.6.21-rc7.orig/kernel/sched.c 2007-05-23 20:46:41.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched.c 2007-05-23 20:48:26.000000000 +0530
@@ -4571,7 +4571,7 @@
 static int __migrate_task(struct task_struct *p, int src_cpu, int dest_cpu)
 {
  struct rq *rq_dest, *rq_src;
- int ret = 0;
+ int ret = 0, on_rq;

```
  if (unlikely(cpu_is_offline(dest_cpu)))
   return ret;
@@ -4587,9 +4587,11 @@
  if (!cpu_isset(dest_cpu, p->cpus_allowed))
   goto out;

- set_task_cpu(p, dest_cpu);
- if (p->on_rq) {
+ on_rq = p->on_rq;
+ if (on_rq)
   deactivate_task(rq_src, p, 0);
+ set_task_cpu(p, dest_cpu);
+ if (on_rq) {
   activate_task(rq_dest, p, 0);
   check_preempt_curr(rq_dest, p);
  }

--
Regards,
vatsa
```

Subject: [RFC] [PATCH 2/3] Introduce two new structures - struct lrq and sched_entity

Posted by Srivatsa Vaddagiri on Wed, 23 May 2007 16:54:48 GMT

View Forum Message <> Reply to Message

This patch groups together fields used by CFS (for SCHED_NORMAL tasks)
in task_struct and runqueue into separate structures so that they can be
reused in a later patch.

'struct sched_entity' represents the attributes used by CFS for every
schedulable entity (task in this case).

'struct lrq' represents the runqueue used to store schedulable entities (tasks
in this case) and to maintain various clocks (ex: fair clock for tasks).

This patch also modifies rest of kernel to reflect these new structures.

Intended effect of this patch is zero on overall functionality of
scheduler.

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

```
---
 fs/proc/array.c           |    2
 include/linux/sched.h     |   44 ++++----
 kernel/exit.c             |    2
 kernel/posix-cpu-timers.c |   19 +--
 kernel/sched.c            |  195 +++++++++++++++++-----------------
 kernel/sched_debug.c      |   85 ++++++++--------
 kernel/sched_fair.c       |  238 ++++++++++++++++++++++-----------------------
 7 files changed, 301 insertions(+), 284 deletions(-)

Index: linux-2.6.21-rc7/fs/proc/array.c
===================================================================
--- linux-2.6.21-rc7.orig/fs/proc/array.c 2007-05-23 20:46:40.000000000 +0530
+++ linux-2.6.21-rc7/fs/proc/array.c 2007-05-23 20:48:34.000000000 +0530
@@ -412,7 +412,7 @@
   * Use CFS's precise accounting, if available:
   */
  if (!has_rt_policy(task)) {
-  utime = nsec_to_clock_t(task->sum_exec_runtime);
+  utime = nsec_to_clock_t(task->se.sum_exec_runtime);
   stime = 0;
  }

Index: linux-2.6.21-rc7/include/linux/sched.h
===================================================================
--- linux-2.6.21-rc7.orig/include/linux/sched.h 2007-05-23 20:46:40.000000000 +0530
+++ linux-2.6.21-rc7/include/linux/sched.h 2007-05-23 20:48:34.000000000 +0530
@@ -838,6 +838,29 @@
  void (*task_new) (struct rq *rq, struct task_struct *p);
 };

+/* CFS scheduling entity (task, user etc) statistics fields: */
+struct sched_entity {
+ int load_weight; /* for niceness load balancing purposes */
+ int on_rq;
+ struct rb_node run_node;
+ u64 wait_start_fair;
+ u64 wait_start;
+ u64 exec_start;
+ u64 sleep_start, sleep_start_fair;
+ u64 block_start;
+ u64 sleep_max;
+ u64 block_max;
+ u64 exec_max;
+ u64 wait_max;
+ u64 last_ran;
+
+ s64 wait_runtime;
```

```
+ u64 sum_exec_runtime;
+ s64 fair_key;
+ s64 sum_wait_runtime, sum_sleep_runtime;
+ unsigned long wait_runtime_overruns, wait_runtime_underruns;
+};
+
 struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
@@ -852,34 +875,15 @@
  int oncpu;
 #endif
 #endif
- int load_weight; /* for niceness load balancing purposes */

  int prio, static_prio, normal_prio;
- int on_rq;
  struct list_head run_list;
- struct rb_node run_node;
+ struct sched_entity se;

  unsigned short ioprio;
 #ifdef CONFIG_BLK_DEV_IO_TRACE
  unsigned int btrace_seq;
 #endif
- /* CFS scheduling class statistics fields: */
- u64 wait_start_fair;
- u64 wait_start;
- u64 exec_start;
- u64 sleep_start, sleep_start_fair;
- u64 block_start;
- u64 sleep_max;
- u64 block_max;
- u64 exec_max;
- u64 wait_max;
- u64 last_ran;
-
- s64 wait_runtime;
- u64 sum_exec_runtime;
- s64 fair_key;
- s64 sum_wait_runtime, sum_sleep_runtime;
- unsigned long wait_runtime_overruns, wait_runtime_underruns;

  unsigned int policy;
  cpumask_t cpus_allowed;
Index: linux-2.6.21-rc7/kernel/exit.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/exit.c 2007-05-23 20:46:40.000000000 +0530
```

```
+++ linux-2.6.21-rc7/kernel/exit.c 2007-05-23 20:48:34.000000000 +0530
@@ -124,7 +124,7 @@
   sig->nivcsw += tsk->nivcsw;
   sig->inblock += task_io_get_inblock(tsk);
   sig->oublock += task_io_get_oublock(tsk);
-  sig->sum_sched_runtime += tsk->sum_exec_runtime;
+  sig->sum_sched_runtime += tsk->se.sum_exec_runtime;
   sig = NULL; /* Marker for below. */
  }

Index: linux-2.6.21-rc7/kernel/posix-cpu-timers.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/posix-cpu-timers.c 2007-05-23 20:46:40.000000000 +0530
+++ linux-2.6.21-rc7/kernel/posix-cpu-timers.c 2007-05-23 20:48:34.000000000 +0530
@@ -161,7 +161,8 @@
 }
 static inline unsigned long long sched_ns(struct task_struct *p)
 {
- return (p == current) ? current_sched_runtime(p) : p->sum_exec_runtime;
+ return (p == current) ? current_sched_runtime(p) :
+     p->se.sum_exec_runtime;
 }

 int posix_cpu_clock_getres(const clockid_t which_clock, struct timespec *tp)
@@ -249,7 +250,7 @@
   cpu->sched = p->signal->sum_sched_runtime;
   /* Add in each other live thread.  */
   while ((t = next_thread(t)) != p) {
-   cpu->sched += t->sum_exec_runtime;
+   cpu->sched += t->se.sum_exec_runtime;
   }
   cpu->sched += sched_ns(p);
   break;
@@ -467,7 +468,7 @@
 void posix_cpu_timers_exit(struct task_struct *tsk)
 {
  cleanup_timers(tsk->cpu_timers,
-        tsk->utime, tsk->stime, tsk->sum_exec_runtime);
+        tsk->utime, tsk->stime, tsk->se.sum_exec_runtime);

 }
 void posix_cpu_timers_exit_group(struct task_struct *tsk)
@@ -475,7 +476,7 @@
  cleanup_timers(tsk->signal->cpu_timers,
         cputime_add(tsk->utime, tsk->signal->utime),
         cputime_add(tsk->stime, tsk->signal->stime),
-        tsk->sum_exec_runtime + tsk->signal->sum_sched_runtime);
+       tsk->se.sum_exec_runtime + tsk->signal->sum_sched_runtime);
```

```
 }


@@ -536,7 +537,7 @@
   nsleft = max_t(unsigned long long, nsleft, 1);
   do {
    if (likely(!(t->flags & PF_EXITING))) {
-    ns = t->sum_exec_runtime + nsleft;
+    ns = t->se.sum_exec_runtime + nsleft;
     if (t->it_sched_expires == 0 ||
         t->it_sched_expires > ns) {
      t->it_sched_expires = ns;
@@ -1004,7 +1005,7 @@
   struct cpu_timer_list *t = list_first_entry(timers,
             struct cpu_timer_list,
             entry);
-  if (!--maxfire || tsk->sum_exec_runtime < t->expires.sched) {
+  if (!--maxfire || tsk->se.sum_exec_runtime < t->expires.sched) {
    tsk->it_sched_expires = t->expires.sched;
    break;
   }
@@ -1049,7 +1050,7 @@
  do {
   utime = cputime_add(utime, t->utime);
   stime = cputime_add(stime, t->stime);
-  sum_sched_runtime += t->sum_exec_runtime;
+  sum_sched_runtime += t->se.sum_exec_runtime;
   t = next_thread(t);
  } while (t != tsk);
  ptime = cputime_add(utime, stime);
@@ -1208,7 +1209,7 @@
     t->it_virt_expires = ticks;
    }

-   sched = t->sum_exec_runtime + sched_left;
+   sched = t->se.sum_exec_runtime + sched_left;
    if (sched_expires && (t->it_sched_expires == 0 ||
          t->it_sched_expires > sched)) {
     t->it_sched_expires = sched;
@@ -1300,7 +1301,7 @@

  if (UNEXPIRED(prof) && UNEXPIRED(virt) &&
      (tsk->it_sched_expires == 0 ||
-      tsk->sum_exec_runtime < tsk->it_sched_expires))
+      tsk->se.sum_exec_runtime < tsk->it_sched_expires))
   return;

 #undef UNEXPIRED
```

```
Index: linux-2.6.21-rc7/kernel/sched.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/sched.c 2007-05-23 20:48:26.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched.c 2007-05-23 20:48:34.000000000 +0530
@@ -114,6 +114,23 @@
  struct list_head queue[MAX_RT_PRIO];
 };

+/* CFS-related fields in a runqueue */
+struct lrq {
+ unsigned long raw_weighted_load;
+ #define CPU_LOAD_IDX_MAX 5
+ unsigned long cpu_load[CPU_LOAD_IDX_MAX];
+ unsigned long nr_load_updates;
+
+ u64 fair_clock, prev_fair_clock;
+ u64 exec_clock, prev_exec_clock;
+ s64 wait_runtime;
+ unsigned long wait_runtime_overruns, wait_runtime_underruns;
+
+ struct rb_root tasks_timeline;
+ struct rb_node *rb_leftmost;
+ struct rb_node *rb_load_balance_curr;
+};
+
 /*
  * This is the main, per-CPU runqueue data structure.
  *
@@ -129,16 +146,13 @@
   * remote CPUs use both these fields when doing load calculation.
   */
  long nr_running;
- unsigned long raw_weighted_load;
- #define CPU_LOAD_IDX_MAX 5
- unsigned long cpu_load[CPU_LOAD_IDX_MAX];
+ struct lrq lrq;

  unsigned char idle_at_tick;
 #ifdef CONFIG_NO_HZ
  unsigned char in_nohz_recently;
 #endif
  u64 nr_switches;
- unsigned long nr_load_updates;

  /*
   * This is part of a global counter where only the total sum
@@ -154,10 +168,6 @@
```

```
  u64 clock, prev_clock_raw;
  s64 clock_max_delta;
- u64 fair_clock, prev_fair_clock;
- u64 exec_clock, prev_exec_clock;
- s64 wait_runtime;
- unsigned long wait_runtime_overruns, wait_runtime_underruns;

  unsigned int clock_warps;
  unsigned int clock_unstable_events;
@@ -168,10 +178,6 @@
  int rt_load_balance_idx;
  struct list_head *rt_load_balance_head, *rt_load_balance_curr;

- struct rb_root tasks_timeline;
- struct rb_node *rb_leftmost;
- struct rb_node *rb_load_balance_curr;
-
  atomic_t nr_iowait;

 #ifdef CONFIG_SMP
@@ -573,13 +579,13 @@
 static inline void
 inc_raw_weighted_load(struct rq *rq, const struct task_struct *p)
 {
- rq->raw_weighted_load += p->load_weight;
+ rq->lrq.raw_weighted_load += p->se.load_weight;
 }

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

 static inline void inc_nr_running(struct task_struct *p, struct rq *rq)
@@ -605,22 +611,22 @@

 static void set_load_weight(struct task_struct *p)
 {
- task_rq(p)->wait_runtime -= p->wait_runtime;
- p->wait_runtime = 0;
+ task_rq(p)->lrq.wait_runtime -= p->se.wait_runtime;
+ p->se.wait_runtime = 0;

  if (has_rt_policy(p)) {
-  p->load_weight = prio_to_weight[0] * 2;
+  p->se.load_weight = prio_to_weight[0] * 2;
```

```
   return;
  }
  /*
   * SCHED_BATCH tasks get minimal weight:
   */
  if (p->policy == SCHED_BATCH) {
-  p->load_weight = 1;
+  p->se.load_weight = 1;
   return;
  }

- p->load_weight = prio_to_weight[p->static_prio - MAX_RT_PRIO];
+ p->se.load_weight = prio_to_weight[p->static_prio - MAX_RT_PRIO];
 }

 static void enqueue_task(struct rq *rq, struct task_struct *p, int wakeup)
@@ -629,7 +635,7 @@

  sched_info_queued(p);
  p->sched_class->enqueue_task(rq, p, wakeup, now);
- p->on_rq = 1;
+ p->se.on_rq = 1;
 }

 static void dequeue_task(struct rq *rq, struct task_struct *p, int sleep)
@@ -637,7 +643,7 @@
  u64 now = rq_clock(rq);

  p->sched_class->dequeue_task(rq, p, sleep, now);
- p->on_rq = 0;
+ p->se.on_rq = 0;
 }

 /*
@@ -725,7 +731,7 @@
 /* Used instead of source_load when we know the type == 0 */
 unsigned long weighted_cpuload(const int cpu)
 {
- return cpu_rq(cpu)->raw_weighted_load;
+ return cpu_rq(cpu)->lrq.raw_weighted_load;
 }

 #ifdef CONFIG_SMP
@@ -742,18 +748,18 @@
  u64 clock_offset, fair_clock_offset;

  clock_offset = old_rq->clock - new_rq->clock;
- fair_clock_offset = old_rq->fair_clock - new_rq->fair_clock;
```

```
+ fair_clock_offset = old_rq->lrq.fair_clock - new_rq->lrq.fair_clock;

- if (p->wait_start)
-  p->wait_start -= clock_offset;
- if (p->wait_start_fair)
-  p->wait_start_fair -= fair_clock_offset;
- if (p->sleep_start)
-  p->sleep_start -= clock_offset;
- if (p->block_start)
-  p->block_start -= clock_offset;
- if (p->sleep_start_fair)
-  p->sleep_start_fair -= fair_clock_offset;
+ if (p->se.wait_start)
+  p->se.wait_start -= clock_offset;
+ if (p->se.wait_start_fair)
+  p->se.wait_start_fair -= fair_clock_offset;
+ if (p->se.sleep_start)
+  p->se.sleep_start -= clock_offset;
+ if (p->se.block_start)
+  p->se.block_start -= clock_offset;
+ if (p->se.sleep_start_fair)
+  p->se.sleep_start_fair -= fair_clock_offset;

  task_thread_info(p)->cpu = new_cpu;

@@ -781,7 +787,7 @@
   * If the task is not on a runqueue (and not running), then
   * it is sufficient to simply update the task's cpu field.
   */
- if (!p->on_rq && !task_running(rq, p)) {
+ if (!p->se.on_rq && !task_running(rq, p)) {
   set_task_cpu(p, dest_cpu);
   return 0;
  }
@@ -812,7 +818,7 @@
 repeat:
  rq = task_rq_lock(p, &flags);
  /* Must be off runqueue entirely, not preempted. */
- if (unlikely(p->on_rq || task_running(rq, p))) {
+ if (unlikely(p->se.on_rq || task_running(rq, p))) {
   /* If it's preempted, we yield.  It could be a while. */
   preempted = !task_running(rq, p);
   task_rq_unlock(rq, &flags);
@@ -860,9 +866,9 @@
  struct rq *rq = cpu_rq(cpu);

  if (type == 0)
-  return rq->raw_weighted_load;
```

```
+  return rq->lrq.raw_weighted_load;

- return min(rq->cpu_load[type-1], rq->raw_weighted_load);
+ return min(rq->lrq.cpu_load[type-1], rq->lrq.raw_weighted_load);
 }

 /*
@@ -874,9 +880,9 @@
  struct rq *rq = cpu_rq(cpu);

  if (type == 0)
-  return rq->raw_weighted_load;
+  return rq->lrq.raw_weighted_load;

- return max(rq->cpu_load[type-1], rq->raw_weighted_load);
+ return max(rq->lrq.cpu_load[type-1], rq->lrq.raw_weighted_load);
 }

 /*
@@ -887,7 +893,7 @@
  struct rq *rq = cpu_rq(cpu);
  unsigned long n = rq->nr_running;

- return n ? rq->raw_weighted_load / n : SCHED_LOAD_SCALE;
+ return n ? rq->lrq.raw_weighted_load / n : SCHED_LOAD_SCALE;
 }

 /*
@@ -1118,7 +1124,7 @@
  if (!(old_state & state))
   goto out;

- if (p->on_rq)
+ if (p->se.on_rq)
   goto out_running;

  cpu = task_cpu(p);
@@ -1173,11 +1179,11 @@
     * of the current CPU:
     */
    if (sync)
-    tl -= current->load_weight;
+    tl -= current->se.load_weight;

    if ((tl <= load &&
-    tl + target_load(cpu, idx) <= tl_per_task) ||
-    100*(tl + p->load_weight) <= imbalance*load) {
+         tl + target_load(cpu, idx) <= tl_per_task) ||
```

```
+          100*(tl + p->se.load_weight) <= imbalance*load) {
      /*
       * This domain has SD_WAKE_AFFINE and
       * p is cache cold in this domain, and
@@ -1211,7 +1217,7 @@
    old_state = p->state;
    if (!(old_state & state))
     goto out;
-   if (p->on_rq)
+   if (p->se.on_rq)
     goto out_running;

    this_cpu = smp_processor_id();
@@ -1275,18 +1281,19 @@
   */
  static void __sched_fork(struct task_struct *p)
  {
- p->wait_start_fair = p->wait_start = p->exec_start = p->last_ran = 0;
- p->sum_exec_runtime = 0;
-
- p->wait_runtime = 0;
-
- p->sum_wait_runtime = p->sum_sleep_runtime = 0;
- p->sleep_start = p->sleep_start_fair = p->block_start = 0;
- p->sleep_max = p->block_max = p->exec_max = p->wait_max = 0;
- p->wait_runtime_overruns = p->wait_runtime_underruns = 0;
+ p->se.wait_start_fair = p->se.wait_start = p->se.exec_start = 0;
+ p->se.last_ran = 0;
+ p->se.sum_exec_runtime = 0;
+
+ p->se.wait_runtime = 0;
+
+ p->se.sum_wait_runtime = p->se.sum_sleep_runtime = 0;
+ p->se.sleep_start = p->se.sleep_start_fair = p->se.block_start = 0;
+ p->se.sleep_max = p->se.block_max = p->se.exec_max = p->se.wait_max = 0;
+ p->se.wait_runtime_overruns = p->se.wait_runtime_underruns = 0;

    INIT_LIST_HEAD(&p->run_list);
- p->on_rq = 0;
+ p->se.on_rq = 0;
    p->nr_switches = 0;

    /*
@@ -1357,7 +1364,7 @@
    p->prio = effective_prio(p);

    if (!sysctl_sched_child_runs_first || (clone_flags & CLONE_VM) ||
-       task_cpu(p) != this_cpu || !current->on_rq) {
```

```
+   task_cpu(p) != this_cpu || !current->se.on_rq) {
    activate_task(rq, p, 0);
  } else {
    /*
@@ -1372,7 +1379,7 @@

 void sched_dead(struct task_struct *p)
 {
- WARN_ON_ONCE(p->on_rq);
+ WARN_ON_ONCE(p->se.on_rq);
 }

 /**
@@ -1584,17 +1591,19 @@
  unsigned long tmp;
  u64 tmp64;

- this_rq->nr_load_updates++;
+ this_rq->lrq.nr_load_updates++;
  if (!(sysctl_sched_load_smoothing & 64)) {
-  this_load = this_rq->raw_weighted_load;
+  this_load = this_rq->lrq.raw_weighted_load;
    goto do_avg;
  }

- fair_delta64 = this_rq->fair_clock - this_rq->prev_fair_clock + 1;
- this_rq->prev_fair_clock = this_rq->fair_clock;
-
- exec_delta64 = this_rq->exec_clock - this_rq->prev_exec_clock + 1;
- this_rq->prev_exec_clock = this_rq->exec_clock;
+ fair_delta64 = this_rq->lrq.fair_clock -
+     this_rq->lrq.prev_fair_clock + 1;
+ this_rq->lrq.prev_fair_clock = this_rq->lrq.fair_clock;
+
+ exec_delta64 = this_rq->lrq.exec_clock -
+     this_rq->lrq.prev_exec_clock + 1;
+ this_rq->lrq.prev_exec_clock = this_rq->lrq.exec_clock;

  if (fair_delta64 > (s64)LONG_MAX)
    fair_delta64 = (s64)LONG_MAX;
@@ -1620,10 +1629,10 @@

    /* scale is effectively 1 << i now, and >> i divides by scale */

-  old_load = this_rq->cpu_load[i];
+  old_load = this_rq->lrq.cpu_load[i];
    new_load = this_load;
```

```
-  this_rq->cpu_load[i] = (old_load*(scale-1) + new_load) >> i;
+  this_rq->lrq.cpu_load[i] = (old_load*(scale-1) + new_load) >> i;
   }
  }

@@ -1879,7 +1888,8 @@
    * skip a task if it will be the highest priority task (i.e. smallest
    * prio value) on its new queue regardless of its load weight
    */
- skip_for_load = (p->load_weight >> 1) > rem_load_move + SCHED_LOAD_SCALE_FUZZ;
+ skip_for_load = (p->se.load_weight >> 1) > rem_load_move +
+         SCHED_LOAD_SCALE_FUZZ;
   if (skip_for_load && p->prio < this_best_prio)
    skip_for_load = !best_prio_seen && p->prio == best_prio;
   if (skip_for_load ||
@@ -1892,7 +1902,7 @@

   pull_task(busiest, p, this_rq, this_cpu);
   pulled++;
- rem_load_move -= p->load_weight;
+ rem_load_move -= p->se.load_weight;

   /*
    * We only want to steal up to the prescribed number of tasks
@@ -1989,7 +1999,7 @@

     avg_load += load;
     sum_nr_running += rq->nr_running;
-    sum_weighted_load += rq->raw_weighted_load;
+    sum_weighted_load += rq->lrq.raw_weighted_load;
    }

    /*
@@ -2223,11 +2233,12 @@

    rq = cpu_rq(i);

-   if (rq->nr_running == 1 && rq->raw_weighted_load > imbalance)
+   if (rq->nr_running == 1 &&
+     rq->lrq.raw_weighted_load > imbalance)
     continue;

-   if (rq->raw_weighted_load > max_load) {
-    max_load = rq->raw_weighted_load;
+   if (rq->lrq.raw_weighted_load > max_load) {
+    max_load = rq->lrq.raw_weighted_load;
     busiest = rq;
    }
```

```
  }
@@ -2830,7 +2841,7 @@
  unsigned long flags;

  local_irq_save(flags);
- ns = p->sum_exec_runtime + sched_clock() - p->last_ran;
+ ns = p->se.sum_exec_runtime + sched_clock() - p->se.last_ran;
  local_irq_restore(flags);

  return ns;
@@ -3518,7 +3529,7 @@
  rq = task_rq_lock(p, &flags);

  oldprio = p->prio;
- on_rq = p->on_rq;
+ on_rq = p->se.on_rq;
  if (on_rq)
   dequeue_task(rq, p, 0);

@@ -3571,7 +3582,7 @@
   p->static_prio = NICE_TO_PRIO(nice);
   goto out_unlock;
  }
- on_rq = p->on_rq;
+ on_rq = p->se.on_rq;
  if (on_rq) {
   dequeue_task(rq, p, 0);
   dec_raw_weighted_load(rq, p);
@@ -3708,7 +3719,7 @@
 static void
 __setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)
 {
- BUG_ON(p->on_rq);
+ BUG_ON(p->se.on_rq);

  p->policy = policy;
  switch (p->policy) {
@@ -3814,7 +3825,7 @@
   spin_unlock_irqrestore(&p->pi_lock, flags);
   goto recheck;
  }
- on_rq = p->on_rq;
+ on_rq = p->se.on_rq;
  if (on_rq)
   deactivate_task(rq, p, 0);
  oldprio = p->prio;
@@ -4468,7 +4479,7 @@
  unsigned long flags;
```

```
  __sched_fork(idle);
- idle->exec_start = sched_clock();
+ idle->se.exec_start = sched_clock();

  idle->prio = idle->normal_prio = MAX_PRIO;
  idle->cpus_allowed = cpumask_of_cpu(cpu);
@@ -4587,7 +4598,7 @@
  if (!cpu_isset(dest_cpu, p->cpus_allowed))
   goto out;

- on_rq = p->on_rq;
+ on_rq = p->se.on_rq;
  if (on_rq)
   deactivate_task(rq_src, p, 0);
  set_task_cpu(p, dest_cpu);
@@ -5986,11 +5997,11 @@
   spin_lock_init(&rq->lock);
   lockdep_set_class(&rq->lock, &rq->rq_lock_key);
   rq->nr_running = 0;
-  rq->tasks_timeline = RB_ROOT;
-  rq->clock = rq->fair_clock = 1;
+  rq->lrq.tasks_timeline = RB_ROOT;
+  rq->clock = rq->lrq.fair_clock = 1;

   for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
-   rq->cpu_load[j] = 0;
+   rq->lrq.cpu_load[j] = 0;
 #ifdef CONFIG_SMP
   rq->sd = NULL;
   rq->active_balance = 0;
@@ -6072,15 +6083,15 @@

  read_lock_irq(&tasklist_lock);
  for_each_process(p) {
-  p->fair_key = 0;
-  p->wait_runtime = 0;
-  p->wait_start_fair = 0;
-  p->wait_start = 0;
-  p->exec_start = 0;
-  p->sleep_start = 0;
-  p->sleep_start_fair = 0;
-  p->block_start = 0;
-  task_rq(p)->fair_clock = 0;
+  p->se.fair_key = 0;
+  p->se.wait_runtime = 0;
+  p->se.wait_start_fair = 0;
+  p->se.wait_start = 0;
```

```
+  p->se.exec_start = 0;
+  p->se.sleep_start = 0;
+  p->se.sleep_start_fair = 0;
+  p->se.block_start = 0;
+  task_rq(p)->lrq.fair_clock = 0;
   task_rq(p)->clock = 0;

   if (!rt_task(p)) {
@@ -6103,7 +6114,7 @@
    goto out_unlock;
 #endif

-  on_rq = p->on_rq;
+  on_rq = p->se.on_rq;
   if (on_rq)
    deactivate_task(task_rq(p), p, 0);
   __setscheduler(rq, p, SCHED_NORMAL, 0);
Index: linux-2.6.21-rc7/kernel/sched_debug.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/sched_debug.c 2007-05-23 20:46:40.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched_debug.c 2007-05-23 20:48:34.000000000 +0530
@@ -40,15 +40,16 @@
  SEQ_printf(m, "%15s %5d %15Ld %13Ld %13Ld %9Ld %5d "
        "%15Ld %15Ld %15Ld %15Ld %15Ld\n",
   p->comm, p->pid,
-  (long long)p->fair_key, (long long)p->fair_key - rq->fair_clock,
-  (long long)p->wait_runtime,
+  (long long)p->se.fair_key,
+  (long long)p->se.fair_key - rq->lrq.fair_clock,
+  (long long)p->se.wait_runtime,
   (long long)p->nr_switches,
   p->prio,
-  (long long)p->sum_exec_runtime,
-  (long long)p->sum_wait_runtime,
-  (long long)p->sum_sleep_runtime,
-  (long long)p->wait_runtime_overruns,
-  (long long)p->wait_runtime_underruns);
+  (long long)p->se.sum_exec_runtime,
+  (long long)p->se.sum_wait_runtime,
+  (long long)p->se.sum_sleep_runtime,
+  (long long)p->se.wait_runtime_overruns,
+  (long long)p->se.wait_runtime_underruns);
 }

 static void print_rq(struct seq_file *m, struct rq *rq, u64 now)
@@ -69,7 +70,7 @@

  curr = first_fair(rq);
```

```
  while (curr) {
-  p = rb_entry(curr, struct task_struct, run_node);
+  p = rb_entry(curr, struct task_struct, se.run_node);
   print_task(m, rq, p, now);

   curr = rb_next(curr);
@@ -86,8 +87,8 @@
  spin_lock_irqsave(&rq->lock, flags);
  curr = first_fair(rq);
  while (curr) {
-  p = rb_entry(curr, struct task_struct, run_node);
-  wait_runtime_rq_sum += p->wait_runtime;
+  p = rb_entry(curr, struct task_struct, se.run_node);
+  wait_runtime_rq_sum += p->se.wait_runtime;

   curr = rb_next(curr);
  }
@@ -106,9 +107,9 @@
  SEQ_printf(m, "  .%-22s: %Ld\n", #x, (long long)(rq->x))

  P(nr_running);
- P(raw_weighted_load);
+ P(lrq.raw_weighted_load);
  P(nr_switches);
- P(nr_load_updates);
+ P(lrq.nr_load_updates);
  P(nr_uninterruptible);
  SEQ_printf(m, "  .%-22s: %lu\n", "jiffies", jiffies);
  P(next_balance);
@@ -119,18 +120,18 @@
  P(clock_unstable_events);
  P(clock_max_delta);
  rq->clock_max_delta = 0;
- P(fair_clock);
- P(prev_fair_clock);
- P(exec_clock);
- P(prev_exec_clock);
- P(wait_runtime);
- P(wait_runtime_overruns);
- P(wait_runtime_underruns);
- P(cpu_load[0]);
- P(cpu_load[1]);
- P(cpu_load[2]);
- P(cpu_load[3]);
- P(cpu_load[4]);
+ P(lrq.fair_clock);
+ P(lrq.prev_fair_clock);
+ P(lrq.exec_clock);
```

```
+ P(lrq.prev_exec_clock);
+ P(lrq.wait_runtime);
+ P(lrq.wait_runtime_overruns);
+ P(lrq.wait_runtime_underruns);
+ P(lrq.cpu_load[0]);
+ P(lrq.cpu_load[1]);
+ P(lrq.cpu_load[2]);
+ P(lrq.cpu_load[3]);
+ P(lrq.cpu_load[4]);
 #undef P
  print_rq_runtime_sum(m, rq);

@@ -190,21 +191,21 @@
 #define P(F) \
  SEQ_printf(m, "%-25s:%20Ld\n", #F, (long long)p->F)

- P(wait_start);
- P(wait_start_fair);
- P(exec_start);
- P(sleep_start);
- P(sleep_start_fair);
- P(block_start);
- P(sleep_max);
- P(block_max);
- P(exec_max);
- P(wait_max);
- P(last_ran);
- P(wait_runtime);
- P(wait_runtime_overruns);
- P(wait_runtime_underruns);
- P(sum_exec_runtime);
+ P(se.wait_start);
+ P(se.wait_start_fair);
+ P(se.exec_start);
+ P(se.sleep_start);
+ P(se.sleep_start_fair);
+ P(se.block_start);
+ P(se.sleep_max);
+ P(se.block_max);
+ P(se.exec_max);
+ P(se.wait_max);
+ P(se.last_ran);
+ P(se.wait_runtime);
+ P(se.wait_runtime_overruns);
+ P(se.wait_runtime_underruns);
+ P(se.sum_exec_runtime);
 #undef P
```

```
 {
@@ -218,7 +219,7 @@

 void proc_sched_set_task(struct task_struct *p)
 {
- p->sleep_max = p->block_max = p->exec_max = p->wait_max = 0;
- p->wait_runtime_overruns = p->wait_runtime_underruns = 0;
- p->sum_exec_runtime = 0;
+ p->se.sleep_max = p->se.block_max = p->se.exec_max = p->se.wait_max = 0;
+ p->se.wait_runtime_overruns = p->se.wait_runtime_underruns = 0;
+ p->se.sum_exec_runtime = 0;
 }
Index: linux-2.6.21-rc7/kernel/sched_fair.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/sched_fair.c 2007-05-23 20:46:40.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched_fair.c 2007-05-23 20:48:34.000000000 +0530
@@ -55,10 +55,10 @@
  */
 static inline void __enqueue_task_fair(struct rq *rq, struct task_struct *p)
 {
- struct rb_node **link = &rq->tasks_timeline.rb_node;
+ struct rb_node **link = &rq->lrq.tasks_timeline.rb_node;
  struct rb_node *parent = NULL;
  struct task_struct *entry;
- s64 key = p->fair_key;
+ s64 key = p->se.fair_key;
  int leftmost = 1;

  /*
@@ -66,12 +66,12 @@
   */
  while (*link) {
   parent = *link;
-  entry = rb_entry(parent, struct task_struct, run_node);
+  entry = rb_entry(parent, struct task_struct, se.run_node);
   /*
    * We dont care about collisions. Nodes with
    * the same key stay together.
    */
-  if ((s64)(key - entry->fair_key) < 0) {
+  if ((s64)(key - entry->se.fair_key) < 0) {
    link = &parent->rb_left;
   } else {
    link = &parent->rb_right;
@@ -84,31 +84,31 @@
   * used):
   */
  if (leftmost)
```

```
-  rq->rb_leftmost = &p->run_node;
+  rq->lrq.rb_leftmost = &p->se.run_node;

- rb_link_node(&p->run_node, parent, link);
- rb_insert_color(&p->run_node, &rq->tasks_timeline);
+ rb_link_node(&p->se.run_node, parent, link);
+ rb_insert_color(&p->se.run_node, &rq->lrq.tasks_timeline);
 }

 static inline void __dequeue_task_fair(struct rq *rq, struct task_struct *p)
 {
- if (rq->rb_leftmost == &p->run_node)
-  rq->rb_leftmost = NULL;
- rb_erase(&p->run_node, &rq->tasks_timeline);
+ if (rq->lrq.rb_leftmost == &p->se.run_node)
+  rq->lrq.rb_leftmost = NULL;
+ rb_erase(&p->se.run_node, &rq->lrq.tasks_timeline);
 }

 static inline struct rb_node * first_fair(struct rq *rq)
 {
- if (rq->rb_leftmost)
-  return rq->rb_leftmost;
+ if (rq->lrq.rb_leftmost)
+  return rq->lrq.rb_leftmost;
   /* Cache the value returned by rb_first() */
- rq->rb_leftmost = rb_first(&rq->tasks_timeline);
- return rq->rb_leftmost;
+ rq->lrq.rb_leftmost = rb_first(&rq->lrq.tasks_timeline);
+ return rq->lrq.rb_leftmost;
 }

 static struct task_struct * __pick_next_task_fair(struct rq *rq)
 {
- return rb_entry(first_fair(rq), struct task_struct, run_node);
+ return rb_entry(first_fair(rq), struct task_struct, se.run_node);
 }

 /**************************************************************/
@@ -125,24 +125,24 @@
  /*
   * Negative nice levels get the same granularity as nice-0:
   */
- if (curr->load_weight >= NICE_0_LOAD)
+ if (curr->se.load_weight >= NICE_0_LOAD)
   return granularity;
  /*
   * Positive nice level tasks get linearly finer
```

```
   * granularity:
   */
- return curr->load_weight * (s64)(granularity / NICE_0_LOAD);
+ return curr->se.load_weight * (s64)(granularity / NICE_0_LOAD);
 }

 unsigned long get_rq_load(struct rq *rq)
 {
- unsigned long load = rq->cpu_load[CPU_LOAD_IDX_MAX-1] + 1;
+ unsigned long load = rq->lrq.cpu_load[CPU_LOAD_IDX_MAX-1] + 1;

  if (!(sysctl_sched_load_smoothing & 1))
-  return rq->raw_weighted_load;
+  return rq->lrq.raw_weighted_load;

  if (sysctl_sched_load_smoothing & 4)
-  load = max(load, rq->raw_weighted_load);
+  load = max(load, rq->lrq.raw_weighted_load);

  return load;
 }
@@ -156,31 +156,31 @@
   * Niced tasks have the same history dynamic range as
   * non-niced tasks, but their limits are offset.
   */
- if (p->wait_runtime > nice_limit) {
-  p->wait_runtime = nice_limit;
-  p->wait_runtime_overruns++;
-  rq->wait_runtime_overruns++;
+ if (p->se.wait_runtime > nice_limit) {
+  p->se.wait_runtime = nice_limit;
+  p->se.wait_runtime_overruns++;
+  rq->lrq.wait_runtime_overruns++;
  }
  limit = (limit << 1) - nice_limit;
- if (p->wait_runtime < -limit) {
-  p->wait_runtime = -limit;
-  p->wait_runtime_underruns++;
-  rq->wait_runtime_underruns++;
+ if (p->se.wait_runtime < -limit) {
+  p->se.wait_runtime = -limit;
+  p->se.wait_runtime_underruns++;
+  rq->lrq.wait_runtime_underruns++;
  }
 }

 static void __add_wait_runtime(struct rq *rq, struct task_struct *p, s64 delta)
 {
```

```
- p->wait_runtime += delta;
- p->sum_wait_runtime += delta;
+ p->se.wait_runtime += delta;
+ p->se.sum_wait_runtime += delta;
  limit_wait_runtime(rq, p);
 }

 static void add_wait_runtime(struct rq *rq, struct task_struct *p, s64 delta)
 {
- rq->wait_runtime -= p->wait_runtime;
+ rq->lrq.wait_runtime -= p->se.wait_runtime;
  __add_wait_runtime(rq, p, delta);
- rq->wait_runtime += p->wait_runtime;
+ rq->lrq.wait_runtime += p->se.wait_runtime;
 }

 /*
@@ -193,15 +193,15 @@
  struct task_struct *curr = rq->curr;

  if (curr->sched_class != &fair_sched_class || curr == rq->idle
-   || !curr->on_rq)
+   || !curr->se.on_rq)
   return;
  /*
   * Get the amount of time the current task was running
   * since the last time we changed raw_weighted_load:
   */
- delta_exec = now - curr->exec_start;
- if (unlikely(delta_exec > curr->exec_max))
-  curr->exec_max = delta_exec;
+ delta_exec = now - curr->se.exec_start;
+ if (unlikely(delta_exec > curr->se.exec_max))
+  curr->se.exec_max = delta_exec;

  if (sysctl_sched_load_smoothing & 1) {
   unsigned long load = get_rq_load(rq);
@@ -211,24 +211,24 @@
    do_div(delta_fair, load);
   } else {
    delta_fair = delta_exec * NICE_0_LOAD;
-   do_div(delta_fair, rq->raw_weighted_load);
+   do_div(delta_fair, rq->lrq.raw_weighted_load);
   }

- delta_mine = delta_exec * curr->load_weight;
+ delta_mine = delta_exec * curr->se.load_weight;
  do_div(delta_mine, load);
```

```
  } else {
   delta_fair = delta_exec * NICE_0_LOAD;
-  delta_fair += rq->raw_weighted_load >> 1;
-  do_div(delta_fair, rq->raw_weighted_load);
+  delta_fair += rq->lrq.raw_weighted_load >> 1;
+  do_div(delta_fair, rq->lrq.raw_weighted_load);

-  delta_mine = delta_exec * curr->load_weight;
-  delta_mine += rq->raw_weighted_load >> 1;
-  do_div(delta_mine, rq->raw_weighted_load);
+  delta_mine = delta_exec * curr->se.load_weight;
+  delta_mine += rq->lrq.raw_weighted_load >> 1;
+  do_div(delta_mine, rq->lrq.raw_weighted_load);
  }

- curr->sum_exec_runtime += delta_exec;
- curr->exec_start = now;
- rq->exec_clock += delta_exec;
+ curr->se.sum_exec_runtime += delta_exec;
+ curr->se.exec_start = now;
+ rq->lrq.exec_clock += delta_exec;

  /*
   * Task already marked for preemption, do not burden
@@ -237,7 +237,7 @@
  if (unlikely(test_tsk_thread_flag(curr, TIF_NEED_RESCHED)))
   goto out_nowait;

- rq->fair_clock += delta_fair;
+ rq->lrq.fair_clock += delta_fair;
  /*
   * We executed delta_exec amount of time on the CPU,
   * but we were only entitled to delta_mine amount of
@@ -253,8 +253,8 @@
 static inline void
 update_stats_wait_start(struct rq *rq, struct task_struct *p, u64 now)
 {
- p->wait_start_fair = rq->fair_clock;
- p->wait_start = now;
+ p->se.wait_start_fair = rq->lrq.fair_clock;
+ p->se.wait_start = now;
 }

 /*
@@ -274,29 +274,29 @@
  /*
   * Update the key:
   */
```

```
- key = rq->fair_clock;
+ key = rq->lrq.fair_clock;

  /*
   * Optimize the common nice 0 case:
   */
- if (likely(p->load_weight == NICE_0_LOAD)) {
-  key -= p->wait_runtime;
+ if (likely(p->se.load_weight == NICE_0_LOAD)) {
+  key -= p->se.wait_runtime;
  } else {
-  int negative = p->wait_runtime < 0;
+  int negative = p->se.wait_runtime < 0;
   u64 tmp;

-  if (p->load_weight > NICE_0_LOAD) {
+  if (p->se.load_weight > NICE_0_LOAD) {
    /* negative-reniced tasks get helped: */

    if (negative) {
-    tmp = -p->wait_runtime;
+    tmp = -p->se.wait_runtime;
     tmp *= NICE_0_LOAD;
-    do_div(tmp, p->load_weight);
+    do_div(tmp, p->se.load_weight);

     key += tmp;
    } else {
-    tmp = p->wait_runtime;
-    tmp *= p->load_weight;
+    tmp = p->se.wait_runtime;
+    tmp *= p->se.load_weight;
     do_div(tmp, NICE_0_LOAD);

     key -= tmp;
@@ -305,16 +305,16 @@
    /* plus-reniced tasks get hurt: */

    if (negative) {
-    tmp = -p->wait_runtime;
+    tmp = -p->se.wait_runtime;

     tmp *= NICE_0_LOAD;
-    do_div(tmp, p->load_weight);
+    do_div(tmp, p->se.load_weight);

     key += tmp;
    } else {
```

```
-    tmp = p->wait_runtime;
+     tmp = p->se.wait_runtime;

-    tmp *= p->load_weight;
+     tmp *= p->se.load_weight;
     do_div(tmp, NICE_0_LOAD);

     key -= tmp;
@@ -322,7 +322,7 @@
   }
  }

- p->fair_key = key;
+ p->se.fair_key = key;
 }

 /*
@@ -333,20 +333,20 @@
 {
  s64 delta_fair, delta_wait;

- delta_wait = now - p->wait_start;
- if (unlikely(delta_wait > p->wait_max))
-  p->wait_max = delta_wait;
-
- if (p->wait_start_fair) {
-  delta_fair = rq->fair_clock - p->wait_start_fair;
-  if (unlikely(p->load_weight != NICE_0_LOAD))
-   delta_fair = (delta_fair * p->load_weight) /
+ delta_wait = now - p->se.wait_start;
+ if (unlikely(delta_wait > p->se.wait_max))
+  p->se.wait_max = delta_wait;
+
+ if (p->se.wait_start_fair) {
+  delta_fair = rq->lrq.fair_clock - p->se.wait_start_fair;
+  if (unlikely(p->se.load_weight != NICE_0_LOAD))
+   delta_fair = (delta_fair * p->se.load_weight) /
         NICE_0_LOAD;
   add_wait_runtime(rq, p, delta_fair);
  }

- p->wait_start_fair = 0;
- p->wait_start = 0;
+ p->se.wait_start_fair = 0;
+ p->se.wait_start = 0;
 }

 static inline void
```

```
@@ -370,7 +370,7 @@
  /*
   * We are starting a new run period:
   */
- p->exec_start = now;
+ p->se.exec_start = now;
 }

 /*
@@ -381,7 +381,7 @@
 {
  update_curr(rq, now);

- p->exec_start = 0;
+ p->se.exec_start = 0;
 }

 /*********************************************************/
@@ -396,7 +396,7 @@
  if (!(sysctl_sched_load_smoothing & 16))
   goto out;

- delta_fair = rq->fair_clock - p->sleep_start_fair;
+ delta_fair = rq->lrq.fair_clock - p->se.sleep_start_fair;
  if ((s64)delta_fair < 0)
   delta_fair = 0;

@@ -406,15 +406,15 @@
   */
  if (sysctl_sched_load_smoothing & 8) {
   delta_fair = delta_fair * load;
-  do_div(delta_fair, load + p->load_weight);
+  do_div(delta_fair, load + p->se.load_weight);
  }

  __add_wait_runtime(rq, p, delta_fair);

 out:
- rq->wait_runtime += p->wait_runtime;
+ rq->lrq.wait_runtime += p->se.wait_runtime;

- p->sleep_start_fair = 0;
+ p->se.sleep_start_fair = 0;
 }

 /*
@@ -433,29 +433,29 @@
  update_curr(rq, now);
```

```
  if (wakeup) {
-  if (p->sleep_start) {
-   delta = now - p->sleep_start;
+  if (p->se.sleep_start) {
+   delta = now - p->se.sleep_start;
    if ((s64)delta < 0)
     delta = 0;

-   if (unlikely(delta > p->sleep_max))
-    p->sleep_max = delta;
+   if (unlikely(delta > p->se.sleep_max))
+    p->se.sleep_max = delta;

-   p->sleep_start = 0;
+   p->se.sleep_start = 0;
   }
-  if (p->block_start) {
-   delta = now - p->block_start;
+  if (p->se.block_start) {
+   delta = now - p->se.block_start;
    if ((s64)delta < 0)
     delta = 0;

-   if (unlikely(delta > p->block_max))
-    p->block_max = delta;
+   if (unlikely(delta > p->se.block_max))
+    p->se.block_max = delta;

-   p->block_start = 0;
+   p->se.block_start = 0;
   }
-  p->sum_sleep_runtime += delta;
+  p->se.sum_sleep_runtime += delta;

-  if (p->sleep_start_fair)
+  if (p->se.sleep_start_fair)
    enqueue_sleeper(rq, p);
  }
  update_stats_enqueue(rq, p, now);
@@ -473,11 +473,11 @@
  update_stats_dequeue(rq, p, now);
  if (sleep) {
   if (p->state & TASK_INTERRUPTIBLE)
-   p->sleep_start = now;
+   p->se.sleep_start = now;
   if (p->state & TASK_UNINTERRUPTIBLE)
-   p->block_start = now;
```

```
-  p->sleep_start_fair = rq->fair_clock;
-  rq->wait_runtime -= p->wait_runtime;
+   p->se.block_start = now;
+  p->se.sleep_start_fair = rq->lrq.fair_clock;
+  rq->lrq.wait_runtime -= p->se.wait_runtime;
  }
  __dequeue_task_fair(rq, p);
 }
@@ -509,9 +509,9 @@
    * position within the tree:
    */
   dequeue_task_fair(rq, p, 0, now);
-  p->on_rq = 0;
+  p->se.on_rq = 0;
   enqueue_task_fair(rq, p, 0, now);
-  p->on_rq = 1;
+  p->se.on_rq = 1;

   /*
    * Reschedule if another task tops the current one.
@@ -526,11 +526,11 @@
   * yield-to support: if we are on the same runqueue then
   * give half of our wait_runtime (if it's positive) to the other task:
   */
- if (p_to && p->wait_runtime > 0) {
-  p_to->wait_runtime += p->wait_runtime >> 1;
-  p->wait_runtime >>= 1;
+ if (p_to && p->se.wait_runtime > 0) {
+  p_to->se.wait_runtime += p->se.wait_runtime >> 1;
+  p->se.wait_runtime >>= 1;
  }
- curr = &p->run_node;
+ curr = &p->se.run_node;
  first = first_fair(rq);
  /*
   * Move this task to the second place in the tree:
@@ -547,25 +547,25 @@
    return;
  }

- p_next = rb_entry(next, struct task_struct, run_node);
+ p_next = rb_entry(next, struct task_struct, se.run_node);
  /*
   * Minimally necessary key value to be the second in the tree:
   */
- yield_key = p_next->fair_key + 1;
+ yield_key = p_next->se.fair_key + 1;
```

```
  now = __rq_clock(rq);
  dequeue_task_fair(rq, p, 0, now);
- p->on_rq = 0;
+ p->se.on_rq = 0;

  /*
   * Only update the key if we need to move more backwards
   * than the minimally necessary position to be the second:
   */
- if (p->fair_key < yield_key)
-  p->fair_key = yield_key;
+ if (p->se.fair_key < yield_key)
+  p->se.fair_key = yield_key;

  __enqueue_task_fair(rq, p);
- p->on_rq = 1;
+ p->se.on_rq = 1;
 }

 /*
@@ -575,7 +575,7 @@
 __check_preempt_curr_fair(struct rq *rq, struct task_struct *p,
      struct task_struct *curr, unsigned long granularity)
 {
- s64 __delta = curr->fair_key - p->fair_key;
+ s64 __delta = curr->se.fair_key - p->se.fair_key;

  /*
   * Take scheduling granularity into account - do not
@@ -631,7 +631,7 @@
   * If the task is still waiting for the CPU (it just got
   * preempted), start the wait period:
   */
- if (prev->on_rq)
+ if (prev->se.on_rq)
   update_stats_wait_start(rq, prev, now);
 }

@@ -654,23 +654,23 @@
  if (!first)
   return NULL;

- p = rb_entry(first, struct task_struct, run_node);
+ p = rb_entry(first, struct task_struct, se.run_node);

- rq->rb_load_balance_curr = rb_next(first);
+ rq->lrq.rb_load_balance_curr = rb_next(first);
```

```
  return p;
 }

 static struct task_struct * load_balance_next_fair(struct rq *rq)
 {
- struct rb_node *curr = rq->rb_load_balance_curr;
+ struct rb_node *curr = rq->lrq.rb_load_balance_curr;
  struct task_struct *p;

  if (!curr)
   return NULL;

- p = rb_entry(curr, struct task_struct, run_node);
- rq->rb_load_balance_curr = rb_next(curr);
+ p = rb_entry(curr, struct task_struct, se.run_node);
+ rq->lrq.rb_load_balance_curr = rb_next(curr);

  return p;
 }
@@ -688,9 +688,9 @@
   * position within the tree:
   */
  dequeue_task_fair(rq, curr, 0, now);
- curr->on_rq = 0;
+ curr->se.on_rq = 0;
  enqueue_task_fair(rq, curr, 0, now);
- curr->on_rq = 1;
+ curr->se.on_rq = 1;

  /*
   * Reschedule if another task tops the current one.
@@ -723,16 +723,16 @@
   * until it reschedules once. We set up the key so that
   * it will preempt the parent:
   */
- p->fair_key = current->fair_key - niced_granularity(rq->curr,
+ p->se.fair_key = current->se.fair_key - niced_granularity(rq->curr,
       sysctl_sched_granularity) - 1;
  /*
   * The first wait is dominated by the child-runs-first logic,
   * so do not credit it with that waiting time yet:
   */
- p->wait_start_fair = 0;
+ p->se.wait_start_fair = 0;

  __enqueue_task_fair(rq, p);
- p->on_rq = 1;
+ p->se.on_rq = 1;
```

inc_nr_running(p, rq);
}

--
Regards,
vatsa

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

Subject: [RFC] [PATCH 3/3] Generalize CFS core and provide per-user fairness
Posted by Srivatsa Vaddagiri on Wed, 23 May 2007 16:56:17 GMT
View Forum Message <> Reply to Message

This patch reuses CFS core to provide inter task-group fairness. For
demonstration purpose, the patch also extends CFS to provide per-user
fairness. The patch is very experimental atm and in particular, SMP LOAD
BALANCE IS DISABLED to keep the patch simple. I think group-based smp
load
balance is more trickier and I intend to look at it next.

Although user id is chosen as the basis of grouping tasks, the patch can
be adapted to work with other task grouping mechanisms (like :
http://lkml.org/lkml/2007/4/27/146).

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

```
---
 arch/i386/Kconfig     |    6
 arch/x86_64/Kconfig   |    6
 include/linux/sched.h |   16
 kernel/sched.c        |  256 +++++++++++----
 kernel/sched_debug.c  |    4
 kernel/sched_fair.c   |  820 ++++++++++++++++++++++++++++++++-----------------
 kernel/sched_rt.c     |    2
 kernel/user.c         |    5
 8 files changed, 753 insertions(+), 362 deletions(-)

Index: linux-2.6.21-rc7/arch/i386/Kconfig
===================================================================
--- linux-2.6.21-rc7.orig/arch/i386/Kconfig 2007-05-23 20:46:38.000000000 +0530
+++ linux-2.6.21-rc7/arch/i386/Kconfig 2007-05-23 20:48:39.000000000 +0530
@@ -307,6 +307,12 @@
   making when dealing with multi-core CPU chips at a cost of slightly
```

```
    increased overhead in some places. If unsure say N here.

+config FAIR_USER_SCHED
+ bool "Fair user scheduler"
+ default n
+ help
+  Fair user scheduler
+
 source "kernel/Kconfig.preempt"

 config X86_UP_APIC
Index: linux-2.6.21-rc7/arch/x86_64/Kconfig
===================================================================
--- linux-2.6.21-rc7.orig/arch/x86_64/Kconfig 2007-05-23 20:46:38.000000000 +0530
+++ linux-2.6.21-rc7/arch/x86_64/Kconfig 2007-05-23 20:48:39.000000000 +0530
@@ -330,6 +330,12 @@
    making when dealing with multi-core CPU chips at a cost of slightly
    increased overhead in some places. If unsure say N here.

+config FAIR_USER_SCHED
+ bool "Fair user scheduler"
+ default n
+ help
+  Fair user scheduler
+
 source "kernel/Kconfig.preempt"

 config NUMA
Index: linux-2.6.21-rc7/include/linux/sched.h
===================================================================
--- linux-2.6.21-rc7.orig/include/linux/sched.h 2007-05-23 20:48:34.000000000 +0530
+++ linux-2.6.21-rc7/include/linux/sched.h 2007-05-23 20:48:39.000000000 +0530
@@ -551,6 +551,16 @@
 #define is_rt_policy(p)  ((p) != SCHED_NORMAL && (p) != SCHED_BATCH)
 #define has_rt_policy(p) unlikely(is_rt_policy((p)->policy))

+#ifdef CONFIG_FAIR_USER_SCHED
+int sched_alloc_user(struct user_struct *user);
+void sched_free_user(struct user_struct *user);
+void sched_move_task(struct user_struct *old);
+#else
+static inline int sched_alloc_user(struct user_struct *user) { return 0; }
+static inline void sched_free_user(struct user_struct *user) { }
+static inline void sched_move_task(struct user_struct *user) { }
+#endif
+
 /*
  * Some day this will be a full-fledged user tracking system..
```

```
  */
@@ -575,6 +585,10 @@
  /* Hash table maintenance information */
  struct list_head uidhash_list;
  uid_t uid;
+#ifdef CONFIG_FAIR_USER_SCHED
+ struct sched_entity *se;        /* per-cpu sched_entity */
+ struct lrq *lrq;        /* per-cpu runqueue for this user */
+#endif
 };

 extern struct user_struct *find_user(uid_t);
@@ -859,6 +873,8 @@
  s64 fair_key;
  s64 sum_wait_runtime, sum_sleep_runtime;
  unsigned long wait_runtime_overruns, wait_runtime_underruns;
+ struct sched_entity *parent;
+ struct lrq *my_q;       /* The queue owned by this entity */
 };

 struct task_struct {
Index: linux-2.6.21-rc7/kernel/sched.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/sched.c 2007-05-23 20:48:34.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched.c 2007-05-23 20:48:39.000000000 +0530
@@ -129,6 +129,14 @@
  struct rb_root tasks_timeline;
  struct rb_node *rb_leftmost;
  struct rb_node *rb_load_balance_curr;
+ struct sched_entity *curr;
+ unsigned int *sched_granularity; /* &sysctl_sched_granularity */
+ struct rq *rq;
+ unsigned long nice_0_load;
+#ifdef CONFIG_FAIR_USER_SCHED
+ struct list_head lrq_list;
+ struct rcu_head rcu;
+#endif
 };

 /*
@@ -164,6 +172,7 @@

  struct task_struct *curr, *idle;
  unsigned long next_balance;
+ unsigned long rt_load;
  struct mm_struct *prev_mm;

  u64 clock, prev_clock_raw;
```

```
@@ -214,6 +223,32 @@
  struct lock_class_key rq_lock_key;
 };

+#define NICE_0_LOAD SCHED_LOAD_SCALE
+#define NICE_0_SHIFT SCHED_LOAD_SHIFT
+
+#ifdef CONFIG_FAIR_USER_SCHED
+static struct sched_entity root_user_se[NR_CPUS];
+static struct lrq root_user_lrq[NR_CPUS];
+
+static inline void init_se(struct sched_entity *se, struct lrq *lrq)
+{
+ se->my_q = lrq;
+ se->load_weight = NICE_0_LOAD;
+}
+
+static inline void init_lrq(struct lrq *lrq, struct rq *rq)
+{
+ lrq->rq = rq;
+ lrq->fair_clock = 1;
+ lrq->tasks_timeline = RB_ROOT;
+ lrq->nice_0_load = NICE_0_LOAD;
+ lrq->sched_granularity = &sysctl_sched_granularity;
+ INIT_LIST_HEAD(&lrq->lrq_list);
+ list_add_rcu(&lrq->lrq_list, &rq->lrq.lrq_list);
+}
+
+#endif
+
 static DEFINE_PER_CPU(struct rq, runqueues) ____cacheline_aligned_in_smp;
 static DEFINE_MUTEX(sched_hotcpu_mutex);

@@ -555,9 +590,6 @@
 #define RTPRIO_TO_LOAD_WEIGHT(rp) \
  (PRIO_TO_LOAD_WEIGHT(MAX_RT_PRIO) + LOAD_WEIGHT(rp))

-#define NICE_0_LOAD SCHED_LOAD_SCALE
-#define NICE_0_SHIFT SCHED_LOAD_SHIFT
-
 /*
  * Nice levels are multiplicative, with a gentle 10% change for every
  * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
@@ -576,16 +608,22 @@
 /*  10 */   110,    87,    70,    56,    45,    36,    29,    23,    18,    15,
 };

+extern struct sched_class rt_sched_class;
```

```
+
 static inline void
 inc_raw_weighted_load(struct rq *rq, const struct task_struct *p)
 {
- rq->lrq.raw_weighted_load += p->se.load_weight;
+ /* Hack - needs better handling */
+ if (p->sched_class == &rt_sched_class)
+  rq->rt_load += p->se.load_weight;
 }

 static inline void
 dec_raw_weighted_load(struct rq *rq, const struct task_struct *p)
 {
- rq->lrq.raw_weighted_load -= p->se.load_weight;
+ /* Hack - needs better handling */
+ if (p->sched_class == &rt_sched_class)
+  rq->rt_load -= p->se.load_weight;
 }

 static inline void inc_nr_running(struct task_struct *p, struct rq *rq)
@@ -728,10 +766,32 @@
  return cpu_curr(task_cpu(p)) == p;
 }

+#ifdef CONFIG_FAIR_USER_SCHED
+
+#define for_each_lrq(rq, lrq) \
+ for (lrq = container_of((rq)->lrq.lrq_list.next, struct lrq, lrq_list);\
+      prefetch(rcu_dereference(lrq->lrq_list.next)), lrq != &(rq)->lrq;\
+      lrq = container_of(lrq->lrq_list.next, struct lrq, lrq_list))
+
+#else
+
+#define for_each_lrq(rq, lrq) \
+  for (lrq = &rq->lrq; lrq != NULL; lrq = NULL)
+
+#endif
+
 /* Used instead of source_load when we know the type == 0 */
 unsigned long weighted_cpuload(const int cpu)
 {
- return cpu_rq(cpu)->lrq.raw_weighted_load;
+ struct lrq *lrq;
+ unsigned long weight = 0;
+
+ for_each_lrq(cpu_rq(cpu), lrq)
+  weight += lrq->raw_weighted_load;
+
```

```
+ weight += cpu_rq(cpu)->rt_load;
+
+ return weight;
 }

 #ifdef CONFIG_SMP
@@ -761,6 +821,10 @@
  if (p->se.sleep_start_fair)
   p->se.sleep_start_fair -= fair_clock_offset;

+#ifdef CONFIG_FAIR_USER_SCHED
+ p->se.parent = &p->user->se[new_cpu];
+#endif
+
  task_thread_info(p)->cpu = new_cpu;

 }
@@ -863,12 +927,18 @@
  */
 static inline unsigned long source_load(int cpu, int type)
 {
- struct rq *rq = cpu_rq(cpu);
+ unsigned long rwl, cpl = 0;
+ struct lrq *lrq;
+
+ rwl = weighted_cpuload(cpu);

  if (type == 0)
-  return rq->lrq.raw_weighted_load;
+  return rwl;
+
+ for_each_lrq(cpu_rq(cpu), lrq)
+  cpl += lrq->cpu_load[type-1];

- return min(rq->lrq.cpu_load[type-1], rq->lrq.raw_weighted_load);
+ return min(cpl, rwl);
 }

 /*
@@ -877,12 +947,18 @@
  */
 static inline unsigned long target_load(int cpu, int type)
 {
- struct rq *rq = cpu_rq(cpu);
+ unsigned long rwl, cpl = 0;
+ struct lrq *lrq;
+
+ rwl = weighted_cpuload(cpu);
```

```
  if (type == 0)
-  return rq->lrq.raw_weighted_load;
+  return rwl;
+
+ for_each_lrq(cpu_rq(cpu), lrq)
+  cpl += lrq->cpu_load[type-1];

- return max(rq->lrq.cpu_load[type-1], rq->lrq.raw_weighted_load);
+ return max(cpl, rwl);
 }

 /*
@@ -893,7 +969,7 @@
  struct rq *rq = cpu_rq(cpu);
  unsigned long n = rq->nr_running;

- return n ? rq->lrq.raw_weighted_load / n : SCHED_LOAD_SCALE;
+ return n ? weighted_cpuload(cpu) / n : SCHED_LOAD_SCALE;
 }

 /*
@@ -1583,59 +1659,6 @@
  return running + uninterruptible;
 }

-static void update_load_fair(struct rq *this_rq)
-{
- unsigned long this_load, fair_delta, exec_delta, idle_delta;
- unsigned int i, scale;
- s64 fair_delta64, exec_delta64;
- unsigned long tmp;
- u64 tmp64;
-
- this_rq->lrq.nr_load_updates++;
- if (!(sysctl_sched_load_smoothing & 64)) {
-  this_load = this_rq->lrq.raw_weighted_load;
-  goto do_avg;
- }
-
- fair_delta64 = this_rq->lrq.fair_clock -
-    this_rq->lrq.prev_fair_clock + 1;
- this_rq->lrq.prev_fair_clock = this_rq->lrq.fair_clock;
-
- exec_delta64 = this_rq->lrq.exec_clock -
-    this_rq->lrq.prev_exec_clock + 1;
- this_rq->lrq.prev_exec_clock = this_rq->lrq.exec_clock;
-
```

```
- if (fair_delta64 > (s64)LONG_MAX)
-  fair_delta64 = (s64)LONG_MAX;
- fair_delta = (unsigned long)fair_delta64;
-
- if (exec_delta64 > (s64)LONG_MAX)
-  exec_delta64 = (s64)LONG_MAX;
- exec_delta = (unsigned long)exec_delta64;
- if (exec_delta > TICK_NSEC)
-  exec_delta = TICK_NSEC;
-
- idle_delta = TICK_NSEC - exec_delta;
-
- tmp = (SCHED_LOAD_SCALE * exec_delta) / fair_delta;
- tmp64 = (u64)tmp * (u64)exec_delta;
- do_div(tmp64, TICK_NSEC);
- this_load = (unsigned long)tmp64;
-
-do_avg:
- /* Update our load: */
- for (i = 0, scale = 1; i < CPU_LOAD_IDX_MAX; i++, scale += scale) {
-  unsigned long old_load, new_load;
-
-  /* scale is effectively 1 << i now, and >> i divides by scale */
-
-  old_load = this_rq->lrq.cpu_load[i];
-  new_load = this_load;
-
-  this_rq->lrq.cpu_load[i] = (old_load*(scale-1) + new_load) >> i;
- }
-}
-
 #ifdef CONFIG_SMP

 /*
@@ -1999,7 +2022,7 @@

    avg_load += load;
    sum_nr_running += rq->nr_running;
-   sum_weighted_load += rq->lrq.raw_weighted_load;
+   sum_weighted_load += weighted_cpuload(i);
   }

   /*
@@ -2227,18 +2250,19 @@
  int i;

  for_each_cpu_mask(i, group->cpumask) {
+  unsigned long rwl;
```

```
   if (!cpu_isset(i, *cpus))
    continue;

   rq = cpu_rq(i);
+  rwl = weighted_cpuload(i);

-  if (rq->nr_running == 1 &&
-    rq->lrq.raw_weighted_load > imbalance)
+  if (rq->nr_running == 1 && rwl > imbalance)
    continue;

-  if (rq->lrq.raw_weighted_load > max_load) {
-   max_load = rq->lrq.raw_weighted_load;
+  if (rwl > max_load) {
+   max_load = rwl;
    busiest = rq;
   }
  }
@@ -5988,6 +6012,12 @@
   */
  rt_sched_class.next = &fair_sched_class;
  fair_sched_class.next = NULL;
+#ifdef CONFIG_FAIR_USER_SCHED
+ root_user.se = root_user_se; /* per-cpu schedulable entities */
+ root_user.lrq = root_user_lrq; /* per-cpu runqueue */
+ root_user_lrq[0].curr = &current->se; /* todo: remove this */
+ cpu_rq(0)->lrq.curr = current->se.parent = &root_user.se[0];
+#endif

  for_each_possible_cpu(i) {
   struct prio_array *array;
@@ -5999,6 +6029,9 @@
   rq->nr_running = 0;
   rq->lrq.tasks_timeline = RB_ROOT;
   rq->clock = rq->lrq.fair_clock = 1;
+  rq->lrq.nice_0_load = NICE_0_LOAD;
+  rq->lrq.sched_granularity = &sysctl_sched_granularity;
+  rq->lrq.rq  = rq;

   for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
    rq->lrq.cpu_load[j] = 0;
@@ -6020,6 +6053,16 @@
   highest_cpu = i;
   /* delimiter for bitsearch: */
   __set_bit(MAX_RT_PRIO, array->bitmap);
+#ifdef CONFIG_FAIR_USER_SCHED
+  INIT_LIST_HEAD(&rq->lrq.lrq_list);
```

```
+  {
+   struct lrq *lrq = &current->user->lrq[i];
+   struct sched_entity *se = &current->user->se[i];
+
+   init_se(se, lrq);
+   init_lrq(lrq, rq);
+  }
+#endif
  }

  set_load_weight(&init_task);
@@ -6176,3 +6219,74 @@
 }

 #endif
+
+#ifdef CONFIG_FAIR_USER_SCHED
+
+int sched_alloc_user(struct user_struct *new)
+{
+ int i = num_possible_cpus();
+
+ new->se = kzalloc(sizeof(struct sched_entity) * i, GFP_KERNEL);
+ if (!new->se)
+  return -ENOMEM;
+
+ new->lrq = kzalloc(sizeof(struct lrq) * i, GFP_KERNEL);
+ if (!new->lrq) {
+  kfree(new->se);
+  return -ENOMEM;
+ }
+
+ for_each_possible_cpu(i) {
+  struct lrq *lrq = &new->lrq[i];
+  struct sched_entity *se = &new->se[i];
+  struct rq *rq = cpu_rq(i);
+
+  init_se(se, lrq);
+  init_lrq(lrq, rq);
+ }
+
+ return 0;
+}
+
+static void free_lrq(struct rcu_head *rhp)
+{
+ struct lrq *lrq = container_of(rhp, struct lrq, rcu);
+
```

```
+ kfree(lrq);
+}
+
+void sched_free_user(struct user_struct *up)
+{
+ int i;
+ struct lrq *lrq;
+
+ for_each_possible_cpu(i) {
+  lrq = &up->lrq[i];
+  list_del_rcu(&lrq->lrq_list);
+ }
+
+ lrq = &up->lrq[0];
+ call_rcu(&lrq->rcu, free_lrq);
+
+ kfree(up->se);
+}
+
+void sched_move_task(struct user_struct *old)
+{
+ unsigned long flags;
+ struct user_struct *new = current->user;
+ struct rq *rq;
+
+ rq = task_rq_lock(current, &flags);
+
+ current->user = old;
+ deactivate_task(rq, current, 0);
+ current->user = new;
+ current->se.parent = &new->se[task_cpu(current)];
+ activate_task(rq, current, 0);
+
+ task_rq_unlock(rq, &flags);
+}
+
+
+#endif
Index: linux-2.6.21-rc7/kernel/sched_debug.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/sched_debug.c 2007-05-23 20:48:34.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched_debug.c 2007-05-23 20:48:39.000000000 +0530
@@ -68,7 +68,7 @@
  "------------------------------------------------"
  "--------------------------------\n");

- curr = first_fair(rq);
+ curr = first_fair(&rq->lrq);
```

```
  while (curr) {
   p = rb_entry(curr, struct task_struct, se.run_node);
   print_task(m, rq, p, now);
@@ -85,7 +85,7 @@
  unsigned long flags;

  spin_lock_irqsave(&rq->lock, flags);
- curr = first_fair(rq);
+ curr = first_fair(&rq->lrq);
  while (curr) {
   p = rb_entry(curr, struct task_struct, se.run_node);
   wait_runtime_rq_sum += p->se.wait_runtime;
Index: linux-2.6.21-rc7/kernel/sched_fair.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/sched_fair.c 2007-05-23 20:48:34.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched_fair.c 2007-05-23 20:48:39.000000000 +0530
@@ -46,19 +46,25 @@

 extern struct sched_class fair_sched_class;

+#define entity_is_task(t)      (!t->my_q)
+#define task_entity(t)         container_of(t, struct task_struct, se)
+static inline void update_curr(struct lrq *lrq, u64 now);
+
 /**************************************************************/
 /* Scheduling class tree data structure manipulation methods:
  */

+/************* Start generic schedulable entity operations ********************/
+
 /*
  * Enqueue a task into the rb-tree:
  */
-static inline void __enqueue_task_fair(struct rq *rq, struct task_struct *p)
+static inline void __enqueue_entity(struct lrq *lrq, struct sched_entity *p)
 {
- struct rb_node **link = &rq->lrq.tasks_timeline.rb_node;
+ struct rb_node **link = &lrq->tasks_timeline.rb_node;
  struct rb_node *parent = NULL;
- struct task_struct *entry;
- s64 key = p->se.fair_key;
+ struct sched_entity *entry;
+ s64 key = p->fair_key;
  int leftmost = 1;

  /*
@@ -66,12 +72,12 @@
   */
```

```
  while (*link) {
   parent = *link;
-  entry = rb_entry(parent, struct task_struct, se.run_node);
+  entry = rb_entry(parent, struct sched_entity, run_node);
   /*
    * We dont care about collisions. Nodes with
    * the same key stay together.
    */
-  if ((s64)(key - entry->se.fair_key) < 0) {
+  if ((s64)(key - entry->fair_key) < 0) {
    link = &parent->rb_left;
   } else {
    link = &parent->rb_right;
@@ -84,31 +90,35 @@
   * used):
   */
  if (leftmost)
-  rq->lrq.rb_leftmost = &p->se.run_node;
+  lrq->rb_leftmost = &p->run_node;

- rb_link_node(&p->se.run_node, parent, link);
- rb_insert_color(&p->se.run_node, &rq->lrq.tasks_timeline);
+ rb_link_node(&p->run_node, parent, link);
+ rb_insert_color(&p->run_node, &lrq->tasks_timeline);
+ lrq->raw_weighted_load += p->load_weight;
+ p->on_rq = 1;
 }

-static inline void __dequeue_task_fair(struct rq *rq, struct task_struct *p)
+static inline void __dequeue_entity(struct lrq *lrq, struct sched_entity *p)
 {
- if (rq->lrq.rb_leftmost == &p->se.run_node)
-  rq->lrq.rb_leftmost = NULL;
- rb_erase(&p->se.run_node, &rq->lrq.tasks_timeline);
+ if (lrq->rb_leftmost == &p->run_node)
+  lrq->rb_leftmost = NULL;
+ rb_erase(&p->run_node, &lrq->tasks_timeline);
+ lrq->raw_weighted_load -= p->load_weight;
+ p->on_rq = 0;
 }

-static inline struct rb_node * first_fair(struct rq *rq)
+static inline struct rb_node * first_fair(struct lrq *lrq)
 {
- if (rq->lrq.rb_leftmost)
-  return rq->lrq.rb_leftmost;
+ if (lrq->rb_leftmost)
+  return lrq->rb_leftmost;
```

```
  /* Cache the value returned by rb_first() */
- rq->lrq.rb_leftmost = rb_first(&rq->lrq.tasks_timeline);
- return rq->lrq.rb_leftmost;
+ lrq->rb_leftmost = rb_first(&lrq->tasks_timeline);
+ return lrq->rb_leftmost;
 }

-static struct task_struct * __pick_next_task_fair(struct rq *rq)
+static struct sched_entity * __pick_next_entity(struct lrq *lrq)
 {
- return rb_entry(first_fair(rq), struct task_struct, se.run_node);
+ return rb_entry(first_fair(lrq), struct sched_entity, run_node);
 }

 /**************************************************************/
@@ -119,125 +129,126 @@
  * We rescale the rescheduling granularity of tasks according to their
  * nice level, but only linearly, not exponentially:
  */
-static u64
-niced_granularity(struct task_struct *curr, unsigned long granularity)
+static u64 niced_granularity(struct lrq *lrq, struct sched_entity *curr,
+     unsigned long granularity)
 {
  /*
   * Negative nice levels get the same granularity as nice-0:
   */
- if (curr->se.load_weight >= NICE_0_LOAD)
+ if (curr->load_weight >= lrq->nice_0_load)
   return granularity;
  /*
   * Positive nice level tasks get linearly finer
   * granularity:
   */
- return curr->se.load_weight * (s64)(granularity / NICE_0_LOAD);
+ return curr->load_weight * (s64)(granularity / lrq->nice_0_load);
 }

-unsigned long get_rq_load(struct rq *rq)
+unsigned long get_lrq_load(struct lrq *lrq)
 {
- unsigned long load = rq->lrq.cpu_load[CPU_LOAD_IDX_MAX-1] + 1;
+ unsigned long load = lrq->cpu_load[CPU_LOAD_IDX_MAX-1] + 1;

  if (!(sysctl_sched_load_smoothing & 1))
-  return rq->lrq.raw_weighted_load;
+  return lrq->raw_weighted_load;
```

```
  if (sysctl_sched_load_smoothing & 4)
-  load = max(load, rq->lrq.raw_weighted_load);
+  load = max(load, lrq->raw_weighted_load);

  return load;
 }

-static void limit_wait_runtime(struct rq *rq, struct task_struct *p)
+static void limit_wait_runtime(struct lrq *lrq, struct sched_entity *p)
 {
- s64 limit = sysctl_sched_runtime_limit;
+ s64 limit = *(lrq->sched_granularity);
  s64 nice_limit = limit; // niced_granularity(p, limit);

  /*
   * Niced tasks have the same history dynamic range as
   * non-niced tasks, but their limits are offset.
   */
- if (p->se.wait_runtime > nice_limit) {
-  p->se.wait_runtime = nice_limit;
-  p->se.wait_runtime_overruns++;
-  rq->lrq.wait_runtime_overruns++;
+ if (p->wait_runtime > nice_limit) {
+  p->wait_runtime = nice_limit;
+  p->wait_runtime_overruns++;
+  lrq->wait_runtime_overruns++;
  }
  limit = (limit << 1) - nice_limit;
- if (p->se.wait_runtime < -limit) {
-  p->se.wait_runtime = -limit;
-  p->se.wait_runtime_underruns++;
-  rq->lrq.wait_runtime_underruns++;
+ if (p->wait_runtime < -limit) {
+  p->wait_runtime = -limit;
+  p->wait_runtime_underruns++;
+  lrq->wait_runtime_underruns++;
  }
 }

-static void __add_wait_runtime(struct rq *rq, struct task_struct *p, s64 delta)
+static void
+__add_wait_runtime(struct lrq *lrq, struct sched_entity *p, s64 delta)
 {
- p->se.wait_runtime += delta;
- p->se.sum_wait_runtime += delta;
- limit_wait_runtime(rq, p);
+ p->wait_runtime += delta;
+ p->sum_wait_runtime += delta;
```

```
+ limit_wait_runtime(lrq, p);
 }

-static void add_wait_runtime(struct rq *rq, struct task_struct *p, s64 delta)
+static void add_wait_runtime(struct lrq *lrq, struct sched_entity *p, s64 delta)
 {
- rq->lrq.wait_runtime -= p->se.wait_runtime;
- __add_wait_runtime(rq, p, delta);
- rq->lrq.wait_runtime += p->se.wait_runtime;
+ lrq->wait_runtime -= p->wait_runtime;
+ __add_wait_runtime(lrq, p, delta);
+ lrq->wait_runtime += p->wait_runtime;
 }

 /*
  * Update the current task's runtime statistics. Skip current tasks that
  * are not in our scheduling class.
  */
-static inline void update_curr(struct rq *rq, u64 now)
+static inline void _update_curr(struct lrq *lrq, u64 now)
 {
  u64 delta_exec, delta_fair, delta_mine;
- struct task_struct *curr = rq->curr;
+ struct sched_entity *curr = lrq->curr;
+ struct task_struct *curtask = lrq->rq->curr;

- if (curr->sched_class != &fair_sched_class || curr == rq->idle
-   || !curr->se.on_rq)
+ if (!curr->on_rq || !curr->exec_start)
   return;
  /*
   * Get the amount of time the current task was running
   * since the last time we changed raw_weighted_load:
   */
- delta_exec = now - curr->se.exec_start;
- if (unlikely(delta_exec > curr->se.exec_max))
-  curr->se.exec_max = delta_exec;
+ delta_exec = now - curr->exec_start;
+ if (unlikely(delta_exec > curr->exec_max))
+  curr->exec_max = delta_exec;

  if (sysctl_sched_load_smoothing & 1) {
-  unsigned long load = get_rq_load(rq);
+  unsigned long load = get_lrq_load(lrq);

   if (sysctl_sched_load_smoothing & 2) {
-   delta_fair = delta_exec * NICE_0_LOAD;
+   delta_fair = delta_exec * lrq->nice_0_load;
```

```
    do_div(delta_fair, load);
   } else {
-   delta_fair = delta_exec * NICE_0_LOAD;
-   do_div(delta_fair, rq->lrq.raw_weighted_load);
+   delta_fair = delta_exec * lrq->nice_0_load;
+   do_div(delta_fair, lrq->raw_weighted_load);
   }

-  delta_mine = delta_exec * curr->se.load_weight;
+  delta_mine = delta_exec * curr->load_weight;
   do_div(delta_mine, load);
  } else {
-  delta_fair = delta_exec * NICE_0_LOAD;
-  delta_fair += rq->lrq.raw_weighted_load >> 1;
-  do_div(delta_fair, rq->lrq.raw_weighted_load);
-
-  delta_mine = delta_exec * curr->se.load_weight;
-  delta_mine += rq->lrq.raw_weighted_load >> 1;
-  do_div(delta_mine, rq->lrq.raw_weighted_load);
+  delta_fair = delta_exec * lrq->nice_0_load;
+  delta_fair += lrq->raw_weighted_load >> 1;
+  do_div(delta_fair, lrq->raw_weighted_load);
+
+  delta_mine = delta_exec * curr->load_weight;
+  delta_mine += lrq->raw_weighted_load >> 1;
+  do_div(delta_mine, lrq->raw_weighted_load);
  }

- curr->se.sum_exec_runtime += delta_exec;
- curr->se.exec_start = now;
- rq->lrq.exec_clock += delta_exec;
+ curr->sum_exec_runtime += delta_exec;
+ curr->exec_start = now;
+ lrq->exec_clock += delta_exec;

  /*
   * Task already marked for preemption, do not burden
   * it with the cost of not having left the CPU yet.
   */
- if (unlikely(test_tsk_thread_flag(curr, TIF_NEED_RESCHED)))
+ if (unlikely(test_tsk_thread_flag(curtask, TIF_NEED_RESCHED)))
   goto out_nowait;

- rq->lrq.fair_clock += delta_fair;
+ lrq->fair_clock += delta_fair;
  /*
   * We executed delta_exec amount of time on the CPU,
   * but we were only entitled to delta_mine amount of
```

```
@@ -245,23 +256,23 @@
   * the two values are equal)
   * [Note: delta_mine - delta_exec is negative]:
   */
- add_wait_runtime(rq, curr, delta_mine - delta_exec);
+ add_wait_runtime(lrq, curr, delta_mine - delta_exec);
 out_nowait:
  ;
 }

 static inline void
-update_stats_wait_start(struct rq *rq, struct task_struct *p, u64 now)
+update_stats_wait_start(struct lrq *lrq, struct sched_entity *p, u64 now)
 {
- p->se.wait_start_fair = rq->lrq.fair_clock;
- p->se.wait_start = now;
+ p->wait_start_fair = lrq->fair_clock;
+ p->wait_start = now;
 }

 /*
  * Task is being enqueued - update stats:
  */
 static inline void
-update_stats_enqueue(struct rq *rq, struct task_struct *p, u64 now)
+update_stats_enqueue(struct lrq *lrq, struct sched_entity *p, u64 now)
 {
  s64 key;

@@ -269,35 +280,35 @@
   * Are we enqueueing a waiting task? (for current tasks
   * a dequeue/enqueue event is a NOP)
   */
- if (p != rq->curr)
-  update_stats_wait_start(rq, p, now);
+ if (p != lrq->curr)
+  update_stats_wait_start(lrq, p, now);
  /*
   * Update the key:
   */
- key = rq->lrq.fair_clock;
+ key = lrq->fair_clock;

  /*
   * Optimize the common nice 0 case:
   */
- if (likely(p->se.load_weight == NICE_0_LOAD)) {
-  key -= p->se.wait_runtime;
```

```
+ if (likely(p->load_weight == lrq->nice_0_load)) {
+  key -= p->wait_runtime;
  } else {
-  int negative = p->se.wait_runtime < 0;
+  int negative = p->wait_runtime < 0;
   u64 tmp;

-  if (p->se.load_weight > NICE_0_LOAD) {
+  if (p->load_weight > lrq->nice_0_load) {
    /* negative-reniced tasks get helped: */

    if (negative) {
-    tmp = -p->se.wait_runtime;
-    tmp *= NICE_0_LOAD;
-    do_div(tmp, p->se.load_weight);
+    tmp = -p->wait_runtime;
+    tmp *= lrq->nice_0_load;
+    do_div(tmp, p->load_weight);

     key += tmp;
    } else {
-    tmp = p->se.wait_runtime;
-    tmp *= p->se.load_weight;
-    do_div(tmp, NICE_0_LOAD);
+    tmp = p->wait_runtime;
+    tmp *= p->load_weight;
+    do_div(tmp, lrq->nice_0_load);

     key -= tmp;
    }
@@ -305,98 +316,98 @@
    /* plus-reniced tasks get hurt: */

    if (negative) {
-    tmp = -p->se.wait_runtime;
+    tmp = -p->wait_runtime;

-    tmp *= NICE_0_LOAD;
-    do_div(tmp, p->se.load_weight);
+    tmp *= lrq->nice_0_load;
+    do_div(tmp, p->load_weight);

     key += tmp;
    } else {
-    tmp = p->se.wait_runtime;
+    tmp = p->wait_runtime;

-    tmp *= p->se.load_weight;
```

```
-    do_div(tmp, NICE_0_LOAD);
+    tmp *= p->load_weight;
+    do_div(tmp, lrq->nice_0_load);

     key -= tmp;
    }
   }
  }

- p->se.fair_key = key;
+ p->fair_key = key;
 }

 /*
  * Note: must be called with a freshly updated rq->fair_clock.
  */
 static inline void
-update_stats_wait_end(struct rq *rq, struct task_struct *p, u64 now)
+update_stats_wait_end(struct lrq *lrq, struct sched_entity *p, u64 now)
 {
  s64 delta_fair, delta_wait;

- delta_wait = now - p->se.wait_start;
- if (unlikely(delta_wait > p->se.wait_max))
-  p->se.wait_max = delta_wait;
-
- if (p->se.wait_start_fair) {
-  delta_fair = rq->lrq.fair_clock - p->se.wait_start_fair;
-  if (unlikely(p->se.load_weight != NICE_0_LOAD))
-   delta_fair = (delta_fair * p->se.load_weight) /
-        NICE_0_LOAD;
-  add_wait_runtime(rq, p, delta_fair);
+ delta_wait = now - p->wait_start;
+ if (unlikely(delta_wait > p->wait_max))
+  p->wait_max = delta_wait;
+
+ if (p->wait_start_fair) {
+  delta_fair = lrq->fair_clock - p->wait_start_fair;
+  if (unlikely(p->load_weight != lrq->nice_0_load))
+   delta_fair = (delta_fair * p->load_weight) /
+        lrq->nice_0_load;
+  add_wait_runtime(lrq, p, delta_fair);
  }

- p->se.wait_start_fair = 0;
- p->se.wait_start = 0;
+ p->wait_start_fair = 0;
+ p->wait_start = 0;
```

```
 }

 static inline void
-update_stats_dequeue(struct rq *rq, struct task_struct *p, u64 now)
+update_stats_dequeue(struct lrq *lrq, struct sched_entity *p, u64 now)
 {
- update_curr(rq, now);
+ update_curr(lrq, now);
  /*
   * Mark the end of the wait period if dequeueing a
   * waiting task:
   */
- if (p != rq->curr)
-  update_stats_wait_end(rq, p, now);
+ if (p != lrq->curr)
+  update_stats_wait_end(lrq, p, now);
 }

 /*
  * We are picking a new current task - update its stats:
  */
 static inline void
-update_stats_curr_start(struct rq *rq, struct task_struct *p, u64 now)
+update_stats_curr_start(struct lrq *lrq, struct sched_entity *p, u64 now)
 {
  /*
   * We are starting a new run period:
   */
- p->se.exec_start = now;
+ p->exec_start = now;
 }

 /*
  * We are descheduling a task - update its stats:
  */
 static inline void
-update_stats_curr_end(struct rq *rq, struct task_struct *p, u64 now)
+update_stats_curr_end(struct lrq *lrq, struct sched_entity *p, u64 now)
 {
- update_curr(rq, now);
+ update_curr(lrq, now);

- p->se.exec_start = 0;
+ p->exec_start = 0;
 }

 /**********************************************************/
 /* Scheduling class queueing methods:
```

```
 */

-static void enqueue_sleeper(struct rq *rq, struct task_struct *p)
+static void enqueue_sleeper(struct lrq *lrq, struct sched_entity *p)
 {
- unsigned long load = get_rq_load(rq);
+ unsigned long load = get_lrq_load(lrq);
  u64 delta_fair = 0;

  if (!(sysctl_sched_load_smoothing & 16))
   goto out;

- delta_fair = rq->lrq.fair_clock - p->se.sleep_start_fair;
+ delta_fair = lrq->fair_clock - p->sleep_start_fair;
  if ((s64)delta_fair < 0)
   delta_fair = 0;

@@ -406,15 +417,15 @@
   */
  if (sysctl_sched_load_smoothing & 8) {
   delta_fair = delta_fair * load;
-  do_div(delta_fair, load + p->se.load_weight);
+  do_div(delta_fair, load + p->load_weight);
  }

- __add_wait_runtime(rq, p, delta_fair);
+ __add_wait_runtime(lrq, p, delta_fair);

 out:
- rq->lrq.wait_runtime += p->se.wait_runtime;
+ lrq->wait_runtime += p->wait_runtime;

- p->se.sleep_start_fair = 0;
+ p->sleep_start_fair = 0;
 }

 /*
@@ -423,43 +434,43 @@
  * then put the task into the rbtree:
  */
 static void
-enqueue_task_fair(struct rq *rq, struct task_struct *p, int wakeup, u64 now)
+enqueue_entity(struct lrq *lrq, struct sched_entity *p, int wakeup, u64 now)
 {
  u64 delta = 0;

  /*
   * Update the fair clock.
```

```
   */
- update_curr(rq, now);
+ update_curr(lrq, now);

  if (wakeup) {
-  if (p->se.sleep_start) {
-   delta = now - p->se.sleep_start;
+  if (p->sleep_start && entity_is_task(p)) {
+   delta = now - p->sleep_start;
    if ((s64)delta < 0)
     delta = 0;

-   if (unlikely(delta > p->se.sleep_max))
-    p->se.sleep_max = delta;
+   if (unlikely(delta > p->sleep_max))
+    p->sleep_max = delta;

-   p->se.sleep_start = 0;
+   p->sleep_start = 0;
   }
-  if (p->se.block_start) {
-   delta = now - p->se.block_start;
+  if (p->block_start && entity_is_task(p)) {
+   delta = now - p->block_start;
    if ((s64)delta < 0)
     delta = 0;

-   if (unlikely(delta > p->se.block_max))
-    p->se.block_max = delta;
+   if (unlikely(delta > p->block_max))
+    p->block_max = delta;

-   p->se.block_start = 0;
+   p->block_start = 0;
   }
-  p->se.sum_sleep_runtime += delta;
+  p->sum_sleep_runtime += delta;

-  if (p->se.sleep_start_fair)
-   enqueue_sleeper(rq, p);
+  if (p->sleep_start_fair)
+   enqueue_sleeper(lrq, p);
  }
- update_stats_enqueue(rq, p, now);
- __enqueue_task_fair(rq, p);
+ update_stats_enqueue(lrq, p, now);
+ __enqueue_entity(lrq, p);
 }
```

```
 /*
@@ -468,18 +479,374 @@
  * update the fair scheduling stats:
  */
 static void
-dequeue_task_fair(struct rq *rq, struct task_struct *p, int sleep, u64 now)
+dequeue_entity(struct lrq *lrq, struct sched_entity *p, int sleep, u64 now)
 {
- update_stats_dequeue(rq, p, now);
+ update_stats_dequeue(lrq, p, now);
  if (sleep) {
-  if (p->state & TASK_INTERRUPTIBLE)
-   p->se.sleep_start = now;
-  if (p->state & TASK_UNINTERRUPTIBLE)
-   p->se.block_start = now;
-  p->se.sleep_start_fair = rq->lrq.fair_clock;
-  rq->lrq.wait_runtime -= p->se.wait_runtime;
+  if (entity_is_task(p)) {
+   struct task_struct *tsk = task_entity(p);
+
+   if (tsk->state & TASK_INTERRUPTIBLE)
+    p->sleep_start = now;
+   if (tsk->state & TASK_UNINTERRUPTIBLE)
+    p->block_start = now;
+  }
+  p->sleep_start_fair = lrq->fair_clock;
+  lrq->wait_runtime -= p->wait_runtime;
+ }
+ __dequeue_entity(lrq, p);
+}
+
+/*
+ * Preempt the current task with a newly woken task if needed:
+ */
+static inline void
+__check_preempt_curr_fair(struct lrq *lrq, struct sched_entity *p,
+     struct sched_entity *curr, unsigned long granularity)
+{
+ s64 __delta = curr->fair_key - p->fair_key;
+
+ /*
+  * Take scheduling granularity into account - do not
+  * preempt the current task unless the best task has
+  * a larger than sched_granularity fairness advantage:
+  */
+ if (__delta > niced_granularity(lrq, curr, granularity))
+  resched_task(lrq->rq->curr);
```

```
+}
+
+static struct sched_entity * pick_next_entity(struct lrq *lrq, 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
+  * runqueue. (note, here we rely on pick_next_task() having
+  * 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);
+ lrq->curr = p;
+
+ return p;
+}
+
+/*
+ * Account for a descheduled task:
+ */
+static void put_prev_entity(struct lrq *lrq, struct sched_entity *prev, u64 now)
+{
+ if (!prev)  /* Don't update idle task's stats */
+  return;
+
+ update_stats_curr_end(lrq, prev, now);
+ /*
+  * If the task is still waiting for the CPU (it just got
+  * preempted), start the wait period:
+  */
+ if (prev->on_rq)
+  update_stats_wait_start(lrq, prev, now);
+}
+
+/*
+ * scheduler tick hitting a task of our scheduling class:
+ */
+static void entity_tick(struct lrq *lrq, struct sched_entity *curr)
+{
+ struct sched_entity *next;
+ u64 now = __rq_clock(lrq->rq);
+
+ /*
+  * Dequeue and enqueue the task to update its
+  * position within the tree:
```

```
+  */
+ dequeue_entity(lrq, curr, 0, now);
+ enqueue_entity(lrq, curr, 0, now);
+
+ /*
+  * Reschedule if another task tops the current one.
+  */
+ next = __pick_next_entity(lrq);
+ if (next == curr)
+  return;
+
+ if (entity_is_task(curr)) {
+  struct task_struct *c = task_entity(curr),
+       *n = task_entity(next);
+
+  if ((c == lrq->rq->idle) || (rt_prio(n->prio) &&
+      (n->prio < c->prio)))
+   resched_task(c);
+ } else
+  __check_preempt_curr_fair(lrq, next, curr,
+       *(lrq->sched_granularity));
+}
+
+static void _update_load(struct lrq *this_rq)
+{
+ unsigned long this_load, fair_delta, exec_delta, idle_delta;
+ unsigned int i, scale;
+ s64 fair_delta64, exec_delta64;
+ unsigned long tmp;
+ u64 tmp64;
+
+ this_rq->nr_load_updates++;
+ if (!(sysctl_sched_load_smoothing & 64)) {
+  this_load = this_rq->raw_weighted_load;
+  goto do_avg;
+ }
+
+ fair_delta64 = this_rq->fair_clock -
+    this_rq->prev_fair_clock + 1;
+ this_rq->prev_fair_clock = this_rq->fair_clock;
+
+ exec_delta64 = this_rq->exec_clock -
+    this_rq->prev_exec_clock + 1;
+ this_rq->prev_exec_clock = this_rq->exec_clock;
+
+ if (fair_delta64 > (s64)LONG_MAX)
+  fair_delta64 = (s64)LONG_MAX;
+ fair_delta = (unsigned long)fair_delta64;
```

```
+
+ if (exec_delta64 > (s64)LONG_MAX)
+  exec_delta64 = (s64)LONG_MAX;
+ exec_delta = (unsigned long)exec_delta64;
+ if (exec_delta > TICK_NSEC)
+  exec_delta = TICK_NSEC;
+
+ idle_delta = TICK_NSEC - exec_delta;
+
+ tmp = (SCHED_LOAD_SCALE * exec_delta) / fair_delta;
+ tmp64 = (u64)tmp * (u64)exec_delta;
+ do_div(tmp64, TICK_NSEC);
+ this_load = (unsigned long)tmp64;
+
+do_avg:
+ /* Update our load: */
+ for (i = 0, scale = 1; i < CPU_LOAD_IDX_MAX; i++, scale += scale) {
+  unsigned long old_load, new_load;
+
+  /* scale is effectively 1 << i now, and >> i divides by scale */
+
+  old_load = this_rq->cpu_load[i];
+  new_load = this_load;
+
+  this_rq->cpu_load[i] = (old_load*(scale-1) + new_load) >> i;
+ }
+}
+
+/*******************  Start task operations ********************************/
+
+static inline struct lrq * task_grp_lrq(const struct task_struct *p)
+{
+#ifdef CONFIG_FAIR_USER_SCHED
+ return &p->user->lrq[task_cpu(p)];
+#else
+ return &task_rq(p)->lrq;
+#endif
+}
+
+/*
+ * The enqueue_task method is called before nr_running is
+ * increased. Here we update the fair scheduling stats and
+ * then put the task into the rbtree:
+ */
+static void
+enqueue_task_fair(struct rq *rq, struct task_struct *p, int wakeup, u64 now)
+{
+ struct lrq *lrq = task_grp_lrq(p);
```

```
+ struct sched_entity *se = &p->se;
+
+ enqueue_entity(lrq, se, wakeup, now);
+ if (p == rq->curr)
+  lrq->curr = se;
+
+ if (likely(!se->parent || se->parent->on_rq))
+  return;
+
+ lrq = &rq->lrq;
+ se = se->parent;
+ if (p == rq->curr)
+  lrq->curr = se;
+ enqueue_entity(lrq, se, wakeup, now);
+ se->on_rq = 1;
+}
+
+/*
+ * The dequeue_task method is called before nr_running is
+ * decreased. We remove the task from the rbtree and
+ * update the fair scheduling stats:
+ */
+static void
+dequeue_task_fair(struct rq *rq, struct task_struct *p, int sleep, u64 now)
+{
+ struct lrq *lrq = task_grp_lrq(p);
+ struct sched_entity *se = &p->se;
+
+ dequeue_entity(lrq, se, sleep, now);
+
+ if (likely(!se->parent || lrq->raw_weighted_load))
+  return;
+
+ se = se->parent;
+ lrq = &rq->lrq;
+ dequeue_entity(lrq, se, sleep, now);
+ se->on_rq = 0;
+}
+
+static struct task_struct * pick_next_task_fair(struct rq *rq, u64 now)
+{
+ struct lrq *lrq;
+ struct sched_entity *se;
+
+ lrq = &rq->lrq;
+ se = pick_next_entity(lrq, now);
+
+ if (se->my_q) {
```

```
+  lrq = se->my_q;
+  se = pick_next_entity(lrq, now);
+ }
+
+ return task_entity(se);
+}
+
+/*
+ * Account for a descheduled task:
+ */
+static void put_prev_task_fair(struct rq *rq, struct task_struct *prev, u64 now)
+{
+ struct lrq *lrq = task_grp_lrq(prev);
+ struct sched_entity *se = &prev->se;
+
+ if (prev == rq->idle)
+  return;
+
+ put_prev_entity(lrq, se, now);
+
+ if (!se->parent)
+  return;
+
+ se = se->parent;
+ lrq = &rq->lrq;
+ put_prev_entity(lrq, se, now);
+}
+
+/*
+ * scheduler tick hitting a task of our scheduling class:
+ */
+static void task_tick_fair(struct rq *rq, struct task_struct *curr)
+{
+ struct lrq *lrq;
+ struct sched_entity *se;
+
+ se = &curr->se;
+ lrq = task_grp_lrq(curr);
+ entity_tick(lrq, se);
+
+ if (likely(!se->parent))
+  return;
+
+ /* todo: reduce tick frequency at higher scheduling levels? */
+ se = se->parent;
+ lrq = &rq->lrq;
+ entity_tick(lrq, se);
+}
```

```
+
+/*
+ * Preempt the current task with a newly woken task if needed:
+ */
+static void check_preempt_curr_fair(struct rq *rq, struct task_struct *p)
+{
+ struct task_struct *curr = rq->curr;
+
+ if ((curr == rq->idle) || rt_prio(p->prio)) {
+  resched_task(curr);
+ } else {
+  struct sched_entity *cse, *nse;
+
+  if (!curr->se.parent || (curr->se.parent == p->se.parent)) {
+   cse = &curr->se;
+   nse = &p->se;
+  } else {
+   cse = curr->se.parent;
+   nse = p->se.parent;
+  }
+
+  __check_preempt_curr_fair(&rq->lrq, cse, nse,
+      sysctl_sched_wakeup_granularity);
+ }
+}
+
+static inline void update_curr(struct lrq *lrq, u64 now)
+{
+ struct task_struct *curtask = lrq->rq->curr;
+ struct lrq *curq;
+
+ if (curtask->sched_class != &fair_sched_class ||
+   curtask == lrq->rq->idle || !curtask->se.on_rq)
+  return;
+
+ /* this is slightly inefficient - need better way of updating clock */
+ curq = task_grp_lrq(curtask);
+ _update_curr(curq, now);
+
+ if (unlikely(curtask->se.parent)) {
+  curq = &lrq->rq->lrq;
+  _update_curr(curq, now);
+ }
+}
+
+void update_load_fair(struct rq *this_rq)
+{
+ struct task_struct *curr = this_rq->curr;
```

```
+ struct lrq *lrq = task_grp_lrq(curr);
+ struct sched_entity *se = &curr->se;
+
+ _update_load(lrq);
+
+ if (!se->parent)
+  return;
+
+ lrq = &this_rq->lrq;
+ _update_load(lrq);
+}
+
+/*
+ * Share the fairness runtime between parent and child, thus the
+ * total amount of pressure for CPU stays equal - new tasks
+ * get a chance to run but frequent forkers are not allowed to
+ * monopolize the CPU. Note: the parent runqueue is locked,
+ * the child is not running yet.
+ */
+static void task_new_fair(struct rq *rq, struct task_struct *p)
+{
+ struct lrq *lrq = task_grp_lrq(p);
+ struct sched_entity *se = &p->se;
+
+ sched_info_queued(p);
+ update_stats_enqueue(lrq, se, rq_clock(rq));
+ /*
+  * Child runs first: we let it run before the parent
+  * until it reschedules once. We set up the key so that
+  * it will preempt the parent:
+  */
+ p->se.fair_key = current->se.fair_key - niced_granularity(lrq,
+     &rq->curr->se, sysctl_sched_granularity) - 1;
+ /*
+  * The first wait is dominated by the child-runs-first logic,
+  * so do not credit it with that waiting time yet:
+  */
+ p->se.wait_start_fair = 0;
+
+ __enqueue_entity(lrq, se);
+ if (unlikely(se && !se->on_rq)) {       /* idle task forking */
+  lrq = &rq->lrq;
+  update_stats_enqueue(lrq, se, rq_clock(rq));
+  __enqueue_entity(lrq, se);
  }
- __dequeue_task_fair(rq, p);
+ inc_nr_running(p, rq);
 }
```

```
 /*
@@ -494,6 +861,8 @@
  struct task_struct *p_next;
  s64 yield_key;
  u64 now;
+ struct lrq *lrq = task_grp_lrq(p);
+ struct sched_entity *se = &p->se;

  /*
   * Bug workaround for 3D apps running on the radeon 3D driver:
@@ -508,15 +877,14 @@
    * Dequeue and enqueue the task to update its
    * position within the tree:
    */
-  dequeue_task_fair(rq, p, 0, now);
-  p->se.on_rq = 0;
-  enqueue_task_fair(rq, p, 0, now);
-  p->se.on_rq = 1;
+  dequeue_entity(lrq, se, 0, now);
+  enqueue_entity(lrq, se, 0, now);

   /*
    * Reschedule if another task tops the current one.
    */
-  p_next = __pick_next_task_fair(rq);
+  se = __pick_next_entity(lrq);
+  p_next = task_entity(se);
   if (p_next != p)
    resched_task(p);
   return;
@@ -531,7 +899,7 @@
   p->se.wait_runtime >>= 1;
  }
  curr = &p->se.run_node;
- first = first_fair(rq);
+ first = first_fair(lrq);
  /*
   * Move this task to the second place in the tree:
   */
@@ -554,8 +922,7 @@
  yield_key = p_next->se.fair_key + 1;

  now = __rq_clock(rq);
- dequeue_task_fair(rq, p, 0, now);
- p->se.on_rq = 0;
+ dequeue_entity(lrq, se, 0, now);
```

```
  /*
   * Only update the key if we need to move more backwards
@@ -564,75 +931,7 @@
  if (p->se.fair_key < yield_key)
   p->se.fair_key = yield_key;

- __enqueue_task_fair(rq, p);
- p->se.on_rq = 1;
-}
-
-/*
- * Preempt the current task with a newly woken task if needed:
- */
-static inline void
-__check_preempt_curr_fair(struct rq *rq, struct task_struct *p,
-     struct task_struct *curr, unsigned long granularity)
-{
- s64 __delta = curr->se.fair_key - p->se.fair_key;
-
- /*
-  * Take scheduling granularity into account - do not
-  * preempt the current task unless the best task has
-  * a larger than sched_granularity fairness advantage:
-  */
- if (__delta > niced_granularity(curr, granularity))
-  resched_task(curr);
-}
-
-/*
- * Preempt the current task with a newly woken task if needed:
- */
-static void check_preempt_curr_fair(struct rq *rq, struct task_struct *p)
-{
- struct task_struct *curr = rq->curr;
-
- if ((curr == rq->idle) || rt_prio(p->prio)) {
-  resched_task(curr);
- } else {
-  __check_preempt_curr_fair(rq, p, curr,
-       sysctl_sched_wakeup_granularity);
- }
-}
-
-static struct task_struct * pick_next_task_fair(struct rq *rq, u64 now)
-{
- struct task_struct *p = __pick_next_task_fair(rq);
-
- /*
```

```
-  * Any task has to be enqueued before it get to execute on
-  * a CPU. So account for the time it spent waiting on the
-  * runqueue. (note, here we rely on pick_next_task() having
-  * done a put_prev_task_fair() shortly before this, which
-  * updated rq->fair_clock - used by update_stats_wait_end())
-  */
- update_stats_wait_end(rq, p, now);
- update_stats_curr_start(rq, p, now);
-
- return p;
-}
-
-/*
- * Account for a descheduled task:
- */
-static void put_prev_task_fair(struct rq *rq, struct task_struct *prev, u64 now)
-{
- if (prev == rq->idle)
-  return;
-
- update_stats_curr_end(rq, prev, now);
- /*
-  * If the task is still waiting for the CPU (it just got
-  * preempted), start the wait period:
-  */
- if (prev->se.on_rq)
-  update_stats_wait_start(rq, prev, now);
+ __enqueue_entity(lrq, se);
 }

 /**************************************************************/
@@ -648,6 +947,7 @@
  */
 static struct task_struct * load_balance_start_fair(struct rq *rq)
 {
+#if 0
  struct rb_node *first = first_fair(rq);
  struct task_struct *p;

@@ -659,10 +959,13 @@
  rq->lrq.rb_load_balance_curr = rb_next(first);

  return p;
+#endif
+ return NULL; /* todo: fix load balance */
 }

 static struct task_struct * load_balance_next_fair(struct rq *rq)
```

```
 {
+#if 0
  struct rb_node *curr = rq->lrq.rb_load_balance_curr;
  struct task_struct *p;

@@ -673,67 +976,8 @@
  rq->lrq.rb_load_balance_curr = rb_next(curr);

  return p;
-}
-
-/*
- * scheduler tick hitting a task of our scheduling class:
- */
-static void task_tick_fair(struct rq *rq, struct task_struct *curr)
-{
- struct task_struct *next;
- u64 now = __rq_clock(rq);
-
- /*
-  * Dequeue and enqueue the task to update its
-  * position within the tree:
-  */
- dequeue_task_fair(rq, curr, 0, now);
- curr->se.on_rq = 0;
- enqueue_task_fair(rq, curr, 0, now);
- curr->se.on_rq = 1;
-
- /*
-  * Reschedule if another task tops the current one.
-  */
- next = __pick_next_task_fair(rq);
- if (next == curr)
-  return;
-
- if ((curr == rq->idle) || (rt_prio(next->prio) &&
-     (next->prio < curr->prio)))
-  resched_task(curr);
- else
-  __check_preempt_curr_fair(rq, next, curr,
-       sysctl_sched_granularity);
-}
-
-/*
- * Share the fairness runtime between parent and child, thus the
- * total amount of pressure for CPU stays equal - new tasks
- * get a chance to run but frequent forkers are not allowed to
- * monopolize the CPU. Note: the parent runqueue is locked,
```

```
- * the child is not running yet.
- */
-static void task_new_fair(struct rq *rq, struct task_struct *p)
-{
- sched_info_queued(p);
- update_stats_enqueue(rq, p, rq_clock(rq));
- /*
-  * Child runs first: we let it run before the parent
-  * until it reschedules once. We set up the key so that
-  * it will preempt the parent:
-  */
- p->se.fair_key = current->se.fair_key - niced_granularity(rq->curr,
-      sysctl_sched_granularity) - 1;
- /*
-  * The first wait is dominated by the child-runs-first logic,
-  * so do not credit it with that waiting time yet:
-  */
- p->se.wait_start_fair = 0;
-
- __enqueue_task_fair(rq, p);
- p->se.on_rq = 1;
- inc_nr_running(p, rq);
+#endif
+ return NULL;
 }

 /*
Index: linux-2.6.21-rc7/kernel/user.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/user.c 2007-05-23 20:46:38.000000000 +0530
+++ linux-2.6.21-rc7/kernel/user.c 2007-05-23 20:48:39.000000000 +0530
@@ -112,6 +112,7 @@
  if (atomic_dec_and_lock(&up->__count, &uidhash_lock)) {
   uid_hash_remove(up);
   spin_unlock_irqrestore(&uidhash_lock, flags);
+  sched_free_user(up);
   key_put(up->uid_keyring);
   key_put(up->session_keyring);
   kmem_cache_free(uid_cachep, up);
@@ -153,6 +154,8 @@
    return NULL;
   }

+  sched_alloc_user(new);
+
   /*
    * Before adding this, check whether we raced
    * on adding the same user already..
```

```
@@ -163,6 +166,7 @@
    key_put(new->uid_keyring);
    key_put(new->session_keyring);
    kmem_cache_free(uid_cachep, new);
+   sched_free_user(new);
   } else {
    uid_hash_insert(new, hashent);
    up = new;
@@ -187,6 +191,7 @@
  atomic_dec(&old_user->processes);
  switch_uid_keyring(new_user);
  current->user = new_user;
+ sched_move_task(old_user);

  /*
   * We need to synchronize with __sigqueue_alloc()
Index: linux-2.6.21-rc7/kernel/sched_rt.c
===================================================================
--- linux-2.6.21-rc7.orig/kernel/sched_rt.c 2007-05-23 09:28:03.000000000 +0530
+++ linux-2.6.21-rc7/kernel/sched_rt.c 2007-05-23 20:48:39.000000000 +0530
@@ -166,7 +166,7 @@
  activate_task(rq, p, 1);
 }

-static struct sched_class rt_sched_class __read_mostly = {
+struct sched_class rt_sched_class __read_mostly = {
  .enqueue_task  = enqueue_task_rt,
  .dequeue_task  = dequeue_task_rt,
  .yield_task  = yield_task_rt,
--
Regards,
vatsa
```

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Peter Williams on Thu, 24 May 2007 03:15:04 GMT
View Forum Message <> Reply to Message

Srivatsa Vaddagiri wrote:
> Here's an attempt to extend CFS (v13) to be fair at a group level, rather than
> just at task level. The patch is in a very premature state (passes
> simple tests, smp load balance not supported yet) at this point. I am sending
> it out early to know if this is a good direction to proceed.
>

> Salient points which needs discussion:
>
> 1. This patch reuses CFS core to achieve fairness at group level also.
>
>   To make this possible, CFS core has been abstracted to deal with generic
>   schedulable "entities" (tasks, users etc).
>
> 2. The per-cpu rb-tree has been split to be per-group per-cpu.
>
>   schedule() now becomes two step on every cpu : pick a group first (from
>   group rb-tree) and a task within that group next (from that group's task
>   rb-tree)
>
> 3. Grouping mechanism - I have used 'uid' as the basis of grouping for
>   timebeing (since that grouping concept is already in mainline today).
>   The patch can be adapted to a more generic process grouping mechanism
>   (like http://lkml.org/lkml/2007/4/27/146) later.
>
> Some results below, obtained on a 4way (with HT) Intel Xeon box. All
> number are reflective of single CPU performance (tests were forced to
> run on single cpu since load balance is not yet supported).
>
>
>   uid "vatsa"   uid "guest"
>   (make -s -j4 bzImage)   (make -s -j20 bzImage)
>
> 2.6.22-rc1   772.02 sec 497.42 sec (real)
> 2.6.22-rc1+cfs-v13   780.62 sec 478.35 sec (real)
> 2.6.22-rc1+cfs-v13+this patch   776.36 sec 776.68 sec (real)

This would seem to indicate that being fair between groups isn't always
a good thing. With 2.6.22-rc1 and 2.6.22-rc1+cfs-v13 "guest" gets his
build done in about 2/3 the time of "vatsa" without seriously
inconveniencing "vatsa". All making scheduling fair between the groups
has done is penalize "guest" without significantly improving matters for
"vatsa" (he gains a mere 4 seconds out of 780).

BUT I imagine that this is an artefact caused by the use of HT
technology and that if the test were run on a computer without HT the
results would be more impressive.

Peter
--
Peter Williams   pwil3058@bigpond.net.au

"Learning, n. The kind of ignorance distinguishing the studious."
  -- Ambrose Bierce

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Ingo Molnar on Fri, 25 May 2007 08:29:51 GMT
View Forum Message <> Reply to Message

* Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:

> Can you repeat your tests with this patch pls? With the patch applied,
> I am now getting the same split between nice 0 and nice 10 task as
> CFS-v13 provides (90:10 as reported by top)
>
> 5418 guest   20   0 2464 304 236 R   90 0.0   5:41.40 3 hog
> 5419 guest   30 10 2460 304 236 R   10 0.0   0:43.62 3 nice10hog

btw., what are you thoughts about SMP?

it's a natural extension of your current code. I think the best approach would be to add a level of 'virtual CPU' objects above struct user. (how to set the attributes of those objects is open - possibly combine it with cpusets?)

That way the scheduler would first pick a "virtual CPU" to schedule, and then pick a user from that virtual CPU, and then a task from the user.

To make group accounting scalable, the accounting object attached to the user struct should/must be per-cpu (per-vcpu) too. That way we'd have a clean hierarchy like:

  CPU #0 => VCPU A [40%] + VCPU B [60%]
  CPU #1 => VCPU C [30%] + VCPU D [70%]

  VCPU A => USER X [10%] + USER Y [90%]
  VCPU B => USER X [10%] + USER Y [90%]
  VCPU C => USER X [10%] + USER Y [90%]
  VCPU D => USER X [10%] + USER Y [90%]

the scheduler first picks a vcpu, then a user from a vcpu. (the actual external structure of the hierarchy should be opaque to the scheduler core, naturally, so that we can use other hierarchies too)

whenever the scheduler does accounting, it knows where in the hierarchy it is and updates all higher level entries too. This means that the accounting object for USER X is replicated for each VCPU it participates in.

SMP balancing is straightforward: it would fundamentally iterate through
the same hierarchy and would attempt to keep all levels balanced - i
abstracted away its iterators already.

Hm?

 Ingo

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

---

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Guillaume Chazarain on Fri, 25 May 2007 09:30:27 GMT
View Forum Message <> Reply to Message

> Can you repeat your tests with this patch pls? With the patch applied, I am
> now getting the same split between nice 0 and nice 10 task as CFS-v13
> provides (90:10 as reported by top)

Yep, this fixes the problem for me too.

Thanks.

--
Guillaume

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

---

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Srivatsa Vaddagiri on Fri, 25 May 2007 10:56:41 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2007 at 10:29:51AM +0200, Ingo Molnar wrote:
> btw., what are you thoughts about SMP?

I was planning on reusing smpnice concepts here, with the difference
that we balance group weights across CPU in addition to total weight of
CPUs.

For ex, assuming weight of each task is 10

CPU0 => USER X (Wt 100) + USER Y (Wt 200) => total weight 300
CPU1 => USER X (Wt 30) + USER Y (Wt 80) => total weight 110

So first we notice that CPU0 and CPU1 are imbalanced by weight 190 and
target at reducing this imbalance by half i.e CPU1 has to pull total
weight of 95 (190/2) from CPU0. However while pulling weights, we apply
the same imbalance/2 rule at group level also. For ex: we cannot pull more than
70/2 = 35 from USER X on CPU0 or more than 120/2 = 60 from USER Y on CPU0.

Using this rule, after balance, the two CPUs may look like:

CPU0 => USER X (Wt 70) + USER Y (Wt 140) => total weight 210
CPU1 => USER X (Wt 60) + USER Y (Wt 140) => total weight 200

I had tried this approach earlier (in
https://lists.linux-foundation.org/pipermail/containers/2007-April/004580.html)
and had obtained decent results. It also required minimal changes to
smpnice.

Compared to this, what better degree of control/flexibilty does virtual
cpu approach give?

> it's a natural extension of your current code. I think the best approach
> would be to add a level of 'virtual CPU' objects above struct user. (how
> to set the attributes of those objects is open - possibly combine it
> with cpusets?)

are these virtual CPUs visible to users (ex: does smp_processor_id()
return virtual cpu id rather than physical id and does DEFINE_PER_CPU
create per-cpu data for virtual CPUs rather than physical cpus)?

> That way the scheduler would first pick a "virtual CPU" to schedule,

are virtual cpus pinned to their physical cpu or can they bounce around?
i.e can CPU #0 schedule VCPU D (in your example below)? If bouncing is
allowed, I am not sure whether that is a good thing for performance. How
do we minimize this performance cost?

> and then pick a user from that virtual CPU, and then a task from the user.
>
> To make group accounting scalable, the accounting object attached to the
> user struct should/must be per-cpu (per-vcpu) too. That way we'd have a
> clean hierarchy like:
>
>   CPU #0 => VCPU A [40%] + VCPU B [60%]
>   CPU #1 => VCPU C [30%] + VCPU D [70%]
>

> VCPU A => USER X [10%] + USER Y [90%]
> VCPU B => USER X [10%] + USER Y [90%]
> VCPU C => USER X [10%] + USER Y [90%]
> VCPU D => USER X [10%] + USER Y [90%]
>
> the scheduler first picks a vcpu, then a user from a vcpu. (the actual
> external structure of the hierarchy should be opaque to the scheduler
> core, naturally, so that we can use other hierarchies too)
>
> whenever the scheduler does accounting, it knows where in the hierarchy
> it is and updates all higher level entries too. This means that the
> accounting object for USER X is replicated for each VCPU it participates
> in.
>
> SMP balancing is straightforward: it would fundamentally iterate through
> the same hierarchy and would attempt to keep all levels balanced - i
> abstracted away its iterators already
>
> Hm?

--
Regards,
vatsa

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

---

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Ingo Molnar on Fri, 25 May 2007 11:11:40 GMT
View Forum Message <> Reply to Message

* Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:

> On Fri, May 25, 2007 at 10:29:51AM +0200, Ingo Molnar wrote:
> > btw., what are you thoughts about SMP?
>
> I was planning on reusing smpnice concepts here, with the difference
> that we balance group weights across CPU in addition to total weight
> of CPUs.

ok, that would be (much) simpler that any explicit vcpu approach. Do you
plan to add this next?

 Ingo

_____
Containers mailing list

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Srivatsa Vaddagiri on Fri, 25 May 2007 11:28:49 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2007 at 01:11:40PM +0200, Ingo Molnar wrote:
> > I was planning on reusing smpnice concepts here, with the difference
> > that we balance group weights across CPU in addition to total weight
> > of CPUs.
>
> ok, that would be (much) simpler that any explicit vcpu approach. Do you
> plan to add this next?

sure, after i first cleanup current group changes a bit. Hopefully I can
send you first version of smp support by early next week.

--
Regards,
vatsa

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

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Ingo Molnar on Fri, 25 May 2007 12:05:36 GMT
View Forum Message <> Reply to Message

* Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:

> On Fri, May 25, 2007 at 01:11:40PM +0200, Ingo Molnar wrote:
> > > I was planning on reusing smpnice concepts here, with the difference
> > > that we balance group weights across CPU in addition to total weight
> > > of CPUs.
> >
> > ok, that would be (much) simpler that any explicit vcpu approach. Do you
> > plan to add this next?
>
> sure, after i first cleanup current group changes a bit. Hopefully I
> can send you first version of smp support by early next week.

great. Btw., could you please keep the "up to this point there should be
no behavioral change in CFS" fundamental splitup of your patches - that

way i can look at the core changes (and possibly apply them) without
having to consider the uid based changes (which do change behavior and
which need more upstream buy-in).

Ingo

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

---

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Srivatsa Vaddagiri on Fri, 25 May 2007 12:41:41 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2007 at 02:05:36PM +0200, Ingo Molnar wrote:
> great. Btw., could you please keep the "up to this point there should be
> no behavioral change in CFS" fundamental splitup of your patches -

sure ..basically the changes required in CFS core is the introduction of two
structures - struct sched_entity and struct lrq and generalization of
cfs routines to operate on these structures rather than on
task_struct/rq structures directly.

In the first split, I will ensure that this generalization is applied
only to tasks (which represents reorganization of core with no
behavioral/functional change in scheduler) and in a subsequent split/patch I
will apply the generalization to uids also (which will add group fairness
aspect to scheduler), as you require.

Thanks for your feedback so far!

> that way i can look at the core changes (and possibly apply them) without
> having to consider the uid based changes (which do change behavior and
> which need more upstream buy-in).

--
Regards,
vatsa

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

---

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by dev on Fri, 25 May 2007 13:05:16 GMT

Ingo Molnar wrote:
> * Srivatsa Vaddagiri <vatsa@in.ibm.com> wrote:
>
>
>>Can you repeat your tests with this patch pls? With the patch applied,
>>I am now getting the same split between nice 0 and nice 10 task as
>>CFS-v13 provides (90:10 as reported by top)
>>
>> 5418 guest   20   0 2464 304 236 R   90 0.0   5:41.40 3 hog
>> 5419 guest   30 10 2460 304 236 R   10 0.0   0:43.62 3 nice10hog
>
>
> btw., what are you thoughts about SMP?
>
> it's a natural extension of your current code. I think the best approach
> would be to add a level of 'virtual CPU' objects above struct user. (how
> to set the attributes of those objects is open - possibly combine it
> with cpusets?)

> That way the scheduler would first pick a "virtual CPU" to schedule, and
> then pick a user from that virtual CPU, and then a task from the user.

don't you mean the vice versa:
first use to scheduler, then VCPU (which is essentially a runqueue or rbtree),
then a task from VCPU?

this is the approach we use in OpenVZ and if you don't mind
I would propose to go this way for fair-scheduling in mainstream.
It has it's own advantages and disatvantages.

This is not the easy way to go and I can outline the problems/disadvantages
which appear on this way:
- tasks which bind to CPU mask will bind to virtual CPUs.
  no problem with user tasks, but some kernel threads
  use this to do CPU-related management (like cpufreq).
  This can be fixed using SMP IPI actually.
- VCPUs should no change PCPUs very frequently,
  otherwise there is some overhead. Solvable.

Advantages:
- High precision and fairness.
- Allows to use different group scheduling algorithms
  on top of VCPU concept.
  OpenVZ uses fairscheduler with CPU limiting feature allowing
  to set maximum CPU time given to a group of tasks.

> To make group accounting scalable, the accounting object attached to the

> user struct should/must be per-cpu (per-vcpu) too. That way we'd have a
> clean hierarchy like:
>
> CPU #0 => VCPU A [40%] + VCPU B [60%]
> CPU #1 => VCPU C [30%] + VCPU D [70%]

how did you select these 40%:60% and 30%:70% split?

> VCPU A => USER X [10%] + USER Y [90%]
> VCPU B => USER X [10%] + USER Y [90%]
> VCPU C => USER X [10%] + USER Y [90%]
> VCPU D => USER X [10%] + USER Y [90%]
>
> the scheduler first picks a vcpu, then a user from a vcpu. (the actual
> external structure of the hierarchy should be opaque to the scheduler
> core, naturally, so that we can use other hierarchies too)
>
> whenever the scheduler does accounting, it knows where in the hierarchy
> it is and updates all higher level entries too. This means that the
> accounting object for USER X is replicated for each VCPU it participates
> in.

So if 2 VCPUs running on 2 physical CPUs do accounting the have to update the same
user X accounting information which is not per-[v]cpu?

> SMP balancing is straightforward: it would fundamentally iterate through
> the same hierarchy and would attempt to keep all levels balanced - i
> abstracted away its iterators already.

Thanks,
Kirill

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Srivatsa Vaddagiri on Fri, 25 May 2007 15:34:50 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2007 at 05:05:16PM +0400, Kirill Korotaev wrote:
> > That way the scheduler would first pick a "virtual CPU" to schedule, and
> > then pick a user from that virtual CPU, and then a task from the user.
>
> don't you mean the vice versa:
> first use to scheduler, then VCPU (which is essentially a runqueue or rbtree),
> then a task from VCPU?

>
> this is the approach we use in OpenVZ [...]

So is this how it looks in OpenVZ?

 CONTAINER1 => VCPU0 + VCPU1
 CONTAINER2 => VCPU2 + VCPU3

PCPU0 picks a container first, a vcpu next and then a task in it
PCPU1 also picks a container first, a vcpu next and then a task in it.

Few questions:

1. Are VCPU runqueues (on which tasks are present) global queues?

   That is, let's say that both PCPU0 and PCPU1 pick CONTAINER1 to schedule (first level) at the same time and next (let's say) they pick same vcpu VCPU0 to schedule (second level). Will the two pcpu's now have to be serialized for scanning task to schedule next (third level) within VCPU0 using a spinlock? Won't that shootup scheduling costs (esp on large systems), compared to (local scheduling + balance across cpus once in a while, the way its done today)?

   Or do you required that two pcpus don't schedule the same vcpu at the same time (the way hypervisors normally work)? Even then I would imagine a fair level of contention to be present in second step (pick a virtual cpu from a container's list of vcpus).

2. How would this load balance at virtual cpu level and sched domain based load balancing interact?

   The current sched domain based balancing code has many HT/MC/SMT related optimizations, which ensure that tasks are spread across physical threads/cores/packages in a most efficient manner - so as to utilize hardware bandwidth to the maximum. You would now need to introduce those optimizations essentially at schedule() time ..? Don't know if that is a wise thing to do.

3. How do you determine the number of VCPUs per container? Is there any relation for number of virtual cpus exposed per user/container and the number of available cpus? For ex: in case of user-driven scheduling, we would want all users to see the same number of cpus (which is the number available in the system).

4. VCPU ids (namespace) - is it different for different containers?

   For ex: can id's of vcpus belonging to different containers (say VCPU0 and VCPU2), as seen by users thr' vgetcpu/smp_processor_id() that is, be same?

If so, then potentially two threads belonging to different users may find
that they are running -truly simultaneously- on /same/ cpu 0 (one on
VCPU0/PCPU0 and another on VCPU2/PCPU1) which normally isn't possible!

This may be ok for containers, with non-overlapping cpu id namespace,
but when applied to group scheduling for, say, users, which require a
global cpu id namespace, wondering how that would be addressed ..

> and if you don't mind I would propose to go this way for fair-scheduling in
> mainstream.
> It has it's own advantages and disatvantages.
>
> This is not the easy way to go and I can outline the problems/disadvantages
> which appear on this way:
> - tasks which bind to CPU mask will bind to virtual CPUs.
> no problem with user tasks, [...]

Why is this not a problem for user tasks? Tasks which bind to different
CPUs for performance reason now can find that they are running on same
(physical) CPU unknowingly.

> but some kernel threads
> use this to do CPU-related management (like cpufreq).
> This can be fixed using SMP IPI actually.
> - VCPUs should no change PCPUs very frequently,
> otherwise there is some overhead. Solvable.
>
> Advantages:
> - High precision and fairness.

I just don't know if this benefit of high degree of fairness is worth the
complexity it introduces. Besides having some data which shows how much better
is is with respect to fairness/overhead when compared with other approaches
(like smpnice) would help I guess. I will however let experts like Ingo make
the final call here :)

--
Regards,
vatsa

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

---

Subject: Re: [RFC] [PATCH 0/3] Add group fairness to CFS

Posted by tong.n.li on Fri, 25 May 2007 17:14:58 GMT
View Forum Message <> Reply to Message

On Fri, 2007-05-25 at 21:44 +0530, Srivatsa Vaddagiri wrote:
> >
> > That assumes per-user scheduling groups; other configurations would
> > make it one step for each level of hierarchy. It may be possible to
> > reduce those steps to only state transitions that change weightings
> > and incremental updates of task weightings. By and large, one needs
> > the groups to determine task weightings as opposed to hierarchically
> > scheduling, so there are alternative ways of going about this, ones
> > that would even make load balancing easier.
>
> Yeah I agree that providing hierarchical group-fairness at the cost of single
> (or fewer) scheduling levels would be a nice thing to target for,
> although I don't know of any good way to do it. Do you have any ideas
> here? Doing group fairness in a single level, using a common rb-tree for tasks
> from all groups is very difficult IMHO. We need atleast two levels.
>
> One possibility is that we recognize deeper hierarchies only in user-space,
> but flatten this view from kernel perspective i.e some user space tool
> will have to distributed the weights accordingly in this flattened view
> to the kernel.

Nice work, Vatsa. When I wrote the DWRR algorithm, I flattened the hierarchies into one level, so maybe that approach can be applied to your code as well. What I did is to maintain task and task group weights and reservations separately from the scheduler, while the scheduler only sees one system-wide weight per task and does not concern about which group a task is in. The key here is the system-wide weight of each task should represent an equivalent share to the share represented by the group hierarchies. To do this, the scheduler looks up the task and group weights/reservations it maintains, and dynamically computes the system-wide weight *only* when it need a weight for a given task while scheduling. The on-demand weight computation makes sure the cost is small (constant time). The computation itself can be seen from an example: assume we have a group of two tasks and the group's total share is represented by a weight of 10. Inside the group, let's say the two tasks, P1 and P2, have weights 1 and 2. Then the system-wide weight for P1 is 10/3 and the weight for P2 is 20/3. In essence, this flattens weights into one level without changing the shares they represent.

tong

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Srivatsa Vaddagiri on Mon, 28 May 2007 16:39:19 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2007 at 10:14:58AM -0700, Li, Tong N wrote:
> Nice work, Vatsa. When I wrote the DWRR algorithm, I flattened the
> hierarchies into one level, so maybe that approach can be applied to
> your code as well. What I did is to maintain task and task group weights
> and reservations separately from the scheduler, while the scheduler only
> sees one system-wide weight per task and does not concern about which
> group a task is in. The key here is the system-wide weight of each task
> should represent an equivalent share to the share represented by the
> group hierarchies. To do this, the scheduler looks up the task and group
> weights/reservations it maintains, and dynamically computes the
> system-wide weight *only* when it need a weight for a given task while
> scheduling. The on-demand weight computation makes sure the cost is
> small (constant time). The computation itself can be seen from an
> example: assume we have a group of two tasks and the group's total share
> is represented by a weight of 10. Inside the group, let's say the two
> tasks, P1 and P2, have weights 1 and 2. Then the system-wide weight for
> P1 is 10/3 and the weight for P2 is 20/3. In essence, this flattens
> weights into one level without changing the shares they represent.

What do these task weights control? Timeslice primarily? If so, I am not
sure how well it can co-exist with cfs then (unless you are planning to
replace cfs with a equally good interactive/fair scheduler :)

I would be very interested if this weight calculation can be used for
smpnice based load balancing purposes too ..

--
Regards,
vatsa

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

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by billh on Wed, 30 May 2007 00:14:06 GMT
View Forum Message <> Reply to Message

On Mon, May 28, 2007 at 10:09:19PM +0530, Srivatsa Vaddagiri wrote:
> On Fri, May 25, 2007 at 10:14:58AM -0700, Li, Tong N wrote:
> > is represented by a weight of 10. Inside the group, let's say the two
> > tasks, P1 and P2, have weights 1 and 2. Then the system-wide weight for
> > P1 is 10/3 and the weight for P2 is 20/3. In essence, this flattens

> > weights into one level without changing the shares they represent.
>
> What do these task weights control? Timeslice primarily? If so, I am not
> sure how well it can co-exist with cfs then (unless you are planning to
> replace cfs with a equally good interactive/fair scheduler :)

It's called SD. From Con Kolivas that got it right the first time around :)

> I would be very interested if this weight calculation can be used for
> smpnice based load balancing purposes too ..

bill

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by William Lee Irwin III on Wed, 30 May 2007 02:51:32 GMT
View Forum Message <> Reply to Message

On Mon, May 28, 2007 at 10:09:19PM +0530, Srivatsa Vaddagiri wrote:
> What do these task weights control? Timeslice primarily? If so, I am not
> sure how well it can co-exist with cfs then (unless you are planning to
> replace cfs with a equally good interactive/fair scheduler :)
> I would be very interested if this weight calculation can be used for
> smpnice based load balancing purposes too ..

Task weights represent shares of CPU bandwidth. If task i has weight w_i
then its share of CPU bandwidth is intended to be w_i/sum_i w_i.

"Load weight" seems to be used more in the scheduler source.

-- wli
