Subject: [RFC][PATCH 3/6] core changes in CFS
Posted by Srivatsa Vaddagiri on Mon, 11 Jun 2007 15:53:45 GMT
View Forum Message <> Reply to Message

This patch introduces core changes in CFS work to operate on generic schedulable entities. The task specific operations (like enqueue, dequeue, task_tick etc) is then rewritten to work off this generic CFS "library".

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

```
---
 kernel/sched_debug.c |    2
 kernel/sched_fair.c  |  574 ++++++++++++++++++++++++++++++++++---------------------
 2 files changed, 345 insertions(+), 231 deletions(-)

Index: current/kernel/sched_fair.c
===================================================================
--- current.orig/kernel/sched_fair.c 2007-06-09 15:07:16.000000000 +0530
+++ current/kernel/sched_fair.c 2007-06-09 15:07:33.000000000 +0530
@@ -42,19 +42,54 @@

 extern struct sched_class fair_sched_class;

+/******************************************************************************/
+/*           BEGIN : CFS operations on generic schedulable entities          */
+/******************************************************************************/
+
+static inline struct rq *lrq_rq(struct lrq *lrq)
+{
+ return container_of(lrq, struct rq, lrq);
+}
+
+static inline struct sched_entity *lrq_curr(struct lrq *lrq)
+{
+ struct rq *rq = lrq_rq(lrq);
+ struct sched_entity *se = NULL;
+
+ if (rq->curr->sched_class == &fair_sched_class)
+  se = &rq->curr->se;
+
+ return se;
+}
+
+static long lrq_nr_running(struct lrq *lrq)
+{
+ struct rq *rq = lrq_rq(lrq);
+
```

```
+ return rq->nr_running;
+}
+
+#define entity_is_task(se) 1
+
+static inline struct task_struct *entity_to_task(struct sched_entity *se)
+{
+ return container_of(se, struct task_struct, se);
+}
+
+
 /**************************************************************/
 /* Scheduling class tree data structure manipulation methods:
  */

 /*
- * Enqueue a task into the rb-tree:
+ * Enqueue a entity 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;

  /*
@@ -62,12 +97,12 @@
   */
  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;
@@ -80,31 +115,31 @@
```

```
   * 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);
 }

-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);
 }

-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);
 }

 /**************************************************************/
@@ -116,21 +151,21 @@
  * nice level, but only linearly, not exponentially:
  */
```

```
 static u64
-niced_granularity(struct task_struct *curr, unsigned long granularity)
+niced_granularity(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 >= 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 / NICE_0_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;

@@ -138,30 +173,31 @@
   * Niced tasks have the same history dynamic range as
   * non-niced tasks:
   */
- if (p->se.wait_runtime > limit) {
-  p->se.wait_runtime = limit;
-  p->se.wait_runtime_overruns++;
-  rq->lrq.wait_runtime_overruns++;
- }
- 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_overruns++;
+  lrq->wait_runtime_overruns++;
+ }
+ 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;
 }

 static s64 div64_s(s64 divident, unsigned long divisor)
@@ -183,49 +219,51 @@
  * 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)
 {
- unsigned long load = rq->lrq.raw_weighted_load;
+ unsigned long load = lrq->raw_weighted_load;
  u64 delta_exec, delta_fair, delta_mine;
- struct task_struct *curr = rq->curr;
+ struct sched_entity *curr = lrq_curr(lrq);
+ struct rq *rq = lrq_rq(lrq);
+ struct task_struct *curtask = rq->curr;

- if (curr->sched_class != &fair_sched_class || curr == rq->idle || !load)
+ if (!curr || curtask == rq->idle || !load)
   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;
+ delta_exec = now - curr->exec_start;
```

```
  if (unlikely((s64)delta_exec < 0))
   delta_exec = 0;
- if (unlikely(delta_exec > curr->se.exec_max))
-  curr->se.exec_max = delta_exec;
+ if (unlikely(delta_exec > curr->exec_max))
+  curr->exec_max = delta_exec;

- 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;

  delta_fair = delta_exec * NICE_0_LOAD;
  delta_fair += load >> 1; /* rounding */
  do_div(delta_fair, load);

  /* Load-balancing accounting. */
- rq->lrq.delta_fair_clock += delta_fair;
- rq->lrq.delta_exec_clock += delta_exec;
+ lrq->delta_fair_clock += delta_fair;
+ lrq->delta_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(sysctl_sched_features & 1))
-  if (unlikely(test_tsk_thread_flag(curr, TIF_NEED_RESCHED)))
+  if (unlikely(test_tsk_thread_flag(curtask, TIF_NEED_RESCHED)))
    return;

- delta_mine = delta_exec * curr->se.load_weight;
+ delta_mine = delta_exec * curr->load_weight;
  delta_mine += load >> 1; /* rounding */
  do_div(delta_mine, load);

- 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
@@ -233,21 +271,21 @@
   * 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);
 }

 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;

@@ -255,86 +293,86 @@
   * 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(lrq))
+  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 == NICE_0_LOAD))
+  key -= p->wait_runtime;
  else {
-  if (p->se.wait_runtime < 0)
-   key -= div64_s(p->se.wait_runtime * NICE_0_LOAD,
-         p->se.load_weight);
+  if (p->wait_runtime < 0)
```

```
+   key -= div64_s(p->wait_runtime * NICE_0_LOAD,
+         p->load_weight);
   else
-   key -= div64_s(p->se.wait_runtime * p->se.load_weight,
+   key -= div64_s(p->wait_runtime * p->load_weight,
          NICE_0_LOAD);
  }

- 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;
+ delta_wait = now - p->wait_start;
+ if (unlikely(delta_wait > p->wait_max))
+  p->wait_max = delta_wait;

- if (p->se.wait_start_fair) {
-  delta_fair = rq->lrq.fair_clock - p->se.wait_start_fair;
+ if (p->wait_start_fair) {
+  delta_fair = lrq->fair_clock - p->wait_start_fair;

-  if (unlikely(p->se.load_weight != NICE_0_LOAD))
-   delta_fair = div64_s(delta_fair * p->se.load_weight,
+  if (unlikely(p->load_weight != NICE_0_LOAD))
+   delta_fair = div64_s(delta_fair * p->load_weight,
          NICE_0_LOAD);
-  add_wait_runtime(rq, p, delta_fair);
+  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(lrq))
+  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)
 {
- p->se.exec_start = 0;
+ p->exec_start = 0;
 }

 /*
@@ -347,39 +385,41 @@
  * manner we move the fair clock back by a proportional
  * amount of the new wait_runtime this task adds to the pool.
  */
-static void distribute_fair_add(struct rq *rq, s64 delta)
+static void distribute_fair_add(struct lrq *lrq, s64 delta)
```

```
 {
- struct task_struct *curr = rq->curr;
+ struct sched_entity *curr = lrq_curr(lrq);
  s64 delta_fair = 0;

  if (!(sysctl_sched_features & 2))
   return;

- if (rq->nr_running) {
-  delta_fair = div64_s(delta, rq->nr_running);
+ if (lrq_nr_running(lrq)) {
+  delta_fair = div64_s(delta, lrq_nr_running(lrq));
   /*
    * The currently running task's next wait_runtime value does
    * not depend on the fair_clock, so fix it up explicitly:
    */
-   if (curr->sched_class == &fair_sched_class)
-   add_wait_runtime(rq, curr, -delta_fair);
+   if (curr)
+   add_wait_runtime(lrq, curr, -delta_fair);
  }
- rq->lrq.fair_clock -= delta_fair;
+ lrq->fair_clock -= delta_fair;
 }

 /**************************************************************/
 /* 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 = rq->lrq.raw_weighted_load;
+ unsigned long load = lrq->raw_weighted_load;
  s64 delta_fair, prev_runtime;
+ struct task_struct *tsk = entity_to_task(p);

- if (p->policy == SCHED_BATCH || !(sysctl_sched_features & 4))
+ if ((entity_is_task(p) && tsk->policy == SCHED_BATCH) ||
+       !(sysctl_sched_features & 4))
   goto out;

- delta_fair = rq->lrq.fair_clock - p->se.sleep_start_fair;
+ delta_fair = lrq->fair_clock - p->sleep_start_fair;

  /*
   * Fix up delta_fair with the effect of us running
@@ -387,69 +427,206 @@
```

```
   */
  if (!(sysctl_sched_features & 32))
   delta_fair = div64_s(delta_fair * load,
-       load + p->se.load_weight);
- delta_fair = div64_s(delta_fair * p->se.load_weight, NICE_0_LOAD);
+       load + p->load_weight);
+ delta_fair = div64_s(delta_fair * p->load_weight, NICE_0_LOAD);

- prev_runtime = p->se.wait_runtime;
- __add_wait_runtime(rq, p, delta_fair);
- delta_fair = p->se.wait_runtime - prev_runtime;
+ prev_runtime = p->wait_runtime;
+ __add_wait_runtime(lrq, p, delta_fair);
+ delta_fair = p->wait_runtime - prev_runtime;

  /*
   * We move the fair clock back by a load-proportional
   * amount of the new wait_runtime this task adds to
   * the 'pool':
   */
- distribute_fair_add(rq, delta_fair);
+ distribute_fair_add(lrq, 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;
 }

-/*
- * 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)
+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) {
+   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) {
+   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);
+}
+
+static void
+dequeue_entity(struct lrq *lrq, struct sched_entity *p, int sleep, u64 now)
+{
```

```
+ update_stats_dequeue(lrq, p, now);
+ if (sleep) {
+  if (entity_is_task(p)) {
+   struct task_struct *tsk = entity_to_task(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(curr, granularity))
+  resched_task(lrq_rq(lrq)->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);
+
```

```
+ return p;
+}
+
+static void put_prev_entity(struct lrq *lrq, struct sched_entity *prev, u64 now)
+{
+ /*
+  * If the task is still waiting for the CPU (it just got
+  * preempted), update its position within the tree and
+  * start the wait period:
+  */
+ if ((sysctl_sched_features & 16) && entity_is_task(prev))  {
+  struct task_struct *prevtask = entity_to_task(prev);
+
+  if (prev->on_rq &&
+   test_tsk_thread_flag(prevtask, TIF_NEED_RESCHED)) {
+
+   dequeue_entity(lrq, prev, 0, now);
+   prev->on_rq = 0;
+   enqueue_entity(lrq, prev, 0, now);
+   prev->on_rq = 1;
+  } else
+   update_curr(lrq, now);
+ } else {
+  update_curr(lrq, now);
+ }
+
+ update_stats_curr_end(lrq, prev, now);
+
+ if (prev->on_rq)
+  update_stats_wait_start(lrq, prev, now);
+}
+
+static void entity_tick(struct lrq *lrq, struct sched_entity *curr)
+{
+ struct sched_entity *next;
+ struct rq *rq = lrq_rq(lrq);
+ u64 now = __rq_clock(rq);
+
+ /*
+  * Dequeue and enqueue the task to update its
+  * position within the tree:
+  */
+ dequeue_entity(lrq, curr, 0, now);
+ curr->on_rq = 0;
+ enqueue_entity(lrq, curr, 0, now);
+ curr->on_rq = 1;
+
+ /*
```

```
+  * 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 *curtask = entity_to_task(curr),
+       *nexttask = entity_to_task(next);
+
+  if ((curtask == rq->idle) || (rt_prio(nexttask->prio) &&
+     (nexttask->prio < curtask->prio))) {
+   resched_task(curtask);
+   return;
+  }
+ }
+ __check_preempt_curr_fair(lrq, next, curr, sysctl_sched_granularity);
+}
+
+
+/**************************************************************************/
+/*                      BEGIN : CFS operations on tasks                   */
+/**************************************************************************/
+
+static inline struct lrq *task_lrq(struct task_struct *p)
+{
+ return &task_rq(p)->lrq;
+}
+
+/*
+ * 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_lrq(p);
+ struct sched_entity *se = &p->se;
+
+ enqueue_entity(lrq, se, wakeup, now);
 }

 /*
@@ -460,16 +637,10 @@
 static void
 dequeue_task_fair(struct rq *rq, struct task_struct *p, int sleep, u64 now)
 {
```

```
- update_stats_dequeue(rq, 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;
- }
- __dequeue_task_fair(rq, p);
+ struct lrq *lrq = task_lrq(p);
+ struct sched_entity *se = &p->se;
+
+ dequeue_entity(lrq, se, sleep, now);
 }

 /*
@@ -482,16 +653,18 @@
 {
  struct task_struct *p_next;
  u64 now;
+ struct lrq *lrq = task_lrq(p);
+ struct sched_entity *se = &p->se;

  now = __rq_clock(rq);
  /*
   * 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);
+ se->on_rq = 0;
+ enqueue_entity(lrq, se, 0, now);
+ se->on_rq = 1;

  /*
   * yield-to support: if we are on the same runqueue then
@@ -503,35 +676,19 @@

   s64 delta = p->se.wait_runtime >> 1;

-  __add_wait_runtime(rq, p_to, delta);
-  __add_wait_runtime(rq, p, -delta);
+  __add_wait_runtime(lrq, &p_to->se, delta);
+  __add_wait_runtime(lrq, &p->se, -delta);
```

```
 }

 /*
  * Reschedule if another task tops the current one.
  */
- p_next = __pick_next_task_fair(rq);
+ se = __pick_next_entity(lrq);
+ p_next = entity_to_task(se);
 if (p_next != p)
  resched_task(p);
}

-/*
- * 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:
@@ -539,12 +696,13 @@
 static void check_preempt_curr_fair(struct rq *rq, struct task_struct *p)
 {
  struct task_struct *curr = rq->curr;
+ struct lrq *lrq = task_lrq(curr);
  unsigned long granularity;

  if ((curr == rq->idle) || rt_prio(p->prio)) {
   if (sysctl_sched_features & 8) {
    if (rt_prio(p->prio))
-    update_curr(rq, rq_clock(rq));
+    update_curr(lrq, rq_clock(rq));
   }
   resched_task(curr);
  } else {
@@ -555,25 +713,18 @@
```

```
   if (unlikely(p->policy == SCHED_BATCH))
    granularity = sysctl_sched_batch_wakeup_granularity;

-  __check_preempt_curr_fair(rq, p, curr, granularity);
+  __check_preempt_curr_fair(lrq, &p->se, &curr->se, granularity);
  }
 }

 static struct task_struct * pick_next_task_fair(struct rq *rq, u64 now)
 {
- struct task_struct *p = __pick_next_task_fair(rq);
+ struct lrq *lrq = &rq->lrq;
+ struct sched_entity *se;

- /*
-  * 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);
+ se = pick_next_entity(lrq, now);

- return p;
+ return entity_to_task(se);
 }

 /*
@@ -581,32 +732,13 @@
  */
 static void put_prev_task_fair(struct rq *rq, struct task_struct *prev, u64 now)
 {
+ struct lrq *lrq = task_lrq(prev);
+ struct sched_entity *se = &prev->se;
+
  if (prev == rq->idle)
   return;

- /*
-  * If the task is still waiting for the CPU (it just got
-  * preempted), update its position within the tree and
-  * start the wait period:
-  */
- if (sysctl_sched_features & 16)  {
-  if (prev->se.on_rq &&
-   test_tsk_thread_flag(prev, TIF_NEED_RESCHED)) {
```

```
-
-   dequeue_task_fair(rq, prev, 0, now);
-   prev->se.on_rq = 0;
-   enqueue_task_fair(rq, prev, 0, now);
-   prev->se.on_rq = 1;
-  } else
-   update_curr(rq, now);
- } else {
-  update_curr(rq, now);
- }
-
- update_stats_curr_end(rq, prev, now);
-
- if (prev->se.on_rq)
-  update_stats_wait_start(rq, prev, now);
+ put_prev_entity(lrq, se, now);
 }

 /**************************************************************/
@@ -636,7 +768,7 @@

 static struct task_struct * load_balance_start_fair(struct rq *rq)
 {
- return __load_balance_iterator(rq, first_fair(rq));
+ return __load_balance_iterator(rq, first_fair(&rq->lrq));
 }

 static struct task_struct * load_balance_next_fair(struct rq *rq)
@@ -649,31 +781,10 @@
  */
 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;
+ struct lrq *lrq = task_lrq(curr);
+ struct sched_entity *se = &curr->se;

- 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);
+ entity_tick(lrq, se);
 }

 /*
@@ -685,14 +796,17 @@
  */
 static void task_new_fair(struct rq *rq, struct task_struct *p)
 {
+ struct lrq *lrq = task_lrq(p);
+ struct sched_entity *se = &p->se;
+
  sched_info_queued(p);
- update_stats_enqueue(rq, p, rq_clock(rq));
+ 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(rq->curr,
+ p->se.fair_key = current->se.fair_key - niced_granularity(&rq->curr->se,
       sysctl_sched_granularity) - 1;
  /*
   * The first wait is dominated by the child-runs-first logic,
@@ -706,7 +820,7 @@
   */
 // p->se.wait_runtime = -(s64)(sysctl_sched_granularity / 2);

- __enqueue_task_fair(rq, p);
+ __enqueue_entity(lrq, se);
  p->se.on_rq = 1;
  inc_nr_running(p, rq);
 }
Index: current/kernel/sched_debug.c
===================================================================
--- current.orig/kernel/sched_debug.c 2007-06-09 15:07:16.000000000 +0530
+++ current/kernel/sched_debug.c 2007-06-09 15:07:33.000000000 +0530
```

```
@@ -87,7 +87,7 @@
  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;
--
```

Regards,
vatsa