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
```

_____

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