

---

Subject: [RFC][PATCH 0/6] Add group fairness to CFS - v1  
Posted by [Srivatsa Vaddagiri](#) on Mon, 11 Jun 2007 15:47:24 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Ingo,

Here's an update of the group fairness patch I have been working on. Its against CFS v16 (sched-cfs-v2.6.22-rc4-mm2-v16.patch).

The core idea is to reuse much of CFS logic to apply fairness at higher hierarchical levels (user, container etc). In this regard CFS engine has been modified to deal with generic 'schedulable entities'. The patches introduce two essential structures in CFS core:

- struct sched\_entity
  - represents a schedulable entity in a hierarchy. Task is the lowest element in this hierarchy. Its ancestors could be user, container etc. This structure stores essential attributes/execution-history (wait\_runtime etc) which is required by CFS engine to provide fairness between 'struct sched\_entities' at the same hierarchy.
- struct lrq
  - represents (per-cpu) runqueue in which ready-to-run 'struct sched\_entities' are queued. The fair clock calculation is split to be per 'struct lrq'.

Here's a brief description of the patches to follow:

Patches 1-3 introduce the essential changes in CFS core to support this concept. They rework existing code w/o any (intended!) change in functionality.

Patch 4 fixes some bad interaction between SCHED\_RT and SCHED\_NORMAL tasks in current CFS.

Patch 5 introduces basic changes in CFS core to support group fairness.

Patch 6 hooks up scheduler with container patches in mm (as an interface for task-grouping functionality).

Changes since last version:

- Preliminary SMP support included (based on the idea outlined at <http://lkml.org/lkml/2007/5/25/146>)
- Task grouping to which fairness is applied is based on Paul Menage's container patches included in -mm tree. Usage of this feature is described in Patch 6/6
- Fix some real time and SCHED\_NORMAL interactions (maintain

separate nr\_running/raw\_weighted counters for SCHED\_NORMAL tasks)

- Support arbitrary levels of hierarchy. Previous version supported only 2 levels. Current version makes no assumption on the number of levels supported.

#### TODO:

- Weighted fair-share support

Currently each group gets "equal" share. Support weighted fair-share so that some groups deemed important get more than this "equal" share. I believe it is possible to use load\_weight to achieve this goal (similar to how niced tasks use it to get differential bandwidth)

- Separate out tunables

Right now tunable are same for all layers of scheduling. I strongly think we will need to separate them, esp sysctl\_sched\_runtime\_limit.

- Flattening hierarchy

This may be useful if we want to avoid cost of deep hierarchical scheduling in core scheduler, but at the same time want deeper hierarchical levels to be supported from user pov. William Lee Irwin has suggested basic technique at <http://lkml.org/lkml/2007/5/26/81> which I need to experiment with. With this technique, for ex, it is possible to have core scheduler support two levels (container, task) but use weight adjustment to support more levels from user pov (container, user, process, task).

- (SMP optimization) during load balance, pick cache-cold tasks first to migrate

- (optimization) reduce frequency of timer tick processing at higher levels (similar to how load balancing frequency varies across scheduling domains).

The patches have been very stable in my tests. There is however one oops I hit just before sending this (!). I think I know the reason for that (some cleanup required in RT<->NORMAL switch) and am currently investigating that.

I am sending the patches largely to get feedback on the direction this is heading.

Some results of the patches below.

Legends used in the results :-

cfs = base cfs performance (sched-cfs-v2.6.22-rc4-mm2-v16.patch)  
cfsc = base cfs + patches 1-3 applied (core changes to cfs core)  
cfscrt = base cfs + patches 1-4 applied (fix RT/NORMAL interactions)  
cfsgpch = base cfs + patches 1-5 applied (group changes applied)  
cfsgpchdi = base cfs + all patches applied (CONFIG\_FAIR\_GROUP\_SCHED disabled)  
cfsgpch = base cfs + all patches applied (CONFIG\_FAIR\_GROUP\_SCHED enabled)

## 1. lat\_ctx (from lmbench):

=====

Context switching - times in microseconds - smaller is better

```
-----  
Host      OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K  
           ctxsw ctxsw  ctxsw  ctxsw  ctxsw  ctxsw  ctxsw  
-----
```

```
cfs      Linux 2.6.22- 6.2060 7.1200 7.7746 7.6880 11.27 8.61400 20.68  
cfsc     Linux 2.6.22- 6.3920 6.9800 7.9320 8.5420 12.1  9.64000 20.46  
cfscrt   Linux 2.6.22- 6.5280 7.1600 7.7640 7.9340 11.35 9.34000 20.34  
cfsgpch  Linux 2.6.22- 6.9400 7.3080 8.0620 8.5660 12.24 9.29200 21.04  
cfsgpchdi Linux 2.6.22- 6.7966 7.4033 8.1833 8.8166 11.76 9.53667 20.33  
cfsgpch  Linux 2.6.22- 7.3366 7.7666 7.9   8.8766 12.06 9.31337 21.03
```

Performance of CFS with all patches applied (but with CONFIG\_FAIR\_GROUP\_SCHED disabled) [cfsgpchdi above] seems to be very close to base cfs performance [cfs above] (delta within tolerable noise level limits?)

## 2. hackbench

=====

hackbench -pipe 10:

```
cfs      0.787  
cfsc     0.7547  
cfscrt   0.9014  
cfsgpch  0.8691  
cfsgpchdi 0.7864  
cfsgpch  0.9229
```

hackbench -pipe 100:

```
cfs          3.726
cfsccl       3.7216
cfscclrt     3.8151
cfsgpchl     3.6107
cfsgpchl     3.8468
cfsgpchl     4.2332
```

3. Fairness result between users 'vatsa' and 'guest':

The two groups were created as below in container filesystem:

```
# mkdir /dev/cpuctl
# mount -t container -o cpuctl none /dev/cpuctl
# cd /dev/cpuctl
# mkdir vatsa
# mkdir guest

# echo vatsa_shell_pid > vatsa/tasks
# echo guest_shell_pid > guest/tasks

# # Start tests now in the two user's shells
```

hackbench -pipe 10:

```
vatsa : 1.0186
guest : 1.0449
```

hackbench -pipe 100:

```
vatsa : 6.9512
guest : 7.5668
```

Note: I have noticed that running lat\_ctx in a loop for 10 times doesn't give me good results. Basically I expected the loop to take same time for both users (when run simultaneously), whereas it was taking different times for different users. I think this can be solved by increasing sysctl\_sched\_runtime\_limit at group level (to remember execution history over a longer period).

--

Regards,  
vatsa

---

Containers mailing list  
Containers@lists.linux-foundation.org

---

Subject: [RFC][PATCH 1/6] Introduce struct sched\_entity and struct irq

Posted by [Srivatsa Vaddagiri](#) on Mon, 11 Jun 2007 15:50:46 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

This patch introduces two new structures:

struct sched\_entity

stores essential attributes/execution-history used by CFS core  
to drive fairness between 'schedulable entities' (tasks, users etc)

struct irq

runqueue used to hold ready-to-run entities

These new structures are formed by grouping together existing fields in existing structures (task\_struct and rq) and hence represents rework with zero functionality change.

Signed-off-by : Srivatsa Vaddagiri <[vatsa@linux.vnet.ibm.com](mailto:vatsa@linux.vnet.ibm.com)>

```
---
fs/proc/array.c      |   4
include/linux/sched.h |  43 ++++++---
kernel/exit.c        |   2
kernel/posix-cpu-timers.c |  16 +--
kernel/sched.c        | 186 ++++++-----
kernel/sched_debug.c  |  86 ++++++---
kernel/sched_fair.c   | 211 ++++++-----
kernel/sched_rt.c     |  14 +--
8 files changed, 289 insertions(+), 273 deletions(-)
```

Index: current/include/linux/sched.h

```
=====
--- current.orig/include/linux/sched.h 2007-06-09 15:01:39.000000000 +0530
+++ current/include/linux/sched.h 2007-06-09 15:04:54.000000000 +0530
@@ -872,6 +872,29 @@
 void (*task_new) (struct rq *rq, struct task_struct *p);
};

+/* CFS stats for a schedulable entity (task, task-group etc) */
+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;
@@ -886,33 +909,15 @@
    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;
-
- 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: current/fs/proc/array.c
=====

```

```

--- current.orig/fs/proc/array.c 2007-06-09 15:01:39.000000000 +0530
+++ current/fs/proc/array.c 2007-06-09 15:04:54.000000000 +0530
@@ -329,7 +329,7 @@
 * Use CFS's precise accounting, if available:
 */
if (!(sysctl_sched_features & 128)) {
- u64 temp = (u64)nsec_to_clock_t(p->sum_exec_runtime);
+ u64 temp = (u64)nsec_to_clock_t(p->se.sum_exec_runtime);

if (total) {
temp *= utime;
@@ -351,7 +351,7 @@
 * by userspace grows monotonically - apps rely on that):
 */
if (!(sysctl_sched_features & 128))
- stime = nsec_to_clock_t(p->sum_exec_runtime) - task_utime(p);
+ stime = nsec_to_clock_t(p->se.sum_exec_runtime) - task_utime(p);

return stime;
}
Index: current/kernel/exit.c
=====
--- current.orig/kernel/exit.c 2007-06-09 14:56:50.000000000 +0530
+++ current/kernel/exit.c 2007-06-09 15:04:54.000000000 +0530
@@ -126,7 +126,7 @@
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: current/kernel/posix-cpu-timers.c
=====
--- current.orig/kernel/posix-cpu-timers.c 2007-06-09 15:01:39.000000000 +0530
+++ current/kernel/posix-cpu-timers.c 2007-06-09 15:04:54.000000000 +0530
@@ -249,7 +249,7 @@
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 +467,7 @@
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 +475,7 @@
    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 +536,7 @@
    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 +1004,7 @@
        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 +1049,7 @@
    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 +1208,7 @@
  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 +1300,7 @@

```

```

  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: current/kernel/sched.c

```

=====
--- current.orig/kernel/sched.c 2007-06-09 15:01:39.000000000 +0530
+++ current/kernel/sched.c 2007-06-09 15:07:17.000000000 +0530
@@ -116,6 +116,23 @@
  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, delta_fair_clock;
+ u64 exec_clock, delta_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.
*
@@ -131,16 +148,13 @@
* 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 Irq Irq;

    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
@@ -156,10 +170,6 @@

    u64 clock, prev_clock_raw;
    s64 clock_max_delta;
- u64 fair_clock, delta_fair_clock;
- u64 exec_clock, delta_exec_clock;
- s64 wait_runtime;
- unsigned long wait_runtime_overruns, wait_runtime_underruns;

    unsigned int clock_warps, clock_overflows;
    unsigned int clock_unstable_events;
@@ -170,10 +180,6 @@
    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
@@ -583,13 +589,13 @@
static inline void
inc_raw_weighted_load(struct rq *rq, const struct task_struct *p)
{
- rq->raw_weighted_load += p->load_weight;
+ rq->Irq.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)
@@ -615,22 +621,22 @@

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_IDLEPRIO tasks get minimal weight:
*/
if (p->policy == SCHED_IDLEPRIO) {
- 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)
@@ -639,7 +645,7 @@

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)
@@ -647,7 +653,7 @@

```

```

u64 now = rq_clock(rq);

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

/*
@@ -735,7 +741,7 @@
/* 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
@@ -752,18 +758,18 @@
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;

@@ -791,7 +797,7 @@

```

```

    * 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;
}
@@ -822,7 +828,7 @@
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);
@@ -870,9 +876,9 @@
    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);
}

/*
@@ -884,9 +890,9 @@
    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);
}

/*
@@ -897,7 +903,7 @@
    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;
}

```

```

/*
@@ -1128,7 +1134,7 @@
    if (!(old_state & state))
        goto out;

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

    cpu = task_cpu(p);
@@ -1183,11 +1189,11 @@
    * 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) {
+   100*(tl + p->se.load_weight) <= imbalance*load) {
/*
    * This domain has SD_WAKE_AFFINE and
    * p is cache cold in this domain, and
@@ -1221,7 +1227,7 @@
    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();
@@ -1285,18 +1291,18 @@
    */
    static void __sched_fork(struct task_struct *p)
    {
- p->wait_start_fair = p->wait_start = p->exec_start = 0;
- p->sum_exec_runtime = 0;
+ p->se.wait_start_fair = p->se.wait_start = p->se.exec_start = 0;
+ p->se.sum_exec_runtime = 0;

- p->wait_runtime = 0;
+ p->se.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.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;

```

```

/*
@@ -1367,7 +1373,7 @@
  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 {
/*
@@ -1382,7 +1388,7 @@

```

```

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

```

```

/**
@@ -1592,17 +1598,17 @@
  u64 fair_delta64, exec_delta64, tmp64;
  unsigned int i, scale;

```

```

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

- fair_delta64 = this_rq->delta_fair_clock + 1;
- this_rq->delta_fair_clock = 0;
+ fair_delta64 = this_rq->lrq.delta_fair_clock + 1;
+ this_rq->lrq.delta_fair_clock = 0;

```

```

- exec_delta64 = this_rq->delta_exec_clock + 1;
- this_rq->delta_exec_clock = 0;
+ exec_delta64 = this_rq->lrq.delta_exec_clock + 1;
+ this_rq->lrq.delta_exec_clock = 0;

    if (fair_delta64 > (u64)LONG_MAX)
        fair_delta64 = (u64)LONG_MAX;
@@ -1627,10 +1633,10 @@

    /* 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;
}
}

@@ -1886,7 +1892,8 @@
    * 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 ||
@@ -1899,7 +1906,7 @@

    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
    @@ -1996,7 +2003,7 @@

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

/*

```



@@ -2230,11 +2237,12 @@

```
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;
  }
}
```

@@ -2838,9 +2846,9 @@

```
struct rq *rq;
```

```
rq = task_rq_lock(p, &flags);
- ns = p->sum_exec_runtime;
+ ns = p->se.sum_exec_runtime;
  if (rq->curr == p) {
-   delta_exec = rq_clock(rq) - p->exec_start;
+   delta_exec = rq_clock(rq) - p->se.exec_start;
    if ((s64)delta_exec > 0)
      ns += delta_exec;
  }
```

@@ -3538,7 +3546,7 @@

```
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);
```

@@ -3591,7 +3599,7 @@

```
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);
```

@@ -3728,7 +3736,7 @@

```
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) {
@@ -3836,7 +3844,7 @@
        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;
@@ -4490,7 +4498,7 @@
    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);
@@ -4633,7 +4641,7 @@
    goto out;

    set_task_cpu(p, dest_cpu);
- if (p->on_rq) {
+ if (p->se.on_rq) {
        deactivate_task(rq_src, p, 0);
        activate_task(rq_dest, p, 0);
        check_preempt_curr(rq_dest, p);
@@ -6100,11 +6108,11 @@
        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;
@@ -6187,15 +6195,15 @@
    read_lock_irq(&tasklist_lock);

```

```

    do_each_thread(g, 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)) {
@@ -6218,7 +6226,7 @@
        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: current/kernel/sched\_debug.c

```

=====
--- current.orig/kernel/sched_debug.c 2007-06-09 15:01:39.000000000 +0530

```

```

+++ current/kernel/sched_debug.c 2007-06-09 15:07:16.000000000 +0530

```

```

@@ -40,16 +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)
@@ -70,7 +70,7 @@
  read_lock_irq(&tasklist_lock);

  do_each_thread(g, p) {
- if (!p->on_rq)
+ if (!p->se.on_rq)
    continue;

    print_task(m, rq, p, now);
@@ -89,8 +89,8 @@
    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);
  }
@@ -109,9 +109,9 @@
  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);

```

```

@@ -122,18 +122,18 @@
P(clock_overflows);
P(clock_unstable_events);
P(clock_max_delta);
- P(fair_clock);
- P(delta_fair_clock);
- P(exec_clock);
- P(delta_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.delta_fair_clock);
+ P(lrq.exec_clock);
+ P(lrq.delta_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);

```

```

@@ -205,21 +205,21 @@
#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(wait_runtime);
- P(wait_runtime_overruns);
- P(wait_runtime_underruns);

```

```

- P(sum_exec_runtime);
- P(load_weight);
+ 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.wait_runtime);
+ P(se.wait_runtime_overruns);
+ P(se.wait_runtime_underruns);
+ P(se.sum_exec_runtime);
+ P(se.load_weight);
  P(policy);
  P(prio);
#undef P
@@ -235,7 +235,7 @@

```

```

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: current/kernel/sched\_fair.c

```

=====
--- current.orig/kernel/sched_fair.c 2007-06-09 15:01:39.000000000 +0530
+++ current/kernel/sched_fair.c 2007-06-09 15:07:16.000000000 +0530
@@ -51,10 +51,10 @@

```

```

  */
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->lq.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;

  /*

```

```

@@ -62,12 +62,12 @@
 */
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;
@@ -80,31 +80,31 @@
    * 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);
}

/*****/
@@ -121,13 +121,13 @@
/*
 * 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);
}

static void limit_wait_runtime(struct rq *rq, struct task_struct *p)
@@ -138,30 +138,30 @@
 * Niced tasks have the same history dynamic range as
 * non-niced tasks:
 */
- if (p->wait_runtime > limit) {
- p->wait_runtime = limit;
- p->wait_runtime_overruns++;
- rq->wait_runtime_overruns++;
- }
- 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_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++;

```



```

}
}

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;
}

static s64 div64_s(s64 dividend, unsigned long divisor)
@@ -185,7 +185,7 @@
*/
static inline void update_curr(struct rq *rq, u64 now)
{
- unsigned long load = rq->raw_weighted_load;
+ unsigned long load = rq->lrq.raw_weighted_load;
  u64 delta_exec, delta_fair, delta_mine;
  struct task_struct *curr = rq->curr;

@@ -195,23 +195,23 @@
  * 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;
+ delta_exec = now - curr->se.exec_start;
  if (unlikely((s64)delta_exec < 0))
    delta_exec = 0;
- if (unlikely(delta_exec > curr->exec_max))
+ if (unlikely(delta_exec > curr->se.exec_max))
- curr->exec_max = delta_exec;
+ curr->se.exec_max = delta_exec;

- 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;

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

/* Load-balancing accounting. */
- rq->delta_fair_clock += delta_fair;
- rq->delta_exec_clock += delta_exec;
+ rq->lrq.delta_fair_clock += delta_fair;
+ rq->lrq.delta_exec_clock += delta_exec;

/*
 * Task already marked for preemption, do not burden
@@ -221,11 +221,11 @@
if (unlikely(test_tsk_thread_flag(curr, TIF_NEED_RESCHED)))
return;

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

- 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
@@ -239,8 +239,8 @@
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;
}

/*
@@ -260,21 +260,23 @@
/*
 * 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 {
- if (p->wait_runtime < 0)
- key -= div64_s(p->wait_runtime * NICE_0_LOAD, p->load_weight);
+ if (p->se.wait_runtime < 0)
+ key -= div64_s(p->se.wait_runtime * NICE_0_LOAD,
+ p->se.load_weight);
else
- key -= div64_s(p->wait_runtime * p->load_weight, NICE_0_LOAD);
+ key -= div64_s(p->se.wait_runtime * p->se.load_weight,
+ NICE_0_LOAD);
}

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

/*
@@ -285,21 +287,21 @@
{
s64 delta_fair, delta_wait;

- delta_wait = now - p->wait_start;
- if (unlikely(delta_wait > p->wait_max))
- p->wait_max = 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->wait_start_fair) {
- delta_fair = rq->fair_clock - p->wait_start_fair;
+ if (p->se.wait_start_fair) {
+ delta_fair = rq->lrq.fair_clock - p->se.wait_start_fair;

- if (unlikely(p->load_weight != NICE_0_LOAD))
- delta_fair = div64_s(delta_fair * p->load_weight,
+ if (unlikely(p->se.load_weight != NICE_0_LOAD))
+ delta_fair = div64_s(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
@@ -323,7 +325,7 @@
/*
 * We are starting a new run period:
 */
- p->exec_start = now;
+ p->se.exec_start = now;
}

/*
@@ -332,7 +334,7 @@
static inline void
update_stats_curr_end(struct rq *rq, struct task_struct *p, u64 now)
{
- p->exec_start = 0;
+ p->se.exec_start = 0;
}

/*
@@ -362,7 +364,7 @@
if (curr->sched_class == &fair_sched_class)
add_wait_runtime(rq, curr, -delta_fair);
}
- rq->fair_clock -= delta_fair;
+ rq->lrq.fair_clock -= delta_fair;
}

/*****
@@ -371,25 +373,26 @@

static void enqueue_sleeper(struct rq *rq, struct task_struct *p)
{
- unsigned long load = rq->raw_weighted_load;
+ unsigned long load = rq->lrq.raw_weighted_load;
s64 delta_fair, prev_runtime;

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

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

/*

```

```

* Fix up delta_fair with the effect of us running
* during the whole sleep period:
*/
if (!(sysctl_sched_features & 32))
- delta_fair = div64_s(delta_fair * load, load + p->load_weight);
- delta_fair = div64_s(delta_fair * p->load_weight, NICE_0_LOAD);
+ 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);

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

/*
* We move the fair clock back by a load-proportional
@@ -399,9 +402,9 @@
  distribute_fair_add(rq, 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;
}

/*
@@ -420,29 +423,29 @@
  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);
@@ -460,11 +463,11 @@
    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);
}
@@ -486,9 +489,9 @@
    * 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;

/*
 * yield-to support: if we are on the same runqueue then
@@ -496,9 +499,9 @@
 */
if (p_to && rq == task_rq(p_to) &&
    p_to->sched_class == &fair_sched_class
- && p->wait_runtime > 0) {
+ && p->se.wait_runtime > 0) {

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

    __add_wait_runtime(rq, p_to, delta);
    __add_wait_runtime(rq, p, -delta);
@@ -519,7 +522,7 @@
    __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
@@ -587,13 +590,13 @@
 * start the wait period:
 */
if (sysctl_sched_features & 16) {
- if (prev->on_rq &&
+ if (prev->se.on_rq &&
    test_tsk_thread_flag(prev, TIF_NEED_RESCHED)) {

    dequeue_task_fair(rq, prev, 0, now);
- prev->on_rq = 0;
+ prev->se.on_rq = 0;
    enqueue_task_fair(rq, prev, 0, now);
- prev->on_rq = 1;
+ prev->se.on_rq = 1;
    } else
        update_curr(rq, now);
    } else {
@@ -602,7 +605,7 @@

    update_stats_curr_end(rq, prev, now);

- if (prev->on_rq)
+ if (prev->se.on_rq)

```

```

    update_stats_wait_start(rq, prev, now);
}

@@ -625,8 +628,8 @@
    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;
}
@@ -638,7 +641,7 @@

static struct task_struct * load_balance_next_fair(struct rq *rq)
{
- return __load_balance_iterator(rq, rq->rb_load_balance_curr);
+ return __load_balance_iterator(rq, rq->lrq.rb_load_balance_curr);
}

/*
@@ -654,9 +657,9 @@
    * 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.
@@ -689,22 +692,22 @@
    * 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;

```



```

/*
 * The statistical average of wait_runtime is about
 * -granularity/2, so initialize the task with that:
 */
-// p->wait_runtime = -(s64)(sysctl_sched_granularity / 2);
+// p->se.wait_runtime = -(s64)(sysctl_sched_granularity / 2);

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

```

Index: current/kernel/sched\_rt.c

```

=====
--- current.orig/kernel/sched_rt.c 2007-06-09 15:01:39.000000000 +0530
+++ current/kernel/sched_rt.c 2007-06-09 15:04:54.000000000 +0530
@@ -15,14 +15,14 @@
  if (!has_rt_policy(curr))
    return;

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

- curr->sum_exec_runtime += delta_exec;
- curr->exec_start = now;
+ curr->se.sum_exec_runtime += delta_exec;
+ curr->se.exec_start = now;
}

static void
@@ -89,7 +89,7 @@
  queue = array->queue + idx;
  next = list_entry(queue->next, struct task_struct, run_list);

- next->exec_start = now;
+ next->se.exec_start = now;

  return next;
}
@@ -97,7 +97,7 @@

```

```
static void put_prev_task_rt(struct rq *rq, struct task_struct *p, u64 now)
{
    update_curr_rt(rq, now);
- p->exec_start = 0;
+ p->se.exec_start = 0;
}

/*
```

--  
Regards,  
vatsa

---

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

---



---

Subject: [RFC][PATCH 2/6] task's cpu information needs to be always correct  
Posted by [Srivatsa Vaddagiri](#) on Mon, 11 Jun 2007 15:52:12 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 runqueue 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@linux.vnet.ibm.com](mailto:vatsa@linux.vnet.ibm.com)>

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

Index: current/kernel/sched.c

```
=====
--- current.orig/kernel/sched.c 2007-06-09 15:07:17.000000000 +0530
+++ current/kernel/sched.c 2007-06-09 15:07:32.000000000 +0530
@@ -4624,7 +4624,7 @@
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;
@@ -4640,9 +4640,11 @@
if (!cpu_isset(dest_cpu, p->cpus_allowed))
    goto out;

```

```

- set_task_cpu(p, dest_cpu);
- if (p->se.on_rq) {
+ on_rq = p->se.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

---

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

---



---

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](mailto: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 dividend, 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;
+ struct rq *rq = lrq->rq;
+ 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 dequeuing 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);
+

```





```

+ /*****
+
+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->lq));
}

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 lq *lq = task_lq(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(lq, se);
}

/*
@@ -685,14 +796,17 @@
    */
static void task_new_fair(struct rq *rq, struct task_struct *p)
{
+ struct lq *lq = task_lq(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

---

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

---

Subject: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED\_RT and SCHED\_NORMAL tasks  
Posted by [Srivatsa Vaddagiri](#) on Mon, 11 Jun 2007 15:55:04 GMT

Currently nr\_running and raw\_weighted\_load fields in runqueue affect some CFS calculations (like distribute\_fair\_add, enqueue\_sleeper etc).

These fields however are shared between tasks of all classes, which can potentially affect those calculations for SCHED\_NORMAL tasks. However I do not know of any bad behaviour caused by not splitting these fields (like this patch does).

This split is nevertheless needed for subsequent patches.

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

---

```
kernel/sched.c      | 134 ++++++-----
kernel/sched_fair.c |  65 ++++++-----
2 files changed, 128 insertions(+), 71 deletions(-)
```

Index: current/kernel/sched.c

=====

--- current.orig/kernel/sched.c 2007-06-09 15:07:32.000000000 +0530

+++ current/kernel/sched.c 2007-06-09 15:07:36.000000000 +0530

@ @ -118,6 +118,7 @ @

/\* CFS-related fields in a runqueue \*/

```
struct lrq {
+ long nr_running;
  unsigned long raw_weighted_load;
  #define CPU_LOAD_IDX_MAX 5
  unsigned long cpu_load[CPU_LOAD_IDX_MAX];
@ @ -125,6 +126,7 @ @
```

```
  u64 fair_clock, delta_fair_clock;
  u64 exec_clock, delta_exec_clock;
+ u64 last_tick; /* when did we last smoothen cpu load? */
  s64 wait_runtime;
  unsigned long wait_runtime_overruns, wait_runtime_underruns;
```

@ @ -148,12 +150,18 @ @

```
  * remote CPUs use both these fields when doing load calculation.
  */
```

```
  long nr_running;
- struct lrq lrq;
+ unsigned long raw_weighted_load;
+ #ifdef CONFIG_SMP
+ #define CPU_LOAD_IDX_MAX 5
+ unsigned long cpu_load[CPU_LOAD_IDX_MAX];
```



```

    unsigned char idle_at_tick;
#ifdef CONFIG_NO_HZ
    unsigned char in_nohz_recently;
#endif
+ #endif
+ struct Irq Irq;
+
    u64 nr_switches;

/*
@@ -589,13 +597,13 @@
static inline void
inc_raw_weighted_load(struct rq *rq, const struct task_struct *p)
{
- rq->Irq.raw_weighted_load += p->se.load_weight;
+ rq->raw_weighted_load += p->se.load_weight;
}

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

static inline void inc_nr_running(struct task_struct *p, struct rq *rq)
@@ -741,7 +749,7 @@
/* Used instead of source_load when we know the type == 0 */
unsigned long weighted_cpuload(const int cpu)
{
- return cpu_rq(cpu)->Irq.raw_weighted_load;
+ return cpu_rq(cpu)->raw_weighted_load;
}

#ifdef CONFIG_SMP
@@ -876,9 +884,9 @@
    struct rq *rq = cpu_rq(cpu);

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

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

/*

```

```

@@ -890,9 +898,9 @@
    struct rq *rq = cpu_rq(cpu);

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

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

/*
@@ -903,7 +911,7 @@
    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 ? rq->raw_weighted_load / n : SCHED_LOAD_SCALE;
}

/*
@@ -1592,54 +1600,6 @@
    return running + uninterruptible;
}

-static void update_load_fair(struct rq *this_rq)
-{
-   unsigned long this_load, fair_delta, exec_delta, idle_delta;
-   u64 fair_delta64, exec_delta64, tmp64;
-   unsigned int i, scale;
-
-   this_rq->lrq.nr_load_updates++;
-   if (!(sysctl_sched_features & 64)) {
-       this_load = this_rq->lrq.raw_weighted_load;
-       goto do_avg;
-   }
-
-   fair_delta64 = this_rq->lrq.delta_fair_clock + 1;
-   this_rq->lrq.delta_fair_clock = 0;
-
-   exec_delta64 = this_rq->lrq.delta_exec_clock + 1;
-   this_rq->lrq.delta_exec_clock = 0;
-
-   if (fair_delta64 > (u64)LONG_MAX)
-       fair_delta64 = (u64)LONG_MAX;
-   fair_delta = (unsigned long)fair_delta64;
-
-   if (exec_delta64 > (u64)TICK_NSEC)

```

```

- exec_delta64 = (u64)TICK_NSEC;
- exec_delta = (unsigned long)exec_delta64;
-
- idle_delta = TICK_NSEC - exec_delta;
-
- tmp64 = SCHED_LOAD_SCALE * exec_delta64;
- do_div(tmp64, fair_delta);
- tmp64 *= exec_delta64;
- 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

/*
@@ -2003,7 +1963,7 @@

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

/*
@@ -2238,11 +2198,11 @@
    rq = cpu_rq(i);

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

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

```

```

+ max_load = rq->raw_weighted_load;
+ busiest = rq;
+ }
+ }
@@ -2576,6 +2536,32 @@
+ spin_unlock(&target_rq->lock);
+ }

+static void update_load(struct rq *this_rq)
+{
+ unsigned long this_load;
+ unsigned int i, scale;
+
+ this_load = this_rq->raw_weighted_load;
+
+ /* 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;
+ /*
+  * Round up the averaging division if load is increasing. This
+  * prevents us from getting stuck on 9 if the load is 10, for
+  * example.
+  */
+ if (new_load > old_load)
+ new_load += scale-1;
+ this_rq->cpu_load[i] = (old_load*(scale-1) + new_load) >> i;
+ }
+ }
+
+ #ifdef CONFIG_NO_HZ
+ static struct {
+ atomic_t load_balancer;
@@ -2822,14 +2808,14 @@
+ if (time_after_eq(jiffies, rq->next_balance))
+ raise_softirq(SCHED_SOFTIRQ);
+ }
+ #else
+ #else /* CONFIG_SMP */
+ /*
+  * on UP we do not need to balance between CPUs:
+  */
+ static inline void idle_balance(int cpu, struct rq *rq)
+ {

```

```

}
-#endif
+#endif /* CONFIG_SMP */

DEFINE_PER_CPU(struct kernel_stat, kstat);

@@ -2953,8 +2939,8 @@

    if (!idle_at_tick)
        task_running_tick(rq, p);
- update_load_fair(rq);
#ifdef CONFIG_SMP
+ update_load(rq);
    rq->idle_at_tick = idle_at_tick;
    trigger_load_balance(cpu);
#endif
@@ -6090,6 +6076,18 @@
    && addr < (unsigned long)__sched_text_end);
}

+static inline void init_lrq(struct lrq *lrq, struct rq *rq)
+{
+ int j;
+
+ lrq->tasks_timeline = RB_ROOT;
+ lrq->fair_clock = 1;
+ lrq->last_tick = rq_clock(rq);
+ lrq->nr_running = 0;
+ for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
+ lrq->cpu_load[j] = 0;
+}
+
void __init sched_init(void)
{
    int highest_cpu = 0;
@@ -6110,12 +6108,12 @@
    spin_lock_init(&rq->lock);
    lockdep_set_class(&rq->lock, &rq->rq_lock_key);
    rq->nr_running = 0;
- rq->lrq.tasks_timeline = RB_ROOT;
- rq->clock = rq->lrq.fair_clock = 1;
+ rq->clock = 1;
+ init_lrq(&rq->lrq, rq);

- for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
- rq->lrq.cpu_load[j] = 0;
#ifdef CONFIG_SMP
+ for (j = 0; j < CPU_LOAD_IDX_MAX; j++)

```

```

+ rq->cpu_load[j] = 0;
  rq->sd = NULL;
  rq->active_balance = 0;
  rq->push_cpu = 0;
Index: current/kernel/sched_fair.c
=====
--- current.orig/kernel/sched_fair.c 2007-06-09 15:07:33.000000000 +0530
+++ current/kernel/sched_fair.c 2007-06-09 15:07:36.000000000 +0530
@@ -64,9 +64,7 @@

static long lrq_nr_running(struct lrq *lrq)
{
- struct rq *rq = lrq_rq(lrq);
-
- return rq->nr_running;
+ return lrq->nr_running;
}

#define entity_is_task(se) 1
@@ -119,6 +117,8 @@

  rb_link_node(&p->run_node, parent, link);
  rb_insert_color(&p->run_node, &lrq->tasks_timeline);
+ lrq->raw_weighted_load += p->load_weight;
+ lrq->nr_running++;
}

static inline void __dequeue_entity(struct lrq *lrq, struct sched_entity *p)
@@ -126,6 +126,8 @@
  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;
+ lrq->nr_running--;
}

static inline struct rb_node * first_fair(struct lrq *lrq)
@@ -570,12 +572,69 @@
  update_stats_wait_start(lrq, prev, now);
}

+static void update_load_fair(struct lrq *this_lrq)
+{
+ unsigned long this_load, fair_delta, exec_delta, idle_delta;
+ u64 fair_delta64, exec_delta64, tmp64;
+ unsigned int i, scale;
+
+ this_lrq->nr_load_updates++;

```

```

+ if (!(sysctl_sched_features & 64)) {
+   this_load = this_lrq->raw_weighted_load;
+   goto do_avg;
+ }
+
+ fair_delta64 = this_lrq->delta_fair_clock + 1;
+ this_lrq->delta_fair_clock = 0;
+
+ exec_delta64 = this_lrq->delta_exec_clock + 1;
+ this_lrq->delta_exec_clock = 0;
+
+ if (fair_delta64 > (u64)LONG_MAX)
+   fair_delta64 = (u64)LONG_MAX;
+ fair_delta = (unsigned long)fair_delta64;
+
+ if (exec_delta64 > (u64)TICK_NSEC)
+   exec_delta64 = (u64)TICK_NSEC;
+ exec_delta = (unsigned long)exec_delta64;
+
+ idle_delta = TICK_NSEC - exec_delta;
+
+ tmp64 = SCHED_LOAD_SCALE * exec_delta64;
+ do_div(tmp64, fair_delta);
+ tmp64 *= exec_delta64;
+ 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_lrq->cpu_load[i];
+   new_load = this_load;
+
+   this_lrq->cpu_load[i] = (old_load*(scale-1) + new_load) >> i;
+ }
+}
+
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);

+ /* replay load smoothening for all ticks we lost */

```





```

- struct task_struct * (*load_balance_start) (struct rq *rq);
- struct task_struct * (*load_balance_next) (struct rq *rq);
#ifdef CONFIG_SMP
+ int (*load_balance) (struct rq *this_rq, int this_cpu,
+ struct rq *busiest,
+ unsigned long max_nr_move, unsigned long max_load_move,
+ struct sched_domain *sd, enum idle_type idle,
+ int *all_pinned, unsigned long *total_load_moved);
#endif
void (*task_tick) (struct rq *rq, struct task_struct *p);
void (*task_new) (struct rq *rq, struct task_struct *p);
};
@@ -893,6 +898,11 @@
s64 fair_key;
s64 sum_wait_runtime, sum_sleep_runtime;
unsigned long wait_runtime_overruns, wait_runtime_underruns;
#ifdef CONFIG_FAIR_GROUP_SCHED
+ struct sched_entity *parent;
+ struct lq *lq, /* runqueue on which this entity is (to be) queued */
+ *my_q; /* runqueue "owned" by this entity/group */
#endif
};

```

```

struct task_struct {

```

Index: current/kernel/sched.c

```

=====
--- current.orig/kernel/sched.c 2007-06-09 15:07:36.000000000 +0530
+++ current/kernel/sched.c 2007-06-09 15:07:37.000000000 +0530
@@ -133,6 +133,20 @@
struct rb_root tasks_timeline;
struct rb_node *rb_leftmost;
struct rb_node *rb_load_balance_curr;
+
#ifdef CONFIG_FAIR_GROUP_SCHED
+ struct sched_entity *curr;
+ struct rq *rq;
+
+ /* leaf lq's are those that hold tasks (lowest schedulable entity in a
+ * hierarchy). Non-leaf lq's hold other higher schedulable entities
+ * (like users, containers etc.)
+ *
+ * leaf_lq_list ties together list of leaf lq's in a cpu. This list
+ * is used during load balance.
+ */
+ struct list_head leaf_lq_list;
#endif
};

```

```

/*
@@ -161,6 +175,9 @@
#endif
#endif
    struct irq irq;
#ifdef CONFIG_FAIR_GROUP_SCHED
+ struct list_head leaf_irq_list; /* list of leaf irqs on this cpu */
#endif

    u64 nr_switches;

@@ -619,6 +636,16 @@
}

static void activate_task(struct rq *rq, struct task_struct *p, int wakeup);
#ifdef CONFIG_SMP
+static int balance_tasks(struct rq *this_rq, int this_cpu, struct rq *busiest,
+    unsigned long max_nr_move, unsigned long max_load_move,
+    struct sched_domain *sd, enum idle_type idle,
+    int *all_pinned, unsigned long *load_moved,
+    int this_best_prio, int best_prio, int best_prio_seen,
+    void *iterator_arg,
+    struct task_struct *(*iterator_start)(void *arg),
+    struct task_struct *(*iterator_next)(void *arg));
#endif

#include "sched_stats.h"
#include "sched_rt.c"
@@ -757,6 +784,9 @@
static inline void __set_task_cpu(struct task_struct *p, unsigned int cpu)
{
    task_thread_info(p)->cpu = cpu;
#ifdef CONFIG_FAIR_GROUP_SCHED
+ p->se.irq = &cpu_rq(cpu)->irq;
#endif
}

void set_task_cpu(struct task_struct *p, unsigned int new_cpu)
@@ -781,6 +811,9 @@

    task_thread_info(p)->cpu = new_cpu;

#ifdef CONFIG_FAIR_GROUP_SCHED
+ p->se.irq = &new_rq->irq;
#endif
}

struct migration_req {

```

```

@@ -1761,89 +1794,28 @@
    return 1;
}

-/*
- * Load-balancing iterator: iterate through the hierarchy of scheduling
- * classes, starting with the highest-prio one:
- */
-
-struct task_struct * load_balance_start(struct rq *rq)
-{
- struct sched_class *class = sched_class_highest;
- struct task_struct *p;
-
- do {
-  p = class->load_balance_start(rq);
-  if (p) {
-   rq->load_balance_class = class;
-   return p;
-  }
-  class = class->next;
- } while (class);
-
- return NULL;
-}
-
-struct task_struct * load_balance_next(struct rq *rq)
-{
- struct sched_class *class = rq->load_balance_class;
- struct task_struct *p;
-
- p = class->load_balance_next(rq);
- if (p)
-  return p;
-
- /*
-  * Pick up the next class (if any) and attempt to start
-  * the iterator there:
-  */
- while ((class = class->next)) {
-  p = class->load_balance_start(rq);
-  if (p) {
-   rq->load_balance_class = class;
-   return p;
-  }
- }
-
- return NULL;
-}
-

```

```

-#define rq_best_prio(rq) (rq)->curr->prio
-
-/*
- * move_tasks tries to move up to max_nr_move tasks and max_load_move weighted
- * load from busiest to this_rq, as part of a balancing operation within
- * "domain". Returns the number of tasks moved.
- *
- * Called with both runqueues locked.
- */
-static int move_tasks(struct rq *this_rq, int this_cpu, struct rq *busiest,
+static int balance_tasks(struct rq *this_rq, int this_cpu, struct rq *busiest,
    unsigned long max_nr_move, unsigned long max_load_move,
    struct sched_domain *sd, enum idle_type idle,
-    int *all_pinned)
+    int *all_pinned, unsigned long *load_moved,
+    int this_best_prio, int best_prio, int best_prio_seen,
+    void *iterator_arg,
+    struct task_struct *(*iterator_start)(void *arg),
+    struct task_struct *(*iterator_next)(void *arg))
{
-    int pulled = 0, pinned = 0, this_best_prio, best_prio,
-    best_prio_seen, skip_for_load;
+    int pulled = 0, pinned = 0, skip_for_load;
    struct task_struct *p;
-    long rem_load_move;
+    long rem_load_move = max_load_move;

    if (max_nr_move == 0 || max_load_move == 0)
        goto out;

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

    /*
     * Start the load-balancing iterator:
     */
-    p = load_balance_start(busiest);
+    p = (*iterator_start)(iterator_arg);

```

```

next:
if (!p)
goto out;
@@ -1860,7 +1832,7 @@
    !can_migrate_task(p, busiest, this_cpu, sd, idle, &pinned)) {

    best_prio_seen |= p->prio == best_prio;
-   p = load_balance_next(busiest);
+   p = (*iterator_next)(iterator_arg);
    goto next;
}

@@ -1875,7 +1847,7 @@
if (pulled < max_nr_move && rem_load_move > 0) {
    if (p->prio < this_best_prio)
        this_best_prio = p->prio;
-   p = load_balance_next(busiest);
+   p = (*iterator_next)(iterator_arg);
    goto next;
}
out:
@@ -1888,10 +1860,39 @@

if (all_pinned)
    *all_pinned = pinned;
+ *load_moved = max_load_move - rem_load_move;
return pulled;
}

/*
+ * move_tasks tries to move up to max_nr_move tasks and max_load_move weighted
+ * load from busiest to this_rq, as part of a balancing operation within
+ * "domain". Returns the number of tasks moved.
+ *
+ * Called with both runqueues locked.
+ */
+static int move_tasks(struct rq *this_rq, int this_cpu, struct rq *busiest,
+    unsigned long max_nr_move, unsigned long max_load_move,
+    struct sched_domain *sd, enum idle_type idle,
+    int *all_pinned)
+{
+ struct sched_class *class = sched_class_highest;
+ unsigned long load_moved, total_nr_moved = 0, nr_moved;
+
+ do {
+     nr_moved = class->load_balance(this_rq, this_cpu, busiest,
+         max_nr_move, max_load_move, sd, idle,
+         all_pinned, &load_moved);

```

```

+ total_nr_moved += nr_moved;
+ max_nr_move -= nr_moved;
+ max_load_move -= load_moved;
+ class = class->next;
+ } while (class && max_nr_move && max_load_move);
+
+ return total_nr_moved;
+}
+
+/*
 * find_busiest_group finds and returns the busiest CPU group within the
 * domain. It calculates and returns the amount of weighted load which
 * should be moved to restore balance via the imbalance parameter.
@@ -4503,6 +4504,9 @@
#else
    task_thread_info(idle)->preempt_count = 0;
#endif
+#ifdef CONFIG_FAIR_GROUP_SCHED
+ idle->se.lrq = &rq->lrq;
+#endif
+
+
+/*
@@ -6086,6 +6090,9 @@
    lrq->nr_running = 0;
    for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
        lrq->cpu_load[j] = 0;
+#ifdef CONFIG_FAIR_GROUP_SCHED
+ lrq->rq = rq;
+#endif
+
+
void __init sched_init(void)
@@ -6110,6 +6117,10 @@
    rq->nr_running = 0;
    rq->clock = 1;
    init_lrq(&rq->lrq, rq);
+#ifdef CONFIG_FAIR_GROUP_SCHED
+ INIT_LIST_HEAD(&rq->leaf_lrq_list);
+ list_add(&rq->lrq.leaf_lrq_list, &rq->leaf_lrq_list);
+#endif

#ifdef CONFIG_SMP
    for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
Index: current/kernel/sched_fair.c
=====
--- current.orig/kernel/sched_fair.c 2007-06-09 15:07:36.000000000 +0530
+++ current/kernel/sched_fair.c 2007-06-09 15:07:37.000000000 +0530

```

```

@@ -46,6 +46,29 @@
/* BEGIN : CFS operations on generic schedulable entities */
/*****

+ #ifdef CONFIG_FAIR_GROUP_SCHED
+
+ /* cpu runqueue to which this Irq is attached */
+ static inline struct rq *Irq_rq(struct Irq *Irq)
+ {
+     return Irq->rq;
+ }
+
+ static inline struct sched_entity *Irq_curr(struct Irq *Irq)
+ {
+     return Irq->curr;
+ }
+
+ /* An entity is a task if it doesn't "own" a runqueue */
+ #define entity_is_task(se) (!se->my_q)
+
+ static inline void set_Irq_curr(struct Irq *Irq, struct sched_entity *se)
+ {
+     Irq->curr = se;
+ }
+
+ #else /* CONFIG_FAIR_GROUP_SCHED */
+
+ static inline struct rq *Irq_rq(struct Irq *Irq)
+ {
+     return container_of(Irq, struct rq, Irq);
@@ -69,6 +92,10 @@

#define entity_is_task(se) 1

+ static inline void set_Irq_curr(struct Irq *Irq, struct sched_entity *se) { }
+
+ #endif /* CONFIG_FAIR_GROUP_SCHED */
+
+ static inline struct task_struct *entity_to_task(struct sched_entity *se)
+ {
+     return container_of(se, struct task_struct, se);
@@ -119,6 +146,7 @@
+ rb_insert_color(&p->run_node, &Irq->tasks_timeline);
+ Irq->raw_weighted_load += p->load_weight;
+ Irq->nr_running++;
+ p->on_rq = 1;
+ }

```

```

static inline void __dequeue_entity(struct lrq *lrq, struct sched_entity *p)
@@ -128,6 +156,7 @@
    rb_erase(&p->run_node, &lrq->tasks_timeline);
    lrq->raw_weighted_load -= p->load_weight;
    lrq->nr_running--;
+ p->on_rq = 0;
}

static inline struct rb_node * first_fair(struct lrq *lrq)
@@ -231,6 +260,9 @@

    if (!curr || curtask == rq->idle || !load)
        return;
+
+ BUG_ON(!curr->exec_start);
+
    /*
     * Get the amount of time the current task was running
     * since the last time we changed raw_weighted_load:
    @@ -539,6 +571,7 @@
    */
    update_stats_wait_end(lrq, p, now);
    update_stats_curr_start(lrq, p, now);
+ set_lrq_curr(lrq, p);

    return p;
}
@@ -557,9 +590,7 @@
    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 {
    @@ -570,6 +601,7 @@

    if (prev->on_rq)
        update_stats_wait_start(lrq, prev, now);
+ set_lrq_curr(lrq, NULL);
    }

static void update_load_fair(struct lrq *this_lrq)
@@ -640,9 +672,7 @@
    * 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.
@@ -669,11 +699,70 @@
/*          BEGIN : CFS operations on tasks          */
/*****/

#ifdef CONFIG_FAIR_GROUP_SCHED
+
+#define for_each_sched_entity(se) \
+ for (; se; se = se->parent)
+
+static inline struct lrq *task_lrq(struct task_struct *p)
+{
+ return p->se.lrq;
+}
+
+/* runqueue on which this entity is (to be) queued */
+static inline struct lrq *sched_entity_lrq(struct sched_entity *se)
+{
+ return se->lrq;
+}
+
+/* runqueue "owned" by this group */
+static inline struct lrq *group_lrq(struct sched_entity *grp)
+{
+ return grp->my_q;
+}
+
+static inline struct lrq *cpu_lrq(struct lrq *lrq, int this_cpu)
+{
+ return &cpu_rq(this_cpu)->lrq;
+}
+
+#define for_each_leaf_lrq(a, b) \
+ list_for_each_entry(b, &a->leaf_lrq_list, leaf_lrq_list)
+
+#else /* CONFIG_FAIR_GROUP_SCHED */
+
+#define for_each_sched_entity(se) \
+ for (; se; se = NULL)
+
+static inline struct lrq *task_lrq(struct task_struct *p)
+{

```

```

    return &task_rq(p)->lrq;
}

+static inline struct lrq *sched_entity_lrq(struct sched_entity *se)
+{
+ struct task_struct *p = entity_to_task(se);
+ struct rq *rq = task_rq(p);
+
+ return &rq->lrq;
+}
+
+/* runqueue "owned" by this group */
+static inline struct lrq *group_lrq(struct sched_entity *grp)
+{
+ return NULL;
+}
+
+static inline struct lrq *cpu_lrq(struct lrq *lrq, int this_cpu)
+{
+ return &cpu_rq(this_cpu)->lrq;
+}
+
+#define for_each_leaf_lrq(a, b) \
+ for (b = &a->lrq; b; b = NULL)
+
+#endif /* CONFIG_FAIR_GROUP_SCHED */
+
+/*
+ * The enqueue_task method is called before nr_running is
+ * increased. Here we update the fair scheduling stats and
@@ -682,10 +771,15 @@
static void
enqueue_task_fair(struct rq *rq, struct task_struct *p, int wakeup, u64 now)
{
- struct lrq *lrq = task_lrq(p);
+ struct lrq *lrq;
+ struct sched_entity *se = &p->se;

- enqueue_entity(lrq, se, wakeup, now);
+ for_each_sched_entity(se) {
+ if (se->on_rq)
+ break;
+ lrq = sched_entity_lrq(se);
+ enqueue_entity(lrq, se, wakeup, now);
+ }
}

/*

```

```

@@ -696,10 +790,16 @@
static void
dequeue_task_fair(struct rq *rq, struct task_struct *p, int sleep, u64 now)
{
- struct lrq *lrq = task_lrq(p);
+ struct lrq *lrq;
  struct sched_entity *se = &p->se;

- dequeue_entity(lrq, se, sleep, now);
+ for_each_sched_entity(se) {
+   lrq = sched_entity_lrq(se);
+   dequeue_entity(lrq, se, sleep, now);
+   /* Don't dequeue parent if it has other entities besides us */
+   if (lrq->raw_weighted_load)
+     break;
+ }
}

/*
@@ -721,9 +821,7 @@
 * position within the tree:
 */
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
@@ -772,7 +870,10 @@
if (unlikely(p->policy == SCHED_BATCH))
granularity = sysctl_sched_batch_wakeup_granularity;

- __check_preempt_curr_fair(lrq, &p->se, &curr->se, granularity);
+ /* todo: check for preemption at higher levels */
+ if (lrq == task_lrq(p))
+   __check_preempt_curr_fair(lrq, &p->se, &curr->se,
+     granularity);
}
}

@@ -781,7 +882,10 @@
struct lrq *lrq = &rq->lrq;
struct sched_entity *se;

- se = pick_next_entity(lrq, now);
+ do {
+   se = pick_next_entity(lrq, now);

```

```

+ lrq = group_lrq(se);
+ } while (lrq);

    return entity_to_task(se);
}
@@ -791,19 +895,26 @@
    */
    static void put_prev_task_fair(struct rq *rq, struct task_struct *prev, u64 now)
    {
- struct lrq *lrq = task_lrq(prev);
+ struct lrq *lrq;
    struct sched_entity *se = &prev->se;

    if (prev == rq->idle)
        return;

- put_prev_entity(lrq, se, now);
+ for_each_sched_entity(se) {
+ lrq = sched_entity_lrq(se);
+ put_prev_entity(lrq, se, now);
+ }
    }

+ #ifdef CONFIG_SMP
+
+ /*****
+  /* Fair scheduling class load-balancing methods:
+  */
+
+ /* todo: return cache-cold tasks first */
+
+ /*
+  * Load-balancing iterator. Note: while the runqueue stays locked
+  * during the whole iteration, the current task might be
+  @@ -812,7 +923,7 @@
+  * the current task:
+  */
+  static inline struct task_struct *
+  __load_balance_iterator(struct rq *rq, struct rb_node *curr)
+  __load_balance_iterator(struct lrq *lrq, struct rb_node *curr)
+  {
+  struct task_struct *p;

+  @@ -820,30 +931,121 @@
+  return NULL;

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

```

```

+ lrq->rb_load_balance_curr = rb_next(curr);

    return p;
}

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

-static struct task_struct * load_balance_next_fair(struct rq *rq)
+static struct task_struct * load_balance_next_fair(void *arg)
{
- return __load_balance_iterator(rq, rq->lrq.rb_load_balance_curr);
+ struct lrq *lrq = arg;
+
+ return __load_balance_iterator(lrq, lrq->rb_load_balance_curr);
}

+static inline int lrq_best_prio(struct lrq *lrq)
+{
+ struct sched_entity *curr;
+ struct task_struct *p;
+
+ if (!lrq->nr_running)
+ return MAX_PRIO;
+
+ curr = __pick_next_entity(lrq);
+ p = entity_to_task(curr);
+
+ return p->prio;
+}
+
+static int
+load_balance_fair(struct rq *this_rq, int this_cpu, struct rq *busiest,
+ unsigned long max_nr_move, unsigned long max_load_move,
+ struct sched_domain *sd, enum idle_type idle,
+ int *all_pinned, unsigned long *total_load_moved)
+{
+ struct lrq *busy_lrq;
+ unsigned long load_moved, total_nr_moved = 0, nr_moved, rem_load_move;
+
+ rem_load_move = max_load_move;
+
+

```

```

+ for_each_leaf_lrq(busiest, busy_lrq) {
+ struct lrq *this_lrq;
+ long imbalance;
+ unsigned long maxload;
+ int this_best_prio, best_prio, best_prio_seen = 0;
+
+ this_lrq = cpu_lrq(busy_lrq, this_cpu);
+
+ imbalance = busy_lrq->raw_weighted_load -
+   this_lrq->raw_weighted_load;
+ /* Don't pull if this_lrq has more load than busy_lrq */
+ if (imbalance <= 0)
+   continue;
+
+ /* Don't pull more than imbalance/2 */
+ imbalance /= 2;
+ maxload = min(rem_load_move, (unsigned long)imbalance);
+
+ this_best_prio = lrq_best_prio(this_lrq);
+ best_prio = lrq_best_prio(busy_lrq);
+
+ /*
+  * Enable handling of the case where there is more than one task
+  * with the best priority. If the current running task is one
+  * of those with prio==best_prio we know it won't be moved
+  * and therefore it's safe to override the skip (based on load)
+  * of any task we find with that prio.
+  */
+ if (lrq_curr(busy_lrq) == &busiest->curr->se)
+   best_prio_seen = 1;
+
+ nr_moved = balance_tasks(this_rq, this_cpu, busiest,
+   max_nr_move, maxload, sd, idle, all_pinned,
+   &load_moved, this_best_prio, best_prio,
+   best_prio_seen,
+   /* pass busy_lrq argument into
+    * load_balance_[start|next]_fair iterators
+    */
+   busy_lrq,
+   load_balance_start_fair,
+   load_balance_next_fair);
+
+ total_nr_moved += nr_moved;
+ max_nr_move -= nr_moved;
+ rem_load_move -= load_moved;
+
+ /* todo: break if rem_load_move is < load_per_task */
+ if (!max_nr_move || !rem_load_move)

```

```

+ break;
+ }
+
+ *total_load_moved = max_load_move - rem_load_move;
+
+ return total_nr_moved;
+}
+
+ #endif /* CONFIG_SMP */
+
+ /*
+  * scheduler tick hitting a task of our scheduling class:
+  */
+ static void task_tick_fair(struct rq *rq, struct task_struct *curr)
+ {
+ - struct lrq *lrq = task_lrq(curr);
+ + struct lrq *lrq;
+   struct sched_entity *se = &curr->se;

+ - entity_tick(lrq, se);
+ + for_each_sched_entity(se) {
+ +   lrq = sched_entity_lrq(se);
+ +   /* todo: reduce tick frequency at higher levels */
+ +   entity_tick(lrq, se);
+ + }
+ }

+ /*
@@ -880,7 +1082,6 @@
// p->se.wait_runtime = -(s64)(sysctl_sched_granularity / 2);

__enqueue_entity(lrq, se);
- p->se.on_rq = 1;
  inc_nr_running(p, rq);
}

@@ -897,8 +1098,9 @@
.pick_next_task = pick_next_task_fair,
.put_prev_task = put_prev_task_fair,

- .load_balance_start = load_balance_start_fair,
- .load_balance_next = load_balance_next_fair,
+ #ifdef CONFIG_SMP
+ .load_balance = load_balance_fair,
+ #endif
+ .task_tick = task_tick_fair,
+ .task_new = task_new_fair,
};

```

Index: current/kernel/sched\_rt.c

```
=====
--- current.orig/kernel/sched_rt.c 2007-06-09 15:04:54.000000000 +0530
+++ current/kernel/sched_rt.c 2007-06-09 15:07:37.000000000 +0530
@@ -100,6 +100,8 @@
    p->se.exec_start = 0;
}

+#ifdef CONFIG_SMP
+
+/*
+ * Load-balancing iterator. Note: while the runqueue stays locked
+ * during the whole iteration, the current task might be
@@ -107,8 +109,9 @@
+ * achieve that by always pre-iterating before returning
+ * the current task:
+ */
-static struct task_struct * load_balance_start_rt(struct rq *rq)
+static struct task_struct * load_balance_start_rt(void *arg)
{
+ struct rq *rq = (struct rq *)arg;
    struct prio_array *array = &rq->active;
    struct list_head *head, *curr;
    struct task_struct *p;
@@ -132,8 +135,9 @@
    return p;
}

-static struct task_struct * load_balance_next_rt(struct rq *rq)
+static struct task_struct * load_balance_next_rt(void *arg)
{
+ struct rq *rq = (struct rq *)arg;
    struct prio_array *array = &rq->active;
    struct list_head *head, *curr;
    struct task_struct *p;
@@ -170,6 +174,42 @@
    return p;
}

+static int
+load_balance_rt(struct rq *this_rq, int this_cpu, struct rq *busiest,
+ unsigned long max_nr_move, unsigned long max_load_move,
+ struct sched_domain *sd, enum idle_type idle,
+ int *all_pinned, unsigned long *load_moved)
+{
+ int this_best_prio, best_prio, best_prio_seen = 0;
+ int nr_moved;
+
+

```



```

+ best_prio = sched_find_first_bit(busiest->active.bitmap);
+ this_best_prio = sched_find_first_bit(this_rq->active.bitmap);
+
+ /*
+  * Enable handling of the case where there is more than one task
+  * with the best priority. If the current running task is one
+  * of those with prio==best_prio we know it won't be moved
+  * and therefore it's safe to override the skip (based on load)
+  * of any task we find with that prio.
+  */
+ if (busiest->curr->prio == best_prio)
+   best_prio_seen = 1;
+
+ nr_moved = balance_tasks(this_rq, this_cpu, busiest, max_nr_move,
+   max_load_move, sd, idle, all_pinned, load_moved,
+   this_best_prio, best_prio, best_prio_seen,
+   /* pass busiest argument into
+    * load_balance_[start|next]_rt iterators
+    */
+   busiest,
+   load_balance_start_rt, load_balance_next_rt);
+
+ return nr_moved;
+}
+
+##endif /* CONFIG_SMP */
+
+static void task_tick_rt(struct rq *rq, struct task_struct *p)
+{
+  /*
+@@ -204,8 +244,9 @@
+   .pick_next_task = pick_next_task_rt,
+   .put_prev_task = put_prev_task_rt,
+
+   .load_balance_start = load_balance_start_rt,
+   .load_balance_next = load_balance_next_rt,
+##ifdef CONFIG_SMP
+   .load_balance = load_balance_rt,
+##endif
+
+   .task_tick = task_tick_rt,
+   .task_new = task_new_rt,
+
+--
+
+Regards,
+vatsa

```

---

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

---

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

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

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

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

1. Define CONFIG\_FAIR\_GROUP\_SCHED (General Setup->Fair Group Scheduler) and compile the kernel
2. After booting:

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

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

...

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

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

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

---

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

Index: current/kernel/sched.c

=====

--- current.orig/kernel/sched.c 2007-06-09 15:07:37.000000000 +0530

+++ current/kernel/sched.c 2007-06-09 15:07:38.000000000 +0530

@@ -116,6 +116,39 @@

```
struct list_head queue[MAX_RT_PRIO];
};
```

+#ifdef CONFIG\_FAIR\_GROUP\_SCHED

+

+#include <linux/container.h>

+

+struct lrq;

+

+struct task\_grp {

+ struct container\_subsys\_state css;

+ /\* schedulable entities of this group on each cpu \*/

+ struct sched\_entity \*\*se;

+ /\* runqueue "owned" by this group on each cpu \*/

+ struct lrq \*\*lrq;

+};

+

+static DEFINE\_PER\_CPU\_SHARED\_ALIGNED(struct sched\_entity, init\_sched\_entity);

+static DEFINE\_PER\_CPU\_SHARED\_ALIGNED(struct lrq, init\_lrq);

+

+static struct sched\_entity \*init\_sched\_entity\_p[CONFIG\_NR\_CPUS];

+static struct lrq \*init\_lrq\_p[CONFIG\_NR\_CPUS];

+

+static struct task\_grp init\_task\_grp = {

+ .se = init\_sched\_entity\_p,

+ .lrq = init\_lrq\_p,

+ };

+

+static inline struct task\_grp \*task\_grp(struct task\_struct \*p)

+{

+ return container\_of(task\_subsys\_state(p, cpuctlr\_subsys\_id),

+ struct task\_grp, css);

```

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

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

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

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

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

@@ -6119,7 +6156,22 @@
+     init_lrq(&rq->lrq, rq);
+ #ifdef CONFIG_FAIR_GROUP_SCHED

```

```

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

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

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

```

```

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

```

```

+ tg->css.container = cont;
+
+ return 0;
+
+err:
+ for_each_possible_cpu(i) {
+   if (tg->lrq && tg->lrq[i])
+     kfree(tg->lrq[i]);
+   if (tg->se && tg->se[i])
+     kfree(tg->se[i]);
+ }
+ if (tg->lrq)
+   kfree(tg->lrq);
+ if (tg->se)
+   kfree(tg->se);
+ if (tg)
+   kfree(tg);
+
+ return -ENOMEM;
+}
+
+
+/* destroy runqueue etc associated with a task group */
+static void sched_destroy_group(struct container_subsys *ss,
+   struct container *cont)
+{
+   struct task_grp *tg = container_tg(cont);
+   struct lrq *lrq;
+   struct sched_entity *se;
+   int i;
+
+   for_each_possible_cpu(i) {
+     lrq = tg->lrq[i];
+     list_del_rcu(&lrq->leaf_lrq_list);
+   }
+
+   /* wait for possible concurrent references to lrqs complete */
+   synchronize_sched();
+
+   /* now it should be safe to free those lrqs */
+   for_each_possible_cpu(i) {
+     lrq = tg->lrq[i];
+     kfree(lrq);
+
+     se = tg->se[i];
+     kfree(se);
+   }
+
+

```

```

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

```





```

+}
+
+static struct sched_entity * pick_next_entity(struct lrq *lrq, u64 now)
+{
+ struct sched_entity *p = __pick_next_entity(lrq);
+
+ set_next_entity(lrq, p, now);

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

```

```

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

```

```

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

```

```

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

```

```

@@ -1098,6 +1118,8 @@
    .pick_next_task = pick_next_task_fair,
    .put_prev_task = put_prev_task_fair,

+ .set_curr_task = set_curr_task_fair,
+
#ifdef CONFIG_SMP
    .load_balance = load_balance_fair,

```

#endif

Index: current/include/linux/sched.h

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

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

#ifdef CONFIG\_SMP

int (\*load\_balance) (struct rq \*this\_rq, int this\_cpu,

Index: current/init/Kconfig

```
=====
--- current.orig/init/Kconfig 2007-06-09 14:56:43.000000000 +0530
+++ current/init/Kconfig 2007-06-09 15:07:38.000000000 +0530
@@ -328,6 +328,14 @@
```

Say N if unsure.

+config FAIR\_GROUP\_SCHED

+ select CONTAINERS

+ help

+ This option enables you to group tasks and control CPU resource  
+ allocation to such groups.

+

+ Say N if unsure.

+

config SYSFS\_DEPRECATED

bool "Create deprecated sysfs files"

default y

--

Regards,

vatsa

---

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>

---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1

Posted by [Srivatsa Vaddagiri](#) on Mon, 11 Jun 2007 16:02:28 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On Mon, Jun 11, 2007 at 09:17:24PM +0530, Srivatsa Vaddagiri wrote:

> TODO:

>  
> - Weighted fair-share support  
> Currently each group gets "equal" share. Support  
> weighted fair-share so that some groups deemed important  
> get more than this "equal" share. I believe it is  
> possible to use load\_weight to achieve this goal  
> (similar to how niced tasks use it to get differential  
> bandwidth)

[snip]

+ Fix CFS debug interface to be group aware

--  
Regards,  
vatsa

---

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

---

---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1  
Posted by [Ingo Molnar](#) on Mon, 11 Jun 2007 19:37:35 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

\* Srivatsa Vaddagiri <[vatsa@linux.vnet.ibm.com](mailto:vatsa@linux.vnet.ibm.com)> wrote:

> Ingo,  
> Here's an update of the group fairness patch I have been  
> working on. Its against CFS v16 (sched-cfs-v2.6.22-rc4-mm2-v16.patch).

thanks!

> The core idea is to reuse much of CFS logic to apply fairness at  
> higher hierarchical levels (user, container etc). In this regard CFS  
> engine has been modified to deal with generic 'schedulable entities'.  
> The patches introduce two essential structures in CFS core:  
>  
> - struct sched\_entity  
> - represents a schedulable entity in a hierarchy. Task  
> is the lowest element in this hierarchy. Its ancestors  
> could be user, container etc. This structure stores  
> essential attributes/execution-history (wait\_runtime etc)  
> which is required by CFS engine to provide fairness between  
> 'struct sched\_entities' at the same hierarchy.  
>  
> - struct lrq

> - represents (per-cpu) runqueue in which ready-to-run  
> 'struct sched\_entities' are queued. The fair clock  
> calculation is split to be per 'struct Irq'.  
>  
> Here's a brief description of the patches to follow:  
>  
> Patches 1-3 introduce the essential changes in CFS core to support  
> this concept. They rework existing code w/o any (intended!) change in  
> functionality.

i currently have these 3 patches applied to the CFS queue and it's looking pretty good so far! If it continues to be problem-free i'll release them as part of -v17, just to check that they truly have no bad side-effects (they shouldnt). Then #4 can go into -v18.

i've attached my current -v17 tree - it should apply mostly cleanly ontop of the -mm queue (with a minor number of fixups). Could you refactor the remaining 3 patches ontop of this base? There's some rejects in the last 3 patches due to the update\_load\_fair() change.

> Patch 4 fixes some bad interaction between SCHED\_RT and SCHED\_NORMAL  
> tasks in current CFS.

btw., the plan here is to turn off 'bit 0' in sched\_features: i.e. to use the precise statistics to calculate Irq->cpu\_load[], not the timer-irq-sampled imprecise statistics. Dmitry has fixed a couple of bugs in it that made it not work too well in previous CFS versions, but now we are ready to turn it on for -v17. (indeed in my tree it's already turned on - i.e. sched\_features defaults to '14')

> Patch 5 introduces basic changes in CFS core to support group  
> fairness.  
>  
> Patch 6 hooks up scheduler with container patches in mm (as an  
> interface for task-grouping functionality).

ok. Kirill, how do you like Srivatsa's current approach? Would be nice to kill two birds with the same stone, if possible :-)

> Note: I have noticed that running lat\_ctx in a loop for 10 times  
> doesnt give me good results. Basically I expected the loop to take  
> same time for both users (when run simultaneously), whereas it was  
> taking different times for different users. I think this can be solved  
> by increasing sysctl\_sched\_runtime\_limit at group level (to remeber  
> execution history over a longer period).

you'll get the best hackbench results by using SCHED\_BATCH:

```
chrt -b 0 ./hackbench 10
```

or indeed increasing the runtime\_limit would work too.

Ingo

Index: linux/Makefile

```
=====
--- linux.orig/Makefile
+++ linux/Makefile
@@ -1,7 +1,7 @@
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 21
-EXTRAVERSION = .4-cfs-v16
+EXTRAVERSION = .4-cfs-v17
NAME = Nocturnal Monster Puppy
```

# \*DOCUMENTATION\*

Index: linux/fs/proc/array.c

```
=====
--- linux.orig/fs/proc/array.c
+++ linux/fs/proc/array.c
@@ -319,7 +319,7 @@ static clock_t task_utime(struct task_st
 * Use CFS's precise accounting, if available:
 */
if (!(sysctl_sched_features & 128)) {
- u64 temp = (u64)nsec_to_clock_t(p->sum_exec_runtime);
+ u64 temp = (u64)nsec_to_clock_t(p->se.sum_exec_runtime);

if (total) {
temp *= utime;
@@ -341,7 +341,7 @@ static clock_t task_stime(struct task_st
 * by userspace grows monotonically - apps rely on that):
*/
if (!(sysctl_sched_features & 128))
- stime = nsec_to_clock_t(p->sum_exec_runtime) - task_utime(p);
+ stime = nsec_to_clock_t(p->se.sum_exec_runtime) - task_utime(p);

return stime;
}
```

Index: linux/include/linux/sched.h

```
=====
--- linux.orig/include/linux/sched.h
+++ linux/include/linux/sched.h
@@ -534,8 +534,7 @@ struct signal_struct {

#define rt_prio(prio) unlikely((prio) < MAX_RT_PRIO)
```

```

#define rt_task(p) rt_prio((p)->prio)
-#define batch_task(p) (unlikely((p)->policy == SCHED_BATCH))
-#define is_rt_policy(p) ((p) != SCHED_NORMAL && (p) != SCHED_BATCH)
+#define is_rt_policy(p) ((p) == SCHED_FIFO || (p) == SCHED_RR)
#define has_rt_policy(p) unlikely(is_rt_policy((p)->policy))

/*
@@ -819,6 +818,29 @@ struct sched_class {
    void (*task_new) (struct rq *rq, struct task_struct *p);
};

+/* CFS stats for a schedulable entity (task, task-group etc) */
+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 */
+ struct thread_info *thread_info;
@@ -833,33 +855,15 @@ struct task_struct {
+ 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;
-
- 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 long policy;
    cpumask_t cpus_allowed;

```

Index: linux/kernel/exit.c

```

=====
--- linux.orig/kernel/exit.c
+++ linux/kernel/exit.c
@@ -112,7 +112,7 @@ static void __exit_signal(struct task_st
    sig->majflt += tsk->majflt;
    sig->nvcsw += tsk->nvcsw;
    sig->nivcsw += tsk->nivcsw;
- sig->sum_sched_runtime += tsk->sum_exec_runtime;
+ sig->sum_sched_runtime += tsk->se.sum_exec_runtime;
    sig = NULL; /* Marker for below. */
}

```

Index: linux/kernel/posix-cpu-timers.c

```

=====
--- linux.orig/kernel/posix-cpu-timers.c
+++ linux/kernel/posix-cpu-timers.c
@@ -249,7 +249,7 @@ static int cpu_clock_sample_group_locked
    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 +467,7 @@ static void cleanup_timers(struct list_h
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 +475,7 @@ void posix_cpu_timers_exit_group(struct
    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 +536,7 @@ static void process_timer_rebalance(stru
    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 +1004,7 @@ static void check_thread_timers(struct t
    struct cpu_timer_list *t = list_entry(timers->next,
        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 +1049,7 @@ static void check_process_timers(struct
    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 +1208,7 @@ static void check_process_timers(struct

```

```

    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 +1300,7 @@ void run_posix_cpu_timers(struct task_st

  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/kernel/sched.c

--- linux.orig/kernel/sched.c

+++ linux/kernel/sched.c

```

@@ -113,6 +113,23 @@ struct prio_array {
    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, delta_fair_clock;
+ u64 exec_clock, delta_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.
+ */
@@ -128,12 +145,9 @@ struct rq {
    * 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 Irq Irq;

    u64 nr_switches;
- unsigned long nr_load_updates;

/*
 * This is part of a global counter where only the total sum
@@ -149,10 +163,6 @@ struct rq {

    u64 clock, prev_clock_raw;
    s64 clock_max_delta;
- u64 fair_clock, delta_fair_clock;
- u64 exec_clock, delta_exec_clock;
- s64 wait_runtime;
- unsigned long wait_runtime_overruns, wait_runtime_underruns;

    unsigned int clock_warps, clock_overflows;
    unsigned int clock_unstable_events;
@@ -163,10 +173,6 @@ struct rq {
    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
@@ -543,13 +549,13 @@ const int prio_to_weight[40] = {
    static inline void
    inc_raw_weighted_load(struct rq *rq, const struct task_struct *p)
    {
- rq->raw_weighted_load += p->load_weight;
+ rq->Irq.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->Irq.raw_weighted_load -= p->se.load_weight;
    }

```

```

static inline void inc_nr_running(struct task_struct *p, struct rq *rq)
@@ -575,22 +581,22 @@ static void activate_task(struct rq *rq,

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_IDLEPRIO tasks get minimal weight:
*/
if (p->policy == SCHED_IDLEPRIO) {
- 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)
@@ -599,7 +605,7 @@ static void enqueue_task(struct rq *rq,

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)
@@ -607,7 +613,7 @@ static void dequeue_task(struct rq *rq,
u64 now = rq_clock(rq);

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

/*
@@ -695,7 +701,7 @@ inline int task_curr(const struct task_s

```

```

/* 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
@@ -712,18 +718,18 @@ void set_task_cpu(struct task_struct *p,
    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;

@@ -751,7 +757,7 @@ migrate_task(struct task_struct *p, int
    * 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;
}
@@ -782,7 +788,7 @@ void wait_task_inactive(struct task_stru

```

```

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);
@@ -830,9 +836,9 @@ static inline unsigned long source_load(
    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);
}

/*
@@ -844,9 +850,9 @@ static inline unsigned long target_load(
    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);
}

/*
@@ -857,7 +863,7 @@ static inline unsigned long cpu_avg_load
    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;
}

/*
@@ -1078,7 +1084,7 @@ static int try_to_wake_up(struct task_st
    if (!(old_state & state))
        goto out;

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

```

```

cpu = task_cpu(p);
@@ -1133,11 +1139,11 @@ static int try_to_wake_up(struct task_st
    * 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) {
+   100*(tl + p->se.load_weight) <= imbalance*load) {
    /*
     * This domain has SD_WAKE_AFFINE and
     * p is cache cold in this domain, and
@@ -1171,7 +1177,7 @@ out_set_cpu:
    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();
@@ -1235,18 +1241,18 @@ static void task_running_tick(struct rq
    */
    static void __sched_fork(struct task_struct *p)
    {
-   p->wait_start_fair = p->wait_start = p->exec_start = 0;
-   p->sum_exec_runtime = 0;
+   p->se.wait_start_fair = p->se.wait_start = p->se.exec_start = 0;
+   p->se.sum_exec_runtime = 0;

-   p->wait_runtime = 0;
+   p->se.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.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;

/*
@@ -1317,7 +1323,7 @@ void fastcall wake_up_new_task(struct ta
  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 {
/*
@@ -1332,7 +1338,7 @@ void fastcall wake_up_new_task(struct ta

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

/**
@@ -1542,17 +1548,17 @@ static void update_load_fair(struct rq *
  u64 fair_delta64, exec_delta64, tmp64;
  unsigned int i, scale;

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

- fair_delta64 = this_rq->delta_fair_clock + 1;
- this_rq->delta_fair_clock = 0;
+ fair_delta64 = this_rq->lrq.delta_fair_clock + 1;
+ this_rq->lrq.delta_fair_clock = 0;

- exec_delta64 = this_rq->delta_exec_clock + 1;
- this_rq->delta_exec_clock = 0;
+ exec_delta64 = this_rq->lrq.delta_exec_clock + 1;
+ this_rq->lrq.delta_exec_clock = 0;

  if (fair_delta64 > (u64)LONG_MAX)
    fair_delta64 = (u64)LONG_MAX;
@@ -1577,10 +1583,10 @@ do_avg:

```



```

/* 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;
}
}

@@ -1836,7 +1842,8 @@ next:
  * 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 ||
@@ -1849,7 +1856,7 @@ next:

  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
@@ -1946,7 +1953,7 @@ find_busiest_group(struct sched_domain *

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

/*
@@ -2178,11 +2185,12 @@ find_busiest_queue(struct sched_group *g

  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;
}
}
@@ -2607,9 +2615,9 @@ unsigned long long task_sched_runtime(st
struct rq *rq;

  rq = task_rq_lock(p, &flags);
- ns = p->sum_exec_runtime;
+ ns = p->se.sum_exec_runtime;
  if (rq->curr == p) {
- delta_exec = rq_clock(rq) - p->exec_start;
+ delta_exec = rq_clock(rq) - p->se.exec_start;
    if ((s64)delta_exec > 0)
      ns += delta_exec;
  }
@@ -3299,7 +3307,7 @@ void rt_mutex_setprio(struct task_struct
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);

@@ -3352,7 +3360,7 @@ void set_user_nice(struct task_struct *p
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);
@@ -3489,12 +3497,13 @@ static inline struct task_struct *find_p
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) {
  case SCHED_NORMAL:

```

```

case SCHED_BATCH:
+ case SCHED_ISO:
case SCHED_IDLEPRIO:
    p->sched_class = &fair_sched_class;
    break;
@@ -3534,12 +3543,12 @@ recheck:
    policy = oldpolicy = p->policy;
    else if (policy != SCHED_FIFO && policy != SCHED_RR &&
        policy != SCHED_NORMAL && policy != SCHED_BATCH &&
-    policy != SCHED_IDLEPRIO)
+    policy != SCHED_ISO && policy != SCHED_IDLEPRIO)
        return -EINVAL;
/*
 * Valid priorities for SCHED_FIFO and SCHED_RR are
 * 1..MAX_USER_RT_PRIO-1, valid priority for SCHED_NORMAL,
- * SCHED_BATCH and SCHED_IDLEPRIO is 0.
+ * SCHED_BATCH, SCHED_ISO and SCHED_IDLEPRIO is 0.
 */
if (param->sched_priority < 0 ||
    (p->mm && param->sched_priority > MAX_USER_RT_PRIO-1) ||
@@ -3570,6 +3579,12 @@ recheck:
    param->sched_priority > rlim_rtprio)
    return -EPERM;
}
+ /*
+  * Like positive nice levels, dont allow tasks to
+  * move out of SCHED_IDLEPRIO either:
+  */
+ if (p->policy == SCHED_IDLEPRIO && policy != SCHED_IDLEPRIO)
+     return -EPERM;

/* can't change other user's priorities */
if ((current->euid != p->euid) &&
@@ -3597,7 +3612,7 @@ recheck:
    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;
@@ -4093,6 +4108,7 @@ asmlinkage long sys_sched_get_priority_m
    break;
case SCHED_NORMAL:
case SCHED_BATCH:
+ case SCHED_ISO:
case SCHED_IDLEPRIO:

```

```

    ret = 0;
    break;
@@ -4118,6 +4134,7 @@ asmlinkage long sys_sched_get_priority_m
    break;
    case SCHED_NORMAL:
    case SCHED_BATCH:
+ case SCHED_ISO:
    case SCHED_IDLEPRIO:
        ret = 0;
    }
@@ -4249,7 +4266,7 @@ void __cpuinit init_idle(struct task_str
    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);
@@ -4352,7 +4369,7 @@ EXPORT_SYMBOL_GPL(set_cpus_allowed);
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;
@@ -4368,9 +4385,11 @@ static int __migrate_task(struct task_st
    if (!cpu_isset(dest_cpu, p->cpus_allowed))
        goto out;

- set_task_cpu(p, dest_cpu);
- if (p->on_rq) {
+ on_rq = p->se.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);
    }
@@ -5752,11 +5771,11 @@ void __init sched_init(void)
    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;
@@ -5836,15 +5855,15 @@ void normalize_rt_tasks(void)

    read_lock_irq(&tasklist_lock);
    do_each_thread(g, 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)) {
@@ -5867,7 +5886,7 @@ void normalize_rt_tasks(void)
        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/kernel/sched_debug.c
=====
--- linux.orig/kernel/sched_debug.c
+++ linux/kernel/sched_debug.c
@@ -40,19 +40,19 @@ print_task(struct seq_file *m, struct rq

```

```

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)
+static void print_rq(struct seq_file *m, struct rq *rq, int rq_cpu, u64 now)
{
    struct task_struct *g, *p;

@@ -70,7 +70,7 @@ static void print_rq(struct seq_file *m,
    read_lock_irq(&tasklist_lock);

    do_each_thread(g, p) {
- if (!p->on_rq)
+ if (!p->se.on_rq || task_cpu(p) != rq_cpu)
        continue;

        print_task(m, rq, p, now);
@@ -87,10 +87,10 @@ static void print_rq_runtime_sum(struct
    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, 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);
}
@@ -109,9 +109,9 @@ static void print_cpu(struct seq_file *m
    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);
@@ -122,22 +122,22 @@ static void print_cpu(struct seq_file *m
    P(clock_overflows);
    P(clock_unstable_events);
    P(clock_max_delta);
- P(fair_clock);
- P(delta_fair_clock);
- P(exec_clock);
- P(delta_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.delta_fair_clock);
+ P(lrq.exec_clock);
+ P(lrq.delta_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);

- print_rq(m, rq, now);
+ print_rq(m, rq, cpu, now);

```

```
}
```

```
static int sched_debug_show(struct seq_file *m, void *v)
```

```
@@ -205,21 +205,21 @@ void proc_sched_show_task(struct task_st
```

```
#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(wait_runtime);
- P(wait_runtime_overruns);
- P(wait_runtime_underruns);
- P(sum_exec_runtime);
- P(load_weight);
+ 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.wait_runtime);
+ P(se.wait_runtime_overruns);
+ P(se.wait_runtime_underruns);
+ P(se.sum_exec_runtime);
+ P(se.load_weight);
  P(policy);
  P(prio);
#undef P
```

```
@@ -235,7 +235,7 @@ void proc_sched_show_task(struct task_st
```

```
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/kernel/sched\_fair.c

```
=====
--- linux.orig/kernel/sched_fair.c
```

```
+++ linux/kernel/sched_fair.c
```

```
@ @ -38,22 +38,57 @ @ unsigned int sysctl_sched_batch_wakeup_g
*/
```

```
unsigned int sysctl_sched_runtime_limit __read_mostly;
```

```
-unsigned int sysctl_sched_features __read_mostly = 1 | 2 | 4 | 8 | 0 | 0;
```

```
+unsigned int sysctl_sched_features __read_mostly = 0 | 2 | 4 | 8 | 0 | 0;
```

```
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->tasks_timeline.rb_node;
+ struct rb_node **link = &lrq->tasks_timeline.rb_node;
  struct rb_node *parent = NULL;
- struct task_struct *entry;
+ struct sched_entity *entry;
  s64 key = p->fair_key;
  int leftmost = 1;

@@ -62,7 +97,7 @@ static inline void __enqueue_task_fair(s
  */
  while (*link) {
    parent = *link;
-   entry = rb_entry(parent, struct task_struct, run_node);
+   entry = rb_entry(parent, struct sched_entity, run_node);
  }
  /*
   * We dont care about collisions. Nodes with
   * the same key stay together.
@@ -80,31 +115,31 @@ static inline void __enqueue_task_fair(s
   * used):
   */
  if (leftmost)
-   rq->rb_leftmost = &p->run_node;
+   lrq->rb_leftmost = &p->run_node;

  rb_link_node(&p->run_node, parent, link);
- rb_insert_color(&p->run_node, &rq->tasks_timeline);
+ 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->rb_leftmost == &p->run_node)
-   rq->rb_leftmost = NULL;

```

```

- rb_erase(&p->run_node, &rq->tasks_timeline);
+ if (lirq->rb_leftmost == &p->run_node)
+   lirq->rb_leftmost = NULL;
+ rb_erase(&p->run_node, &lirq->tasks_timeline);
}

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

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

/*****
@@ -115,8 +150,8 @@ static struct task_struct * __pick_next_
 * 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 s64
+niced_granularity(struct sched_entity *curr, unsigned long granularity)
{
  /*
   * Negative nice levels get the same granularity as nice-0:
@@ -130,7 +165,7 @@ niced_granularity(struct task_struct *cu
   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 lirq *lirq, struct sched_entity *p)
{
  s64 limit = sysctl_sched_runtime_limit;

```

```

@@ -141,27 +176,28 @@ static void limit_wait_runtime(struct rq
    if (p->wait_runtime > limit) {
        p->wait_runtime = limit;
        p->wait_runtime_overruns++;
-   rq->wait_runtime_overruns++;
+   lrq->wait_runtime_overruns++;
    }
    if (p->wait_runtime < -limit) {
        p->wait_runtime = -limit;
        p->wait_runtime_underruns++;
-   rq->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->wait_runtime += delta;
    p->sum_wait_runtime += delta;
-   limit_wait_runtime(rq, p);
+   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->wait_runtime -= p->wait_runtime;
-   __add_wait_runtime(rq, p, delta);
-   rq->wait_runtime += p->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 dividend, unsigned long divisor)
@@ -183,13 +219,15 @@ static s64 div64_s(s64 dividend, unsigne
 * 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->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
@@ -203,29 +241,29 @@ static inline void update_curr(struct rq

    curr->sum_exec_runtime += delta_exec;
    curr->exec_start = now;
- rq->exec_clock += delta_exec;
+ 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->delta_fair_clock += delta_fair;
- rq->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->load_weight;
    delta_mine += load >> 1; /* rounding */
    do_div(delta_mine, load);

- rq->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
@@ -233,13 +271,13 @@ static inline void update_curr(struct rq
 * 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->wait_start_fair = rq->fair_clock;
+ p->wait_start_fair = lrq->fair_clock;
  p->wait_start = now;
}
```

```
@@ -247,7 +285,7 @@ update_stats_wait_start(struct rq *rq, s
 * 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;
```

```
@@ -255,12 +293,12 @@ update_stats_enqueue(struct rq *rq, stru
 * 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->fair_clock;
+ key = lrq->fair_clock;
```

```
/*
 * Optimize the common nice 0 case:
```

```
@@ -269,9 +307,11 @@ update_stats_enqueue(struct rq *rq, stru
  key -= p->wait_runtime;
  else {
    if (p->wait_runtime < 0)
-   key -= div64_s(p->wait_runtime * NICE_0_LOAD, p->load_weight);
+   key -= div64_s(p->wait_runtime * NICE_0_LOAD,
+   p->load_weight);
  else
-   key -= div64_s(p->wait_runtime * p->load_weight, NICE_0_LOAD);
+   key -= div64_s(p->wait_runtime * p->load_weight,
```

```

+     NICE_0_LOAD);
}

p->fair_key = key;
@@ -281,7 +321,7 @@ update_stats_enqueue(struct rq *rq, stru
* 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;

@@ -290,12 +330,12 @@ update_stats_wait_end(struct rq *rq, str
    p->wait_max = delta_wait;

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

    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->wait_start_fair = 0;
@@ -303,22 +343,22 @@ update_stats_wait_end(struct rq *rq, str
}

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 dequeuing 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:
@@ -330,7 +370,7 @@ update_stats_curr_start(struct rq *rq, s
* 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->exec_start = 0;
}
@@ -345,50 +385,53 @@ update_stats_curr_end(struct rq *rq, str
* 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->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 lq *lq, struct sched_entity *p)
{
- unsigned long load = rq->raw_weighted_load;
+ unsigned long load = lq->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->fair_clock - p->sleep_start_fair;
+ delta_fair = lq->fair_clock - p->sleep_start_fair;

/*
 * Fix up delta_fair with the effect of us running
 * during the whole sleep period:
 */
  if (!(sysctl_sched_features & 32))
- delta_fair = div64_s(delta_fair * load, load + p->load_weight);
+ delta_fair = div64_s(delta_fair * load,
+     load + p->load_weight);
  delta_fair = div64_s(delta_fair * p->load_weight, NICE_0_LOAD);

  prev_runtime = p->wait_runtime;
- __add_wait_runtime(rq, p, delta_fair);
+ __add_wait_runtime(lq, p, delta_fair);
  delta_fair = p->wait_runtime - prev_runtime;

/*
@@ -396,28 +439,23 @@ static void enqueue_sleeper(struct rq *r
 * amount of the new wait_runtime this task adds to
 * the 'pool':
 */
- distribute_fair_add(rq, delta_fair);
+ distribute_fair_add(lq, delta_fair);

out:
- rq->wait_runtime += p->wait_runtime;
+ lq->wait_runtime += p->wait_runtime;

  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->sleep_start) {
@@ -443,10 +481,152 @@ enqueue_task_fair(struct rq *rq, struct
            p->sum_sleep_runtime += delta;

            if (p->sleep_start_fair)
                - enqueue_sleeper(rq, p);
                + enqueue_sleeper(lrq, 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->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;
+ }
+ }
- update_stats_enqueue(rq, p, now);
- __enqueue_task_fair(rq, p);
+ __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);
+}

/*
@@ -457,16 +637,10 @@ enqueue_task_fair(struct rq *rq, struct
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->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;
- }
- __dequeue_task_fair(rq, p);
+ struct lrq *lrq = task_lrq(p);

```

```

+ struct sched_entity *se = &p->se;
+
+ dequeue_entity(lrq, se, sleep, now);
}

/*
@@ -479,16 +653,18 @@ yield_task_fair(struct rq *rq, struct ta
{
    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->on_rq = 0;
- enqueue_task_fair(rq, p, 0, now);
- p->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
@@ -496,39 +672,23 @@ yield_task_fair(struct rq *rq, struct ta
    */
    if (p_to && rq == task_rq(p_to) &&
        p_to->sched_class == &fair_sched_class
-    && p->wait_runtime > 0) {
+    && p->se.wait_runtime > 0) {

- s64 delta = p->wait_runtime >> 1;
+ 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->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(curr);
-}

/*
 * Preempt the current task with a newly woken task if needed:
@@ -536,12 +696,13 @@ __check_preempt_curr_fair(struct rq *rq,
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 {
@@ -552,25 +713,18 @@ static void check_preempt_curr_fair(stru
  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);
}

/*
@@ -578,32 +732,13 @@ static struct task_struct * pick_next_ta
*/
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->on_rq &&
-       test_tsk_thread_flag(prev, TIF_NEED_RESCHED)) {
-
-     dequeue_task_fair(rq, prev, 0, now);
-     prev->on_rq = 0;
-     enqueue_task_fair(rq, prev, 0, now);
-     prev->on_rq = 1;

```



```

- } else
- update_curr(rq, now);
- } else {
- update_curr(rq, now);
- }
-
- update_stats_curr_end(rq, prev, now);
-
- if (prev->on_rq)
- update_stats_wait_start(rq, prev, now);
+ put_prev_entity(lrq, se, now);
}

/*****/
@@ -625,20 +760,20 @@ __load_balance_iterator(struct rq *rq, s
    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;
}

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)
{
- return __load_balance_iterator(rq, rq->rb_load_balance_curr);
+ return __load_balance_iterator(rq, rq->lrq.rb_load_balance_curr);
}

/*
@@ -646,31 +781,10 @@ static struct task_struct * load_balance
*/
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->on_rq = 0;
- enqueue_task_fair(rq, curr, 0, now);
- curr->on_rq = 1;
+ struct lrq *lrq = task_lrq(curr);
+ struct sched_entity *se = &curr->se;

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

/*
@@ -682,29 +796,32 @@ static void task_tick_fair(struct rq *rq
*/
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->fair_key = current->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,
* so do not credit it with that waiting time yet:
*/

```

```

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

/*
 * The statistical average of wait_runtime is about
 * -granularity/2, so initialize the task with that:
 */
-// p->wait_runtime = -(s64)(sysctl_sched_granularity / 2);
+// p->se.wait_runtime = -(s64)(sysctl_sched_granularity / 2);

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

```

Index: linux/kernel/sched\_rt.c

```

=====
--- linux.orig/kernel/sched_rt.c
+++ linux/kernel/sched_rt.c
@@ -15,14 +15,14 @@ static inline void update_curr_rt(struct
  if (!has_rt_policy(curr))
    return;

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

- curr->sum_exec_runtime += delta_exec;
- curr->exec_start = now;
+ curr->se.sum_exec_runtime += delta_exec;
+ curr->se.exec_start = now;
}

static void
@@ -89,7 +89,7 @@ static struct task_struct * pick_next_ta
  queue = array->queue + idx;
  next = list_entry(queue->next, struct task_struct, run_list);

- next->exec_start = now;
+ next->se.exec_start = now;

```

```

    return next;
}
@@ -97,7 +97,7 @@ static struct task_struct * pick_next_ta
static void put_prev_task_rt(struct rq *rq, struct task_struct *p, u64 now)
{
    update_curr_rt(rq, now);
- p->exec_start = 0;
+ p->se.exec_start = 0;
}

/*

```

---

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

---

## File Attachments

1) [sched-cfs-v17-rc4.patch](#), downloaded 269 times

---



---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1  
Posted by [Ingo Molnar](#) on Mon, 11 Jun 2007 19:39:31 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

\* Ingo Molnar <mingo@elte.hu> wrote:

> > Patch 4 fixes some bad interaction between SCHED\_RT and SCHED\_NORMAL  
> > tasks in current CFS.  
>  
> btw., the plan here is to turn off 'bit 0' in sched\_features: i.e. to  
> use the precise statistics to calculate lrq->cpu\_load[], not the  
> timer-irq-sampled imprecise statistics. [...]

i mean bit 6, value 64. I flipped around its meaning in -v17-rc4, so the  
new precise stats code there is now default-enabled - making SMP  
load-balancing more accurate.

Ingo

---

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

---



---

Subject: Re: [ckrm-tech] [RFC][PATCH 1/6] Introduce struct sched\_entity and struct lrq

Posted by [Balbir Singh](#) on Tue, 12 Jun 2007 02:15:59 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

Srivatsa Vaddagiri wrote:

> This patch introduces two new structures:

>

> struct sched\_entity

>     stores essential attributes/execution-history used by CFS core

>     to drive fairness between 'schedulable entities' (tasks, users etc)

>

> struct lrq

>     runqueue used to hold ready-to-run entities

>

> These new structures are formed by grouping together existing fields in

> existing structures (task\_struct and rq) and hence represents rework

> with zero functionality change.

>

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

[snip]

>

> +/\* 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, delta\_fair\_clock;

> + u64 exec\_clock, delta\_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;

> +};

> +

Shouldn't the rq->lock move into lrq?

--

Warm Regards,

Balbir Singh

Linux Technology Center

IBM, ISTL

---

Containers mailing list

[Containers@lists.linux-foundation.org](mailto:Containers@lists.linux-foundation.org)

---

Subject: Re: [ckrm-tech] [RFC][PATCH 2/6] task's cpu information needs to be always correct

Posted by [Balbir Singh](#) on Tue, 12 Jun 2007 02:17:08 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

Srivatsa Vaddagiri wrote:

> We rely very much on task\_cpu(p) to be correct at all times, so that we  
> can correctly find the runqueue 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@linux.vnet.ibm.com>

>

Acked-by: Balbir Singh <balbir@linux.vnet.ibm.com>

> ---

> kernel/sched.c | 8 ++++++---

> 1 files changed, 5 insertions(+), 3 deletions(-)

>

> Index: current/kernel/sched.c

> =====

> --- current.orig/kernel/sched.c 2007-06-09 15:07:17.000000000 +0530

> +++ current/kernel/sched.c 2007-06-09 15:07:32.000000000 +0530

> @@ -4624,7 +4624,7 @@

> 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;

> @@ -4640,9 +4640,11 @@

> if (!cpu\_isset(dest\_cpu, p->cpus\_allowed))

> goto out;

>

> - set\_task\_cpu(p, dest\_cpu);

> - if (p->se.on\_rq) {

> + on\_rq = p->se.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);
> }
```

--

Warm Regards,  
Balbir Singh  
Linux Technology Center  
IBM, ISTL

---

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

---

---

Subject: Re: [RFC][PATCH 3/6] core changes in CFS  
Posted by [Balbir Singh](#) on Tue, 12 Jun 2007 02:29:22 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Srivatsa Vaddagiri wrote:  
> +#define entity\_is\_task(se) 1

Could you add some comments as to what this means? Should be it boolean instead (true)

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

Enqueue an entity

```
> -static void limit_wait_runtime(struct rq *rq, struct task_struct *p)
> +static void limit_wait_runtime(struct lrq *lrq, struct sched_entity *p)
```

p is a general convention for tasks in the code, could we use something different -- may be "e"?

```
>
> static s64 div64_s(s64 dividend, 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);

```

How about curr\_entity?

```

> + 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)

```

Can !curr ever be true? shouldn't we look into the sched\_class of the task that the entity belongs to?

--

Warm Regards,  
Balbir Singh  
Linux Technology Center  
IBM, ISTL

---

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

---

Subject: Re: [ckrm-tech] [RFC][PATCH 1/6] Introduce struct sched\_entity and struct lrq

Posted by [Srivatsa Vaddagiri](#) on Tue, 12 Jun 2007 03:52:45 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On Tue, Jun 12, 2007 at 07:45:59AM +0530, Balbir Singh wrote:

```

> > +/* 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, delta_fair_clock;
> > + u64 exec_clock, delta_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;
> > +};
> > +
>
> Shouldn't the rq->lock move into irq?
```

Right now, the per-cpu rq lock protects all (local) runqueues attached with the cpu. At some point, for scalability reasons, we may want to split that to be per-cpu per-local runqueue (as you point out). I will put that in my todo list of things to consider. Thanks for the review!

--  
Regards,  
vatsa

---

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

---

---

Subject: Re: [RFC][PATCH 3/6] core changes in CFS  
Posted by [Srivatsa Vaddagiri](#) on Tue, 12 Jun 2007 04:22:47 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Tue, Jun 12, 2007 at 07:59:22AM +0530, Balbir Singh wrote:  
> > + #define entity\_is\_task(se) 1  
>  
> Could you add some comments as to what this means?

sure. Basically this macro tests whether a given schedulable entity is task or not. Other possible schedulable entities could be process, user, container etc. These various entities form a hierarchy with task being at the bottom of the hierarchy.

> Should be it boolean instead (true)

I don't have a good opinion on this. Would it make sparse friendly?

```
> > + * Enqueue a entity into the rb-tree:
>
> Enqueue an entity
```

yes

```
>
> > -static void limit_wait_runtime(struct rq *rq, struct task_struct *p)
> > +static void limit_wait_runtime(struct lrq *lrq, struct sched_entity *p)
>
> p is a general convention for tasks in the code, could we use something
> different -- may be "e"?
```

'se' perhaps as is used elsewhere. I avoided making that change so that people will see less diff o/p in the patch :) I agree though a better name is needed.

```
> > static s64 div64_s(s64 dividend, 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);
>
> How about curr_entity?
```

I prefer its current name, but will consider your suggestion in next iteration.

```
> > + 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)
>
> Can !curr ever be true? shouldn't we look into the sched_class of the task
> that the entity belongs to?
```

Couple of cases that we need to consider here:

CONFIG\_FAIR\_GROUP\_SCHED disabled:

lrq\_curr() essentially returns NULL if currently running task doesn't belong to fair\_sched\_class, else it returns &rq->curr->se. So the check for fair\_sched\_class is taken care in that function.

CONFIG\_FAIR\_GROUP\_SCHED enabled:

Irq\_curr() returns Irq->curr. I introduced ->curr field in Irq to optimize on not having to update Irq's fair\_clock (update\_curr upon enqueue/dequeue task) if it was not currently "active".

Lets say that there are two groups 'vatsa' and 'guest' with their own Irqs on each cpu. If CPU0 is currently running a task from group 'vatsa', then Irq\_vatsa->curr will point to the currently running task, while Irq\_guest->curr will be NULL. While the task from 'vatsa' is running, if we were to enqueue/dequeue task from group 'guest', we need not update Irq\_guest's fair\_clock (as it is not active currently). This optimization in update\_curr is made possible by maintaining a 'curr' field in Irq.

Hope this answers your question.

--

Regards,  
vatsa

---

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

---

---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1  
Posted by [Srivatsa Vaddagiri](#) on Tue, 12 Jun 2007 05:50:24 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Mon, Jun 11, 2007 at 09:37:35PM +0200, Ingo Molnar wrote:

> > Patches 1-3 introduce the essential changes in CFS core to support  
> > this concept. They rework existing code w/o any (intended!) change in  
> > functionality.  
>  
> i currently have these 3 patches applied to the CFS queue and it's  
> looking pretty good so far! If it continues to be problem-free i'll  
> release them as part of -v17, just to check that they truly have no bad  
> side-effects (they shouldnt). Then #4 can go into -v18.

ok. I am also most concerned about not upsetting current performance of CFS when CONFIG\_FAIR\_GROUP\_SCHED is turned off. Staging these patches in incremental versions of CFS is a good idea to test that.

> i've attached my current -v17 tree - it should apply mostly cleanly

> ontop of the -mm queue (with a minor number of fixups). Could you  
> refactor the remaining 3 patches ontop of this base? There's some  
> rejects in the last 3 patches due to the update\_load\_fair() change.

sure, i will rework them on this -v17 snapshot.

> > Patch 4 fixes some bad interaction between SCHED\_RT and SCHED\_NORMAL  
> > tasks in current CFS.

>

> btw., the plan here is to turn off 'bit 0' in sched\_features: i.e. to  
> use the precise statistics to calculate lrq->cpu\_load[], not the  
> timer-irq-sampled imprecise statistics. Dmitry has fixed a couple of  
> bugs in it that made it not work too well in previous CFS versions, but  
> now we are ready to turn it on for -v17. (indeed in my tree it's already  
> turned on - i.e. sched\_features defaults to '14')

On Mon, Jun 11, 2007 at 09:39:31PM +0200, Ingo Molnar wrote:

> i mean bit 6, value 64. I flipped around its meaning in -v17-rc4, so the  
> new precise stats code there is now default-enabled - making SMP  
> load-balancing more accurate.

I must be missing something here. AFAICS, cpu\_load calculation still is  
timer-interrupt driven in the -v17 snapshot you sent me. Besides, there  
is no change in default value of bit 6 b/n v16 and v17:

```
-unsigned int sysctl_sched_features __read_mostly = 1 | 2 | 4 | 8 | 0 | 0;  
+unsigned int sysctl_sched_features __read_mostly = 0 | 2 | 4 | 8 | 0 | 0;
```

So where's this precise stats based calculation of cpu\_load?

Anyway, do you agree that splitting the cpu\_load/nr\_running fields so that:

```
rq->nr_running      = total count of -all- tasks in runqueue  
rq->raw_weighted_load = total weight of -all- tasks in runqueue  
rq->lrq.nr_running   = total count of SCHED_NORMAL/BATCH tasks in runqueue  
rq->lrq.raw_weighted_load = total weight of SCHED_NORMAL/BATCH tasks in runqueue
```

is a good thing to avoid SCHED\_RT<->SCHED\_NORMAL/BATCH mixup (as accomplished  
in Patch #4)?

If you don't agree, then I will make this split dependent on  
CONFIG\_FAIR\_GROUP\_SCHED

> > Patch 5 introduces basic changes in CFS core to support group  
> > fairness.

> >

> > Patch 6 hooks up scheduler with container patches in mm (as an  
> > interface for task-grouping functionality).

Just to be clear, by container patches, I am referring to "process" container patches from Paul Menage [1]. They aren't necessarily tied to "virtualization-related" container support in -mm tree, although I believe that "virtualization-related" container patches will make use of the same "process-related" container patches for their task-grouping requirements. Phew ..we need better names!

> ok. Kirill, how do you like Srivatsa's current approach? Would be nice  
> to kill two birds with the same stone, if possible :-)

One thing the current patches don't support is the notion of virtual cpus (which Kirill and other "virtualization-related" container folks would perhaps want). IMHO, the current patches can still be usefull for containers to load balance between those virtual cpus (as and when it is introduced).

> you'll get the best hackbench results by using SCHED\_BATCH:  
>  
> chrt -b 0 ./hackbench 10

thanks for this tip. Will try out and let you know how it fares for me.

> or indeed increasing the runtime\_limit would work too.

References:

1. <https://lists.linux-foundation.org/pipermail/containers/2007-May/005261.html>

--

Regards,  
vatsa

---

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

---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1  
Posted by [Ingo Molnar](#) on Tue, 12 Jun 2007 06:26:12 GMT  
[View Forum Message](#) <> [Reply to Message](#)

\* Srivatsa Vaddagiri <[vatsa@linux.vnet.ibm.com](mailto:vatsa@linux.vnet.ibm.com)> wrote:

> On Mon, Jun 11, 2007 at 09:39:31PM +0200, Ingo Molnar wrote:  
> > i mean bit 6, value 64. I flipped around its meaning in -v17-rc4, so the  
> > new precise stats code there is now default-enabled - making SMP

> > load-balancing more accurate.  
>  
> I must be missing something here. AFAICS, cpu\_load calculation still  
> is timer-interrupt driven in the -v17 snapshot you sent me. Besides,  
> there is no change in default value of bit 6 b/n v16 and v17:  
>  
> -unsigned int sysctl\_sched\_features \_\_read\_mostly = 1 | 2 | 4 | 8 | 0 | 0;  
> +unsigned int sysctl\_sched\_features \_\_read\_mostly = 0 | 2 | 4 | 8 | 0 | 0;  
>  
> So where's this precise stats based calculation of cpu\_load?

but there's a change in the interpretation of bit 6:

```
-    if (!(sysctl_sched_features & 64)) {
-        this_load = this_rq->raw_weighted_load;
+    if (sysctl_sched_features & 64) {
+        this_load = this_rq->lrq.raw_weighted_load;
```

the update of the cpu\_load[] value is timer interrupt driven, but the \_value\_ that is sampled is not. Previously we used ->raw\_weighted\_load (at whatever value it happened to be at the moment the timer irq hit the system), now we basically use a load derived from the fair-time passed since the last scheduler tick. (Mathematically it's close to an integral of load done over that period) So it takes all scheduling activities and all load values into account to calculate the average, not just the value that was sampled by the scheduler tick.

this, besides being more precise (it for example correctly samples short-lived, timer-interrupt-driven workloads too, which were largely 'invisible' to the previous load calculation method), also enables us to make the scheduler tick hrtimer based in the (near) future. (in essence making the scheduler tick-less even when there are tasks running)

> Anyway, do you agree that splitting the cpu\_load/nr\_running fields so  
> that:  
>  
> rq->nr\_running = total count of -all- tasks in runqueue  
> rq->raw\_weighted\_load = total weight of -all- tasks in runqueue  
> rq->lrq.nr\_running = total count of SCHED\_NORMAL/BATCH tasks in runqueue  
> rq->lrq.raw\_weighted\_load = total weight of SCHED\_NORMAL/BATCH tasks in runqueue  
>  
> is a good thing to avoid SCHED\_RT<->SCHED\_NORMAL/BATCH mixup (as  
> accomplished in Patch #4)?

yes, i agree in general, even though this causes some small overhead.  
This also has another advantage: the inter-policy isolation and load balancing is similar to what fair group scheduling does, so even 'plain' Linux will use the majority of the framework.

> If you don't agree, then I will make this split dependent on  
> CONFIG\_FAIR\_GROUP\_SCHED

no, i'd rather avoid that #ifdefery.

> > Patch 6 hooks up scheduler with container patches in mm (as an  
> > interface for task-grouping functionality).

>

> Just to be clear, by container patches, I am referring to "process"  
> container patches from Paul Menage [1]. They aren't necessarily tied  
> to "virtualization-related" container support in -mm tree, although I  
> believe that "virtualization-related" container patches will make use  
> of the same "process-related" container patches for their  
> task-grouping requirements. Phew ..we need better names!

i'd still like to hear back from Kirill & co whether this framework is  
flexible enough for their work (OpenVZ, etc.) too.

Ingo

---

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

---

---

Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED\_RT and  
SCHED\_NORMAL tasks

Posted by [Dmitry Adamushko](#) on Tue, 12 Jun 2007 09:03:36 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On 11/06/07, Srivatsa Vaddagiri <[vatsa@linux.vnet.ibm.com](mailto:vatsa@linux.vnet.ibm.com)> wrote:

> Currently nr\_running and raw\_weighted\_load fields in runqueue affect  
> some CFS calculations (like distribute\_fair\_add, enqueue\_sleeper etc).

[ briefly looked.. a few comments so far ]

(1)

I had an idea of per-sched-class 'load balance' calculator. So that  
update\_load() (as in your patch) would look smth like :

...

```
struct sched_class *class = sched_class_highest;  
unsigned long total = 0;
```

```
do {
```

```

        total += class->update_load(..., now);
        class = class->next;
    } while (class);
...

```

and e.g. `update_load_fair()` would become a `fair_sched_class :: update_load()`.

That said, all the `sched_classes` would report a load created by their entities (tasks) over the last sampling period. Ideally, the calculation should not be merely based on the 'raw\_weighted\_load' but rather done in a similar way to `update_load_fair()` as in v17.

I'll take a look at how it can be mapped on the current v17 codebase (including your patches #1-3) and come up with some real code so we would have a base for discussion.

(2)

```

> 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);
>
> +     /* replay load smoothening for all ticks we lost */
> +     while (time_after_eq64(now, lrq->last_tick)) {
> +         update_load_fair(lrq);
> +         lrq->last_tick += TICK_NSEC;
> +     }

```

I think, it won't work properly this way. The first call returns a load for last `TICK_NSEC` and all the consequent ones report zero load ('this\_load = 0' internally).. as a result, we will get a lower load than it likely was.

I guess, `update_load_fair()` (as it's in v17) could be slightly changed to report the load for an interval of time over which the load statistics have been accumulated (`delta_exec_time` and `fair_exec_time`):

```
update_load_fair(lrq, now - lrq->last_tick)
```

This new (second) argument would be used instead of `TICK_NSEC` (internally in `update_load_fair()`) ... but again, I'll come up with some code for further discussion.

```
> --
```



> Regards,  
> vatsa  
>

--

Best regards,  
Dmitry Adamushko

---

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

---

---

Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED\_RT and SCHED\_NORMAL tasks

Posted by [Srivatsa Vaddagiri](#) on Tue, 12 Jun 2007 10:26:22 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On Tue, Jun 12, 2007 at 11:03:36AM +0200, Dmitry Adamushko wrote:

> I had an idea of per-sched-class 'load balance' calculator. So that  
> update\_load() (as in your patch) would look smth like :  
>  
> ...  
> struct sched\_class \*class = sched\_class\_highest;  
> unsigned long total = 0;  
>  
> do {  
> total += class->update\_load(..., now);  
> class = class->next;  
> } while (class);  
> ...  
>  
> and e.g. update\_load\_fair() would become a fair\_sched\_class ::  
> update\_load().  
>  
> That said, all the sched\_classes would report a load created by their  
> entities (tasks) over the last sampling period. Ideally, the  
> calculation should not be merely based on the 'raw\_weighted\_load' but  
> rather done in a similar way to update\_load\_fair() as in v17.

I like this idea. It neatly segregates load calculation across classes.  
It effectively replaces what update\_load() function I introduced in  
Patch #4.

Btw what will update\_load\_rt() return?

> > static void entity\_tick(struct Irq \*Irq, struct sched\_entity \*curr)  
> > {

```

> > struct sched_entity *next;
> > struct rq *rq = lrq_rq(lrq);
> > u64 now = __rq_clock(rq);
> >
> >+ /* replay load smoothening for all ticks we lost */
> >+ while (time_after_eq64(now, lrq->last_tick)) {
> >+     update_load_fair(lrq);
> >+     lrq->last_tick += TICK_NSEC;
> >+ }
>
> I think, it won't work properly this way. The first call returns a
> load for last TICK_NSEC and all the consequent ones report zero load
> ('this_load = 0' internally)..

```

mm ..

```

exec_delta64 = this_lrq->delta_exec_clock + 1;
this_lrq->delta_exec_clock = 0;

```

So exec\_delta64 (and fair\_delta64) should be min 1 in successive calls. How can that lead to this\_load = 0?

The idea behind 'replay lost ticks' is to avoid load smoothening of -every- lrq -every- tick. Lets say that there are ten lrqs (corresponding to ten different users). We load smoothen only the currently active lrq (whose task is currently running). Other lrqs load get smoothened as soon as they become active next time (thus catching up with all lost ticks).

--  
Regards,  
vatsa

---

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

---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1  
Posted by [Srivatsa Vaddagiri](#) on Tue, 12 Jun 2007 10:56:37 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

[ resending ..my earlier reply doesn't seem to have made it to lkml ]

On Tue, Jun 12, 2007 at 08:26:12AM +0200, Ingo Molnar wrote:  
> > So where's this precise stats based calculation of cpu\_load?  
>  
> but there's a change in the interpretation of bit 6:

```
>
> -   if (!(sysctl_sched_features & 64)) {
> -       this_load = this_rq->raw_weighted_load;
> +   if (sysctl_sched_features & 64) {
> +       this_load = this_rq->lrq.raw_weighted_load;
>
> the update of the cpu_load[] value is timer interrupt driven, but the
> _value_ that is sampled is not. [...]
```

Ah ..ok. Should have realized it earlier. Thanks for the education, but:

```
> Previously we used ->raw_weighted_load
> (at whatever value it happened to be at the moment the timer irq hit the
> system), now we basically use a load derived from the fair-time passed
> since the last scheduler tick. [...]
```

Isn't that biasing the overall cpu load to be dependent on SCHED\_NORMAL task load (afaics update\_curr\_rt doesn't update fair\_clock at all)?

What if a CPU had just real-time tasks and no SCHED\_NORMAL/BATCH tasks? Would the cpu\_load be seen to be very low?

[ Dmitry's proposal for a per-class update\_load() callback seems to be a good thing in this regard ]

```
> > Just to be clear, by container patches, I am referring to "process"
> > container patches from Paul Menage [1]. They aren't necessarily tied
> > to "virtualization-related" container support in -mm tree, although I
> > believe that "virtualization-related" container patches will make use
> > of the same "process-related" container patches for their
> > task-grouping requirements. Phew ..we need better names!
>
> i'd still like to hear back from Kirill & co whether this framework is
> flexible enough for their work (OpenVZ, etc.) too.
```

sure .. i would love to hear their feedback as well on the overall approach of these patches, which is:

1. Using Paul Menage's process container patches as the basis of task-grouping functionality. I think there is enough consensus on this already

(more importantly)

2. Using CFS core to achieve fairness at higher hierarchical levels (including at a container level). It would be nice to reuse much of the CFS logic which is driving fairness between tasks currently.

3. Using smpnice mechanism for SMP load-balance between CPUs  
(also largely based on what is there currently in CFS). Basic idea behind  
this is described at <http://lkml.org/lkml/2007/5/25/146>

Kirill/Herbert/Eric?

--  
Regards,  
vatsa

--  
Regards,  
vatsa

---

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

---

---

Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED\_RT and  
SCHED\_NORMAL tasks

Posted by [Dmitry Adamushko](#) on Tue, 12 Jun 2007 12:23:38 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On 12/06/07, Srivatsa Vaddagiri <[vatsa@linux.vnet.ibm.com](mailto:vatsa@linux.vnet.ibm.com)> wrote:

> On Tue, Jun 12, 2007 at 11:03:36AM +0200, Dmitry Adamushko wrote:

> > I had an idea of per-sched-class 'load balance' calculator. So that

> > update\_load() (as in your patch) would look smth like :

> >

> > ...

> > struct sched\_class \*class = sched\_class\_highest;

> > unsigned long total = 0;

> >

> > do {

> >       total += class->update\_load(..., now);

> >       class = class->next;

> > } while (class);

> > ...

> >

> > and e.g. update\_load\_fair() would become a fair\_sched\_class ::

> > update\_load().

> >

> > That said, all the sched\_classes would report a load created by their

> > entities (tasks) over the last sampling period. Ideally, the

> > calculation should not be merely based on the 'raw\_weighted\_load' but

> > rather done in a similar way to update\_load\_fair() as in v17.

>

> I like this idea. It neatly segregates load calculation across classes.

> It effectively replaces what update\_load() function I introduced in  
> Patch #4.

Good.

(a minor disclaimer :)

We discussed it a bit with Ingo and I don't remember who first expressed this idea in written words (although I seem to remember, I did have it in mind before -- it's not rocket science after all :)

>  
> Btw what will update\_load\_rt() return?

Well, as a `_temporary_` stub - just return the 'raw\_weighted\_load' contributed by the RT tasks..

Ideally, we'd like a similar approach to the `update_fair_load()` -- i.e. we need the run-time history of `rt_sched_class`'s (like of any other class) tasks over the last sampling period, so e.g. we do account periodic RT tasks which happen to escape accounting through 'raw\_weighted\_load' due to the fact that they are not active at the moment of timer interrupts (when 'raw\_weighted\_load' snapshots are taken).

```
>
> > static void entity_tick(struct irq *irq, struct sched_entity *curr)
> > {
> >     struct sched_entity *next;
> >     struct rq *rq = irq_rq(irq);
> >     u64 now = __rq_clock(rq);
> >
> >+    /* replay load smoothening for all ticks we lost */
> >+    while (time_after_eq64(now, irq->last_tick)) {
> >+        update_load_fair(irq);
> >+        irq->last_tick += TICK_NSEC;
> >+    }
> >
> > I think, it won't work properly this way. The first call returns a
> > load for last TICK_NSEC and all the consequent ones report zero load
> > ('this_load = 0' internally)..
>
> mm ..
>
>     exec_delta64 = this_irq->delta_exec_clock + 1;
>     this_irq->delta_exec_clock = 0;
>
> So exec_delta64 (and fair_delta64) should be min 1 in successive calls. How can that lead to
this_load = 0?
```

just substitute {exec,fair}\_delta == 1 in the following code:

```
tmp64 = SCHED_LOAD_SCALE * exec_delta64;
do_div(tmp64, fair_delta);
tmp64 *= exec_delta64;
do_div(tmp64, TICK_NSEC);
this_load = (unsigned long)tmp64;
```

we'd get

```
tmp64 = 1024 * 1;
tmp64 /= 1;
tmp64 *= 1;
tmp64 /= 1000000;
```

as a result, this\_load = 1024/1000000; which is 0 (no floating point calc.).

> The idea behind 'replay lost ticks' is to avoid load smoothening of  
> -every- Irq -every- tick. Lets say that there are ten Irqs  
> (corresponding to ten different users). We load smoothen only the currently  
> active Irq (whose task is currently running).

The raw idea behind update\_load\_fair() is that it evaluates the run-time history between 2 consequent calls to it (which is now at timer freq. --- that's a sapling period). So if you call update\_fair\_load() in a loop, the sampling period is actually an interval between 2 consequent calls. IOW, you can't say "3 ticks were lost" so at first evaluate the load for the first tick, then the second one, etc. ...

Anyway, I'm missing the details regarding the way you are going to do per-group 'load balancing' so refrain from further commenting so far... it's just that the current implementation of update\_load\_fair() is unlikely to work as you expect in your 'replay lost ticks' loop :-)

> Other Irqs load get smoothened  
> as soon as they become active next time (thus catching up with all lost ticks).

Ok, let's say user1 tasks were highly active till T1 moment of time..  
cpu\_load[] of user's Irq  
has accumulated this load.  
now user's tasks were not active for an interval of dT.. so you don't  
update its cpu\_load[] in the mean time? Let's say 'load balancing'  
takes place at the moment T2 = T1 + dT

Are you going to do any 'load balancing' between users? Based on what?  
If it's user's Irq :: cpu\_load[] .. then it \_still\_ shows the load at  
the moment of T1 while we are at the moment T2 (and user1 was not

active during dT)..

>  
> --  
> Regards,  
> vatsa  
>

--  
Best regards,  
Dmitry Adamushko

---

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

---

---

Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED\_RT and SCHED\_NORMAL tasks  
Posted by [Srivatsa Vaddagiri](#) on Tue, 12 Jun 2007 13:30:45 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Tue, Jun 12, 2007 at 02:23:38PM +0200, Dmitry Adamushko wrote:

> >mm ..  
> >  
> >     exec\_delta64 = this\_lrq->delta\_exec\_clock + 1;  
> >     this\_lrq->delta\_exec\_clock = 0;  
> >  
> >So exec\_delta64 (and fair\_delta64) should be min 1 in successive calls.  
> >How can that lead to this\_load = 0?  
>  
> just substitute {exec,fair}\_delta == 1 in the following code:  
>  
>     tmp64 = SCHED\_LOAD\_SCALE \* exec\_delta64;  
>     do\_div(tmp64, fair\_delta);  
>     tmp64 \*= exec\_delta64;  
>     do\_div(tmp64, TICK\_NSEC);  
>     this\_load = (unsigned long)tmp64;  
>  
> we'd get  
>  
>     tmp64 = 1024 \* 1;  
>     tmp64 /= 1;  
>     tmp64 \*= 1;  
>     tmp64 /= 1000000;  
>  
> as a result, this\_load = 1024/1000000; which is 0 (no floating point calc.).

Ok ..

But isn't that the same result we would have obtained anyways had we called `update_load_fair()` on all `Irq`'s on every timer tick? If a user's `Irq` was inactive for several ticks, then its `exec_delta` will be seen as zero for those several ticks, which means we would compute its '`this_load`' to be zero as well for those several ticks?

Basically what I want to know is, are we sacrificing any accuracy here because of "deferring" smoothening of `cpu_load` for a (inactive) `Irq` (apart from the inaccurate figure used during `load_balance` as you point out below).

> >The idea behind 'replay lost ticks' is to avoid load smoothening of  
> >-every- `Irq` -every- tick. Lets say that there are ten `Irqs`  
> >(corresponding to ten different users). We load smoothen only the currently  
> >active `Irq` (whose task is currently running).  
>  
> The raw idea behind `update_load_fair()` is that it evaluates the  
> run-time history between 2 consequent calls to it (which is now at  
> timer freq. --- that's a sapling period). So if you call  
> `update_fair_load()` in a loop, the sampling period is actually an  
> interval between 2 consequent calls. IOW, you can't say "3 ticks were  
> lost" so at first evaluate the load for the first tick, then the  
> second one, etc. ...

Assuming the `Irq` was inactive for all those 3 ticks and became active at 4th tick, would the end result of `cpu_load` (as obtained in my code) be any different than calling `update_load_fair()` on all `Irq` on each tick?

> Anyway, I'm missing the details regarding the way you are going to do  
> per-group 'load balancing' so refrain from further commenting so  
> far... it's just that the current implementation of `update_load_fair()`  
> is unlikely to work as you expect in your 'replay lost ticks' loop :-)

Even though this lost ticks loop is easily triggered with user-based `Irqs`, I think the same "loop" can be seen in current CFS code (i.e say v16) when low level timer interrupt handler replays such lost timer ticks (say we were in a critical section for some time with timer interrupt disabled). As an example see `arch/powerpc/kernel/time.c:timer_interrupt()` calling `account_process_time->scheduler_tick` in a loop.

If there is any bug in 'replay lost ticks' loop in the patch I posted, then it should already be present in current (i.e v16) implementation of `update_load_fair()`?

> >Other `Irqs` load get smoothened



> >as soon as they become active next time (thus catching up with all lost  
> >ticks).  
>  
> Ok, let's say user1 tasks were highly active till T1 moment of time..  
> cpu\_load[] of user's Irq  
> has accumulated this load.  
> now user's tasks were not active for an interval of dT.. so you don't  
> update its cpu\_load[] in the mean time? Let's say 'load balancing'  
> takes place at the moment T2 = T1 + dT  
>  
> Are you going to do any 'load balancing' between users? Based on what?

Yes, patch #5 introduces group-aware load-balance. It is two-step:

First, we identify busiest group and busiest queue, based on  
rq->raw\_weighted\_load/cpu\_load (which is accumulation of weight from all  
clases on a CPU). This part of the code is untouched.

Next when loadbalancing between two chosen CPUs (busiest and this cpu),  
move\_tasks() is iteratively called on each user/group's Irq on both cpus, with  
the max\_load\_move argument set to 1/2 the imabalnce between that user's Irqs  
on both cpus. For this Irq imbalance calculation, I was using  
Irq->raw\_weighted\_load from both cpus, though I agree using  
Irq->cpu\_load is a better bet.

> If it's user's Irq :: cpu\_load[] .. then it \_still\_ shows the load at  
> the moment of T1 while we are at the moment T2 (and user1 was not  
> active during dT)..

Good point. So how do we solve this? I really really want to avoid  
running update\_load\_fair() on all Irq's every tick (it will be a massive  
overhead). I am assuming that Irqs don't remain inactive for a long time  
(given CFS's fairness promise!) and hence probably their cpu\_load[] also  
won't be -that- stale in practice?

--

Regards,  
vatsa

---

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

---

---

Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED\_RT and  
SCHED\_NORMAL tasks

---

On 12/06/07, Srivatsa Vaddagiri <vatsa@linux.vnet.ibm.com> wrote:

> > [ ... ]

> >

> > just substitute {exec,fair}\_delta == 1 in the following code:

> >

> > tmp64 = SCHED\_LOAD\_SCALE \* exec\_delta64;

> > do\_div(tmp64, fair\_delta);

> > tmp64 \*= exec\_delta64;

> > do\_div(tmp64, TICK\_NSEC);

> > this\_load = (unsigned long)tmp64;

> >

> > we'd get

> >

> > tmp64 = 1024 \* 1;

> > tmp64 /= 1;

> > tmp64 \*= 1;

> > tmp64 /= 1000000;

> >

> > as a result, this\_load = 1024/1000000; which is 0 (no floating point calc.).

>

> Ok ..

>

> But isn't that the same result we would have obtained anyways had we

> called update\_load\_fair() on all Irq's on every timer tick? If a user's

> Irq was inactive for several ticks, then its exec\_delta will be seen as

> zero for those several ticks, which means we would compute its 'this\_load' to be

> zero as well for those several ticks?

Yeah.. seems to be so. But let's consider whether these 'inactive ticks' are really inactive [1] :

The fact that user's tasks are not active at the moment of a timer interrupt doesn't mean

they were not active \_during\_ the last tick. That's why another

approach in update\_load\_fair() which doesn't depend on a snapshot of

rq->raw\_weighted\_load

at timer tick's time. I guess, we'd lose this with 'inactive ticks',

right? ok, maybe

it's not that important for per-user cpu\_load, duno at the moment.

>

> Basically what I want to know is, are we sacrificing any accuracy here

> because of "deferring" smoothening of cpu\_load for a (inactive) Irq

> (apart from the inaccurate figure used during load\_balance as you point

> out below).

At least, we are getting some inaccuracy (not in a generic case though) due to the

```
if (exec_delta64 > (u64)TICK_NSEC)
    exec_delta64 = (u64)TICK_NSEC;  [*]
```

in `update_load_fair()`.. and that's smth I want to try changing...

>

> Assuming the Irq was inactive for all those 3 ticks and became active at  
> 4th tick, would the end result of `cpu_load` (as obtained in my code) be  
> any different than calling `update_load_fair()` on all Irq on each tick?

With the current code, yes - it may be. In case, [\*] condition (see above) comes into play (and these 'inactive' ticks were not really inactive as described above).

> Even though this lost ticks loop is easily triggered with user-based Irqs,  
> I think the same "loop" can be seen in current CFS code (i.e say v16)  
> when low level timer interrupt handler replays such lost timer ticks (say we  
> were in a critical section for some time with timer interrupt disabled).  
> As an example see `arch/powerpc/kernel/time.c:timer_interrupt()` calling  
> `account_process_time->scheduler_tick` in a loop.  
>  
> If there is any bug in 'replay lost ticks' loop in the patch I posted, then  
> it should already be present in current (i.e v16) implementation of  
> `update_load_fair()`?

I think, you are right.

>

> Yes, patch #5 introduces group-aware load-balance. It is two-step:  
>  
> First, we identify busiest group and busiest queue, based on  
> `rq->raw_weighted_load/cpu_load` (which is accumulation of weight from all  
> clases on a CPU). This part of the code is untouched.

I'll take a look (e.g. I guess, we have got a notion of "user's weght"... so does/how a user's weight contribute to his tasks weight.. otherwise, I think, the approach of determining the busiest CPU based only on pure tasks' weight would be wrong.. will look at it first).

> > If it's user's Irq :: `cpu_load[]` .. then it `_still_` shows the load at  
> > the moment of T1 while we are at the moment T2 (and user1 was not  
> > active during dT)..

>  
> Good point. So how do we solve this? I really really want to avoid  
> running `update_load_fair()` on all `Irq`'s every tick (it will be a massive  
> overhead).

yeahh.. have to think about it.

btw, I recall the patch #4 adds some light but noticeable overhead,  
right? did you look at where exactly the overhead comes from?

> I am assuming that `Irqs` don't remain inactive for a long time  
> (given CFS's fairness promise!) and hence probably their `cpu_load[]` also  
> won't be -that- stale in practice?

I guess, it's not only about CFS but about the users' behavior, which  
is something

we can't control and so can't rely on it.

Say, a user was active till the moment `T1` and then just gone.. - all  
his tasks are really inactive.

So at the moment `T2` user's `Irq` :: `cpu_load` will still express the  
situation at the moment `T1`?

As long as user's `Irq` is not involved in 'load balancing', this  
inaccuracy can be revealed only if the info is exported via `/proc`.

But say, user's task becomes finally active after \_a lot\_ of inactive  
ticks (the user came back).. now it's in the `rq` and waiting for its  
turn (which can be easily `> 1` tick).. in the mean time 'load  
balancing' is triggered.. and it considers the old `Irq` :: `cpu_load[]`

...

P.S.

just a personal impression.. I'm quite confused by this '`Irq`' name...  
it looks pretty similar to '`Irq`' (with a big '`i`') and I can't stop  
reading it as '`IRQ`' [ chores: so stop it! ]

would be smth like '`cfs_rq`' or even '`sched_rq`' better? :-)

> --  
> Regards,  
> vatsa

--  
Best regards,  
Dmitry Adamushko

---

Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED\_RT and SCHED\_NORMAL tasks

Posted by [Srivatsa Vaddagiri](#) on Tue, 12 Jun 2007 15:43:32 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On Tue, Jun 12, 2007 at 04:31:38PM +0200, Dmitry Adamushko wrote:

> >But isn't that the same result we would have obtained anyways had we  
> >called update\_load\_fair() on all Irq's on every timer tick? If a user's  
> >Irq was inactive for several ticks, then its exec\_delta will be seen as  
> >zero for those several ticks, which means we would compute its 'this\_load'  
> >to be  
> >zero as well for those several ticks?  
>  
> Yeah.. seems to be so. But let's consider whether these 'inactive ticks' are  
> really inactive [1] :  
>  
> The fact that user's tasks are not active at the moment of a timer  
> interrupt doesn't mean  
> they were not active \_during\_ the last tick.

sure

> That's why another  
> approach in update\_load\_fair() which doesn't depend on a snapshot of  
> rq->raw\_weighted\_load  
> at timer tick's time. I guess, we'd lose this with 'inactive ticks',  
> right?

Sorry this is not clear. We'd lose what with 'inactive' ticks?

If you are referring to the delta execution time a user's Irq consumed in the middle of a tick and whether we would lose them during subsequent update\_load(), the answer IMO is no. Because put\_prev\_task\_fair() would account for the small delta execution time when the task/Irq got descheduled.

> ok, maybe  
> it's not that important for per-user cpu\_load, duno at the moment.

I would say any lossy accounting would be bad in long run. If you think put\_prev\_task\_fair()->update\_curr() still leaves open some problem, I would be interested in knowing that.

> > Basically what I want to know is, are we sacrificing any accuracy here  
 > > because of "deferring" smoothening of cpu\_load for a (inactive) Irq  
 > > (apart from the inaccurate figure used during load\_balance as you point  
 > > out below).  
 >  
 > At least, we are getting some inaccuracy (not in a generic case  
 > though) due to the  
 >  
 > if (exec\_delta64 > (u64)TICK\_NSEC)  
 >     exec\_delta64 = (u64)TICK\_NSEC;  [\*]  
 >  
 > in update\_load\_fair()..

If that is a problem (and I tend to agree that it is), then it is not unique to group Irq accounting. So we have common problems to solve :)

> and that's smth I want to try changing...

good.

> > Assuming the Irq was inactive for all those 3 ticks and became active at  
 > > 4th tick, would the end result of cpu\_load (as obtained in my code) be  
 > > any different than calling update\_load\_fair() on all Irq on each tick?  
 >  
 > With the current code, yes - it may be. In case, [\*] condition (see  
 > above) comes into play (and these 'inactive' ticks were not really  
 > inactive as described above).

Yes sure, we need to fix that assumption that exec\_delta64 can't be greater than TICK\_NSEC. And I assume you will fix that?

> > If there is any bug in 'replay lost ticks' loop in the patch I posted, then  
 > > it should already be present in current (i.e v16) implementation of  
 > > update\_load\_fair()?  
 >  
 > I think, you are right.

good :)

> > Yes, patch #5 introduces group-aware load-balance. It is two-step:  
 > >  
 > > First, we identify busiest group and busiest queue, based on  
 > > rq->raw\_weighted\_load/cpu\_load (which is accumulation of weight from all  
 > > classes on a CPU). This part of the code is untouched.  
 >  
 > I'll take a look (e.g. I guess, we have got a notion of "user's  
 > weight"... so does/how a user's weight contribute to his tasks weight..

A user's weight controls fraction of CPU the user's tasks as a whole receive.

A task's weight controls fraction of CPU the task will receive -within- the fraction allotted to that user.

Strictly speaking, a task's weight need not have to depend on its user's weight. This is true if scheduler core recognizes both user and task levels of scheduling in the hierarchy. If the scheduler were to recognize fewer levels of hierarchy, then we will have to take into account a user's weight in calculation task weight. See thread anchored at <http://lkml.org/lkml/2007/5/26/81> for a description of this idea.

> otherwise, I think, the approach of determining  
> the busiest CPU based only on pure tasks' weight would be wrong.. will  
> look at it first).

The load considered for determining busiest group/queue is the summation of -all- task's load on a CPU. That's why I introduced `update_load()` in Patch #4 which captures load from real-time tasks as well as `SCHED_NORMAL` tasks. When you are changing that `update_load()` function (based on `class->update_load` callback), it would be nice to keep this in mind (that I need a weight field representing summation of all tasks weights).

> >> If it's user's `Irq :: cpu_load[]` .. then it `_still_` shows the load at  
> >> the moment of T1 while we are at the moment T2 (and user1 was not  
> >> active during `dT`)..  
> >  
> >Good point. So how do we solve this? I really really want to avoid  
> >running `update_load_fair()` on all `Irq`'s every tick (it will be a massive  
> >overhead).  
>  
> yeahh.. have to think about it.  
> btw, I recall the patch #4 adds some light but noticeable overhead,  
> right? did you look at where exactly the overhead comes from?

This probably comes from the split up `raw_weighted_load/nr_running` fields. Although I don't know if the overhead is that noticeable in practice. Let me know if you feel any difference with Patch #4 applied!

> > I am assuming that `Irqs` don't remain inactive for a long time  
> >(given CFS's fairness promise!) and hence probably their `cpu_load[]` also  
> >won't be -that- stale in practice?  
>  
> I guess, it's not only about CFS but about the users' behavior, which  
> is something  
> we can't control and so can't rely on it.

> Say, a user was active till the moment T1 and then just gone.. - all  
> his tasks are really inactive.  
> So at the moment T2 user's Irq :: cpu\_load will still express the  
> situation at the moment T1?  
> As long as user's Irq is not involved in 'load balancing', this  
> inaccuracy can be revealed only if the info is exported via /proc.  
>  
> But say, user's task becomes finally active after \_a lot\_ of inactive  
> ticks (the user came back).. now it's in the rq and waiting for its  
> turn (which can be easily > 1 tick).. in the mean time 'load  
> balancing' is triggered.. and it considers the old Irq :: cpu\_load[]

I still think that this 'stale' cpu\_load data won't last long enough to  
serious affect load balance decisions. But something I agree definitely to  
watch for during tests. Thanks for the heads up on this possibility!

> P.S.  
>  
> just a personal impression.. I'm quite confused by this 'Irq' name...  
> it looks pretty similar to 'Irq' (with a big 'i') and I can't stop  
> reading it as 'IRQ' [ chores: so stop it! ]

:~)

> would be smth like 'cfs\_rq' or even 'sched\_rq' better? :-)

I chose Irq to mean local run queue. Other names I can think of are  
entity\_rq or ...actually cfs\_rq (as you suggest) sounds better. I will  
make this change unless Ingo thinks otherwise.

--

Regards,  
vatsa

---

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

---

Subject: Re: [RFC][PATCH 5/6] core changes for group fairness  
Posted by [Srivatsa Vaddagiri](#) on Thu, 14 Jun 2007 12:06:05 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Wed, Jun 13, 2007 at 10:56:06PM +0200, Dmitry Adamushko wrote:

```
> >+static int balance_tasks(struct rq *this_rq, int this_cpu, struct rq  
> >*busiest,  
> >+          unsigned long max_nr_move, unsigned long  
> >max_load_move,
```



```

> >+      struct sched_domain *sd, enum idle_type idle,
> >+      int *all_pinned, unsigned long *load_moved,
> >+      int this_best_prio, int best_prio, int
> >best_prio_seen,
> >+      void *iterator_arg,
> >+      struct task_struct *(*iterator_start)(void *arg),
> >+      struct task_struct *(*iterator_next)(void *arg));
>
> IMHO, it looks a bit frightening :)

```

I agree :) It is taking (ooops) 15 args (8 perhaps was the previous record in sched.c (move\_tasks)!

```

> maybe it would be possible to
> create a structure that combines some relevant arguments .. at least,
> the last 3 ones.

```

How does this look?

```

struct balance_tasks_args {
    struct rq *this_rq, struct rq *busiest;
    unsigned long max_nr_move, unsigned long max_load_move;
    struct sched_domain *sd, enum idle_type idle;
    int this_best_prio, best_prio, best_prio_seen;
    int *all_pinned;
    unsigned long *load_moved;
    void *iterator_arg;
    struct task_struct *(*iterator_start)(void *arg);
    struct task_struct *(*iterator_next)(void *arg));
};

```

```

static int balance_tasks(struct balance_tasks_args *arg);

```

[ down to one argument now! ]

?

I will try this in my next iteration ..

```

> >-static int move_tasks(struct rq *this_rq, int this_cpu, struct rq
> >*busiest,
> >+static int balance_tasks(struct rq *this_rq, int this_cpu, struct rq
> >*busiest,
> >      unsigned long max_nr_move, unsigned long
> >      max_load_move,
> >      struct sched_domain *sd, enum idle_type idle,
> >-      int *all_pinned)

```

```
> >+      int *all_pinned, unsigned long *load_moved,
> >+      int this_best_prio, int best_prio, int
> >best_prio_seen,
> >+      void *iterator_arg,
> >+      struct task_struct *(*iterator_start)(void *arg),
> >+      struct task_struct *(*iterator_next)(void *arg))
>
> I think, there is a possible problem here. If I'm not complete wrong,
> this function (move_tasks()) in the current mainline) can move more
> 'load' than specified by the 'max_load_move'..
```

Yes I think you are right. I will tackle this in next iteration.

Thanks for all your review so far!

--

Regards,  
vatsa

---

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

---

---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1  
Posted by [dev](#) on Fri, 15 Jun 2007 12:46:13 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Ingo Molnar wrote:

```
> i'd still like to hear back from Kirill & co whether this framework is
> flexible enough for their work (OpenVZ, etc.) too.
```

My IMHO is that so far the proposed group scheduler doesn't look ready/suitable.  
We need to have a working SMP version before it will be clear  
whether the whole approach is good and works correct on variety of load patterns.

Thanks,  
Kirill

---

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

---

---

Subject: Re: [RFC][PATCH 0/6] Add group fairness to CFS - v1

---

Posted by [Srivatsa Vaddagiri](#) on Fri, 15 Jun 2007 14:06:45 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On Fri, Jun 15, 2007 at 04:46:13PM +0400, Kirill Korotaev wrote:

> Ingo Molnar wrote:

>

> > i'd still like to hear back from Kirill & co whether this framework is

> > flexible enough for their work (OpenVZ, etc.) too.

>

> My IMHO is that so far the proposed group scheduler doesn't look ready/suitable.

Hi Kirill,

Yes its work-in-progress and hence is not ready/fully-functional

(yet). The patches I posted last gives an idea of the direction it is

heading. For ex: <http://lkml.org/lkml/2007/6/11/162> and

<http://lkml.org/lkml/2007/5/25/146> gives an idea of how SMP load balance will works.

IMHO the nice thing about this approach is it (re)uses lot of code in scheduler which is there already to achieve fairness between tasks and higher schedulable elements (users/containers etc).

Also with CFS engine's precise nanosecond accurate accounting and time-sorted list of tasks/entities, I feel we will get much tighter control over distribution of CPU between tasks/users/containers.

> We need to have a working SMP version before it will be clear

> whether the whole approach is good and works correct on variety of load patterns.

If you have any heads up thoughts on areas/workloads where this may pose problems for container/user scheduling, I would be glad to hear them. Otherwise I would greatly wellcome any help in developing/reviewing these patches which meets both our goals!

--

Regards,

vatsa

---

Containers mailing list

[Containers@lists.linux-foundation.org](mailto:Containers@lists.linux-foundation.org)

<https://lists.linux-foundation.org/mailman/listinfo/containers>

---