
Subject: [PATCH 0/2] Add group awareness to CFS - v2
Posted by [Srivatsa Vaddagiri](#) on Sat, 23 Jun 2007 13:15:45 GMT
[View Forum Message](#) <> [Reply to Message](#)

Hi Ingo,

Here's an update for the group-aware CFS scheduler that I have been working on.

(For those reading these patches for the first time:)

The basic idea is to reuse CFS core and other pieces of scheduler like smprnice-driven load balance for driving fairness between 'schedulable entities' other than tasks, for ex: users or containers.

The time-sorted rb-tree and nanosecond accurate accounting aspects of CFS are "repeated" for schedulable entities other than tasks.

For ex: there could be N task-level rb-trees for N users (which stores tasks) and a single user-level rb-tree which stores user-level entities. CFS operations on each user's task-level rb-tree drives fairness between tasks of that user, while CFS operations on user-level rb-tree drives fairness between users.

v17 CFS introduced basic changes in CFS to support group scheduling. The two patches to follow build upon them as follows:

Patch 1 => introduces a notion of scheduler hierarchy (of entities) and applies CFS operations at all levels of this hierarchy.

Patch 2 => hooks up the cpu scheduler with task grouping feature in mm tree (CONFIG_CONTAINERS) as an interface to task-grouping functionality.

A single config option CONFIG_FAIR_GROUP_SCHED allows the group-scheduling feature to be turned on/off at compile time.

I have tried my best to ensure there is no impact to existing CFS performance when CONFIG_FAIR_GROUP_SCHED is disabled. Some results in this regard are provided at the end.

One noticeable change in functionality may be the /proc/sched_debug output (I had to rearrange that code a bit to dump group cfs_rq information also).

Changes since last version:

- Fixed some bugs in SMP load balance (pointed by Dmitry)
- Modified sched_debug.c to dump all cfs_rq stats

Todo:

- Weighted fair-share
Currently all groups get "equal" cpu bandwidth. I plan to support weighted fair-sharing on the lines of task niceness.
- Separate out tunable
Right now tunable are same for all layers of scheduling. I strongly think we will need to separate them, esp `sysctl_sched_runtime_limit`.
- Optimization
 - reduce frequency of timer tick processing at higher levels
 - during load balance, pick cache-cold tasks first to migrate
- hierarchy flattening
Experiment with this (to reduce number of hierarchical levels) as per <http://lkml.org/lkml/2007/5/26/81>

Some results follows. Legends used in them are:

cfs = base cfs performance (sched-cfs-v2.6.22-rc4-mm2-v18.patch)
 cfsgrpdi = base cfs + patches 1-2 applied (CONFIG_FAIR_GROUP_SCHED disabled)
 cfsgrpdi = base cfs + patches 1-2 applied (CONFIG_FAIR_GROUP_SCHED enabled)

All tests run on a 4-cpu Intel Xeon (x86_64) box:

A. Overhead Test

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-	6.7400	7.8200	8.0100	8.7900	10.90	8.20000	19.88	
cfsgrpdi	Linux 2.6.22-	6.7000	7.6700	8.0700	9.0100	11.54	9.34000	18.71	
cfsgrpdi	Linux 2.6.22-	7.8600	7.8700	8.6500	9.4600	10.27	9.44000	19.74	

hackbench -pipe 100

=====

Average of 10 runs was taken. Smaller numbers are better.

cfs 4.0171
 cfsgrpdi 4.154
 cfsgrpdi 4.7749

B. UP Group fairness test

These tests were forced to run on a single CPU by making using of exclusive cpusets.

hackbench
=====

The two user's shell were put in different groups (as explained in Patch 2/2). Each user then ran this script:

```
i=0
while [ $i -lt 10 ]
do
./hackbench -pipe 100 >> log
i=`expr $i + 1`
done
```

Time taken to complete this script was measured as follows (note that both the scripts were made to run simultaneously on /same/ cpu).

```
vatsa 103.51 s (real)
guest 103.37 s (real)
```

Inference: Both users completed the same amount of work in (nearly) same time.

kernel compilation
=====

Again the two user's shell were put in different groups.

User vatsa ran "make -s -j4 bzImage", while
User guest ran "make -s -j20 bzImage"

Both are compiling the same sources (and hence should effectively be doing the same amount of work). Time taken to complete kernel-compile by both users:

```
vatsa 777.46 s (real)
guest 778.30 s (real)
```

Inference: Both users completed the same amount of work in nearly same time, even though one had higher number of threads dedicated to the job.

C. SMP Fairness test
=====

I used a simple cpu-intensive program which measures how much CPU time it got (using getrusage) over a minute. N (=4*NUM_CPUS) such tasks were spawned with N/2 in one group and N/2 in another group. Total CPU time obtained by one group was compared with total cpu time obtained by another group. While the test was running, I observed distribution of all tasks across CPUs. I am quite happy with the results obtained and with the load distribution. I can share the sources/results of the program/script upon request.

Looking forward to your feedback on these patches!

[P.S : Since I am travelling this weekend, I may not respond promptly]

--

Regards,
vatsa

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

Subject: [PATCH 1/2] Introduce notion of scheduler hierarchy
Posted by [Srivatsa Vaddagiri](#) on Sat, 23 Jun 2007 13:18:07 GMT
[View Forum Message](#) <> [Reply to Message](#)

This patch introduces the core changes in CFS required to accomplish group fairness at higher levels. It also modifies load balance interface between classes a bit, so that move_tasks (which is centric to load balance) can be reused to balance between runqueues of various types (struct rq in case of SCHED_RT tasks, struct lrq in case of SCHED_NORMAL/BATCH tasks).

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

```
include/linux/sched.h | 17 ++
kernel/sched.c         | 174 ++++++-----
kernel/sched_debug.c   | 31 +++-
kernel/sched_fair.c     | 295 ++++++-----
kernel/sched_idletask.c | 11 +
kernel/sched_rt.c       | 47 +++++-
6 files changed, 464 insertions(+), 111 deletions(-)
```

Index: current/include/linux/sched.h

=====

```

--- current.orig/include/linux/sched.h
+++ current/include/linux/sched.h
@@ -134,8 +134,11 @@ extern unsigned long nr_iowait(void);
extern unsigned long weighted_cpuload(const int cpu);

struct seq_file;
+struct cfs_rq;
extern void proc_sched_show_task(struct task_struct *p, struct seq_file *m);
extern void proc_sched_set_task(struct task_struct *p);
+extern void
+print_cfs_rq(struct seq_file *m, int cpu, struct cfs_rq *cfs_rq, u64 now);

/*
 * Task state bitmask. NOTE! These bits are also
@@ -865,8 +868,13 @@ struct sched_class {
    struct task_struct * (*pick_next_task) (struct rq *rq, u64 now);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p, u64 now);

- struct task_struct * (*load_balance_start) (struct rq *rq);
- struct task_struct * (*load_balance_next) (struct rq *rq);
+ 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 cpu_idle_type idle,
+   int *all_pinned, unsigned long *total_load_moved);
+
+ void (*set_curr_task) (struct rq *rq);
    void (*task_tick) (struct rq *rq, struct task_struct *p);
    void (*task_new) (struct rq *rq, struct task_struct *p);
};
@@ -899,6 +907,11 @@ struct sched_entity {
    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 cfs_rq *cfs_rq, /* rq on which this entity is (to be) queued */
+   +   *my_rq; /* rq "owned" by this entity/group */
+ #endif
};

```

```

struct task_struct {

```

```

Index: current/kernel/sched.c

```

```

=====

```

```

--- current.orig/kernel/sched.c
+++ current/kernel/sched.c
@@ -133,6 +133,22 @@ struct cfs_rq {
    struct rb_root tasks_timeline;

```

```

    struct rb_node *rb_leftmost;
    struct rb_node *rb_load_balance_curr;
+ #ifdef CONFIG_FAIR_GROUP_SCHED
+ /* 'curr' points to currently running entity on this cfs_rq.
+  * It is set to NULL otherwise (i.e when none are currently running).
+  */
+ struct sched_entity *curr;
+ struct rq *rq; /* cpu runqueue to which this cfs_rq is attached */
+
+ /* leaf cfs_rqs are those that hold tasks (lowest schedulable entity in
+  * a hierarchy). Non-leaf lrgs hold other higher schedulable entities
+  * (like users, containers etc.)
+  *
+  * leaf_cfs_rq_list ties together list of leaf cfs_rq's in a cpu. This
+  * list is used during load balance.
+  */
+ struct list_head leaf_cfs_rq_list; /* Better name : task_cfs_rq_list? */
+ #endif
};

```

```
@@ -168,6 +184,9 @@ struct rq {
    u64 nr_switches;
```

```
/*
@@ -342,6 +361,16 @@ static inline unsigned long long rq_cloc
#define task_rq(p)  cpu_rq(task_cpu(p))
#define cpu_curr(cpu) (cpu_rq(cpu)->curr)
```

```

@@ -738,6 +767,21 @@ static inline void dec_nr_running(struct
}

static void activate_task(struct rq *rq, struct task_struct *p, int wakeup);
#ifdef CONFIG_SMP
+
+struct rq_iterator {
+ void *arg;
+ struct task_struct *(*start)(void *);
+ struct task_struct *(*next)(void *);
+};
+
+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 cpu_idle_type idle,
+ int *all_pinned, unsigned long *load_moved,
+ int this_best_prio, int best_prio, int best_prio_seen,
+ struct rq_iterator *iterator);
#endif

#include "sched_stats.h"
#include "sched_rt.c"
@@ -894,6 +938,7 @@ unsigned long weighted_cpuload(const int
static inline void __set_task_cpu(struct task_struct *p, unsigned int cpu)
{
    task_thread_info(p)->cpu = cpu;
+ set_task_cfs_rq(p);
}

void set_task_cpu(struct task_struct *p, unsigned int new_cpu)
@@ -919,6 +964,7 @@ void set_task_cpu(struct task_struct *p,

    task_thread_info(p)->cpu = new_cpu;

+ set_task_cfs_rq(p);
}

struct migration_req {
@@ -2003,89 +2049,26 @@ int can_migrate_task(struct task_struct
    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 cpu_idle_type idle,
-    int *all_pinned)
+    int *all_pinned, unsigned long *load_moved,
+    int this_best_prio, int best_prio, int best_prio_seen,
+    struct rq_iterator *iterator)
{
- 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;
@@ -2102,7 +2085,7 @@ next:
    !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;
    }

```

```

@@ -2117,7 +2100,7 @@ next:
    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:
@@ -2130,10 +2113,40 @@ out:

    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 cpu_idle_type idle,
+    int *all_pinned)
+{
+   struct sched_class *class = sched_class_highest;
+   unsigned long load_moved, total_nr_moved = 0, nr_moved;
+   long rem_load_move = max_load_move;
+   do {
+       nr_moved = class->load_balance(this_rq, this_cpu, busiest,
+           max_nr_move, (unsigned long)rem_load_move,
+           sd, idle, all_pinned, &load_moved);
+       total_nr_moved += nr_moved;
+       max_nr_move -= nr_moved;
+       rem_load_move -= load_moved;
+       class = class->next;
+   } while (class && max_nr_move && rem_load_move > 0);
+   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.
@@ -6184,6 +6197,15 @@ int in_sched_functions(unsigned long add
    && addr < (unsigned long)__sched_text_end);
}

```

```

+static inline void init_cfs_rq(struct cfs_rq *cfs_rq, struct rq *rq)
+{
+ cfs_rq->tasks_timeline = RB_ROOT;
+ cfs_rq->fair_clock = 1;
+#ifdef CONFIG_FAIR_GROUP_SCHED
+ cfs_rq->rq = rq;
+#endif
+}
+

```

```

void __init sched_init(void)
{
    int highest_cpu = 0;
@@ -6204,10 +6226,14 @@ void __init sched_init(void)
    spin_lock_init(&rq->lock);
    lockdep_set_class(&rq->lock, &rq->rq_lock_key);
    rq->nr_running = 0;
- rq->cfs.tasks_timeline = RB_ROOT;
- rq->clock = rq->cfs.fair_clock = 1;
+ rq->clock = 1;
    rq->ls.load_update_last = sched_clock();
    rq->ls.load_update_start = sched_clock();
+ init_cfs_rq(&rq->cfs, rq);
+#ifdef CONFIG_FAIR_GROUP_SCHED
+ INIT_LIST_HEAD(&rq->leaf_cfs_rq_list);
+ list_add(&rq->cfs.leaf_cfs_rq_list, &rq->leaf_cfs_rq_list);
+#endif

```

```

    for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
        rq->cpu_load[j] = 0;

```

Index: current/kernel/sched_debug.c

```

=====
--- current.orig/kernel/sched_debug.c
+++ current/kernel/sched_debug.c
@@ -80,15 +80,17 @@ static void print_rq(struct seq_file *m,
    read_unlock_irq(&tasklist_lock);
}

```

```

-static void print_rq_runtime_sum(struct seq_file *m, struct rq *rq)
+static void
+print_cfs_rq_runtime_sum(struct seq_file *m, int cpu, struct cfs_rq *cfs_rq)
{
    s64 wait_runtime_rq_sum = 0;

```

```

    struct task_struct *p;
    struct rb_node *curr;
    unsigned long flags;
+ struct rq *rq = &per_cpu(runqueues, cpu);

    spin_lock_irqsave(&rq->lock, flags);
- curr = first_fair(&rq->cfs);
+ curr = first_fair(cfs_rq);
    while (curr) {
        p = rb_entry(curr, struct task_struct, se.run_node);
        wait_runtime_rq_sum += p->se.wait_runtime;
@@ -101,6 +103,23 @@ static void print_rq_runtime_sum(struct
    (long long)wait_runtime_rq_sum);
}

+void print_cfs_rq(struct seq_file *m, int cpu, struct cfs_rq *cfs_rq, u64 now)
+{
+ SEQ_printf(m, "\ncfs_rq %p\n", cfs_rq);
+
+ #define P(x) \
+ SEQ_printf(m, " .%-30s: %Ld\n", #x, (long long)(cfs_rq->x))
+
+ P(fair_clock);
+ P(exec_clock);
+ P(wait_runtime);
+ P(wait_runtime_overruns);
+ P(wait_runtime_underruns);
+ #undef P
+
+ print_cfs_rq_runtime_sum(m, cpu, cfs_rq);
+}
+
static void print_cpu(struct seq_file *m, int cpu, u64 now)
{
    struct rq *rq = &per_cpu(runqueues, cpu);
@@ -136,18 +155,14 @@ static void print_cpu(struct seq_file *m
    P(clock_overflows);
    P(clock_unstable_events);
    P(clock_max_delta);
- P(cfs.fair_clock);
- P(cfs.exec_clock);
- P(cfs.wait_runtime);
- P(cfs.wait_runtime_overruns);
- P(cfs.wait_runtime_underruns);
    P(cpu_load[0]);
    P(cpu_load[1]);
    P(cpu_load[2]);
    P(cpu_load[3]);

```

```

    P(cpu_load[4]);
#undef P
- print_rq_runtime_sum(m, rq);
+
+ print_cfs_stats(m, cpu, now);

    print_rq(m, rq, cpu, now);
}
Index: current/kernel/sched_fair.c
=====
--- current.orig/kernel/sched_fair.c
+++ current/kernel/sched_fair.c
@@ -70,6 +70,31 @@ extern struct sched_class fair_sched_cla
 * CFS operations on generic schedulable entities:
 */

#ifdef CONFIG_FAIR_GROUP_SCHED
+
+/* cpu runqueue to which this cfs_rq is attached */
+static inline struct rq *rq_of(struct cfs_rq *cfs_rq)
+{
+ return cfs_rq->rq;
+}
+
+/* currently running entity (if any) on this cfs_rq */
+static inline struct sched_entity *cfs_rq_curr(struct cfs_rq *cfs_rq)
+{
+ return cfs_rq->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_cfs_rq_curr(struct cfs_rq *cfs_rq, struct sched_entity *se)
+{
+ cfs_rq->curr = se;
+}
+
+#else /* CONFIG_FAIR_GROUP_SCHED */
+
static inline struct rq *rq_of(struct cfs_rq *cfs_rq)
{
    return container_of(cfs_rq, struct rq, cfs);
}
@@ -87,6 +112,11 @@ static inline struct sched_entity *cfs_r

#define entity_is_task(se) 1

```



```

+ #ifdef CONFIG_FAIR_GROUP_SCHED
+
+ /* Walk up scheduling entities hierarchy */
+ #define for_each_sched_entity(se) \
+ for (; se; se = se->parent)
+
+ static inline struct cfs_rq *task_cfs_rq(struct task_struct *p)
+ {
+ return p->se.cfs_rq;
+ }
+
+ /* runqueue on which this entity is (to be) queued */
+ static inline struct cfs_rq *cfs_rq_of(struct sched_entity *se)
+ {
+ return se->cfs_rq;
+ }
+
+ /* runqueue "owned" by this group */
+ static inline struct cfs_rq *group_cfs_rq(struct sched_entity *grp)
+ {
+ return grp->my_rq;
+ }
+
+ /* Given a group's cfs_rq on one cpu, return its corresponding cfs_rq on
+  * another cpu ('this_cpu')
+  */
+ static inline struct cfs_rq *cpu_cfs_rq(struct cfs_rq *cfs_rq, int this_cpu)
+ {
+ /* A later patch will take group into account */
+ return &cpu_rq(this_cpu)->cfs;
+ }
+
+ /* Iterate thr' all leaf cfs_rq's on a runqueue */
+ #define for_each_leaf_cfs_rq(rq, cfs_rq) \
+ list_for_each_entry(cfs_rq, &rq->leaf_cfs_rq_list, leaf_cfs_rq_list)
+
+ /* Do the two (enqueued) tasks belong to the same group ? */
+ static inline int is_same_group(struct task_struct *curr, struct task_struct *p)
+ {
+ if (curr->se.cfs_rq == p->se.cfs_rq)
+ return 1;
+
+ return 0;
+ }
+
+ #else /* CONFIG_FAIR_GROUP_SCHED */
+
+ #define for_each_sched_entity(se) \

```

```

+ for (; se; se = NULL)
+
+ static inline struct cfs_rq *task_cfs_rq(struct task_struct *p)
+ {
+     return &task_rq(p)->cfs;
+ }

+static inline struct cfs_rq *cfs_rq_of(struct sched_entity *se)
+{
+ struct task_struct *p = task_of(se);
+ struct rq *rq = task_rq(p);
+
+ return &rq->cfs;
+}
+
+/* runqueue "owned" by this group */
+static inline struct cfs_rq *group_cfs_rq(struct sched_entity *grp)
+{
+ return NULL;
+}
+
+static inline struct cfs_rq *cpu_cfs_rq(struct cfs_rq *cfs_rq, int this_cpu)
+{
+ return &cpu_rq(this_cpu)->cfs;
+}
+
+#define for_each_leaf_cfs_rq(rq, cfs_rq) \
+ for (cfs_rq = &rq->cfs; cfs_rq; cfs_rq = NULL)
+
+static inline int is_same_group(struct task_struct *curr, struct task_struct *p)
+{
+ return 1;
+}
+
+#endif /* CONFIG_FAIR_GROUP_SCHED */
+
+/*
+ * The enqueue_task method is called before nr_running is
+ * increased. Here we update the fair scheduling stats and
+@@ -690,10 +807,15 @@ static inline struct cfs_rq *task_cfs_rq
+ static void
+ enqueue_task_fair(struct rq *rq, struct task_struct *p, int wakeup, u64 now)
+ {
+ - struct cfs_rq *cfs_rq = task_cfs_rq(p);
+ + struct cfs_rq *cfs_rq;
+   struct sched_entity *se = &p->se;
+
+ - enqueue_entity(cfs_rq, se, wakeup, now);

```

```

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

/*
@@ -704,10 +826,16 @@ enqueue_task_fair(struct rq *rq, struct
static void
dequeue_task_fair(struct rq *rq, struct task_struct *p, int sleep, u64 now)
{
- struct cfs_rq *cfs_rq = task_cfs_rq(p);
+ struct cfs_rq *cfs_rq;
  struct sched_entity *se = &p->se;

- dequeue_entity(cfs_rq, se, sleep, now);
+ for_each_sched_entity(se) {
+   cfs_rq = cfs_rq_of(se);
+   dequeue_entity(cfs_rq, se, sleep, now);
+   /* Don't dequeue parent if it has other entities besides us */
+   if (cfs_rq->load.weight)
+     break;
+ }
}

/*
@@ -755,7 +883,8 @@ static void check_preempt_curr_fair(stru
if (unlikely(p->policy == SCHED_BATCH))
  gran = sysctl_sched_batch_wakeup_granularity;

- __check_preempt_curr_fair(cfs_rq, &p->se, &curr->se, gran);
+ if (is_same_group(curr, p))
+   __check_preempt_curr_fair(cfs_rq, &p->se, &curr->se, gran);
}

static struct task_struct * pick_next_task_fair(struct rq *rq, u64 now)
@@ -766,7 +895,10 @@ static struct task_struct * pick_next_ta
if (unlikely(!cfs_rq->nr_running))
  return NULL;

- se = pick_next_entity(cfs_rq, now);
+ do {
+   se = pick_next_entity(cfs_rq, now);
+   cfs_rq = group_cfs_rq(se);
+ } while (cfs_rq);

```

```

    return task_of(se);
}
@@ -776,7 +908,13 @@ static struct task_struct * pick_next_ta
*/
static void put_prev_task_fair(struct rq *rq, struct task_struct *prev, u64 now)
{
- put_prev_entity(task_cfs_rq(prev), &prev->se, now);
+ struct cfs_rq *cfs_rq;
+ struct sched_entity *se = &prev->se;
+
+ for_each_sched_entity(se) {
+ cfs_rq = cfs_rq_of(se);
+ put_prev_entity(cfs_rq, se, now);
+ }
}

/*****
@@ -791,7 +929,7 @@ static void put_prev_task_fair(struct rq
* the current task:
*/
static inline struct task_struct *
- __load_balance_iterator(struct rq *rq, struct rb_node *curr)
+ __load_balance_iterator(struct cfs_rq *cfs_rq, struct rb_node *curr)
{
    struct task_struct *p;

@@ -799,19 +937,104 @@ __load_balance_iterator(struct rq *rq, s
    return NULL;

    p = rb_entry(curr, struct task_struct, se.run_node);
- rq->cfs.rb_load_balance_curr = rb_next(curr);
+ cfs_rq->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->cfs));
+ struct cfs_rq *cfs_rq = arg;
+
+ return __load_balance_iterator(cfs_rq, first_fair(cfs_rq));
}

-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->cfs.rb_load_balance_curr);
+ struct cfs_rq *cfs_rq = arg;
+
+ return __load_balance_iterator(cfs_rq, cfs_rq->rb_load_balance_curr);
+}
+
+static inline int cfs_rq_best_prio(struct cfs_rq *cfs_rq)
+{
+ struct sched_entity *curr;
+ struct task_struct *p;
+
+ if (!cfs_rq->nr_running)
+ return MAX_PRIO;
+
+ curr = __pick_next_entity(cfs_rq);
+ p = task_of(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 cpu_idle_type idle,
+ int *all_pinned, unsigned long *total_load_moved)
+{
+ struct cfs_rq *busy_cfs_rq;
+ unsigned long load_moved, total_nr_moved = 0, nr_moved;
+ long rem_load_move = max_load_move;
+ struct rq_iterator cfs_rq_iterator;
+
+ cfs_rq_iterator.start = load_balance_start_fair;
+ cfs_rq_iterator.next = load_balance_next_fair;
+
+ for_each_leaf_cfs_rq(busiest, busy_cfs_rq) {
+ struct cfs_rq *this_cfs_rq;
+ long imbalance;
+ unsigned long maxload;
+ int this_best_prio, best_prio, best_prio_seen = 0;
+
+ this_cfs_rq = cpu_cfs_rq(busy_cfs_rq, this_cpu);
+
+ imbalance = busy_cfs_rq->load.weight -
+ this_cfs_rq->load.weight;
+ /* Don't pull if this_cfs_rq has more load than busy_cfs_rq */
+ if (imbalance <= 0)
+ continue;
+

```

```

+ /* Don't pull more than imbalance/2 */
+ imbalance /= 2;
+ maxload = min(rem_load_move, imbalance);
+
+ this_best_prio = cfs_rq_best_prio(this_cfs_rq);
+ best_prio = cfs_rq_best_prio(busy_cfs_rq);
+
+ /*
+  * 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 (cfs_rq_curr(busy_cfs_rq) == &busiest->curr->se)
+   best_prio_seen = 1;
+
+ /* pass busy_cfs_rq argument into
+  * load_balance_[start|next]_fair iterators
+  */
+ cfs_rq_iterator.arg = busy_cfs_rq;
+ 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, &cfs_rq_iterator);
+
+ total_nr_moved += nr_moved;
+ max_nr_move -= nr_moved;
+ rem_load_move -= load_moved;
+
+ if (max_nr_move <= 0 || rem_load_move <= 0)
+   break;
+ }
+
+ *total_load_moved = max_load_move - rem_load_move;
+
+ return total_nr_moved;
+ }

/*
@@ -819,7 +1042,13 @@ static struct task_struct * load_balance
*/
static void task_tick_fair(struct rq *rq, struct task_struct *curr)
{
- entity_tick(task_cfs_rq(curr), &curr->se);
+ struct cfs_rq *cfs_rq;
+ struct sched_entity *se = &curr->se;
+

```

```

+ for_each_sched_entity(se) {
+   cfs_rq = cfs_rq_of(se);
+   entity_tick(cfs_rq, se);
+ }
}

/*
@@ -863,6 +1092,24 @@ static void task_new_fair(struct rq *rq,
    inc_nr_running(p, rq, now);
}

+/* Account for a task changing its policy or group.
+ *
+ * This routine is mostly called to set cfs_rq->curr field when a task
+ * migrates between groups/classes.
+ */
+static void set_curr_task_fair(struct rq *rq)
+{
+   struct task_struct *curr = rq->curr;
+   struct sched_entity *se = &curr->se;
+   struct cfs_rq *cfs_rq;
+   u64 now = rq_clock(rq);
+   +
+   for_each_sched_entity(se) {
+     cfs_rq = cfs_rq_of(se);
+     set_next_entity(cfs_rq, se, now);
+   }
+}
+
+/*
+ * All the scheduling class methods:
+ */
@@ -876,8 +1123,18 @@ struct sched_class fair_sched_class __re
    .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,
+   .load_balance = load_balance_fair,
+
+   .set_curr_task = set_curr_task_fair,
+   .task_tick = task_tick_fair,
+   .task_new = task_new_fair,
+};
+
+void print_cfs_stats(struct seq_file *m, int cpu, u64 now)
+{
+   struct rq *rq = cpu_rq(cpu);

```

```

+ struct cfs_rq *cfs_rq;
+
+ for_each_leaf_cfs_rq(rq, cfs_rq)
+ print_cfs_rq(m, cpu, cfs_rq, now);
+}
Index: current/kernel/sched_idletask.c
=====
--- current.orig/kernel/sched_idletask.c
+++ current/kernel/sched_idletask.c
@@ -37,9 +37,13 @@ static void put_prev_task_idle(struct rq
{
}

-static struct task_struct *load_balance_start_idle(struct rq *rq)
+static int
+load_balance_idle(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 cpu_idle_type idle,
+ int *all_pinned, unsigned long *total_load_moved)
{
- return NULL;
+ return 0;
}

static void task_tick_idle(struct rq *rq, struct task_struct *curr)
@@ -60,8 +64,7 @@ struct sched_class idle_sched_class __re
.pick_next_task = pick_next_task_idle,
.put_prev_task = put_prev_task_idle,

- .load_balance_start = load_balance_start_idle,
- /* no .load_balance_next for idle tasks */
+ .load_balance = load_balance_idle,

.task_tick = task_tick_idle,
/* no .task_new for idle tasks */
Index: current/kernel/sched_rt.c
=====
--- current.orig/kernel/sched_rt.c
+++ current/kernel/sched_rt.c
@@ -107,8 +107,9 @@ static void put_prev_task_rt(struct rq *
* 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 = arg;
struct prio_array *array = &rq->rt.active;

```

```

    struct list_head *head, *curr;
    struct task_struct *p;
@@ -132,8 +133,9 @@ static struct task_struct * load_balance
    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 = arg;
    struct prio_array *array = &rq->rt.active;
    struct list_head *head, *curr;
    struct task_struct *p;
@@ -170,6 +172,44 @@ static struct task_struct * load_balance
    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 cpu_idle_type idle,
+ int *all_pinned, unsigned long *load_moved)
+{
+ int this_best_prio, best_prio, best_prio_seen = 0;
+ int nr_moved;
+ struct rq_iterator rt_rq_iterator;
+
+ best_prio = sched_find_first_bit(busiest->rt.active.bitmap);
+ this_best_prio = sched_find_first_bit(this_rq->rt.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;
+
+ rt_rq_iterator.start = load_balance_start_rt;
+ rt_rq_iterator.next = load_balance_next_rt;
+ /* pass 'busiest' rq argument into
+  * load_balance_[start|next]_rt iterators
+  */
+ rt_rq_iterator.arg = busiest;
+
+ 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,
+ &rt_rq_iterator);
+
+ return nr_moved;
+}
+
static void task_tick_rt(struct rq *rq, struct task_struct *p)
{
/*
@@ -207,8 +247,7 @@ static struct sched_class rt_sched_class
 .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,
+ .load_balance = load_balance_rt,

 .task_tick = task_tick_rt,
 .task_new = task_new_rt,
--

```

Regards,
vatsa

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

Subject: [PATCH 2/2] Hook up to (process) container feature in mm tree
Posted by [Srivatsa Vaddagiri](#) on Sat, 23 Jun 2007 13:20:04 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 is being working upon) 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
init/Kconfig                      | 10 +
kernel/sched.c                   | 250 ++++++
kernel/sched_fair.c              | 3
4 files changed, 256 insertions(+), 13 deletions(-)
```

Index: current/include/linux/container_subsys.h

=====

```
--- current.orig/include/linux/container_subsys.h
+++ current/include/linux/container_subsys.h
@@ -24,3 +24,9 @@ SUBSYS(debug)
#endif
```

```
/* */
+
+#ifdef CONFIG_FAIR_GROUP_SCHED
+SUBSYS(cpuctlr)
+#endif
+
+/* */
```

Index: current/init/Kconfig

=====

```
--- current.orig/init/Kconfig
+++ current/init/Kconfig
@@ -328,6 +328,16 @@ config CPUSETS
```

Say N if unsure.

```
+config FAIR_GROUP_SCHED
+ bool "Fair group scheduler"
+ depends on EXPERIMENTAL
+ 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
Index: current/kernel/sched.c
```

=====

```
--- current.orig/kernel/sched.c
+++ current/kernel/sched.c
@@ -120,6 +120,56 @@ struct load_stat {
     u64 delta_fair, delta_exec, delta_stat;
 };

+#ifdef CONFIG_FAIR_GROUP_SCHED
+
+#include <linux/container.h>
+
+struct cfs_rq;
+
+/* task container/group related information */
+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 cfs_rq **cfs_rq;
+};
+
+static DEFINE_PER_CPU(struct sched_entity, init_sched_entity);
+static DEFINE_PER_CPU(struct cfs_rq, init_cfs_rq) ____cacheline_aligned_in_smp;
+
+static struct sched_entity *init_sched_entity_p[CONFIG_NR_CPUS];
+static struct cfs_rq *init_cfs_rq_p[CONFIG_NR_CPUS];
```



```

+
+/* 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 cfs_rq *cfs_rq;
+ struct sched_entity *se;
+ int i;
+
+ if (!cont->parent) {
+ /* This is early initialization for the top container */
+ cont->subsys[cputr_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->cfs_rq = kzalloc(sizeof(cfs_rq) * num_possible_cpus(), GFP_KERNEL);
+ if (!tg->cfs_rq)
+ 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);
+
+ cfs_rq = kmalloc_node(sizeof(struct cfs_rq), GFP_KERNEL,
+ cpu_to_node(i));
+ if (!cfs_rq)
+ goto err;
+
+ se = kmalloc_node(sizeof(struct sched_entity), GFP_KERNEL,
+ cpu_to_node(i));
+ if (!se)
+ goto err;
+
+ memset(cfs_rq, 0, sizeof(struct cfs_rq));
+ memset(se, 0, sizeof(struct sched_entity));
+
+

```

```

+ tg->cfs_rq[i] = cfs_rq;
+ init_cfs_rq(cfs_rq, rq);
+ cfs_rq->tg = tg;
+ list_add_rcu(&cfs_rq->leaf_cfs_rq_list, &rq->leaf_cfs_rq_list);
+
+ tg->se[i] = se;
+ se->cfs_rq = &rq->cfs;
+ se->my_q = cfs_rq;
+ se->load.weight = NICE_0_LOAD;
+ se->parent = NULL;
+ }
+
+ /* Bind the container to task_grp object we just created */
+ cont->subsys[cputlr_subsys_id] = &tg->css;
+ tg->css.container = cont;
+
+ return 0;
+
+err:
+ for_each_possible_cpu(i) {
+   if (tg->cfs_rq && tg->cfs_rq[i])
+     kfree(tg->cfs_rq[i]);
+   if (tg->se && tg->se[i])
+     kfree(tg->se[i]);
+ }
+ if (tg->cfs_rq)
+   kfree(tg->cfs_rq);
+ 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 cfs_rq *cfs_rq;
+   struct sched_entity *se;
+   int i;
+
+   for_each_possible_cpu(i) {
+     cfs_rq = tg->cfs_rq[i];
+     list_del_rcu(&cfs_rq->leaf_cfs_rq_list);

```

```

+ }
+
+ /* wait for possible concurrent references to cfs_rqs complete */
+ synchronize_sched();
+
+ /* now it should be safe to free those cfs_rqs */
+ for_each_possible_cpu(i) {
+   cfs_rq = tg->cfs_rq[i];
+   kfree(cfs_rq);
+
+   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;
+
+   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));
+
+   set_task_cfs_rq(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 = "cpuctlr",
+ .create = sched_create_group,
+ .destroy = sched_destroy_group,
+ .attach = sched_move_task,
+ .populate = sched_populate,
+ .subsys_id = cpuctlr_subsys_id,
+ .early_init = 1,
+};
+
+#endif /* CONFIG_FAIR_GROUP_SCHED */
```

Index: current/kernel/sched_fair.c

=====

--- current.orig/kernel/sched_fair.c

+++ current/kernel/sched_fair.c

```
@@ -743,8 +743,7 @@ static inline struct cfs_rq *group_cfs_r
 */
```

```
static inline struct cfs_rq *cpu_cfs_rq(struct cfs_rq *cfs_rq, int this_cpu)
{
- /* A later patch will take group into account */
- return &cpu_rq(this_cpu)->cfs;
+ return cfs_rq->tg->cfs_rq[this_cpu];
}
```

```
/* Iterate thr' all leaf cfs_rq's on a runqueue */
```

--

Regards,
vatsa

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

Subject: Re: [PATCH 0/2] Add group awareness to CFS - v2
Posted by [Ingo Molnar](#) on Tue, 26 Jun 2007 08:52:33 GMT
[View Forum Message](#) <> [Reply to Message](#)

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

> Hi Ingo,
> Here's an update for the group-aware CFS scheduler that I have
> been working on.

thanks. I've carried your patch #1 (core changes) for the past few days
and it's looking pretty good so far! This makes the container-scheduling
patch (patch #2) a pretty painless add-on.

Ingo

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