Process Scheduling in Linux

This document contains notes about how the Linux kernel handles process scheduling. They cover the general scheduler skeleton, scheduling classes, the completely fair scheduling (CFS) algorithm, soft-real-time scheduling and load balancing for both real time and CFS.

The Linux kernel version looked at in this document is 3.1.10-g05b777c which is used in Android 4.2 Grouper for the Nexus 7.

1. Process Scheduling

The current Linux kernel is a multi-tasking kernel. Therefore, more than one process are allowed to exist at any given time and every process is allowed to run as if it were the only process on the system. The process scheduler coordinates which process runs when. In that context, it has the following tasks:

- share CPU equally among all currently running processes

- pick appropriate process to run next if required, considering scheduling class/policy and process priorities

- balance processes between multiple cores in SMP systems

1.2 Linux Processes/Threads

Processes in Linux are a group of threads that share a *thread group ID (TGID)* and whatever resources necessary and does not differentiate between the two. The kernel schedules individual threads, not processes. Therefore, the term "task" will be used for the remainder of the document to refer to a thread.

task_struct (include/linux/sched.h) is the data structure used in Linux that contains all the information about a specific task.

2 Task Classification

2.1 CPU-bound vs. I/O bound

Tasks tend to be either CPU-bound or I/O bound. That is, some threads spend a lot of time using the CPU to do computations, and others spend a lot of time waiting for relatively slow I/O operations to complete. I/O operations, in that context, can be waiting for user input, disk or network accesses.

This also strongly depends on the system the tasks are running on. A server or HPC (high performance computing) workload generally consists mostly of CPU-bound tasks whereas desktop or mobile workloads are mainly I/O bound.

The Linux OS runs on all those system and is therefore designed to handle all types of tasks. To cope with that, its scheduler needs to be responsive for I/O bound tasks and efficient for CPU-bound ones. If tasks run for longer periods of time, they can accomplish more work but responsiveness suffers. If time periods per task get shorter, the system can react faster on I/O events but more time is spent on running the scheduling algorithm between task switches and efficiency suffers. This puts the scheduler in a give and take situation which needs it to find a balance between those two requirements.

2.2 Real Time vs. Normal

Furthermore, tasks running on Linux can explicitly be classified as *real time* (RT) or *normal* tasks. Real time tasks have strict timing requirements and therefore priority over any other task on the system. Different scheduling policies are used for RT and normal tasks which are implemented in the scheduler by using *scheduling classes*.

2.3 Task Priority Values

Higher priorities in the kernel have a numerical smaller value. Real time priorities range from 1 (highest) – 99 whereas normal priorities range from 100 – 139 (lowest). There can, however, be confusion when using system calls or scheduler library functions to set priorities. There, the numerical order can be reversed and/or mapped to different values (nice values).

3 Scheduling Classes

The Linux scheduler is modular, enabling different algorithms/policies to schedule different types of tasks. An algorithm's implementation is wrapped in a so called *scheduling class*. A scheduling class offers an interface to the main scheduler skeleton which it can use to handle tasks according to the implemented algorithm.

The sched_class data structure can be found in include/linux/sched.h:

```
struct sched_class {
        const struct sched_class *next;

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*yield_task) (struct rq *rq);
        bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool preempt);

        void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);

        struct task_struct * (*pick_next_task) (struct rq *rq);
        void (*put_prev_task) (struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
        int  (*select_task_rq)(struct task_struct *p, int sd_flag, int flags);

        void (*pre_schedule) (struct rq *this_rq, struct task_struct *task);
        void (*post_schedule) (struct rq *this_rq);
        void (*task_waking) (struct task_struct *task);
        void (*task_woken) (struct rq *this_rq, struct task_struct *task);

        void (*set_cpus_allowed)(struct task_struct *p,
                                 const struct cpumask *newmask);

        void (*rq_online)(struct rq *rq);
        void (*rq_offline)(struct rq *rq);
#endif

        void (*set_curr_task) (struct rq *rq);
        void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
        void (*task_fork) (struct task_struct *p);

        void (*switched_from) (struct rq *this_rq, struct task_struct *task);
        void (*switched_to) (struct rq *this_rq, struct task_struct *task);
        void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                             int oldprio);

        unsigned int (*get_rr_interval) (struct rq *rq,
                                         struct task_struct *task);

#ifdef CONFIG_FAIR_GROUP_SCHED
        void (*task_move_group) (struct task_struct *p, int on_rq);
#endif
};
```

Except for the first one, all members of this struct are function pointers which are used by the scheduler skeleton to call the corresponding policy implementation hook.

All existing scheduling classes in the kernel are in a list which is ordered by the priority of the scheduling class. The first member in the struct called *next* is a pointer to the next scheduling class with a lower priority in that list.

The list is used to prioritise tasks of different types before others. In the Linux versions described in this document, the complete list looks like the following:

stop_sched_class → **rt_sched_class** → **fair_sched_class** → **idle_sched_class** → **NULL**

Stop and Idle are special scheduling classes. Stop is used to schedule the per-cpu stop task which

pre-empts everything and can be pre-empted by nothing, and Idle is used to schedule the per-cpu idle task (also called *swapper* task) which is run if no other task is runnable. The other two are for the previously mentioned real time and normal tasks.

4 Main Runqueue

The main per-CPU runqueue data structure is defined in kernel/sched.c. It keeps track of all runnable tasks assigned to a particular CPU and manages various scheduling statistics about CPU load or scheduling domains for load balancing. Furthermore it has:

- a lock to synchronize scheduling operations for this CPU

  ```
  raw_spinlock_t lock;
  ```
- pointers to the task_structs of the currently running, the idle and the stop task

  ```
  struct task_struct *curr, *idle, *stop;
  ```

- runqueue data structures for fair and real time scheduling classes

  ```
  struct cfs_rq cfs;
  struct rt_rq rt;
  ```

5 Scheduler Skeleton

5.1 The Scheduler Entry Point

The main entry point into the process scheduler is the function *schedule()*, defined in kernel/sched.c. This is the function that the rest of the kernel uses to invoke the process scheduler, deciding which process to run and then running it.

Its main goal is to find the next task to be run and assign it to the local variable *next*. In the end, it then executes a context switch to that new task. If no other task than *prev* is found and *prev* is still runnable, it is rescheduled which basically means *schedule()* changes nothing.

Lets look at it in more detail:

```c
/*
 * __schedule() is the main scheduler function.
 */
static void __sched __schedule(void)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq *rq;
        int cpu;

need_resched:
        preempt_disable();
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
        rcu_note_context_switch(cpu);
        prev = rq->curr;

        schedule_debug(prev);

        if (sched_feat(HRTICK))
                hrtick_clear(rq);

        raw_spin_lock_irq(&rq->lock);

        switch_count = &prev->nivcsw;
        if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
                if (unlikely(signal_pending_state(prev->state, prev))) {
                        prev->state = TASK_RUNNING;
                } else {
                        deactivate_task(rq, prev, DEQUEUE_SLEEP);
                        prev->on_rq = 0;

                        /*
                         * If a worker went to sleep, notify and ask workqueue
                         * whether it wants to wake up a task to maintain
                         * concurrency.
                         */
                        if (prev->flags & PF_WQ_WORKER) {
                                struct task_struct *to_wakeup;

                                to_wakeup = wq_worker_sleeping(prev, cpu);
                                if (to_wakeup)
                                        try_to_wake_up_local(to_wakeup);
                        }
                }
                switch_count = &prev->nvcsw;
        }

        pre_schedule(rq, prev);

        if (unlikely(!rq->nr_running))
                idle_balance(cpu, rq);

        put_prev_task(rq, prev);
```

```
        next = pick_next_task(rq);
        clear_tsk_need_resched(prev);
        rq->skip_clock_update = 0;

        if (likely(prev != next)) {
                rq->nr_switches++;
                rq->curr = next;
                ++*switch_count;

                context_switch(rq, prev, next); /* unlocks the rq */
                /*
                 * The context switch have flipped the stack from under us
                 * and restored the local variables which were saved when
                 * this task called schedule() in the past. prev == current
                 * is still correct, but it can be moved to another cpu/rq.
                 */
                cpu = smp_processor_id();
                rq = cpu_rq(cpu);
        } else
                raw_spin_unlock_irq(&rq->lock);

        post_schedule(rq);

        preempt_enable_no_resched();
        if (need_resched())
                goto need_resched;
}
```

Since the Linux kernel is pre-emptive, it can happen that a task executing code in kernel space is involuntarily pre-empted by a higher priority task. This pauses the pre-empted task within an unfinished kernel space operation that is only continued when it is scheduled next. Therefore, the first thing, the schedule function does is disabling pre-emption by calling *preempt_disable()* so the scheduling thread can not be pre-empted during critical operations.

Secondly, it establishes another protection by locking the current CPU's runqueue lock since only one thread at a time is allowed to modify the runqueue.

Next, *schedule()* examines the state of the previously executed task in *prev*. If it is not runnable and has not been pre-empted in kernel mode, then it should be removed from the runqueue. However, if it has nonblocked pending signals, its state is set to *TASK_RUNNING* and it is left in the runqueue. This means *prev* gets another chance to be selected for execution.

To remove a task from the runqueue, *deactivate_task()* is called which internally calls the *dequeue_task()* hook of the task's scheduling class.

```
static void dequeue_task(struct rq *rq, struct task_struct *p, int flags)
{
        update_rq_clock(rq);
        sched_info_dequeued(p);
        p->sched_class->dequeue_task(rq, p, flags);
}
```

The next action is to check if any runnable tasks exist in the CPU's runqueue. If not, *idle_balance()* is called to get some runnable tasks from other CPUs (see Load Balancing).

put_prev_task() is a scheduling class hook that informs the task's class that the given task is about to be switched out of the CPU.

Now the corresponding class is asked to pick the next suitable task to be scheduled on the CPU by calling the hook *pick_next_task()*. This is followed by clearing the *need_resched* flag which might have been set previously to invoke the *schedule()* function call in the first place.

The *need_resched* is a flag in the *task_struct* and regularly checked by the kernel. It is a way of telling the kernel that another task deserves to run and *schedule()* should be executed as soon as possible.

```
/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
        const struct sched_class *class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in
         * the fair class we can call that function directly:
         */
        if (likely(rq->nr_running == rq->cfs.nr_running)) {
                p = fair_sched_class.pick_next_task(rq);
                if (likely(p))
                        return p;
        }

        for_each_class(class) {
                p = class->pick_next_task(rq);
                if (p)
                        return p;
        }

        BUG(); /* the idle class will always have a runnable task */
}
```

pick_next_task() is also implemented in sched.c. It iterates through our list of scheduling classes to
find the class with the highest priority that has a runnable task (see Scheduling Classes above). If
the class is found, the scheduling class hook is called. Since most tasks are handled by the
sched_fair class, a short cut to this class is implemented in the beginning of the function.

Now *schedule()* checks if *pick_next_task()* found a new task or if it picked the same task again that
was running before. If the latter is the case, no task switch is performed and the current task just
keeps running. If a new task is found, which is the more likely case, the actual task switch is
executed by calling *context_switch()*. Internally, *context_switch()* switches to the new task's
memory map and swaps register state and stack.

To finish up, the runqueue is unlocked and pre-emption is reenabled. In case pre-emption was
requested during the time in which it was disabled, *schedule()* is run again right away.

5.2 Calling the Scheduler

After seeing the entry point into the scheduler, lets now have a look at when the *schedule()* function
is actually called. There are three main occasions when that happens in kernel code:

1. Regular runtime update of currently scheduled task

The function scheduler_tick() is called regularly by a timer interrupt. Its purpose is to update
runqueue clock, CPU load and runtime counters of the currently running task.

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(void)
{
        int cpu = smp_processor_id();
        struct rq *rq = cpu_rq(cpu);
        struct task_struct *curr = rq->curr;

        sched_clock_tick();

        raw_spin_lock(&rq->lock);
        update_rq_clock(rq);
        update_cpu_load_active(rq);
        curr->sched_class->task_tick(rq, curr, 0);
```

```
        raw_spin_unlock(&rq->lock);

        perf_event_task_tick();
#ifdef CONFIG_SMP
        rq->idle_at_tick = idle_cpu(cpu);
        trigger_load_balance(rq, cpu);
#endif
}
```

You can see, that *scheduler_tick()* calls the scheduling class hook *task_tick()* which runs the regular task update for the corresponding class. Internally, the scheduling class can decide if a new task needs to be scheduled an would set the *need_resched* flag for the task which tells the kernel to invoke *schedule()* as soon as possible.

At the end of *scheduler_tick()* you can also see that load balancing is invoked if SMP is configured.

2. Currently running task goes to sleep

The process of going to sleep to wait for a specific event to happen is implemented in the Linux kernel for multiple occasions. It usually follows a certain pattern:

```
/* 'q' is the wait queue we wish to sleep on */
DEFINE_WAIT(wait);

add_wait_queue(q, &wait);
while (!condition) { /* condition is the event that we are waiting for */
        prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
        if (signal_pending(current))
                /* handle signal */
        schedule();
}
finish_wait(&q, &wait);
```

The calling task would create a wait queue and add itself to it. It would then start a loop that waits until a certain condition becomes true. In the loop, it would set its own task state to either *TASK_INTERRUPTIBLE* or *TASK_UNITERRUPTIBLE*. If the former is the case, it can be woken up for a pending signal which can then be handled.

If the needed event did not occur yet, it would call *schedule()* and go to sleep. *schedule()* would then remove the task from the runqueue (See Scheduler Entry Point). If the condition becomes true, the loop is exited and the task is removed from the wait queue.

You can see here, that *schedule()* is always called right before a task goes to sleep, to pick another task to run next.

3. Sleeping task wakes up

The code that causes the event the sleeping task is waiting for typically calls *wake_up()* on the corresponding wait queue which eventually ends up in the scheduler function *try_to_wake_up()* (ttwu).

This function does three things:

1. It puts the task to be woken back into the runqueue.

2. It wakes the task up by setting its state to TASK_RUNNING.

3. If the the awakened task has higher priority than the currently running task, the *need_resched* flag is set to invoke *schedule()*.

```
/**
 * try_to_wake_up - wake up a thread
 * @p: the thread to be awakened
 * @state: the mask of task states that can be woken
```

```
 * @wake_flags: wake modifier flags (WF_*)
 *
 * Put it on the run-queue if it's not already there. The "current"
 * thread is always on the run-queue (except when the actual
 * re-schedule is in progress), and as such you're allowed to do
 * the simpler "current->state = TASK_RUNNING" to mark yourself
 * runnable without the overhead of this.
 *
 * Returns %true if @p was woken up, %false if it was already running
 * or @state didn't match @p's state.
 */
static int
try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
{
        unsigned long flags;
        int cpu, success = 0;

        smp_wmb();
        raw_spin_lock_irqsave(&p->pi_lock, flags);
        if (!(p->state & state))
                goto out;

        success = 1; /* we're going to change ->state */
        cpu = task_cpu(p);

        if (p->on_rq && ttwu_remote(p, wake_flags))
                goto stat;

#ifdef CONFIG_SMP
        /*
         * If the owning (remote) cpu is still in the middle of schedule() with
         * this task as prev, wait until its done referencing the task.
         */
        while (p->on_cpu) {
#ifdef __ARCH_WANT_INTERRUPTS_ON_CTXSW
                /*
                 * In case the architecture enables interrupts in
                 * context_switch(), we cannot busy wait, since that
                 * would lead to deadlocks when an interrupt hits and
                 * tries to wake up @prev. So bail and do a complete
                 * remote wakeup.
                 */
                if (ttwu_activate_remote(p, wake_flags))
                        goto stat;
#else
                cpu_relax();
#endif
        }
        /*
         * Pairs with the smp_wmb() in finish_lock_switch().
         */
        smp_rmb();

        p->sched_contributes_to_load = !!task_contributes_to_load(p);
        p->state = TASK_WAKING;

        if (p->sched_class->task_waking)
                p->sched_class->task_waking(p);

        cpu = select_task_rq(p, SD_BALANCE_WAKE, wake_flags);
        if (task_cpu(p) != cpu) {
                wake_flags |= WF_MIGRATED;
                set_task_cpu(p, cpu);
        }
#endif /* CONFIG_SMP */

        ttwu_queue(p, cpu);
stat:
        ttwu_stat(p, cpu, wake_flags);
out:
        raw_spin_unlock_irqrestore(&p->pi_lock, flags);

        return success;
}
```

After some initial error checking and some SMP magic, the function *ttwu_queue()* is called which does the wake up work.

```
static void ttwu_queue(struct task_struct *p, int cpu)
{
        struct rq *rq = cpu_rq(cpu);

#if defined(CONFIG_SMP)
        if (sched_feat(TTWU_QUEUE) && cpu != smp_processor_id()) {
                sched_clock_cpu(cpu); /* sync clocks x-cpu */
                ttwu_queue_remote(p, cpu);
                return;
        }
#endif

        raw_spin_lock(&rq->lock);
        ttwu_do_activate(rq, p, 0);
        raw_spin_unlock(&rq->lock);
}

static void
ttwu_do_activate(struct rq *rq, struct task_struct *p, int wake_flags)
{
#ifdef CONFIG_SMP
        if (p->sched_contributes_to_load)
                rq->nr_uninterruptible--;
#endif

        ttwu_activate(rq, p, ENQUEUE_WAKEUP | ENQUEUE_WAKING);
        ttwu_do_wakeup(rq, p, wake_flags);
}
```

This function locks the runqueue and calls *ttwu_do_activate()* which further calls *ttwu_activate()* to perform step 1 and *ttwu_do_wakeup()* to perform step 2 and 3.

```
static void ttwu_activate(struct rq *rq, struct task_struct *p, int en_flags)
{
        activate_task(rq, p, en_flags);
        p->on_rq = 1;

        /* if a worker is waking up, notify workqueue */
        if (p->flags & PF_WQ_WORKER)
                wq_worker_waking_up(p, cpu_of(rq));
}

/*
 * activate_task - move a task to the runqueue.
 */
static void activate_task(struct rq *rq, struct task_struct *p, int flags)
{
        if (task_contributes_to_load(p))
                rq->nr_uninterruptible--;

        enqueue_task(rq, p, flags);
        inc_nr_running(rq);
}

static void enqueue_task(struct rq *rq, struct task_struct *p, int flags)
{
        update_rq_clock(rq);
        sched_info_queued(p);
        p->sched_class->enqueue_task(rq, p, flags);
}
```

If you follow the chain of *ttwu_activate()* you end up in a call to the corresponding scheduling class hook to *enqueue_task()* which we already saw in *schedule()* to put the task back into the runqueue.

```
/*
 * Mark the task runnable and perform wakeup-preemption.
 */
```

```c
static void
ttwu_do_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
{
        trace_sched_wakeup(p, true);
        check_preempt_curr(rq, p, wake_flags);

        p->state = TASK_RUNNING;
#ifdef CONFIG_SMP
        if (p->sched_class->task_woken)
                p->sched_class->task_woken(rq, p);

        if (rq->idle_stamp) {
                u64 delta = rq->clock - rq->idle_stamp;
                u64 max = 2*sysctl_sched_migration_cost;

                if (delta > max)
                        rq->avg_idle = max;
                else
                        update_avg(&rq->avg_idle, delta);
                rq->idle_stamp = 0;
        }
#endif
}

static void check_preempt_curr(struct rq *rq, struct task_struct *p, int flags)
{
        const struct sched_class *class;

        if (p->sched_class == rq->curr->sched_class) {
                rq->curr->sched_class->check_preempt_curr(rq, p, flags);
        } else {
                for_each_class(class) {
                        if (class == rq->curr->sched_class)
                                break;
                        if (class == p->sched_class) {
                                resched_task(rq->curr);
                                break;
                        }
                }
        }

        /*
         * A queue event has occurred, and we're going to schedule.  In
         * this case, we can save a useless back to back clock update.
         */
        if (rq->curr->on_rq && test_tsk_need_resched(rq->curr))
                rq->skip_clock_update = 1;
}
```

ttwu_do_wakeup() checks if the current task needs to be pre-empted by the task being woken up which is now in the runqueue. The function *check_preempt_curr()* ends up calling the corresponding hook into the scheduling class internally might set the *need_resched* flag. Afterwards the task's state is set to *TASK_RUNNING*, which completes the wake up process.

6 Short Scheduling Algorithm History

This is a short history of the development of the scheduling algorithm used in Linux.

- 1995 1.2 Circular Runqueue with processes scheduled in a Round Robin system

- 1999 2.2 introduced scheduling classes and with that rt-tasks, non-preemptible tasks and non-rt-tasks, introduced support for SMP

- 2001 2.4 O(N) scheduler, split time into epochs where each task was allowed a certain time slice, iterating through N runnable tasks and applying goodness function to determine next task

- 2003 2.6 O(1) scheduler used multiple runqueues for each priority, it was a more efficient and scalable version of O(N), it introduced a bonus system for interactive vs. batch tasks

- 2008 2.6.23 Completely Fair Scheduler

7 Completely Fair Scheduler (CFS)

Lets now describe the scheduling algorithm used for the NORMAL category tasks: The current scheduler is the completely fair scheduler by Ingo Molnar based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas. CFS' implementation hides behind the *fair_sched_class* in /kernel/sched_fair.c.

The CFS algorithm is based on the idea of an ideal multi-tasking processor. Such a processor would run all currently active tasks literally at the same time while each task gets a portion of its processing power. For example, if two tasks would be runnable, they would run at the same time with 50% processing power each. That means, assuming the two tasks started at the same time, the execution time or runtime of each task is at any moment in time exactly the same, therefore completely fair. You could also say, each tasks runs for an infinitesimal small amount of time but with full processing power.

Since it is physically not possible to drive current processors in that way and it is highly inefficient to run for very short times due to switching cost, CFS tries to approximate this behaviour as closely as possible. It keeps track of the runtime of each runnable task, called virtual runtime, and tries to maintain an overall balance between all runnable tasks at all times.

The concept is based on giving all runnable tasks a fair share of the processor. The total share grows and shrinks with the total number of running processes and the relative share is inflated or deflated based on the tasks' priorities.

7.1 Time Slices vs. Virtual Runtime

In CFS, the traditional model of an epoch divided into fixed time slice for each task does not exist any more. Instead, a *virtual runtime (vruntime)* was introduced for each task. In every call of *scheduler_tick()* for a scheduled CFS task, its *vruntime* is updated with the elapsed time since it was scheduled. As soon as the *vruntime* of another task in the runqueue is smaller than the current tasks', a rescheduling is executed and the task with the smallest *vruntime* is selected to run next.

To avoid an immense overhead of scheduling runs due to frequent task switches, a scheduling latency and a minimum task runtime granularity were introduced.

The *target scheduling latency* (TSL) is a constant which is used to calculate a minimum task runtime. Lets assume TSL is 20ms and we have 2 running tasks of the same priority. Each task would then run for a total of 10ms before pre-empting in favour of the other.

If the number of tasks grows, this portion of runtime would shrink towards zero which again would lead to an inefficient switching rate. To counter this, a constant *minimum granularity* is used to stretch the TSL depending on the amount of running tasks.

7.2 Priorities

A task is prioritised above another by weighting how fast its *vruntime* grows while being scheduled. The time delta since the task was scheduled is weighted by the task's priority. That means, a high priority tasks' *vruntime* grows slower than the *vruntime* of a low priority one.

7.3 Handling I/O and CPU bound tasks

The previous scheduling algorithm, the O(1) scheduler, tried to use heuristic of sleep and runtimes to determine if a task is I/O or CPU bound. It would then benefit one or the other. As it turned out

later, this concept did not work quite as satisfying as expected due to the complex and error prone heuristics.

CFS' concept of giving tasks a fair share of the processor works quite well and is easy to apply. I would like to explain how this works using the example given in Robert Love's *Linux Kernel Development*:

Consider a system with two runnable tasks: a text editor and a video encoder. The text editor is I/O-bound because it spends nearly all its time waiting for user key presses. However, when the text editor does receive a key press, the user expects the editor to respond immediately. Conversely, the video encoder is processor-bound.

Aside from reading the raw data stream from the disk and later writing the resulting video, the encoder spends all its time applying the video codec to the raw data, easily consuming 100% of the processor. The video encoder does not have any strong time constraints on when it runs—if it started running now or in half a second, the user could not tell and would not care. Of course, the sooner it finishes the better, but latency is not a primary concern. In this scenario, ideally the scheduler gives the text editor a larger proportion of the available processor than the video encoder, because the text editor is interactive. We have two goals for the text editor. First, we want it to have a large amount of processor time available to it; not because it needs a lot of processor (it does not) but because we want it to always have processor time available the moment it needs it. Second, we want the text editor to pre-empt the video encoder the moment it wakes up (say, when the user presses a key). This can ensure the text editor has good interactive performance and is responsive to user input.

CFS solves this issue in the following way: Instead of assigning the text editor a specific priority and timeslice, it guarantees the text editor a specific proportion of the processor. If the video encoder and text editor are the only running processes and both are at the same nice level, this proportion would be 50% — each would be guaranteed half of the processor's time. Because the text editor spends most of its time blocked, waiting for user key presses, it does not use anywhere near 50% of the processor. Conversely, the video encoder is free to use more than its allotted 50%, enabling it to finish the encoding quickly.

The crucial concept is what happens when the text editor wakes up. Our primary goal is to ensure it runs immediately upon user input. In this case, when the editor wakes up, CFS notes that it is allotted 50% of the processor but has used considerably less. Specifically, CFS determines that the text editor has run for less time than the video encoder. Attempting to give all processes a fair share of the processor, it then pre-empts the video encoder and enables the text editor to run. The text editor runs, quickly processes the user's key press, and again sleeps, waiting for more input. As the text editor has not consumed its allotted 50%, we continue in this manner, with CFS always enabling the text editor to run when it wants and the video encoder to run the rest of the time.

7.4 Spawning a new Task

CFS maintains a *vruntime_min* value, which is a monotonic increasing value tracking the smallest *vruntime* among all tasks in the runqueue. This value is used for new tasks that are added to the runqueue to give them a chance of being scheduled quickly.

7.5 The Runqueue – Red-Black-Tree

As a runqueue, CFS uses a Red-Black-Tree data structure. This data structure is a self balancing binary search tree. Following a certain set of rules, no path in the tree will ever be more than twice as long as any other. Furthermore, operations on the tree occur in O(log n) time, which allows

insertion and deletion of nodes to be quick and efficient.

Each node represents a task and they are ordered by the task's *vruntime*. That means the left most node in the tree always points to the task with the smallest *vruntime*, say the task that has the gravest need for the processor. Caching of the left most node makes it quick and easy to access.

7.6 Fair Group Scheduling

Group scheduling is another way to bring fairness to scheduling, particularly in the face of tasks that spawn many other tasks. Consider a server that spawns many tasks to parallelise incoming connections. Instead of all tasks being treated fairly uniformly, CFS introduces groups to account for this behaviour. The server process that spawns tasks share their *vruntimes* across the group (in a hierarchy), while the single task maintains its own independent *vruntime*. In this way, the single task receives roughly the same scheduling time as the group.

Let's say that next to the server tasks another user has only one task running on the machine. Without group scheduling, the second user would be treated unfairly in opposition to the server. With group scheduling, CFS would first try to be fair to the group and then fair to the processes in the group. Therefore, both users would get 50% share of the CPU.

Groups can be configured via a /proc interface.

8 CFS Implementation Details

8.1 Data Structures

With the CFS scheduler, a structure called *sched_entity* was introduced into the Linux scheduler. It is mainly used for time accounting for the single tasks and was added as the *se* member to each task's *task_struct*. Its defined in include/linux/sched.h:

```
struct sched_entity {
        struct load_weight      load;           /* for load-balancing */
        struct rb_node          run_node;
        struct list_head        group_node;
        unsigned int            on_rq;

        u64                     exec_start;
        u64                     sum_exec_runtime;
        u64                     vruntime;
        u64                     prev_sum_exec_runtime;

        u64                     nr_migrations;

#ifdef CONFIG_SCHEDSTATS
        struct sched_statistics statistics;
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
        struct sched_entity    *parent;
        /* rq on which this entity is (to be) queued: */
        struct cfs_rq          *cfs_rq;
        /* rq "owned" by this entity/group: */
        struct cfs_rq          *my_q;
#endif
};
```

Another CFS related field, called *cfs,* was added to the main runqueue data structure. It is of the type *cfs_rq,* which is implemented in kernel/sched.c. It contains a list of pointers to all running CFS tasks, the root of CFS' red-black-tree, a pointer to the left most node, *min_vruntime,* pointers to previously and currently scheduled tasks and additional members for group and smp scheduling and load balancing. The priority of the task is encoded in the *load_weight* data structure.

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
        struct load_weight load;
        unsigned long nr_running;

        u64 exec_clock;
        u64 min_vruntime;
#ifndef CONFIG_64BIT
        u64 min_vruntime_copy;
#endif

        struct rb_root tasks_timeline;
        struct rb_node *rb_leftmost;

        struct list_head tasks;
        struct list_head *balance_iterator;

        /*
         * '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, *next, *last, *skip;

#ifdef CONFIG_SCHED_DEBUG
        unsigned int nr_spread_over;
#endif
```

```
#ifdef CONFIG_FAIR_GROUP_SCHED
        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 lrqs 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.
         */
        int on_list;
        struct list_head leaf_cfs_rq_list;
        struct task_group *tg;/* group that "owns" this runqueue */

#ifdef CONFIG_SMP
        /*
         * the part of load.weight contributed by tasks
         */
        unsigned long task_weight;

        /*
         *   h_load = weight * f(tg)
         *
         * Where f(tg) is the recursive weight fraction assigned to
         * this group.
         */
        unsigned long h_load;

        /*
         * Maintaining per-cpu shares distribution for group scheduling
         *
         * load_stamp is the last time we updated the load average
         * load_last is the last time we updated the load average and saw load
         * load_unacc_exec_time is currently unaccounted execution time
         */
        u64 load_avg;
        u64 load_period;
        u64 load_stamp, load_last, load_unacc_exec_time;

        unsigned long load_contribution;
#endif
#endif
};
```

8.2 Time Accounting

As mentioned above, *vruntime* is used to track the virtual runtime of runnable tasks in CFS' red-black-tree. The *scheduler_tick()* function of the scheduler skeleton regularly calls the *task_tick()* hook into CFS. This hook internally calls *task_tick_fair()* which is the entry point into the CFS task update:

```
/*
 * scheduler tick hitting a task of our scheduling class:
 */
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
        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, queued);
        }
}
```

task_tick_fair() calls *entity_tick()* for the tasks scheduling entity and corresponding runqueue. *entity_tick()* executes two main tasks: First, it updates runtime statistics for the currently scheduled task and secondly, it checks if the current task needs to be pre-empted.

```
static void
```

```
entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
        /*
         * Update run-time statistics of the 'current'.
         */
        update_curr(cfs_rq);

        /*
         * Update share accounting for long-running entities.
         */
        update_entity_shares_tick(cfs_rq);

#ifdef CONFIG_SCHED_HRTICK
        /*
         * queued ticks are scheduled to match the slice, so don't bother
         * validating it and just reschedule.
         */
        if (queued) {
                resched_task(rq_of(cfs_rq)->curr);
                return;
        }
        /*
         * don't let the period tick interfere with the hrtick preemption
         */
        if (!sched_feat(DOUBLE_TICK) &&
                        hrtimer_active(&rq_of(cfs_rq)->hrtick_timer))
                return;
#endif

        if (cfs_rq->nr_running > 1 || !sched_feat(WAKEUP_PREEMPT))
                check_preempt_tick(cfs_rq, curr);
}
```

update_curr() is the responsible function to update the current task's runtime statistics. It calculates the elapsed time since the current task was scheduled last and passes the result, *delta_exec*, on to *__update_curr()*.

```
static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_of(cfs_rq)->clock_task;
        unsigned long delta_exec;

        if (unlikely(!curr))
                return;

        /*
         * Get the amount of time the current task was running
         * since the last time we changed load (this cannot
         * overflow on 32 bits):
         */
        delta_exec = (unsigned long)(now - curr->exec_start);
        if (!delta_exec)
                return;

        __update_curr(cfs_rq, curr, delta_exec);
        curr->exec_start = now;

        if (entity_is_task(curr)) {
                struct task_struct *curtask = task_of(curr);

                trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
                cpuacct_charge(curtask, delta_exec);
                account_group_exec_runtime(curtask, delta_exec);
        }
}
```

The runtime delta is weighted by current task's priority, which is encoded in *load_weigth*, and the result is added to the current task's *vruntime*. This is the location where *vruntime* grows faster or slower, depending on the task' priority. You can also see that *__update_curr()* updates

min_vruntime.

```
/*
 * Update the current task's runtime statistics. Skip current tasks that
 * are not in our scheduling class.
 */
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
              unsigned long delta_exec)
{
        unsigned long delta_exec_weighted;

        schedstat_set(curr->statistics.exec_max,
                        max((u64)delta_exec, curr->statistics.exec_max));

        curr->sum_exec_runtime += delta_exec;
        schedstat_add(cfs_rq, exec_clock, delta_exec);
        delta_exec_weighted = calc_delta_fair(delta_exec, curr);

        curr->vruntime += delta_exec_weighted;
        update_min_vruntime(cfs_rq);

#if defined CONFIG_SMP && defined CONFIG_FAIR_GROUP_SCHED
        cfs_rq->load_unacc_exec_time += delta_exec;
#endif
}
```

In addition to this regular update, *update_current()* is also called if the corresponding task becomes runnable or goes to sleep.

Back to *entity_tick()*. After the current task was updated, *check_preempt_tick()* is called to satisfy CFS' concept of giving each task a fair share. *vruntime* of the current process is checked against *vruntime* of the left most task in the red-black-tree to decide if a process switch is necessary.

```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        unsigned long ideal_runtime, delta_exec;

        ideal_runtime = sched_slice(cfs_rq, curr);
        delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
        if (delta_exec > ideal_runtime) {
                resched_task(rq_of(cfs_rq)->curr);
                /*
                 * The current task ran long enough, ensure it doesn't get
                 * re-elected due to buddy favours.
                 */
                clear_buddies(cfs_rq, curr);
                return;
        }

        /*
         * Ensure that a task that missed wakeup preemption by a
         * narrow margin doesn't have to wait for a full slice.
         * This also mitigates buddy induced latencies under load.
         */
        if (!sched_feat(WAKEUP_PREEMPT))
                return;

        if (delta_exec < sysctl_sched_min_granularity)
                return;

        if (cfs_rq->nr_running > 1) {
                struct sched_entity *se = __pick_first_entity(cfs_rq);
                s64 delta = curr->vruntime - se->vruntime;

                if (delta < 0)
                        return;
```

```
                if (delta > ideal_runtime)
                        resched_task(rq_of(cfs_rq)->curr);
        }
}
```

sched_slice() returns the target latency runtime for the current task, depending on the number of runnable processes. If its delta runtime is larger than this amount, the *need_resched* flag is set for this task.

If not, the runtime is checked against the minimum granularity. If the task ran longer than that and more than one tasks are in total in the red-black-tree, a comparison with the left most node is done. If the *vruntime* difference between those two is positive, that means the current task ran longer than the left most one, and larger than the target latency runtime calculated above, *need_resched* flag is set as well to reschedule ASAP.

8.3 Modifying the Red-Black-Tree

In Scheduler Skeleton you could see how tasks where deactivated when they were removed from the runqueue or activated when woken up in *try_to_wake_up()*. For the CFS scheduling class, *enqueue_task_fair() and dequeue_task_fair()* are called which end up in *enqueue_entity()* and *dequeue_entity()* to modify the red-black-tree.

In *enqueue_entity()*, you can see how the tasks *vruntime* is updated, as mentioned above, and that *__enqueue_entity()* is called. This function is responsible for the red-black-tree insert magic that fulfils the data structures properties. You can also find caching operations of the left most node there.

```
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
        /*
         * Update the normalized vruntime before updating min_vruntime
         * through callig update_curr().
         */
        if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
                se->vruntime += cfs_rq->min_vruntime;

        /*
         * Update run-time statistics of the 'current'.
         */
        update_curr(cfs_rq);
        update_cfs_load(cfs_rq, 0);
        account_entity_enqueue(cfs_rq, se);
        update_cfs_shares(cfs_rq);

        if (flags & ENQUEUE_WAKEUP) {
                place_entity(cfs_rq, se, 0);
                enqueue_sleeper(cfs_rq, se);
        }

        update_stats_enqueue(cfs_rq, se);
        check_spread(cfs_rq, se);
        if (se != cfs_rq->curr)
                __enqueue_entity(cfs_rq, se);
        se->on_rq = 1;

        if (cfs_rq->nr_running == 1)
                list_add_leaf_cfs_rq(cfs_rq);
}
```

Removing a node works in a similar way. First, *vruntime* is updated and then the task is removed from the tree using *__dequeue_entity()*.

```
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
```

```
{
        /*
         * Update run-time statistics of the 'current'.
         */
        update_curr(cfs_rq);

        update_stats_dequeue(cfs_rq, se);
        if (flags & DEQUEUE_SLEEP) {
#ifdef CONFIG_SCHEDSTATS
                if (entity_is_task(se)) {
                        struct task_struct *tsk = task_of(se);

                        if (tsk->state & TASK_INTERRUPTIBLE)
                                se->statistics.sleep_start = rq_of(cfs_rq)->clock;
                        if (tsk->state & TASK_UNINTERRUPTIBLE)
                                se->statistics.block_start = rq_of(cfs_rq)->clock;
                }
#endif
        }

        clear_buddies(cfs_rq, se);

        if (se != cfs_rq->curr)
                __dequeue_entity(cfs_rq, se);
        se->on_rq = 0;
        update_cfs_load(cfs_rq, 0);
        account_entity_dequeue(cfs_rq, se);

        /*
         * Normalize the entity after updating the min_vruntime because the
         * update can refer to the ->curr item and we need to reflect this
         * movement in our normalized position.
         */
        if (!(flags & DEQUEUE_SLEEP))
                se->vruntime -= cfs_rq->min_vruntime;

        update_min_vruntime(cfs_rq);
        update_cfs_shares(cfs_rq);
}
```

The additional calls you see are necessary for group scheduling, scheduling statistic updates and the buddy system which is used for picking the next task to run.

8.4 Picking the next runnable task

The main scheduler function *schedule()* calls *pick_next_task()* of the scheduling class with the highest priority that has runnable tasks. If this is called for the CFS class, the class hook calls *pick_next_task_fair()*.

If no tasks are in this class, *NULL* is returned immediately. Otherwise *pick_next_entity()* is called to select the next task from the tree. This is then forwarded to *set_next_entity()* which removes it from the tree since scheduled processes are not allowed in there. The while loop is used to account for fair group scheduling.

```
static struct task_struct *pick_next_task_fair(struct rq *rq)
{
        struct task_struct *p;
        struct cfs_rq *cfs_rq = &rq->cfs;
        struct sched_entity *se;

        if (!cfs_rq->nr_running)
                return NULL;

        do {
                se = pick_next_entity(cfs_rq);
                set_next_entity(cfs_rq, se);
                cfs_rq = group_cfs_rq(se);
        } while (cfs_rq);
```

```
        p = task_of(se);
        hrtick_start_fair(rq, p);

        return p;
}
```

In *pick_next_entity()* not necessarily the left most task is picked to run next. A buddy system gives a certain degree of freedom in selecting the next task to run. This is supposed to benefit cache locality and group scheduling.

```
/*
 * Pick the next process, keeping these things in mind, in this order:
 * 1) keep things fair between processes/task groups
 * 2) pick the "next" process, since someone really wants that to run
 * 3) pick the "last" process, for cache locality
 * 4) do not run the "skip" process, if something else is available
 */
static struct sched_entity *pick_next_entity(struct cfs_rq *cfs_rq)
{
        struct sched_entity *se = __pick_first_entity(cfs_rq);
        struct sched_entity *left = se;

        /*
         * Avoid running the skip buddy, if running something else can
         * be done without getting too unfair.
         */
        if (cfs_rq->skip == se) {
                struct sched_entity *second = __pick_next_entity(se);
                if (second && wakeup_preempt_entity(second, left) < 1)
                        se = second;
        }

        /*
         * Prefer last buddy, try to return the CPU to a preempted task.
         */
        if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1)
                se = cfs_rq->last;

        /*
         * Someone really wants this to run. If it's not unfair, run it.
         */
        if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1)
                se = cfs_rq->next;

        clear_buddies(cfs_rq, se);

        return se;
}
```

__pick_first_entity() picks the left most node from the tree, which is very easy and fast as you can see below:

```
static struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
        struct rb_node *left = cfs_rq->rb_leftmost;

        if (!left)
                return NULL;

        return rb_entry(left, struct sched_entity, run_node);
}
```

9 Soft Real-Time Scheduling

The Linux scheduler supports soft real-time (RT) scheduling. This means that is can effectively schedule tasks that have strict timing requirements. However, while the kernel is usually capable of meeting very strict RT scheduling deadlines, it does not guarantee that deadlines will be met.

The corresponding scheduling class is *rt_sched_class* which is implemented in kernel/sched_rt.c. RT tasks have priority over CFS tasks.

9.1 Scheduling Modes

Tasks handled by the RT scheduler can be configured in two different scheduling modes:

SCHED_FIFO: A scheduled FIFO task has no time slice and will be scheduled until it terminates, yields the processor voluntarily or a higher priority task becomes runnable.

SCHED_RR: An RR task will be scheduled for a fixed time slice and then pre-empted in a round robin fashion by tasks with the same priority. That means, as soon as the task's time slice is over, it is set to the end of the queue and its slice is refilled. It can also be pre-empted by tasks with higher priority before the time slice is over.

9.2 Priorities

With its priority implementation, the RT class follows the same concept, the previous O(1) scheduler did. It uses multiple runqueues where one is reserved for each priority. That way, operations like adding, removing or finding task with highest priority can be achieved in O(1) time.

9.3 Real Time Throttling

Occasionally, you can see throttling or bandwith operations in the RT scheduler implementation. These were added to add some safety to SCHED_FIFO tasks. They assign an RT task group with FIFO tasks a certain bandwith for a processor (95% by default) before they are pre-empted if they want to or not. This is supposed to add more security to the kernel against blocking FIFO tasks. It, however, started a discussion among kernel developers and its future is not entirely set.

9.4 Implementation Details

Data Structures

Like CFS, the RT scheduler has its own scheduling entity and runqueue data structure which are added as members to *task_struct* and the main runqueue.

sched_rt_entity is implemented in /include/linux/sched.h. It has fields for time slice accounting, a pointer to the priority list it belongs to and group scheduling related members.

```
struct sched_rt_entity {
        struct list_head run_list;
        unsigned long timeout;
        unsigned int time_slice;
        int nr_cpus_allowed;

        struct sched_rt_entity *back;
#ifdef CONFIG_RT_GROUP_SCHED
        struct sched_rt_entity*parent;
        /* rq on which this entity is (to be) queued: */
```

```
        struct rt_rq            *rt_rq;
        /* rq "owned" by this entity/group: */
        struct rt_rq            *my_q;
#endif
};
```

rt_rq is implemented in kernel/sched.c. The first field holds the priority arrays. Almost all other fields are for SMP and group scheduling.

```
/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
        struct rt_prio_array active;
        unsigned long rt_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
        struct {
                int curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
                int next; /* next highest */
#endif
        } highest_prio;
#endif
#ifdef CONFIG_SMP
        unsigned long rt_nr_migratory;
        unsigned long rt_nr_total;
        int overloaded;
        struct plist_head pushable_tasks;
#endif
        int rt_throttled;
        u64 rt_time;
        u64 rt_runtime;
        /* Nests inside the rq lock: */
        raw_spinlock_t rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
        unsigned long rt_nr_boosted;

        struct rq *rq;
        struct list_head leaf_rt_rq_list;
        struct task_group *tg;
#endif
};
```

Time Accounting

scheduler_tick() calls *task_tick_rt()* to update the current task's time slice. It is pretty straight forward: In the beginning runtime statistics of the current task and its runqueue are updated in *update_curr_rt()*. Then the function returns if current is a FIFO task.

If not, the RR task's timeslice is reduced by one. If it reaches 0, it is set back to a default value and put back to the end of its runqueue if other tasks are available in the queue. Additionally the *need_resched* flag is set.

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
        update_curr_rt(rq);

        watchdog(rq, p);

        /*
         * RR tasks need a special form of timeslice management.
         * FIFO tasks have no timeslices.
         */
        if (p->policy != SCHED_RR)
                return;

        if (--p->rt.time_slice)
                return;
```

```
        p->rt.time_slice = DEF_TIMESLICE;

        /*
         * Requeue to the end of queue if we are not the only element
         * on the queue:
         */
        if (p->rt.run_list.prev != p->rt.run_list.next) {
                requeue_task_rt(rq, p, 0);
                set_tsk_need_resched(p);
        }
}
```

Picking the Next Task

As said earlier, picking the next task to run can be done with constant time complexity. It starts in *pick_next_task_rt()* which immediately calls *_pick_next_task_rt()* to do the actual work.

```
static struct task_struct *pick_next_task_rt(struct rq *rq)
{
        struct task_struct *p = _pick_next_task_rt(rq);

        /* The running task is never eligible for pushing */
        if (p)
                dequeue_pushable_task(rq, p);

#ifdef CONFIG_SMP
        /*
         * We detect this state here so that we can avoid taking the RQ
         * lock again later if there is no need to push
         */
        rq->post_schedule = has_pushable_tasks(rq);
#endif

        return p;
}
```

If no tasks are runnable, NULL is returned and a different scheduling class will be searched for tasks. *pick_next_rt_entity()* gets the next task with the highest priority from the runqueue.

```
static struct task_struct *_pick_next_task_rt(struct rq *rq)
{
        struct sched_rt_entity *rt_se;
        struct task_struct *p;
        struct rt_rq *rt_rq;

        rt_rq = &rq->rt;

        if (!rt_rq->rt_nr_running)
                return NULL;

        if (rt_rq_throttled(rt_rq))
                return NULL;

        do {
                rt_se = pick_next_rt_entity(rq, rt_rq);
                BUG_ON(!rt_se);
                rt_rq = group_rt_rq(rt_se);
        } while (rt_rq);

        p = rt_task_of(rt_se);
        p->se.exec_start = rq->clock_task;

        return p;
}
```

In *pick_next_rt_entity()* you can see how a bitmap is used for all priority levels to quickly find the highest priority queue with a runnable task. The priority queue itself is a simple linked list and the next runnable task can be found at its head.

```c
static struct sched_rt_entity *pick_next_rt_entity(struct rq *rq,
                                                   struct rt_rq *rt_rq)
{
        struct rt_prio_array *array = &rt_rq->active;
        struct sched_rt_entity *next = NULL;
        struct list_head *queue;
        int idx;

        idx = sched_find_first_bit(array->bitmap);
        BUG_ON(idx >= MAX_RT_PRIO);

        queue = array->queue + idx;
        next = list_entry(queue->next, struct sched_rt_entity, run_list);

        return next;
}
```

Adding and removing a task from the priority queues is also pretty straight forward. There is merely the right queue to be found and a bit to be set or removed from the priority bitmap. Therefore, I will not supply any further details here.

10 Load Balancing on SMP Systems

Load balancing was introduced with the main goal of improving the performance of SMP systems by offloading tasks from busy CPUs to less busy or idle ones. The Linux scheduler checks regularly how the task load is spread throughout the system and performs load balancing if necessary.

The complexity lies in serving a variety of SMP system topologies. There are systems with multiple physical cores where tasks might suffer more from a cache flush due to being moved to a different CPU than from being scheduled on a more busy CPU. Other systems might support hyper threading with shared caches where this scenario is more flexible towards task migration due to shared caches. NUMA architectures create situations where different nodes have different access speeds to different areas of main memory.

In order to tackle this topology variety, *scheduling domains* were introduced in the 2.6 Linux kernel. They are a way of hierarchically grouping all available CPUs in the system which gives the kernel a way of characterising the underlying core topology.

10.1 Scheduling Domains and Groups

A *scheduling domain* is a set of CPUs which share properties and scheduling policies, and which can be balanced against each other.

Each domain can contain one ore more *scheduling groups* which are treated as a single unit by the domain. When the scheduler tries to balance the load within a domain, it tries to even out the load carried by each *scheduling group* without worrying directly about what is happening within the group.
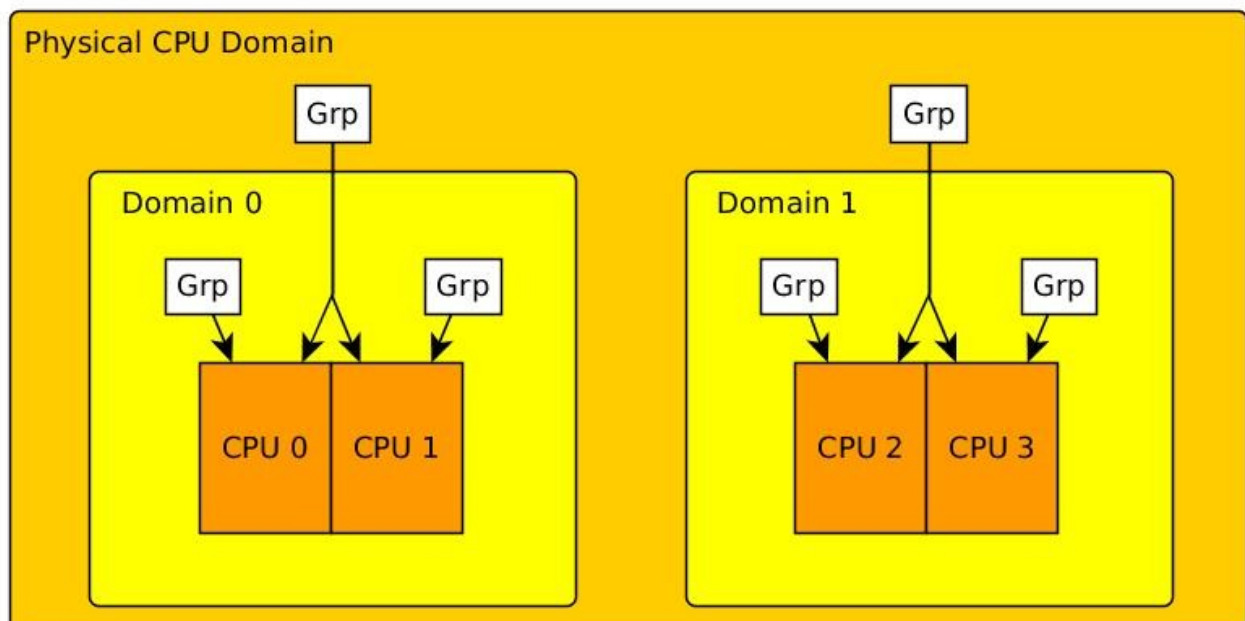
Imagine a system with two physical CPUs which are both hyperthreaded which gives us in total four logical CPUs. If the system starts up, the kernel would divide the logical cores into the two level domain hierarchy you can see in the picture.

Each hyperthreaded processor is put into exactly one group while either two are in a single domain. These two domains are then again put into a top level domain which holds the complete processor.

It contains two groups which both have two hyperthreaded cores each.

If this were a NUMA system, it would have multiple domains which look like the above diagram; each of those domains would represent one NUMA node. The hierarchy would have a third, system-level domain which contains all of the NUMA nodes.

10.2 Load Balancing

Each scheduling domain has a **balancing policy** set which is valid on that level of the hierarchy. The policy parameters include how often attempts should be made to balance loads across the domain, how far the loads on the component processors are allowed to get out of sync before a balancing attempt is made, how long a process can sit idle before it is considered to no longer have any significant cache affinity.

On top of that, various **policy flags** specify the behaviour for certain situations, such as: A CPU goes idle; should the balancing code look for a task to pull? A task wakes up or is spawned; which CPU should it be scheduled on?

An **active load balancing** is executed regularly which moves up the scheduling domain hierarchy and checks all groups along the way if they got out of balance. If so it does a balancing attempt considering the policy rules of the corresponding domain.

10.3 Implementation Details

Data Structures

Two data structures were added to include/linux/sched.h to divide the cores into a balancing hierarchy. *sched_domain* represents a scheduling domain and *sched_group* a scheduling group:

```
struct sched_domain {
        /* These fields must be setup */
        struct sched_domain *parent;  /* top domain must be null terminated */
        struct sched_domain *child;   /* bottom domain must be null terminated */
        struct sched_group *groups;   /* the balancing groups of the domain */
        unsigned long min_interval;   /* Minimum balance interval ms */
        unsigned long max_interval;   /* Maximum balance interval ms */
        unsigned int busy_factor;     /* less balancing by factor if busy */
        unsigned int imbalance_pct;   /* No balance until over watermark */
        unsigned int cache_nice_tries;      /* Leave cache hot tasks for # tries */
        unsigned int busy_idx;
        unsigned int idle_idx;
        unsigned int newidle_idx;
        unsigned int wake_idx;
        unsigned int forkexec_idx;
        unsigned int smt_gain;
        int flags;                    /* See SD_* */
        int level;

        /* Runtime fields. */
        unsigned long last_balance;   /* init to jiffies. units in jiffies */
        unsigned int balance_interval;      /* initialise to 1. units in ms. */
        unsigned int nr_balance_failed; /* initialise to 0 */

        u64 last_update;

...

        unsigned int span_weight;
        /*
         * Span of all CPUs in this domain.
         *
         * NOTE: this field is variable length. (Allocated dynamically
         * by attaching extra space to the end of the structure,
```

```
         * depending on how many CPUs the kernel has booted up with)
         */
        unsigned long span[0];
};

struct sched_group {
        struct sched_group *next;     /* Must be a circular list */
        atomic_t ref;

        unsigned int group_weight;
        struct sched_group_power *sgp;

        /*
         * The CPUs this group covers.
         *
         * NOTE: this field is variable length. (Allocated dynamically
         * by attaching extra space to the end of the structure,
         * depending on how many CPUs the kernel has booted up with)
         */
        unsigned long cpumask[0];
};
```

In include/linux/topology.h or the corresponding architecture version, you can see how flags and values for the scheduling domain policy are set.

In *sched_group* you can see a field called sgp which is of the type *sched_group_power*. The concept of total CPU power of a group was introduced to further specify the topology of a processor. Even though a hyperthreaded core appears as an independent unit, it has in reality significantly less processing power than a second physical core. Two separate processors would have a CPU power of two, while a hyperthreaded pair would have something closer to 1.1. During load balancing, the kernel tries to maximise the CPU power value to increase the overall throughput of the system.

Active Balancing

Active balancing is performed regularly on each CPU. During active balancing, the kernel walks up the domain hierarchy, starting at the current CPU's domain, and checks each scheduling domain to see if it is due to be balanced, and initiates a balancing operation if so.

During scheduler initialisation, a soft irq handler is registered which performs the regular load balancing. It is triggered in *scheduler_tick()* with a call to *trigger_load_balance()* (see Scheduler Skeleton). *trigger_load_balance()* checks a timer and if balancing is due, it fires the soft irq with the corresponding flag SCHED_SOFTIRQ.

```
/*
 * Trigger the SCHED_SOFTIRQ if it is time to do periodic load balancing.
 */
static inline void trigger_load_balance(struct rq *rq, int cpu)
{
        /* Don't need to rebalance while attached to NULL domain */
        if (time_after_eq(jiffies, rq->next_balance) &&
            likely(!on_null_domain(cpu)))
                raise_softirq(SCHED_SOFTIRQ);
#ifdef CONFIG_NO_HZ
        else if (nohz_kick_needed(rq, cpu) && likely(!on_null_domain(cpu)))
                nohz_balancer_kick(cpu);
#endif
}
```

The function registered as irq handler is *run_rebalance_domains()* which calls *rebalance_domains()* to perform the actual work.

```
/*
 * run_rebalance_domains is triggered when needed from the scheduler tick.
 * Also triggered for nohz idle balancing (with nohz_balancing_kick set).
 */
```

```c
static void run_rebalance_domains(struct softirq_action *h)
{
        int this_cpu = smp_processor_id();
        struct rq *this_rq = cpu_rq(this_cpu);
        enum cpu_idle_type idle = this_rq->idle_at_tick ?
                                                CPU_IDLE : CPU_NOT_IDLE;

        rebalance_domains(this_cpu, idle);

        /*
         * If this cpu has a pending nohz_balance_kick, then do the
         * balancing on behalf of the other idle cpus whose ticks are
         * stopped.
         */
        nohz_idle_balance(this_cpu, idle);
}
```

rebalance_domains() then walks up the domain hierarchy and calls *load_balance()* if the domain
has the SD_LOAD_BALANCE flag set and its balancing interval is expired. The balancing interval
of a domain is in jiffies and updated after each balancing run.

Note that active balancing is a pull operation for the executing CPU. It would pull a task from an
overloaded CPU to the current one to rebalance tasks but it would not push one of. The function
executing this pull operation is *load_balance()*. If it is able to find an imbalanced group, it moves
one or more tasks over to the current CPU and returns a value larger than 0.

```c
/*
 * It checks each scheduling domain to see if it is due to be balanced,
 * and initiates a balancing operation if so.
 *
 * Balancing parameters are set up in arch_init_sched_domains.
 */
static void rebalance_domains(int cpu, enum cpu_idle_type idle)
{
        int balance = 1;
        struct rq *rq = cpu_rq(cpu);
        unsigned long interval;
        struct sched_domain *sd;
        /* Earliest time when we have to do rebalance again */
        unsigned long next_balance = jiffies + 60*HZ;
        int update_next_balance = 0;
        int need_serialize;

        update_shares(cpu);

        rcu_read_lock();
        for_each_domain(cpu, sd) {
                if (!(sd->flags & SD_LOAD_BALANCE))
                        continue;

                interval = sd->balance_interval;
                if (idle != CPU_IDLE)
                        interval *= sd->busy_factor;

                /* scale ms to jiffies */
                interval = msecs_to_jiffies(interval);
                interval = clamp(interval, 1UL, max_load_balance_interval);

                need_serialize = sd->flags & SD_SERIALIZE;

                if (need_serialize) {
                        if (!spin_trylock(&balancing))
                                goto out;
                }

                if (time_after_eq(jiffies, sd->last_balance + interval)) {
                        if (load_balance(cpu, rq, sd, idle, &balance)) {
                                /*
                                 * We've pulled tasks over so either we're no
                                 * longer idle.
                                 */
```

```
                                    idle = CPU_NOT_IDLE;
                            }
                            sd->last_balance = jiffies;
                    }
                    if (need_serialize)
                            spin_unlock(&balancing);
out:
                    if (time_after(next_balance, sd->last_balance + interval)) {
                            next_balance = sd->last_balance + interval;
                            update_next_balance = 1;
                    }

                    /*
                     * Stop the load balance at this level. There is another
                     * CPU in our sched group which is doing load balancing more
                     * actively.
                     */
                    if (!balance)
                            break;
            }
            rcu_read_unlock();

            /*
             * next_balance will be updated only when there is a need.
             * When the cpu is attached to null domain for ex, it will not be
             * updated.
             */
            if (likely(update_next_balance))
                    rq->next_balance = next_balance;
}
```

load_balance() calls *find_busiest_group()* which looks for an imbalance in the given *sched_domain* and returns the busiest group if it finds one. If the system is in balance and no group is found, *load_balance()* returns.

If a group was returned, it is passed on to *find_busiest_queue()* which returns the runqueue of the busiest logical CPU in that group.

load_balance() then searches the resulting runqueue for tasks to swap over to the current CPU's queue by calling *move_tasks()*. The imbalance parameter, which was set in *find_busiest_group()*, specifies the amount of tasks that should be moved. It can happen that all tasks on the queue are pinned to it due to cache affinity. In that case, *load_balance()* searches again but excludes the previously found CPU.

```
/*
 * Check this_cpu to ensure it is balanced within domain. Attempt to move
 * tasks if there is an imbalance.
 */
static int load_balance(int this_cpu, struct rq *this_rq,
                        struct sched_domain *sd, enum cpu_idle_type idle,
                        int *balance)
{
        int ld_moved, all_pinned = 0, active_balance = 0;
        struct sched_group *group;
        unsigned long imbalance;
        struct rq *busiest;
        unsigned long flags;
        struct cpumask *cpus = __get_cpu_var(load_balance_tmpmask);

        cpumask_copy(cpus, cpu_active_mask);

        schedstat_inc(sd, lb_count[idle]);

redo:
        group = find_busiest_group(sd, this_cpu, &imbalance, idle,
                                   cpus, balance);

        if (*balance == 0)
                goto out_balanced;
```

```c
        if (!group) {
                schedstat_inc(sd, lb_nobusyg[idle]);
                goto out_balanced;
        }

        busiest = find_busiest_queue(sd, group, idle, imbalance, cpus);
        if (!busiest) {
                schedstat_inc(sd, lb_nobusyq[idle]);
                goto out_balanced;
        }

        BUG_ON(busiest == this_rq);

        schedstat_add(sd, lb_imbalance[idle], imbalance);

        ld_moved = 0;
        if (busiest->nr_running > 1) {
                /*
                 * Attempt to move tasks. If find_busiest_group has found
                 * an imbalance but busiest->nr_running <= 1, the group is
                 * still unbalanced. ld_moved simply stays zero, so it is
                 * correctly treated as an imbalance.
                 */
                all_pinned = 1;
                local_irq_save(flags);
                double_rq_lock(this_rq, busiest);
                ld_moved = move_tasks(this_rq, this_cpu, busiest,
                                      imbalance, sd, idle, &all_pinned);
                double_rq_unlock(this_rq, busiest);
                local_irq_restore(flags);

                /*
                 * some other cpu did the load balance for us.
                 */
                if (ld_moved && this_cpu != smp_processor_id())
                        resched_cpu(this_cpu);

                /* All tasks on this runqueue were pinned by CPU affinity */
                if (unlikely(all_pinned)) {
                        cpumask_clear_cpu(cpu_of(busiest), cpus);
                        if (!cpumask_empty(cpus))
                                goto redo;
                        goto out_balanced;
                }
        }

...

        goto out;

out_balanced:
        schedstat_inc(sd, lb_balanced[idle]);

        sd->nr_balance_failed = 0;

out_one_pinned:
        /* tune up the balancing interval */
        if ((all_pinned && sd->balance_interval < MAX_PINNED_INTERVAL) ||
                        (sd->balance_interval < sd->max_interval))
                sd->balance_interval *= 2;

        ld_moved = 0;
out:
        return ld_moved;
}
```

An energy saving related tweak is hidden in *find_busiest_group()*. If the SD_POWERSAVINGS_BALANCE flag is set in the domains policy and no busiest group is found, *find_busiest_group()* would look for the least loaded group in the *sched_domain*, so that it's CPUs can be put to idle. This feature, however, is not activated in the Android kernel and removed from the Ubuntu one.

Idle Balancing

Idle balancing is invoked as soon as a CPU goes idle. Therefore, it is called by *schedule()* for the CPU executing the current scheduling thread if its runqueue becomes empty (see Scheduler Sekeleton).

Like active balancing, *idle_balance()* is implemented in sched_fair.c. First, it checks if the average idle period of the idle runqueue is larger than the cost of migrating a task over to it, that means it checks if it is worth getting a task from somewhere else or if it is better to just wait since the next task is likely to wake up soon anyway.

If migrating a task makes sense, *idle_balance()* pretty much works like *rebalance_domains().* It walks up the domain hierarchy and calls *idle_balance()* for domains that have the SD_LOAD_BALANCE and the idle balance specific SD_BALANCE_NEWIDLE flag set.

If one or more tasks were pulled over, the hierarchy walk is terminated and *idle_balance()* returns.

```c
/*
 * idle_balance is called by schedule() if this_cpu is about to become
 * idle. Attempts to pull tasks from other CPUs.
 */
static void idle_balance(int this_cpu, struct rq *this_rq)
{
        struct sched_domain *sd;
        int pulled_task = 0;
        unsigned long next_balance = jiffies + HZ;

        this_rq->idle_stamp = this_rq->clock;

        if (this_rq->avg_idle < sysctl_sched_migration_cost)
                return;

        /*
         * Drop the rq->lock, but keep IRQ/preempt disabled.
         */
        raw_spin_unlock(&this_rq->lock);

        update_shares(this_cpu);
        rcu_read_lock();
        for_each_domain(this_cpu, sd) {
                unsigned long interval;
                int balance = 1;

                if (!(sd->flags & SD_LOAD_BALANCE))
                        continue;

                if (sd->flags & SD_BALANCE_NEWIDLE) {
                        /* If we've pulled tasks over stop searching: */
                        pulled_task = load_balance(this_cpu, this_rq,
                                                sd, CPU_NEWLY_IDLE, &balance);
                }

                interval = msecs_to_jiffies(sd->balance_interval);
                if (time_after(next_balance, sd->last_balance + interval))
                        next_balance = sd->last_balance + interval;
                if (pulled_task) {
                        this_rq->idle_stamp = 0;
                        break;
                }
        }
        rcu_read_unlock();

        raw_spin_lock(&this_rq->lock);

        if (pulled_task || time_after(jiffies, this_rq->next_balance)) {
                /*
                 * We are going idle. next_balance may be set based on
                 * a busy processor. So reset next_balance.
                 */
                this_rq->next_balance = next_balance;
```

```
        }
}
```

Selecting a runqueue for a new task

A third spot where balancing decisions need to be made is when a task wakes up or is created and needs to be placed on a runqueue. This runqueue should be selected considering the overall task balance of the system.

Each scheduling class implements its own strategy to handle their tasks and provides a hook (*select_task_rq()*) that can be called by the scheduler skeleton (kernel/sched.c) to execute it. It is called for three different occasions which are each marked by a corresponding domain flag.

1. **SD_BALANCE_EXEC** flag is used in *sched_exec()*. This function is called if a task starts a new one by using the *exec()* system call. A new task at this point has a small memory and cache footprint which gives the kernel a good balancing opportunity.

2. **SD_BALANCE_FORK** flag is used in *wake_up_new_task()*. This function is called if a newly created task is woken up for the first time.

3. **SD_BALANCE_WAKE** flag is used in *try_to_wake_up().* If a task that was running before wakes up it usually has some kind of cache affinity which needs to be considered while selecting a good queue to schedule it on.

11 Real Time Load Balancing

The active and idle balancing implementation in the CFS class makes sure that only CFS tasks are affected. Real time tasks are handled by the real time scheduler. It applies push and pull operations for overloaded RT queues considering the following cases:

1. **A task wakes up and needs to be scheduled**: This scenario is handled by the *select_task_rq()* implementation of the RT algorithm.

2. **A lower prio task wakes up on a runqueue with a higher prio task running**: a push operation is applied for the lower prio task.

3. **A high prio task wakes up on a runqueue with a lower prio task running and pre-empts it**: also a push operation is applied for the lower prio task.

4. **Priorities change on a runqueue because a task lowers its prio and thereby causes a previously lower-prio task to have the higher prio**: A pull operation will look for higher prio tasks to pull to this runqueue.

The design goal the real time load balancing pursues is that of a system-wide strict real-time priority scheduling. That means, the real time scheduler needs to make sure that the N highest-priority RT tasks are running at any given point in time, where N is the number of CPUs.

11.1 Root Domains and CPU Priorities

The given design goal requires the scheduler to get a quick and efficient overview of all runqueues in the system to make scheduling decisions. This leads to scalability issues with a growing number of CPUs. Therefore, the notion of a root domain was introduced which partitions CPUs into subsets that are used by a process or a group of processes. All real-time scheduling decisions are made only within the scope of a root domain.

Another concept that was introduced to deal with the given design goal are CPU priorities. CPU priorities mirror the priority of the highest priority RT task in a root domain. A two-dimensional bitmap is used to manage the CPU priorities. One dimension for the different priorities and one for the CPUs in that priority. CPU priorities are implemented in /kernel/sched_cpupri.c and /kernel/sched_cpupri.h.

A *root_domain* struct has a bit array for overloaded CPUs it contains and a *cpupri* struct with a bitmap of CPU priorities. A CPU counts as overloaded with RT task if it has more than one runnable RT task in its runqueue.

```
/*
 * We add the notion of a root-domain which will be used to define per-domain
 * variables. Each exclusive cpuset essentially defines an island domain by
 * fully partitioning the member cpus from any other cpuset. Whenever a new
 * exclusive cpuset is created, we also create and attach a new root-domain
 * object.
 *
 */
struct root_domain {
        atomic_t refcount;
        atomic_t rto_count;
        struct rcu_head rcu;
        cpumask_var_t span;
        cpumask_var_t online;

        /*
         * The "RT overload" flag: it gets set if a CPU has more than
         * one runnable RT task.
         */
```

```
        cpumask_var_t rto_mask;
        struct cpupri cpupri;
};

struct cpupri_vec {
        raw_spinlock_t lock;
        int            count;
        cpumask_var_t mask;
};

struct cpupri {
        struct cpupri_vec pri_to_cpu[CPUPRI_NR_PRIORITIES];
        long              pri_active[CPUPRI_NR_PRI_WORDS];
        int               cpu_to_pri[NR_CPUS];
};
```

With the cpupri bitmask above properly maintained, the function *cpupri_find()* can be used to
quickly find a low priority CPU to push a higher priority task to. If a priority level is non-empty and
lower than the priority of the task being pushed, the lowest_mask is set to the mask corresponding
to the priority level selected. This mask is then used by the push algorithm to compute the best CPU
to which to push the task, based on affinity, topology and cache characteristics.

```
/**
 * cpupri_find - find the best (lowest-pri) CPU in the system
 * @cp: The cpupri context
 * @p: The task
 * @lowest_mask: A mask to fill in with selected CPUs (or NULL)
 *
 * Note: This function returns the recommended CPUs as calculated during the
 * current invocation.  By the time the call returns, the CPUs may have in
 * fact changed priorities any number of times.  While not ideal, it is not
 * an issue of correctness since the normal rebalancer logic will correct
 * any discrepancies created by racing against the uncertainty of the current
 * priority configuration.
 *
 * Returns: (int)bool - CPUs were found
 */
int cpupri_find(struct cpupri *cp, struct task_struct *p,
                struct cpumask *lowest_mask)
{
        int                 idx     = 0;
        int                 task_pri = convert_prio(p->prio);

        for_each_cpupri_active(cp->pri_active, idx) {
                struct cpupri_vec *vec  = &cp->pri_to_cpu[idx];

                if (idx >= task_pri)
                        break;

                if (cpumask_any_and(&p->cpus_allowed, vec->mask) >= nr_cpu_ids)
                        continue;

                if (lowest_mask) {
                        cpumask_and(lowest_mask, &p->cpus_allowed, vec->mask);

                        /*
                         * We have to ensure that we have at least one bit
                         * still set in the array, since the map could have
                         * been concurrently emptied between the first and
                         * second reads of vec->mask.  If we hit this
                         * condition, simply act as though we never hit this
                         * priority level and continue on.
                         */
                        if (cpumask_any(lowest_mask) >= nr_cpu_ids)
                                continue;
                }

                return 1;
        }

        return 0;
}
```

The function electing a final cpu to push a task to from the lowest mask is called *find_lowest_rq()*.
It is implemented in kernel/sched_rt.c. If *cpupri_find()* returns a mask with possible CPU targets,
find_lowest_rq() would first look if the CPU is among them that last executed the task to be pushed.
This CPU is most likely to have the hottest cache and therefore the best choice.

If not, *find_lowest_rq()* walks up the scheduling domain hierarchy to find a CPU that is logically
closest to the hot cache data of the current CPU and also in the lowest prio map.

If this search also does not return any usable results, just any of the CPUs in the mask is selected
and returned.

```
static int find_lowest_rq(struct task_struct *task)
{
        struct sched_domain *sd;
        struct cpumask *lowest_mask = __get_cpu_var(local_cpu_mask);
        int this_cpu = smp_processor_id();
        int cpu      = task_cpu(task);

        /* Make sure the mask is initialized first */
        if (unlikely(!lowest_mask))
                return -1;

        if (task->rt.nr_cpus_allowed == 1)
                return -1; /* No other targets possible */

        if (!cpupri_find(&task_rq(task)->rd->cpupri, task, lowest_mask))
                return -1; /* No targets found */

        /*
         * At this point we have built a mask of cpus representing the
         * lowest priority tasks in the system.  Now we want to elect
         * the best one based on our affinity and topology.
         *
         * We prioritize the last cpu that the task executed on since
         * it is most likely cache-hot in that location.
         */
        if (cpumask_test_cpu(cpu, lowest_mask))
                return cpu;

        /*
         * Otherwise, we consult the sched_domains span maps to figure
         * out which cpu is logically closest to our hot cache data.
         */
        if (!cpumask_test_cpu(this_cpu, lowest_mask))
                this_cpu = -1; /* Skip this_cpu opt if not among lowest */

        rcu_read_lock();
        for_each_domain(cpu, sd) {
                if (sd->flags & SD_WAKE_AFFINE) {
                        int best_cpu;

                        /*
                         * "this_cpu" is cheaper to preempt than a
                         * remote processor.
                         */
                        if (this_cpu != -1 &&
                            cpumask_test_cpu(this_cpu, sched_domain_span(sd))) {
                                rcu_read_unlock();
                                return this_cpu;
                        }

                        best_cpu = cpumask_first_and(lowest_mask,
                                                     sched_domain_span(sd));
                        if (best_cpu < nr_cpu_ids) {
                                rcu_read_unlock();
                                return best_cpu;
                        }
                }
        }
        rcu_read_unlock();
```

```
        /*
         * And finally, if there were no matches within the domains
         * just give the caller *something* to work with from the compatible
         * locations.
         */
        if (this_cpu != -1)
                return this_cpu;

        cpu = cpumask_any(lowest_mask);
        if (cpu < nr_cpu_ids)
                return cpu;
        return -1;
}
```

11.2 *Push Operation*

The task push operation in the RT scheduler is invoked after the scheduler skeleton scheduled a new task on the current CPU. The scheduling class hook *post_schedule()* is called which is currently only implemented by the RT scheduling class. Internally, it calls *push_rt_task()* for all tasks on the current CPU's runqueue.

This function looks if the runqueue is overloaded and, if so, tries to move non running RT tasks to another applicable CPU until moving a task fails. If a task was moved, a future call to *schedule()* is invoked by setting the *need_resched* flag.

It uses *find_lock_lowest_rq()* which internally calls *find_lowest_rq()* but additionally locks the queue if found.

```
/*
 * If the current CPU has more than one RT task, see if the non
 * running task can migrate over to a CPU that is running a task
 * of lesser priority.
 */
static int push_rt_task(struct rq *rq)
{
        struct task_struct *next_task;
        struct rq *lowest_rq;

        if (!rq->rt.overloaded)
                return 0;

        next_task = pick_next_pushable_task(rq);
        if (!next_task)
                return 0;

retry:
        if (unlikely(next_task == rq->curr)) {
                WARN_ON(1);
                return 0;
        }

        /*
         * It's possible that the next_task slipped in of
         * higher priority than current. If that's the case
         * just reschedule current.
         */
        if (unlikely(next_task->prio < rq->curr->prio)) {
                resched_task(rq->curr);
                return 0;
        }

        /* We might release rq lock */
        get_task_struct(next_task);

        /* find_lock_lowest_rq locks the rq if found */
        lowest_rq = find_lock_lowest_rq(next_task, rq);
        if (!lowest_rq) {
                struct task_struct *task;
                /*
```

```
                    * find lock_lowest_rq releases rq->lock
                    * so it is possible that next_task has migrated.
                    *
                    * We need to make sure that the task is still on the same
                    * run-queue and is also still the next task eligible for
                    * pushing.
                    */
                   task = pick_next_pushable_task(rq);
                   if (task_cpu(next_task) == rq->cpu && task == next_task) {
                           /*
                            * If we get here, the task hasn't moved at all, but
                            * it has failed to push.  We will not try again,
                            * since the other cpus will pull from us when they
                            * are ready.
                            */
                           dequeue_pushable_task(rq, next_task);
                           goto out;
                   }

                   if (!task)
                           /* No more tasks, just exit */
                           goto out;

                   /*
                    * Something has shifted, try again.
                    */
                   put_task_struct(next_task);
                   next_task = task;
                   goto retry;
           }

           deactivate_task(rq, next_task, 0);
           set_task_cpu(next_task, lowest_rq->cpu);
           activate_task(lowest_rq, next_task, 0);

           resched_task(lowest_rq->curr);

           double_unlock_balance(rq, lowest_rq);

out:
           put_task_struct(next_task);

           return 1;
}
```

11.3 *Pull Operation*

The pull operation is called by the *pre_schedule()* hook. It is currently also only implemented by the
RT class and invoked at the beginning of the *schedule()* function. It will check if the highest priority
of the tasks in the current CPU's runqueue is smaller than the priority of the task than ran last. If that
is the case, it will call *pull_rt_task()* to pull another RT task with a higher priority than the to-be-
scheduled task from a different CPU.

```
static void pre_schedule_rt(struct rq *rq, struct task_struct *prev)
{
        /* Try to pull RT tasks here if we lower this rq's prio */
        if (rq->rt.highest_prio.curr > prev->prio)
                pull_rt_task(rq);
}
```

pull_rt_task() will go through each CPU of the same root domain as the current CPU. It looks for
tasks that are the next highest priority task on the potential source CPU to pull them over to the
current CPU. If a task is found, it is pulled over. *Schedule()* is not invoked since it is about to be
executed anyway.

```
static int pull_rt_task(struct rq *this_rq)
{
        int this_cpu = this_rq->cpu, ret = 0, cpu;
        struct task_struct *p;
        struct rq *src_rq;
```

```c
        if (likely(!rt_overloaded(this_rq)))
                return 0;

        for_each_cpu(cpu, this_rq->rd->rto_mask) {
                if (this_cpu == cpu)
                        continue;

                src_rq = cpu_rq(cpu);

                /*
                 * Don't bother taking the src_rq->lock if the next highest
                 * task is known to be lower-priority than our current task.
                 * This may look racy, but if this value is about to go
                 * logically higher, the src_rq will push this task away.
                 * And if its going logically lower, we do not care
                 */
                if (src_rq->rt.highest_prio.next >=
                    this_rq->rt.highest_prio.curr)
                        continue;

                /*
                 * We can potentially drop this_rq's lock in
                 * double_lock_balance, and another CPU could
                 * alter this_rq
                 */
                double_lock_balance(this_rq, src_rq);

                /*
                 * Are there still pullable RT tasks?
                 */
                if (src_rq->rt.rt_nr_running <= 1)
                        goto skip;

                p = pick_next_highest_task_rt(src_rq, this_cpu);

                /*
                 * Do we have an RT task that preempts
                 * the to-be-scheduled task?
                 */
                if (p && (p->prio < this_rq->rt.highest_prio.curr)) {
                        WARN_ON(p == src_rq->curr);
                        WARN_ON(!p->on_rq);

                        /*
                         * There's a chance that p is higher in priority
                         * than what's currently running on its cpu.
                         * This is just that p is wakeing up and hasn't
                         * had a chance to schedule. We only pull
                         * p if it is lower in priority than the
                         * current task on the run queue
                         */
                        if (p->prio < src_rq->curr->prio)
                                goto skip;

                        ret = 1;

                        deactivate_task(src_rq, p, 0);
                        set_task_cpu(p, this_cpu);
                        activate_task(this_rq, p, 0);
                        /*
                         * We continue with the search, just in
                         * case there's an even higher prio task
                         * in another runqueue. (low likelihood
                         * but possible)
                         */
                }
skip:
                double_unlock_balance(this_rq, src_rq);
        }

        return ret;
}
```

11.4 Select a Runqueue for a Waking Task

As described in the load balancing chapter for CFS tasks, the *select_task_rq()* scheduling class hook is called as soon as a task wakes up again or for the first time. Apart from the additional push and pull operations, this hook is also implemented by the RT scheduler.

If the currently running task on this CPU's runqueue is a RT task, if its priority is higher and if we can move the task to be woken up, then try to find another runqueue we can wake the task up on. If not, wake it up on the same CPU and let the RT scheduler do the rest.

This function also uses *find_lowest_rq()* to find a CPU applicable for the task.

```
static int
select_task_rq_rt(struct task_struct *p, int sd_flag, int flags)
{
        struct task_struct *curr;
        struct rq *rq;
        int cpu;

        if (sd_flag != SD_BALANCE_WAKE)
                return smp_processor_id();

        cpu = task_cpu(p);
        rq = cpu_rq(cpu);

        rcu_read_lock();
        curr = ACCESS_ONCE(rq->curr); /* unlocked access */

        if (curr && unlikely(rt_task(curr)) &&
            (curr->rt.nr_cpus_allowed < 2 ||
             curr->prio <= p->prio) &&
            (p->rt.nr_cpus_allowed > 1)) {
                int target = find_lowest_rq(p);

                if (target != -1)
                        cpu = target;
        }
        rcu_read_unlock();

        return cpu;
}
```

12. Resources

Robert Love, *Linux Kernel Development*, 3rd edition, 2010

Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*, 3rd edition, Nov 2005

Tim Jones, Inside the Linux 2.6 Completely Fair Scheduler, 15.12.2009 -
http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/

Chandandeep Singh Pabla, Completely Fair Scheduler, 01.08.2009 -
http://www.linuxjournal.com/magazine/completely-fair-scheduler

CFS Scheduler - https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

Josh Aas, Understanding the Linux 2.6.8.1 CPU Scheduler, 17.02.2005 -
http://joshaas.net/linux/linux_cpu_scheduler.pdf

Jonathan Corbet, Schedulers: the plot thickens, 17.04.2007 - http://lwn.net/Articles/230574/

Jonathan Corbet, SCHED_FIFO and realtime throttling, 01.09.2008 -
http://lwn.net/Articles/296419/

Jonathan Corbet, Scheduling domains, 19.04.2004 - http://lwn.net/Articles/80911/

Ankita Garg, Real-Time Linux Kernel Scheduler, 01.08.2009 -
http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler?page=0,4