Blue Volker Seeker – University of Edinburgh 05.12.2013

**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 CPUbound 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:

/\*

\* \_\_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.

\*/

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/Obound 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' redblack-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.

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, systemlevel 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).

\*/

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.

/\*

\* 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;

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.

/\*

\* 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 preempts 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\_activ e , 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-bescheduled 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;

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;

}