Linux Scheduler

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University
Outline

1. Introduction to Linux Scheduler
2. Completely Fair Scheduler (CFS)
3. CFS Implementation in the Kernel
4. Q & A
Introduction to Linux Scheduler
# Evolution of Linux Scheduler

<table>
<thead>
<tr>
<th>Version</th>
<th>CPU Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ Linux 2.4</td>
<td>• Priority-based scheduling and Round-robin scheduling</td>
</tr>
<tr>
<td></td>
<td>• Multilevel feedback queue</td>
</tr>
<tr>
<td></td>
<td>• SMP support was introduced in Linux 2.2</td>
</tr>
<tr>
<td>Linux 2.4 ~ 2.6</td>
<td>• O(n) scheduler</td>
</tr>
<tr>
<td>Linux 2.6 ~ 2.6.22</td>
<td>• O(1) scheduler introduced by Ingo Molnar</td>
</tr>
<tr>
<td>Linux 2.6.23 ~</td>
<td>• CFS (Completely Fair Scheduler) by Ingo Molnar</td>
</tr>
<tr>
<td></td>
<td>• O(log N) complexity</td>
</tr>
</tbody>
</table>
Since 2.6.23 kernel, the Linux scheduler has been made modular

- The modularity is called scheduler classes
- Scheduler classes enable different, pluggable algorithms to coexist, scheduling their own types of processes
- Each scheduler class has a priority
  - The base scheduler code iterates over each scheduler class in order of priority

```
Base Scheduler
```

```
realtime

rt_scheduled.c
```

```
deadline

deadline_scheduled.c
```

```
cfs

cfs_scheduled.c
```

```
idle

idle_task_scheduled.c
```

```
custom

custom_scheduler.c
```

...
Scheduling Class

- Each task belongs to one specific scheduling class

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>dl_sched_class</td>
<td>for real-time task with deadline</td>
<td>SCHED_DEADLINE</td>
</tr>
<tr>
<td>rt_sched_class</td>
<td>for real-time task</td>
<td>SCHED_FIFO SCHED_RR</td>
</tr>
<tr>
<td>fair_sched_class</td>
<td>for time-sharing task</td>
<td>SCHED_NORMAL SCHED_BATCH</td>
</tr>
<tr>
<td>idle_sched_class</td>
<td>for tasks that avoid to disturb other tasks</td>
<td>SCHED_IDLE</td>
</tr>
</tbody>
</table>

- Each scheduling class is linked with one another in a singly linked list, allowing the classes to be iterated

```c
#define sched_class_highest (&stop_sched_class)
#define for_each_class(class)
    for (class = sched_class_highest; class; class = class->next)

sched_class_highest
     └ stop_sched_class └ dl_sched_class └ rt_sched_class └ fair_sched_class └ idle_sched_class
             .next        .next        .next        .next        .next = NULL
```
## Functions in Scheduling Class

<table>
<thead>
<tr>
<th>Function</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue_task</td>
<td>• Put the task into the run queue</td>
</tr>
<tr>
<td></td>
<td>• Increment the nr_running variable</td>
</tr>
<tr>
<td>dequeue_task</td>
<td>• Remove the task from the run queue</td>
</tr>
<tr>
<td></td>
<td>• Decrement the nr_running variable</td>
</tr>
<tr>
<td>yield_task</td>
<td>• Dequeue the task and then enqueue it</td>
</tr>
<tr>
<td>check_preempt_curr</td>
<td>• Check whether the currently running task can be preempted by a new task</td>
</tr>
<tr>
<td>pick_next_task</td>
<td>• Choose the most appropriated task eligible to run next</td>
</tr>
<tr>
<td>set_curr_task</td>
<td>• Change task’s scheduling class or group</td>
</tr>
<tr>
<td>load_balance</td>
<td>• Trigger load balancing code</td>
</tr>
</tbody>
</table>
Tasks in Linux

- Linux internally refers to processes as tasks

- Process descriptor: task_struct

The kernel stores the list of tasks in a circular doubly linked list called the task list.
Task State in Linux

- Existing task calls fork() and creates a new process.
- Task forks.
- Task is terminated.
- Task exits via do_exit.
- Task is preempted by higher priority task.
- Task sleeps on wait queue for a specific event.
- Schedule dispatches task to run: schedule() calls context_switch().
- Event occurs and task is woken up and placed back on the run queue.
Threads in Linux

- There is no concept of a thread
  - Linux implements all threads as standard tasks

- The Linux kernel does not provide any special scheduling semantics or data structures to represent threads
  - A thread is merely a task that shares certain resources with other tasks
  - Each thread has a unique task_struct and appears to the kernel as a normal process—threads just happen to share resources, such as an address space, with other processes
Task Structure

- All tasks in Linux are represented by task_struct
  - Describes the task’s state, its stack, process flags, priority, and much more

```c
struct task_struct {
    volatile long state;  
    void *stack;           
    atomic_t usage;        
    unsigned int flags;    
    unsigned int ptrace;   
    ... 
    int prio, static_prio, normal_prio; 
    unsigned int rt_priority;  
    const struct sched_class *sched_class; 
    struct sched_entity se;     
    struct sched_rt_entity rt;  
    ... 
};
```

```
kernel/sched/rt.c
```11
const struct sched_class rt_sched_class = {

    .next = &fair_sched_class,
    .enqueue_task = enqueue_task_rt,
    .dequeue_task = dequeue_task_rt,
    .yield_task = yield_task_rt,
    .check_preempt_curr = check_preempt_curr_rt,
    ...
};
```

```
kernel/sched/fair.c
```11
const struct sched_class fair_sched_class = {

    .next = &idle_sched_class,
    .enqueue_task = enqueue_task_fair,
    .dequeue_task = dequeue_task_fair,
    .yield_task = yield_task_fair,
    .yield_to_task = yield_to_task_fair,
    .check_preempt_curr = check_preempt_wakeup,
    ...
};
```

```
kernel/sched/idle_task.c
```11
const struct sched_class idle_sched_class = {

    .dequeue_task = dequeue_task_idle,
    .check_preempt_curr = check_preempt_curr_idle,
    ...
};
```
Task Priority

- **Non-real-time priority**
  - Priority range: 100 ~ 139
    - Nice value (-20 ~ 19, default of 0)
  - Priority for normal time-sharing tasks
    - Formerly, SCHED_OTHER
  - A large nice value corresponds to a lower priority

- **Real-time priority**
  - Priority range: 0 ~ 99
  - Priority for real-time tasks
    - SCHED_FIFO, SCHED_RR
  - A smaller value corresponds to a higher priority
Nice Values

- Nice values are the standard priority range used in all Unix systems
  - Different Unix systems apply them in different ways
  - In Mac OS X, the nice value is a control over the absolute time slice allotted to a process

- Linux uses nice values to control the proportion of time slice (weight)
  - Larger nice values correspond to a lower priority—you are being “nice” to the other processes on the system
  - Processes with a lower nice value (higher priority) receive a larger proportion of the system’s processor compared to processes with a higher nice value (lower priority)
Completely Fair Scheduler
Completely Fair Scheduler (CFS)

- CFS is based on a simple concept
  - The goal is to achieve an ideal multitasking system
  - Provide each task CPU time proportional to its weight

\[ C_{\tau_i}(t_1, t_2) = \frac{W(\tau_i)}{S_{\Phi}} \times (t_2 - t_1) \]

- Inspired by Con Kolivas’s work
  - RSD (Rotating Staircase Deadline) scheduler
  - Released BFS (Brain Fuck Scheduler) in August 2009
    - O(1) complexity
Task Weight

- include/linux/sched.c

```c
struct load_weight {
    unsigned long weight, inv_weight;
};
```

- kernel/sched.c

```c
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */ 9548, 7620, 6100, 4904, 3906,
    /* -5 */ 3121, 2501, 1991, 1586, 1277,
    /* 0 */ 1024, 820, 655, 526, 423,
    /* 5 */ 335, 272, 215, 172, 137,
    /* 10 */ 110, 87, 70, 56, 45,
    /* 15 */ 36, 29, 23, 18, 15,
};
```
Run Queue Structure

- Each CPU owns its run queues
  - Partitioned scheduling with task migration
  - Red-black tree for the normal tasks, array for the RT tasks

```
struct rt_rq {
    struct rt_prio_array active;
    unsigned long rt_nr_running;
    u64 rt_runtime;
    ...
}
```
**cfs_rq Structure**

```c
struct cfs_rq {
    struct load_weight load;    // Load of the CFS run queue
    unsigned long nr_running;   // The number of tasks in the run queue
    u64 exec_clock;
    u64 min_vruntime;

    struct rb_root tasks_timeline;  // Pointer to root node of red black tree
    struct rb_node *rb_leftmost;    // Pointer to leftmost node of red black tree

    struct sched_entity *curr, *next, *last, *skip;
    ... int on_list;
    struct list_head leaf_cfs_rq_list;
    struct task_group *tg;
    ...
    u64 load_avg;
    u64 load_period;
    u64 load_stamp, load_last, load_unacc_exec_time;
    ...
};
```

- Load of the CFS run queue
- The number of tasks in the run queue
- Pointer to root node of red black tree
- Pointer to leftmost node of red black tree
- Per-CPU load information for CPU group scheduling
CFS Algorithm: Virtual Runtime

Virtual Runtime

- Approximate the “ideal multitasking” that CFS is modeling
- To account for
  - How long a process has run
  - How much longer it ought to run

\[
VR(\tau, t) = \frac{W_0}{W(\tau)} \times PR(t)
\]

- \(W_0\): the weight of nice value 0
- \(W(\tau)\): the weight of the task
- \(PR(t)\): actual runtime received in time interval [0, t]
CFS Algorithm: Virtual Runtime

- **Meaning**
  - Small virtual runtime
    - The task has received less CPU time than what it should have received
  - Large virtual runtime
    - The task has received more CPU time than what it should have received

- **Scheduling decision**
  - Choose the smallest virtual runtime (\textit{vruntime})
CFS Algorithm: Timeslice

- Timeslice is the time a task runs before it is preempted
  - It gives each runnable task a slice of the CPU’s time
  - The length of timeslice of a task is proportional to its weight

\[ TS = \frac{weight}{runqueue's\ total\ weight} \times P \]

\[ P = \begin{cases} \text{sched\_latency}, & \text{if } n < \text{nr\_latency} \\ \text{min\_granularity} \times n, & \text{otherwise} \end{cases} \]

\[ n = \text{the number of tasks} \]
CFS Algorithm: Red Black Tree

- CFS maintains a time-ordered red black tree where all runnable tasks are sorted by virtual runtime
  - No path in the tree will ever be more than twice as long as any other (self-balancing)
  - Operations on the tree occur in $O(\log n)$ time
Group Scheduling and Processor Affinity

- **Group scheduling**
  - A collection of tasks can share a virtual time
  - If a sched entity is a group, it has own run queue
  - Tasks in a group can be distributed on multiple CPUs

- **Processor affinity**
  - A task can run on all cpus by default
  - It can run only on a set of some cpus
  - Each task remembers its cpu running on
    - (cpumask_t) task_struct->cpus_allowed
CFS Implementation in the Kernel
Time Management

- Timer interrupts are used to gauge the passing of time
  - The system timer goes off at a preprogrammed frequency, called the tick rate (HZ hertz defined in <asm/param.h>)
  - The time between any two successive timer interrupts is called a tick and is equal to 1/(tick rate) seconds

- With the initial version of Linux, the i386 has had a timer interrupt frequency of 100 Hz
  - During the 2.5 development series, however, the frequency was raised to 1000 Hz and was controversial
  - Although the frequency is again 100 Hz, it is now a configuration option
Overview of Scheduling Flow

- On each scheduling tick, CFS
  - Subtracts the currently running task’s time slice by tick period
    - When the time slice reaches 0, NEED_RESCHED flag is set
  - Updates the virtual runtime of the currently running task
  - Checks NEED_RESCHED flag
    - If set, schedules the task with the smallest virtual runtime in the run queue
    - Enqueue the currently running task into the RB tree
    - Dequeue the task at the left-most node in the RB tree
Setting NEED_RESCHED Flag

- **scheduler_tick()**
  - **task_tick_fair()**
  - **update_curr()**
  - If \( cfs_rq->nr_running > 1 \):
    - **Y**
      - **check_preempt_tick()**
      - **Y**
        - **need to preempt?**
          - **Y**
            - **resched_task()**
            - Set NEED_RESCHED flag
          - **N**
    - **N**

Scheduler tick invokes the scheduler function
scheduler_tick invokes CFS scheduler function
Update the current task’s timing information (vruntime)
if current task has run beyond its time slice or if current task vruntime exceeds the next task’s vruntime (the left-most task in RB tree)
static void update_curr(struct cfs_rq *cfs_rq) {
    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;
    curr->exec_start = now;
    curr->sum_exec_runtime += delta_exec;
    curr->vruntime += calc_delta_fair(delta_exec, curr);
}

delta_exec = now - curr->exec_start;
if (unlikely((s64)delta_exec <= 0))
    return;
curr->exec_start = now;
curr->sum_exec_runtime += delta_exec;
curr->vruntime += calc_delta_fair(delta_exec, curr);

static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se) {
    if (unlikely(se->load.weight != NICE_0_LOAD))
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);
    return delta;
}

= delta_exec * NICE_0_LOAD / weight
static void check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr) {
    ...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);\hline Set NEED_RESCHED flag
    clear_buddies(cfs_rq, curr);
    return;
}
...}

static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se) {
    u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);
...}

\[ TS = \frac{weight}{runqueue's\ weight} \times P \]
Scheduling by schedule()

- NEED_RESCHED is set
  - schedule()
  - preempt_disable()
    - put_prev_task()
      - pick_next_task()
      - __enqueue_entity()
      - __dequeue_entity()
    - Need more rescheduling
      - Y
        - Return the current task to the run queue
      - N
        - Change the current task to the next task
Real-Time Computing and Communications Lab.
Hanyang University

schedule

<kernel/sched/core.c>

asmlinkage __visible void __sched schedule(void)
{
    struct task_struct *tsk = current;
    sched_submit_work(tsk);
    do {
        preempt_disable();
        __schedule(false);
        sched_preempt_enable_no_resched();
    } while (need_resched());
}
EXPORT_SYMBOL(schedule);

#include/linux/sched.h

static __always_inline bool need_resched(void)
{
    return unlikely(tif_need_resched());
}

#include/linux/thread_info.h
#define tif_need_resched() test_thread_flag(TIF_NEED_RESCHED)
`__schedule()`

```c
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    rcu_note_context_switch();
    prev = rq->curr;
    ...
    next = pick_next_task(rq, prev);
    clear_task_need_resched(prev);
    clear_preempt_need_resched();
    rq->clock_skip_update = 0;

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

        trace_sched_switch(preempt, prev, next);
        rq = context_switch(rq, prev, next); /* unlocks the rq */
        cpu = cpu_of(rq);
    } else {
        lockdep_unpin_lock(&rq->lock);
        raw_spin_unlock_irq(&rq->lock);
    }

    balance_callback(rq);
}
```

Enqueue and dequeue task
Save current task as prev
pick_next_task_fair

static struct task_struct *pick_next_task_fair(struct rq *rq, struct task_struct *prev)
{
    ...  
    put_prev_task(rq, prev);  \(\text{Put previous task by enqueue}\)
    do {
        se = pick_next_entity(cfs_rq, NULL);  \(\text{Pick and set next task for scheduling}\)
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);
    p = task_of(se);
    if (hrtick_enabled(rq))
        hrtick_start_fair(rq, p);  \(\text{Set high-resolution timer for setting time slice}\)
    (It allows for the task to check preemption point before 1/HZ seconds)
    return p;
    ...
}
put_prev_entity()

- CFS adds processes to the RB tree when a process becomes runnable (wake up) or is created via fork()

```c
static void put_prev_entity(struct cfs_rq *cfs_rq, struct sched_entity *prev)
{
    if (prev->on_rq)
        update_curr(cfs_rq);
    ...
    if (prev->on_rq) {
        update_stats_wait_start(cfs_rq, prev);
        __enqueue_entity(cfs_rq, prev);
        update_entity_load_avg(prev, 1);
    }
    cfs_rq->curr = NULL;
}
```

Update the runtime and other statistics

Perform inserting the entry into the red black tree
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se) {
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    int leftmost = 1;
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }
    if (leftmost)
        cfs_rq->rb_leftmost = &se->run_node;
    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
set_next_entity()

- CFS removes processes from the RB tree when a process blocks (becomes unrunnable) or terminates

```c
static void set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    if (se->on_rq) {
        update_stats_wait_end(cfs_rq, se);
        __dequeue_entity(cfs_rq, se);
    }

    update_stats_curr_start(cfs_rq, se);
    cfs_rq->curr = se;
    ...  
    se->prev_sum_exec_runtime = se->sum_exec_runtime;
}
```

Perform removing the entry into the red black tree
__dequeue_entity()

static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se) {
    if (cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;
        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }
    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
Deadline Scheduler
SCHED_DEADLINE

- Scheduling class for real-time tasks with deadlines

- Earliest Deadline First (EDF) algorithm
  - Using three parameters
    - Runtime, period, and deadline
  - Augmented with Constant Bandwidth Server (CBS)
    - Isolate behavior of tasks

- Release
  - Added to version 3.14 of the Linux kernel mainline (March 30, 2014)
Scheduling Parameters

- A SCHED_DEADLINE task is guaranteed to receive runtime **every** period, and **these runtime is available within** deadline from the beginning of the period
  - Runtime, period, and deadline are copied from sched_attr to sched_dl_entity during sched_setattr() system call

```
struct sched_dl_entity {
    struct rb_node rb_node;       // a node of red-black tree
    u64 dl_runtime;
    u64 dl_deadline;
    u64 dl_period;
    u64 dl_bw;

    s64 runtime;
    u64 deadline;
    unsigned int flags;

    int dl_throttled, dl_new, dl_boosted, dl_yielded; // Some bool flags

    struct hrtimer dl_timer;     // Per-task bandwidth enforcement timer

};
```

- Original scheduling parameters
- Actual scheduling parameters (continuously update during task execution)
When a task wakes up,

- If `deadline < currentTime` or \[ \frac{runtime}{deadline - currentTime} > \frac{dl\_runtime}{dl\_period} \], then current budget is re-initialized
  - `deadline = currentTime + dl\_deadline`
  - `runtime = dl\_runtime`
- Otherwise, `deadline` and `runtime` are left unchanged

\[ deadline \] and \[ runtime \] are initialized as 0 at first

```
static inline void setup_new_dl_entity(struct sched_dl_entity *dl_se, struct sched_dl_entity *pi_se)
{
    struct dl_rq *dl_rq = dl_rq_of_se(dl_se);
    struct rq *rq = rq_of_dl_rq(dl_rq);
    WARN_ON(!dl_se->dl_new || dl_se->dl_throttled);
    dl_se->deadline = rq_clock(rq) + pi_se->dl_deadline;
    dl_se->runtime = pi_se->dl_runtime;
    dl_se->dl_new = 0;
}
```
Earliest Deadline First

- EDF algorithm selects the task with the smallest *deadline* as the one to be executed first.

- Linux sorts tasks by using red-black tree
  - `__enqueue_dl_entity()`
  - Leftmost node has the smallest *deadline*
  - Scheduler just selects the leftmost node
    - `__dequeue_dl_entry()`
Update Runtime

- When a task executes for an amount of time \( t \), its runtime is decreased as
  - \( \text{runtime} = \text{runtime} - t \)

```c
static void update_curr_dl(struct rq *rq)
{
    struct task_struct *curr = rq->curr;
    struct sched_dl_entity *dl_se = &curr->dl;
    u64 delta_exec;
    if (!dl_task(curr) || !on_dl_rq(dl_se))
        return;
    delta_exec = rq_clock_task(rq) - curr->se.exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;
    ...
    dl_se->runtime -= dl_se->dl_yielded ? 0 : delta_exec;
    ...
}
```
Throttling

- When the `runtime` becomes less than or equal to 0, the task cannot be scheduled until its `deadline`
  - It is called throttling
  - It sets timer for budget replenishment (to `deadline`)
    - `dl_task_timer`
    - `enqueue_task_dl`
    - `enqueue_dl_entity`
    - `replenish_dl_entity`

```c
static void update_curr_dl(struct rq *rq)
{
    ...
    dl_se->runtime -= dl_se->dl_yielded ? 0 : delta_exec;
    if (dl_runtime_exceeded(dl_se)) {
        dl_se->dl_throttled = 1;
        __dequeue_task_dl(rq, curr, 0);
        if (unlikely(dl_se->dl_boosted || !start_dl_timer(curr)))
            enqueue_task_dl(rq, curr, ENQUEUE_REPLENISH);
        if (!is_leftmost(curr, &rq->dl))
            resched_curr(rq);
    }
    ...
}
```

When the replenish time is up (at `deadline`),
`deadline = deadline + dl_period`
`runtime = dl_runtime`

- check `runtime ≤ 0`
- set timer for replenishment
- the task is throttled
- If the expiry time already passed
- If leftmost task is changed by dequeue, then re-schedule to execute new task