CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 8

Process Scheduling II

February 20, 2025

Huaicheng Li

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Recap: The Many Facets of Process Scheduling

- Which task to run, when, and how long
  - Moreover, *how to implement such policies* ...
- Goal #1: Fairness
  - priorities (nice) → *weighted* fairness: A (3), B (1) → A (75% CPU time), B (25% CPU time)
    - » high priority tasks get more CPU share →
    - » without starving low priority tasks
    - » *avoiding priority inversion*

For Task (T_i), weight (w_i):

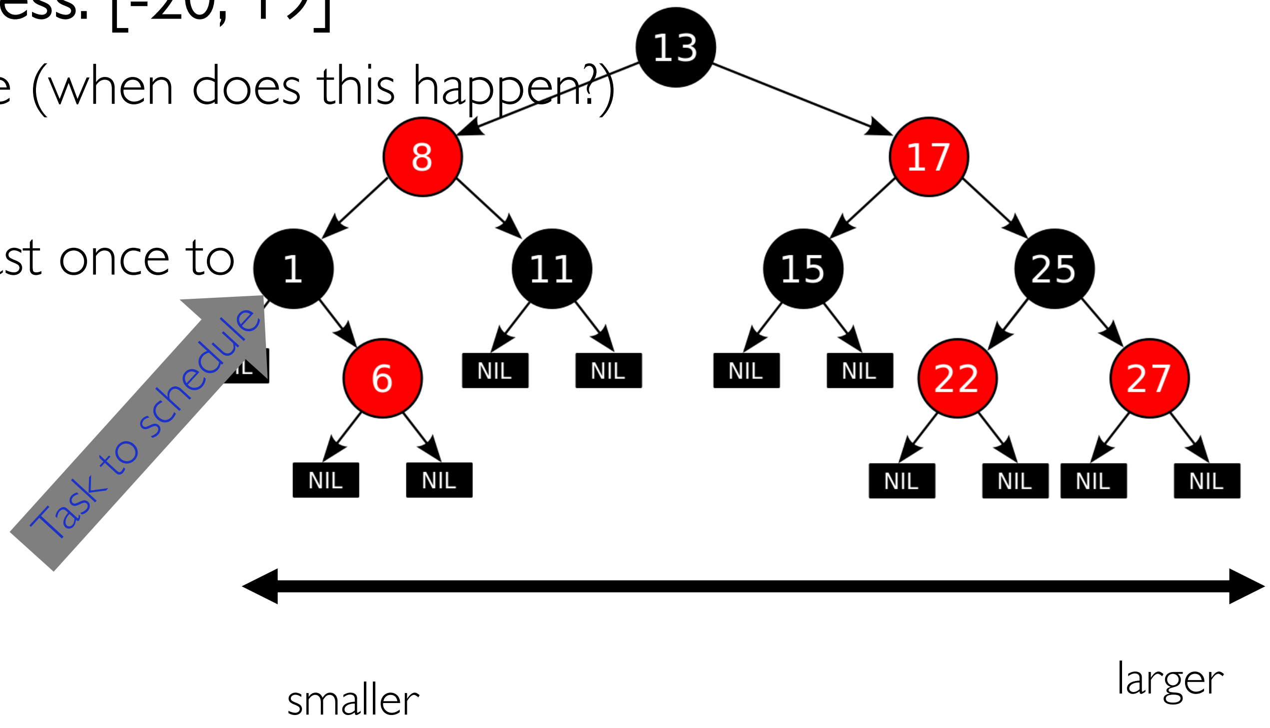
$$runtime(T_i) = \int_{t_0}^{t_1} \frac{w_i}{\sum_{j=0}^{N} w_j} dt \simeq \frac{w_i}{\sum_{j=0}^{N} w_j} \cdot (t_1 - t_0)$$

  - Implementation via virtual runtime (vruntime) in CFS
- Goal #2: Performance (low overhead)
  - e.g., frequencies of context switches → length of timeslice is critical (static vs dynamic)
  - ...

CFS Key Design Choices

The CFS

- It uses a rbtree to organize all the runnable processes based on vruntime
  - Leftmost node in the tree is therefore the task to schedule next.
  - O(log N), but it can be cached, thus O(1) to figure out what to schedule
  - Non-runnable processes will be taken out of the rbtree (dequeue)
    - e.g., blocked due to I/Os (TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE)
    - Insertion, deletion → rebalancing the rbtree
- Timeslice based on process niceness: [-20, 19]
  - Keep track of past CPU time usage (when does this happen?)
  - weighted time share
  - guarantees each task will run at least once to avoid starvation
  - I/O vs CPU bound
  - low vs. high-priority scheduling



CFS Virtual Runtime (vruntime)

- Charge each task a runtime proportional to w_{base} and inversely proportional to its weight w_i (vruntime)
- e.g., $w_{base} = 100$
- Tasks are scheduled in order of increasing vruntime

$$vruntime(T_i) = \frac{w_{base}}{w_i} \cdot (t_1 - t_0) = \frac{100}{w_i} \cdot (t_1 - t_0)$$

- vruntime vs. physical time

Let's Understand How CFS Achieves the Goals

- CPU-bound tasks use lots of CPU → will eventually be deprioritized by CFS than tasks spending a lot of time on I/O
  - Thus, response to interactive-tasks (e.g., text editor) is fast
- vruntime of high priority task decays faster … thus, it get more chance to run

More about CFS

- Group scheduling
  - Being fair to who? each process, or a group of processes
  - Within each group, the scheduler can treat each threads fairly
- Load balancing for multi-core
  - What if the runqueue of one core is empty while another core is busy with too many tasks?
  - Allow a core to ''steal'' tasks from other cores
  - Be careful to not violate fairness guarantees!
    - » a low priority task on a less-busy core can get more CPU time than a high-priority task on a busy core, priority inversion!

CFS is not Perfect

- Task migrations should be minimized …
  - for cache reuse
  - staying close to data in memory (for NUMA machines)
- Energy efficiency / Power efficiency
- Various heuristics, parameters to fine-tune for best-case workload performance
- A Decade of Wasted Cores
  - The Linux Scheduler: a Decade of Wasted Cores
- Let's look at interactive applications again
  - CFS can sometimes be unfair
  - In general, interactive tasks can be responded to pretty quickly, but it might not ensure quickly-enough without sacrificing fairness
  - Need to further improve responsiveness

BFS: https://en.wikipedia.org/wiki/Brain_Fuck_Scheduler

A Decade of Wasted Cores

- 2016 research paper
- Expose significant bugs in Linux multicore scheduling such that threads would wait to run when cores are sitting idle, leading to 13-24% performance degradation (138x for corner cases)
  - Moving a thread for load balancing doesn't always work well (or simply, is it worth migrating the thread across cores or simply let it wait to be executed on the original core?)
    - » How long to wait?
    - » Balancing the load and also maintain fairness based on priorities
    - » Idle core → it's okay to trigger immediate load balancing
    - » Where to migrate?
  - #1: Group imbalance bug → average
  - #2: scheduling group construction: if the groups are two hops aparts, the load balancing thread might not steal them
  - #3: overload on wakeup: pinned thread, sleep and then wake up, will be put back to the original core
  - #4: core re-adding, an important function was no longer invoked …

EEVDF: Earliest Eligible Virtual Deadline First

- **Lag:** difference between the ideal runtime and the actual runtime of a task
- **Eligibility:** a task is eligible to run if its lag >= 0
- **Virtual deadline:** vruntime + requested vruntime

$$lag_T(t_1) = V_{avg}(t_1) - V_T(t_1) \geq 0$$

$$D_T(t_1) = V_T(t_1) + \Delta t_T \cdot \frac{w_{base}}{w_i}$$

EEVDF paper (1995): here

Scheduler Class Implementation

- sched_class: an abstract class for all scheduler classes

```
/* linux/kernel/sched/sched.h */
struct sched_class {
    /* Called when a task enters a runnable state */
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Called when a task becomes unrunnable */
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Yield the processor (dequeue then enqueue back immediately) */
    void (*yield_task) (struct rq *rq);
    /* Preempt the current task with a newly woken task if needed */
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
    /* Choose a next task to run */
    struct task_struct * (*pick_next_task) (struct rq *rq,
                                            struct task_struct *prev,
                                            struct rq_flags *rf);
    /* Called periodically (e.g., 10 msec) by a system timer tick handler */
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    /* Update the current task's runtime statistics */
    void (*update_curr) (struct rq *rq);
};
```

- Each scheduler class implements its own functions

```
/* linux/kernel/sched/fair.c */
DEFINE_SCHED_CLASS(fair) = {
    /* const struct sched_class fair_sched_class = { */
    .enqueue_task       = enqueue_task_fair,
    .dequeue_task       = dequeue_task_fair,
    .yield_task         = yield_task_fair,
    .check_preempt_curr = check_preempt_wakeup,
    .pick_next_task     = pick_next_task_fair,
    .task_tick          = task_tick_fair,
    .update_curr        = update_curr_fair, /* ... */
};
/* 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_struct

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
struct sched_entity {
    /* For load-balancing: */
    struct load_weight  load;
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
```

- The base scheduler code triggers scheduling operations in two cases
  - when processing a timer interrupt (schedule_tick())
  - when the kernel calls schedule()

```c
/* linux/kernel/sched/core.c */
/* This function gets called by the timer code, with HZ frequency. */
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    struct rq_flags rf;

    /* call task_tick handler for the current process */
    sched_clock_tick();
    rq_lock(rq, &rf);
    update_rq_clock(rq);
    curr->sched_class->task_tick(rq, curr, 0); /* e.g., task_tick_fair in CFS */
    cpu_load_update_active(rq);
    calc_global_load_tick(rq);
    rq_unlock(rq, &rf);

    /* load balancing among CPUs */
    rq->idle_balance = idle_cpu(cpu);
    trigger_load_balance(rq);
    rq_last_tick_reset(rq);
}
```

```c
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
    /* ... */
}
```

```c
/* linux/kernel/sched/core.c */
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }

    /* The idle class should always have a runnable task: */
    BUG();
}
```

Time Accounting in CFS

- virtual runtime: how much CPU time a process has used

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
struct sched_entity {
    /* For load-balancing: */
    struct load_weight  load;
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
```

- Upon every timer interrupt, CFS accounts for the task's execution time

```
/* linux/kernel/sched/fair.c */
/* scheduler_tick() calls task_tick_fair() for CFS.
 * task_tick_fair() calls update_curr() for time accounting. */
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start; /* Step 1. calc exec duration */
    if (unlikely((s64)delta_exec <- 0))
        return;

    curr->exec_start = now;
    /* continue in a next slide ... */
}
```

```c
static void update_curr(struct cfs_rq *cfs_rq)
{
    /* continue from the previous slide ... */

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

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

    /* update vruntime with delta_exec and nice value */
    curr->vruntime += calc_delta_fair(delta_exec, curr); /* CODE */
    update_min_vruntime(cfs_rq);

    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);
    }
```

Process Selection in CFS

- CFS maintains a rbtree of tasks indexed by vruntime
- Always pick a task with the smallest vruntime, the left-most node

```
/* linux/kernel/sched/fair.c */
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq) /* CODE */
{
    struct rb_node *left = cfs_rq->rb_leftmost;

    if (!left)
        return NULL;

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

Add a Task to Runqueue

- When a task is woken up or migrated, it's added to a runqueue

```
/* linux/kernel/sched/fair.c */
void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /* Update run-time statistics */
    update_curr(cfs_rq);

    update_load_avg(se, UPDATE_TG);
    enqueue_entity_load_avg(cfs_rq, se);
    update_cfs_shares(se);
    account_entity_enqueue(cfs_rq, se);
    /* ... */

    /* Add this to the rbtree */
    if (!curr)
        __enqueue_entity(cfs_rq, se);
```

```c
/* linux/kernel/sched/fair.c */
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;
    /* Find the right place in the rbtree: */
    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;
        }
    }
    /* Maintain a cache of leftmost tree entries (it is frequently used): */
    if (leftmost)
        cfs_rq->rb_leftmost = &se->run_node;
    rb_link_node(&se->run_node, parent, link);
```

Remove a Task from Runqueue

- When a task goes to sleep or is migrated, it is removed from a runqueue

```
/* linux/kernel/sched/fair.c */
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_load_avg(se, UPDATE_TG);
    dequeue_entity_load_avg(cfs_rq, se);
    update_stats_dequeue(cfs_rq, se, flags);
    clear_buddies(cfs_rq, se);

    /* Remove this to the rbtree */
    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);

 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);
 }
```

Entry Point: schedule()

```
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
    /* ... */
}
```

```c
/* linux/kernel/sched/core.c */
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called.
         * pick_next_task_fair() eventually calls __pick_first_entity() */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
    /* The idle class should always have a runnable task: */
    BUG();
}
```

Sleep and Wake-up

- Reasons for a task to sleep
  - waiting for I/O, blocking on a mutex, etc.
- Steps to sleep
  - Mark a task sleeping
  - Put the task into a waitqueue
  - Dequeue the task from the rbtree
  - The task calls schedule() to select a new process to run
- Waking up a process is the reverse
- Two states associated with sleeping
  - TASK_INTERRUPTIBLE: wake up the sleeping task upon signal
  - TASK_UNINTERRUPTIBLE: defer signal delivery until wake up

Waitqueue: Sleeping

- List of processes waiting for an event to occur (similar to concept of condition variable)

```
/* linux/include/linux/wait.h */
struct wait_queue_entry {
    unsigned int        flags;
    void               *private;
    wait_queue_func_t   func;
    struct list_head    entry;
};
struct wait_queue_head {
    spinlock_t       lock;
    struct list_head    head;
};
typedef struct wait_queue_head wait_queue_head_t;
#define DEFINE_WAIT(name) ...
void add_wait_queue(struct wait_queue_head *wq_head,
                    struct wait_queue_entry *wq_entry);
void prepare_to_wait(struct wait_queue_head *wq_head,
       struct wait_queue_entry *wq_entry, int state);
void finish_wait(struct wait_queue_head *wq_head,
       struct wait_queue_entry *wq_entry);
```

```
DEFINE_WAIT(wait); /* Initialize a wait queue entry */

/* 'q' is the wait queue that we wish to sleep on */
add_wait_queue(q, &wait); /* Add itself to a wait queue */
while (!condition) { /* event we are waiting for */
    /* Change process status to TASK_INTERRUPTIBLE */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);/* prevent the lost wake-up */
    /* Since the state is TASK_INTERRUPTIBLE, a signal can wake up the task.
     * If there is a pending signal, handle signals */
    if(signal_pending(current)) {
        /* This is a spurious wake up, not caused
         * by the oocurance of the waiting event */
        /* Handle signal */
    }
    /* Go to sleep */
    schedule();
    /* Now, the task is woken up.
     * Check condition if the event occurs */
}

/* Set the process status to TASK_RUNNING
 * and remove itself from the wait queue */
finish_wait(&q, &wait);
```

- Or use one of `wait_event*()` macros

```
/* linux/include/linux/wait.h */

/**
 * wait_event_interruptible - sleep until a condition gets true
 * @wq: the waitqueue to wait on
 * @condition: a C expression for the event to wait for
 *
 * The process is put to sleep (TASK_INTERRUPTIBLE) until the
 * @condition evaluates to true or a signal is received.
 * The @condition is checked each time the waitqueue @wq is woken up.
 */
#define wait_event_interruptible(wq, condition)            \
({                                                          \
    int __ret = 0;                                          \
    might_sleep();                                          \
    if (!(condition))                                       \
        __ret = __wait_event_interruptible(wq, condition);  \
    __ret;                                                  \
})
```

Wake up

- Waking up is taken care of by `wake_up()`
  - By default, wake up *all* the processes on a waitqueue
  - Exclusive tasks are added using prepare_to_wait_exclusive()

```
#define wake_up(x)          __wake_up(x, TASK_NORMAL, 1, NULL)

/* __wake_up() calls __wake_up_common() */
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;
    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;
        if (curr->func(curr, mode, wake_flags, key) && /* wake-up function */
                (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

- A wait queue entry contains a pointer to a wake-up function

```
/* linux/include/linux/wait.h */

typedef struct __wait_queue wait_queue_t;
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode,
                                 int flags, void *key);
int default_wake_function(wait_queue_t *wait, unsigned mode,
                          int flags, void *key);
struct wait_queue_entry {
    unsigned int        flags;
    void               *private;
    wait_queue_func_t   func;
    struct list_head    entry;
};
```