Process Scheduling II

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
  - syscall, module, kernel data structures
- Process management
- Process scheduling I

Today's agenda

- Linux Completely Fair Scheduler (CFS)
- Preemption and context switching
- Real-time scheduling policies
- Scheduling related system calls

Linux CFS design

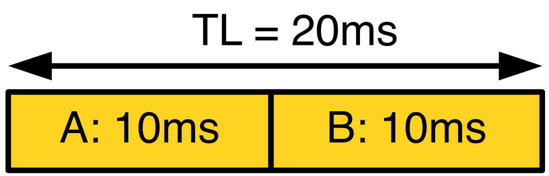
- Completely Fair Scheduler (CFS)
- Evolution of rotating staircase deadline scheduler (RSDL)
- At each moment, each process of the same priority has received an exact same amount of the CPU time
- If we could run `n` tasks in parallel on the CPU, give each `1/n` of the CPU processing power
- CFS runs a process for some times, then swaps it for the runnable process that has run the least

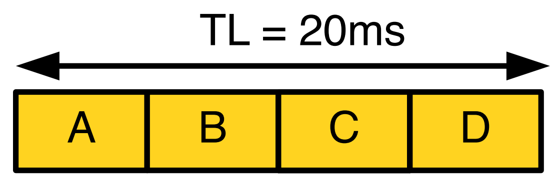
Linux CFS design

- No default timeslice, CFS calculates how long a process should run according to the number of runnable processes
  - That dynamic timeslice is weighted by the process priority (nice)
  - timeslice = weight of a task / total weight of runnable tasks
- To calculate the actual timeslice, CFS sets a **targeted latency**
  - Targeted latency: period during which all runnable processes should be scheduled at least once
  - Minimum granularity: floor at 1 ms (default)

Linux CFS design

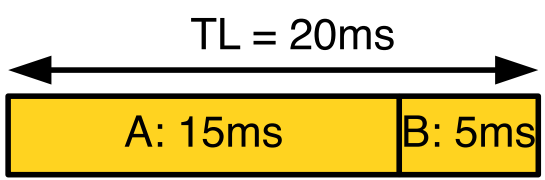
- Example: processes with the same priority



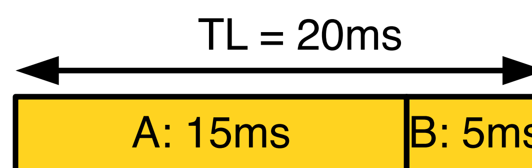


($A_{nice} = B_{nice} = C_{nice} = D_{nice}$)

- Example: processes with different priorities



($A_{nice} = 0$; $B_{nice} = 5$)



($A_{nice} = 10$; $B_{nice} = 15$)

CFS implementation

- Four main components of CFS
  - Time accounting
  - Process selection
  - Scheduler entry point: `schedule()`, `scheduler_tick()`
  - Sleeping and waking up

Time accounting in CFS

- **Virtual runtime**: how much time a process has been executed

```
/* 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 {
    /* ... */
    struct rb_node      run_node;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
                                   * has been executed (ns) */
    struct cfs_rq       *cfs_rq; /* CFS run queue */
    /* ... */
};
```

Time accounting in CFS

- Upon every timer interrupt, CFS accounts the task's execution time
- `scheduler_tick()` → `task_tick_fair()` → `update_curr()`

```
/* linux/kernel/sched/fair.c */
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 ... */
}
```

Time accounting in CFS

```
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);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

Time accounting in CFS

- CFS checks if the currently running task needs to be preempted (i.e., the tasks uses up the timeslice). If so, set the `TIF_NEED_RESCHED` flag
- `scheduler_tick()` → `check_preempt_tick()`

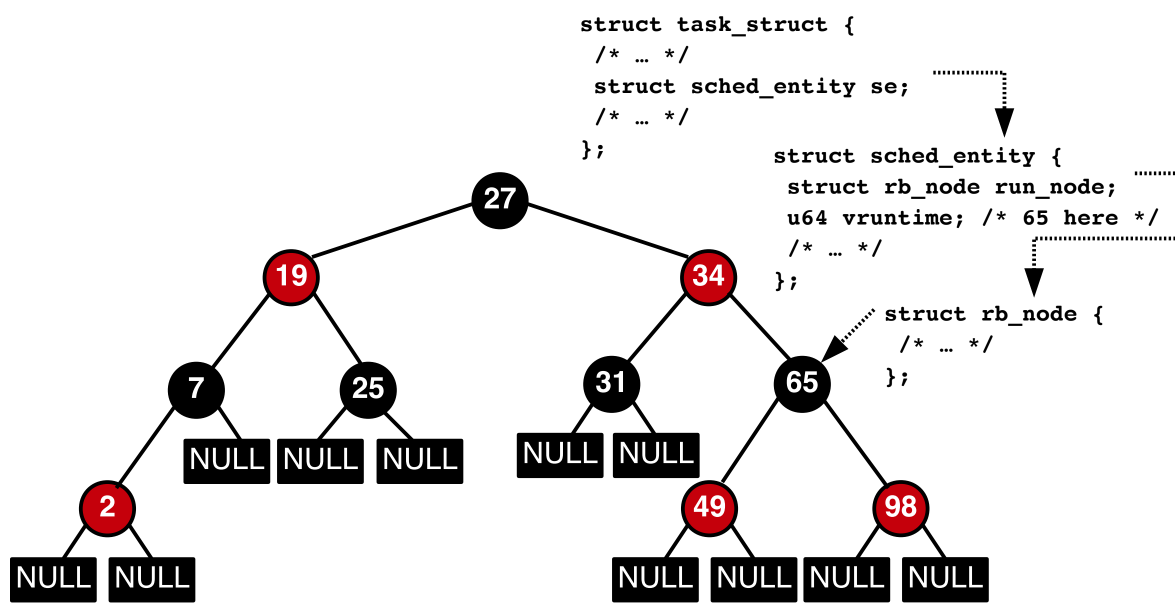
```
/* linux/kernel/sched/fair.c */
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    /* ... */
    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    if (delta_exec > ideal_runtime) {
        resched_curr(rq_of(cfs_rq));
        clear_buddies(cfs_rq, curr);
        return;
    }
    /* ... */
}
```

Time accounting in CFS: QEMU-gdb (Lec 6)

```
   3092    void scheduler_tick(void)
B+>3093    {
   3094            int cpu = smp_processor_id();
   3095            struct rq *rq = cpu_rq(cpu);
   3096            struct task_struct *curr = rq->curr;
   3097            struct rq_flags rf;
   3098
   3099            sched_clock_tick();
   3100
   3101            rq_lock(rq, &rf);
   3102
   3103            update_rq_clock(rq);
   3104            curr->sched_class->task_tick(rq, curr, 0);
   3105            cpu_load_update_active(rq);
   3106            calc_global_load_tick(rq);
   3107
   3108            rq_unlock(rq, &rf);
   3109
   3110            perf_event_task_tick();
remote Thread 2 In: scheduler_tick                                  L3093 PC: 0xffffffff810807c0
Thread 2 hit Breakpoint 1, scheduler_tick () at kernel/sched/core.c:3093
(gdb) bt
#0  scheduler_tick () at kernel/sched/core.c:3093
#1  0xffffffff810bc302 in update_process_times (user_tick=0) at kernel/time/timer.c:1576
#2  0xffffffff810caacd in tick_sched_handle (regs=<optimized out>, ts=<optimized out>,
    ts=<optimized out>) at kernel/time/tick-sched.c:155
#3  0xffffffff810cafd8 in tick_sched_timer (timer=0xffff88007fd135c0) at kernel/time/tick-sched.c:1174
#4  0xffffffff810bcbe2 in __run_hrtimer (now=<optimized out>, timer=<optimized out>,
    base=<optimized out>, cpu_base=<optimized out>) at kernel/time/hrtimer.c:1212
#5  __hrtimer_run_queues (cpu_base=0xffff88007fd13180, now=<optimized out>) at kernel/time/hrtimer.c:1276
#6  0xffffffff810bd23b in hrtimer_interrupt (dev=<optimized out>) at kernel/time/hrtimer.c:1310
#7  0xffffffff8103d9a3 in local_apic_timer_interrupt () at arch/x86/kernel/apic/apic.c:941
#8  0xffffffff8103e383 in smp_apic_timer_interrupt (regs=<optimized out>)
    at arch/x86/kernel/apic/apic.c:965
#9  0xffffffff8194f9c6 in apic_timer_interrupt () at arch/x86/entry/entry_64.S:701
#10 0xffffc9000036fdf8 in ?? ()
#11 0x0000000000000000 in ?? ()
(gdb)
```

CFS maintains cfs_rq (runqueue) as a rbtree

- CFS maintains a rbtree of tasks indexed by `vruntime` (i.e., runqueue)
- Always pick a task with the smallest `vruntime`, the left-most node



Adding a task to a runqueue

- When a task is woken up or migrated, it is 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);
    /* ... */
}
```

Adding a task to a runqueue

```
/* 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);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Removing a task from a 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);
    /* ... */
}
```

Removing a task from a runqueue

```
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);
}
```

Scheduler entry point: `schedule()`

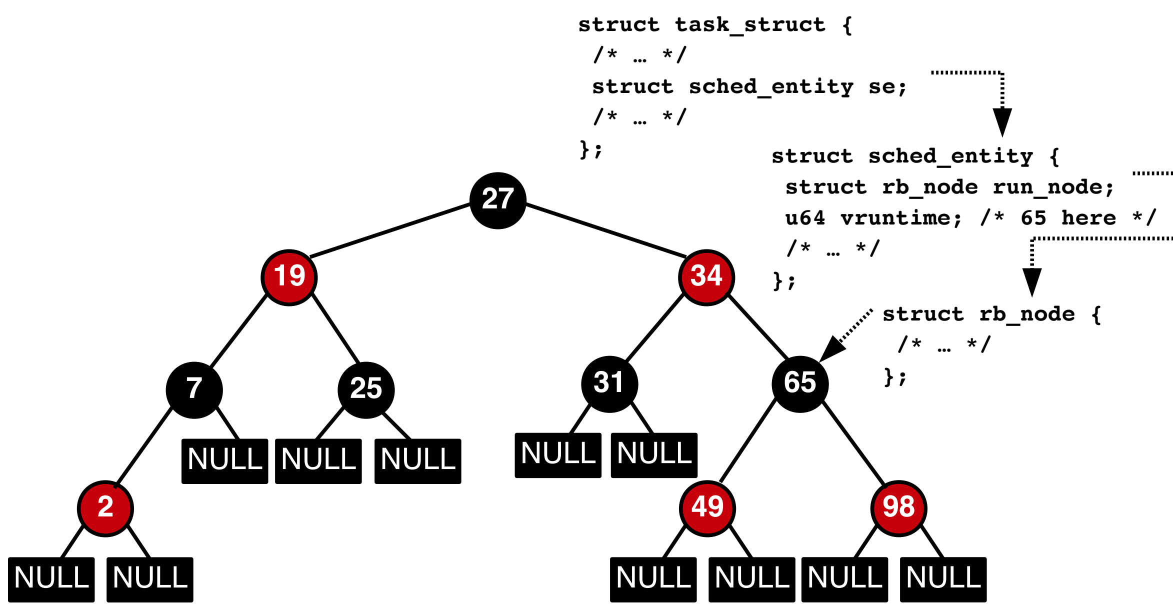
```
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    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);
    }
    /* ... */
}
```

Scheduler entry point: `schedule()`

```
/* 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();
}
```

Process selection in CFS

- CFS maintains an rbtree of tasks indexed by `vruntime` (i.e., runqueue)
- Always pick a task with the smallest `vruntime`, the left-most node



Process selection in CFS

- `schedule()` calls `pick_next_task_fair()` in CFS
- `pick_next_task_fair()` calls `__pick_first_entity()` that returns the left-most node of the rbtree (runqueue)

```
/* 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);
}
```

Sleeping and waking up

- Reasons for a task to sleep:
  - Specified amount of time, 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 of runnable tasks
  - The task calls `schedule()` to select a new task to run
- Waking up a task is the inverse of sleeping

Sleeping and waking up

- Two states associated with sleeping:
  - `TASK_INTERRUPTIBLE`
    - Wake up the sleeping task upon signal
  - `TASK_UNINTERRUPTIBLE`
    - Defer signal delivery until wake up

Wait queue: sleeping

- List of tasks waiting for an event to occur

```
/* 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;
};

#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 s
void finish_wait(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry);
```

Wait queue: sleeping

```
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);
```

Wait queue: sleeping

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

```c
/* 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;                                                     \
})
```

Wait queue: waking-up

- Waking up waiting tasks by `wake_up()`
  - By default, wake up *all* the tasks 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;
    }
}
```

Wait queue: waking-up

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

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

typedef struct wait_queue_entry wait_queue_entry_t;

typedef int (*wait_queue_func_t)(struct wait_queue_entry *wq_entry,
                                 unsigned mode, int flags, void *key);
int default_wake_function(struct wait_queue_entry *wq_entry,
                          unsigned mode, int flags, void *key);

struct wait_queue_entry {
    unsigned int        flags;
    void                *private;
    wait_queue_func_t   func;
    struct list_head    entry;
};
```

Wait queue: waking-up

- `default_wake_function()` calls `try_to_wake_up()`
  - calls `ttwu_queue()`
  - calls `ttwu_do_activate()`
    - puts the task back on runqueue
  - calls `ttwu_do_wakeup()`
    - calls `check_preempt_curr()`
      - sets the `TIF_NEED_RESCHED` flag (as needed)
    - sets the task state to `TASK_RUNNING`

CFS on multi-core machines

- Per-CPU runqueues (rbtrees)
  - To avoid costly accesses to shared data structures
- Runqueues must be kept balanced
  - E.g., dual-core with one long runqueue of high-priority processes, and a short one with low-priority processes
  - High-priority processes get less CPU time than low-priority ones
- A load balancer runs periodically based on priority and CPU usage

Preemption and context switching

- A **context switch** is the action of swapping the process currently running on the CPU to another one
- Performed by `context_switch()`, which is called by `schedule()`
  - Switch the address space through `switch_mm()`
  - Switch the CPU state (registers) through `switch_to()`

Preemption and context switching

- Then, when `schedule()` will be called?
    - A task can voluntarily relinquish the CPU by calling `schedule()`
    - A current task needs to be preempted if
        1. it runs long enough
            - by `scheduler_tick()`
        2. a task with a higher priority is woken up
            - by `try_to_wake_up()`

`need_resched`

- The `TIF_NEED_RESCHED` flag (in `thread_info`)
  - specifies whether a (preemptive) reschedule should be performed
  - `set_tsk_need_resched()`
  - `clear_tsk_need_resched()`
  - `need_resched()`
- `TIF_NEED_RESCHED` is set by
  - `scheduler_tick()`: the currently running task needs to be preempted
  - `try_to_wake_up()`: a process with higher priority wakes up

`need_resched`

- Then `TIF_NEED_RESCHED` flag is checked:
  - Upon returning to user space (from a syscall or an interrupt)
  - Upon returning from an interrupt
- If the flag is set, `schedule()` is called

`need_resched`

Process #100

```
long count = 0;
void foo(void) {
 while(1) {
 count++;
 }
}
```

Process #200

```
long val = 2;
void bar(void) {
 while(1) {
 val *= 3;
 }
}
```

Process #300

```
void baz(void) {
 while(1) {
 printf("hi");
 }
}
```

Operating system: scheduler

CPU0

- **Q: how can the preemptive scheduler take the control of infinite loop?**

Kernel preemption

- In most of UNIX-like operating systems, kernel code is non-preemptive
- In Linux, the kernel code is also preemptive
  - A task can be preempted in the kernel as long as execution is in a safe state without holding any lock
- `preempt_count` in the `thread_info` structure indicates the current lock depth
- If `need_resched && !preempt_count` then, it is safe to preempt
  - Checked when returning to the kernel from interrupt
  - Checked when releasing a lock

Kernel preemption

- Kernel preemption can occur:
  - On return from interrupt
  - When kernel code becomes preemptible again
  - If a task in the kernel blocks (e.g., mutex)

```
/* linux/include/linux/preempt.h */
#define preempt_disable() \
do {                        \
    preempt_count_inc();  \
    barrier();             \
} while (0)
#define preempt_enable()  \
do {                        \
    barrier();             \
    if (unlikely(preempt_count_dec_and_test())) \
        __preempt_schedule(); \
} while (0)
```

Real-time scheduling policies

- Linux provides two *soft real-time* scheduling classes
  - `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE`
  - Best effort, no guarantee
- Real-time task of any scheduling class will always run before non-realtime ones (CFS, `SCHED_OTHER`)
  - `schedule()` → `pick_next_task()` → `for_each_class()`

Real-time scheduling policies

- `SCHED_FIFO`
  - Tasks run until it blocks/yield
  - Only a higher priority RT task can preempt it
  - Round-robin for tasks of same priority
- `SCHED_RR`
  - Same as `SCHED_FIFO`, but with a fixed timeslice

Real-time scheduling policies

- `SCHED_DEADLINE`
  - Real-time policies mainlined in v3.14 enabling predictable RT scheduling
  - Early deadline first (EDF) scheduling based on a period of activation and a worst case execution time (WCET) for each task
  - Ref: kernel Documentation
- `SCHED_BATCH`: non-real-time, low priority background jobs
- `SCHED_IDLE`: non-real-time, very low priority background jobs

Scheduling related system calls

- `sched_getscheduler`, `sched_setscheduler`
- `nice`
- `sched_getparam`, `sched_setparam`
- `sched_get_priority_max`, `sched_get_priority_min`
- `sched_getaffinity`, `sched_setaffinity`
- `sched yield`

Scheduling related system calls: example

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sched.h>
#include <assert.h>

void handle_err(int ret, char *func)
{
    perror(func);
    exit(EXIT_FAILURE);
}

int main(void)
{
    pid_t pid = -1;
    int ret = -1;
    struct sched_param sp;
    int max_rr_prio, min_rr_prio = -42;
    size_t cpu_set_size = 0;
    cpu_set_t cs;
```

Scheduling related system calls: example

```
/* Get the PID of the calling process */
pid = getpid();
printf("My pid is: %d\n", pid);

/* Get the scheduling class */
ret = sched_getscheduler(pid);
if(ret == -1)
    handle_err(ret, "sched_getscheduler");
printf("sched_getscheduler returns: %d\n", ret);
assert(ret == SCHED_OTHER);

/* Get the priority (nice/RT) */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);

/* Set the priority (nice value) */
ret = nice(1);
if(ret == -1)
    handle_err(ret, "nice");
```

Scheduling related system calls: example

```
/* Get the priority again */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);

/* Switch scheduling class to FIFO and the priority to 99 */
sp.sched_priority = 99;
ret = sched_setscheduler(pid, SCHED_FIFO, &sp);
if(ret == -1)
    handle_err(ret, "sched_setscheduler");

/* Get the scheduling class */
ret = sched_getscheduler(pid);
if(ret == -1)
    handle_err(ret, "sched_getscheduler");
printf("sched_getscheduler returns: %d\n", ret);
assert(ret == SCHED_FIFO);
```

Scheduling related system calls: example

```
/* Get the priority again */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);

/* Set the RT priority */
sp.sched_priority = 42;
ret = sched_setparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_setparam");
printf("Priority changed to %d\n", sp.sched_priority);

/* Get the priority again */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);
```

Scheduling related system calls: example

```
/* Get the max priority value for SCHED_RR */
max_rr_prio = sched_get_priority_max(SCHED_RR);
if(max_rr_prio == -1)
    handle_err(max_rr_prio, "sched_get_priority_max");
printf("Max RR prio: %d\n", max_rr_prio);

/* Get the min priority value for SCHED_RR */
min_rr_prio = sched_get_priority_min(SCHED_RR);
if(min_rr_prio == -1)
    handle_err(min_rr_prio, "sched_get_priority_min");
printf("Min RR prio: %d\n", min_rr_prio);

cpu_set_size = sizeof(cpu_set_t);
CPU_ZERO(&cs);  /* clear the mask */
CPU_SET(0, &cs);
CPU_SET(1, &cs);
/* Set the affinity to CPUs 0 and 1 only */
ret = sched_setaffinity(pid, cpu_set_size, &cs);
if(ret == -1)
    handle_err(ret, "sched_setaffinity");
```

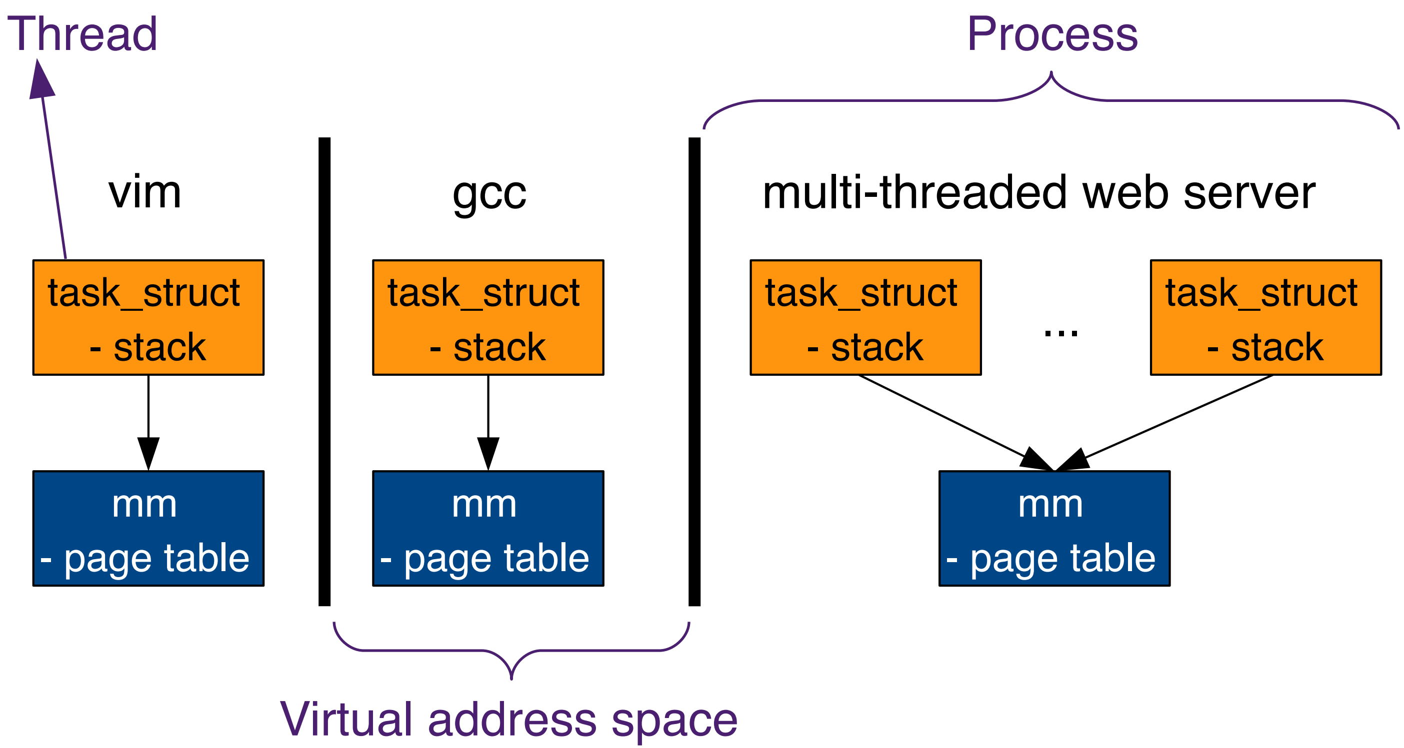
Scheduling related system calls: example

```
    /* Get the CPU affinity */
    CPU_ZERO(&cs);
    ret = sched_getaffinity(pid, cpu_set_size, &cs);
    if(ret == -1)
        handle_err(ret, "sched_getaffinity");
    assert(CPU_ISSET(0, &cs));
    assert(CPU_ISSET(1, &cs));
    printf("Affinity tests OK\n");

    /* Yield the CPU */
    ret = sched_yield();
    if(ret == -1)
        handle_err(ret, "sched_yield");

    return EXIT_SUCCESS;
}
```

Summary: `task = process | thread`

- `struct task_struct`
  - a process or a thread
- `struct mm`
  - a virtual address space

`task = process | thread`

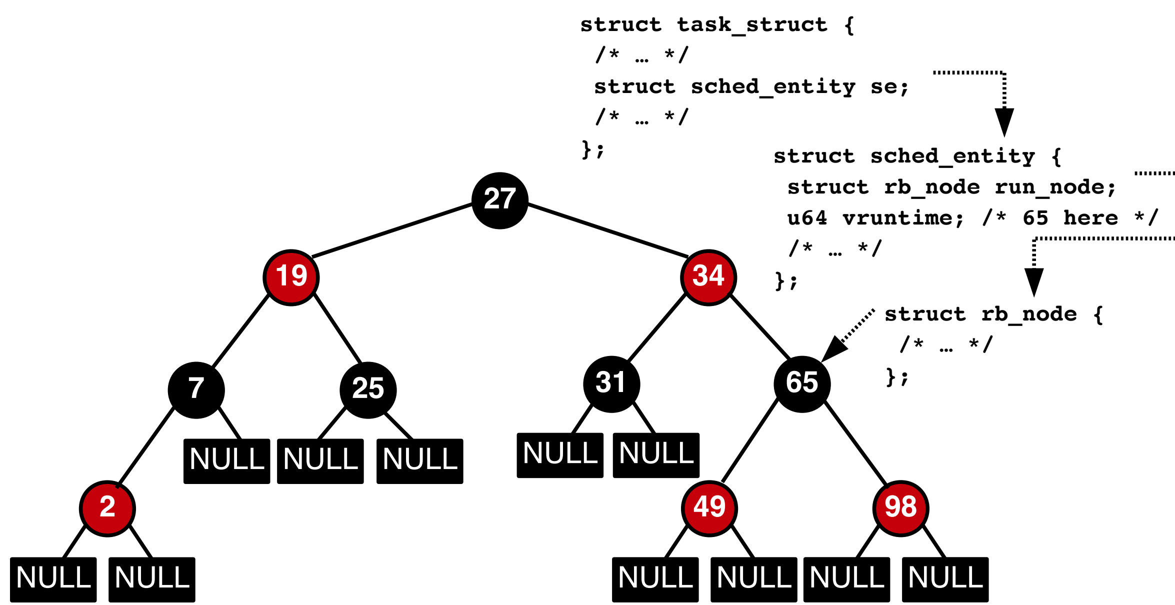


Summary: Completely Fair Scheduling (CFS)

- `struct sched_entiry`
  - embedded to `task_struct`
  - has `vruntime` value
- `struct cfs_rq`
  - a queue of runnable tasks in a `TASK_RUNNING` status
  - has `struct rb_root tasks_timeline`

Summary: Completely Fair Scheduling (CFS)

- CFS maintains an rbtree of tasks indexed by `vruntime` (i.e., runqueue)
- Always pick a task with the smallest `vruntime`, the left-most node



Next lecture

- Interrupt Handler: Top Half

Further readings

- The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS, USENIX ATC18
- The Rotating Staircase Deadline Scheduler