Process Scheduling

Advanced Operating Systems and Virtualization
Alessandro Pellegrini
A.Y. 2018/2019

Linux Scheduler

- The scheduler is a fundamental subsystem of the kernel

- Different scheduling strategies exist
  - Take into account *priority*
  - Take into account *responsiveness*
  - Take into account *fairness*

- The history of Linux has seen different algorithms

Process Priority

- Unix demands for priority based scheduling
  - This relates to the *nice* of a process in [-20, 19]
  - The higher the nice, the lower the priority
  - This tells how nice a process is towards others
- There is also the notion of "real time" processes
  - Hard real time: bound to strict time limits in which a task must be completed (not supported in mainstream Linux)
  - Soft real time: there are boundaries, but don't make your life depend on it
    - Examples: burning data to a CD ROM, VoIP

Process Priority

- In Linux, real time priorieties are in [0, 99]
  - Here higher value means higher priority
- Implemented according to the Real-Time Extensions of POSIX

```
ps -eo pid,rtprio,cmd ('-' = no realtime)
chrt -p pid
chrt -p prio pid
```

Process Priority in the Kernel

- Both nice and rt priorities are mapped to a single value in [0, 139] in the kernel

- 0 to 99 are reserved to rt priorities

- 100 to 139 for nice priorities (mapping exactly to [-20, 19])

- Priorities are defined in `include/linux/sched/prio.h`

Process Priority in the Kernel

```
#define MAX_NICE 19
#define MIN_NICE -20
#define NICE_WIDTH (MAX_NICE - MIN_NICE
                    + 1)


#define MAX_USER_RT_PRIO 100
#define MAX_RT_PRIO MAX_USER_RT_PRIO
#define MAX_PRIO (MAX_RT_PRIO +
NICE_WIDTH)
#define DEFAULT_PRIO (MAX_RT_PRIO +
NICE_WIDTH / 2)
```

Process Priority in the Kernel

```c
/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice) ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio) ((prio) - DEFAULT_PRIO)


/*
 * 'User priority' is the nice value converted to something we
 * can work with better when scaling various scheduler parameters,
 * it's a [ 0 ... 39 ] range.
 */
#define USER_PRIO(p) ((p)-MAX_RT_PRIO)
#define TASK_USER_PRIO(p) USER_PRIO((p)->static_prio)
#define MAX_USER_PRIO (USER_PRIO(MAX_PRIO))
```

Process Priority in `task_struct`

- `static_prio`: priority given "statically" by a user (and mapped into kernel's representation)

- `normal_priority`: based on `static_prio` and scheduling policy of a process: Tasks with the same static priority that belong to different policies will get different normal priorities. Child processes inherit the normal priorities from their parent processes when forked.

- `prio`: "dynamic priority". It can change in certain situations, e.g. to preempt a process with higher priority

- `rt_priority`: the realtime priority for realtime tasks in [0, 99]

Computing `prio`

- In `kernel/sched/core.c`

```
p->prio = effective_prio(p);


static int effective_prio(struct task_struct
*p) {
    p->normal_prio = normal_prio(p);
    if (!rt_prio(p->prio))
        return p->normal_prio;
    return p->prio;
}
```

Returns `static_priority` or maps
`rt_priority` to kernel representation

Load Weights

- `task_struct->se` is a `struct sched_entity` (in `include/linux/sched.h`):
  - It keeps a `struct load_weight load`:

```
struct load_weight {
   unsigned long weight;
   u32 inv_weight;
};
```

- Load weights are used to scale the time slice assigned to a scheduled process

Load Weights

- **From** `kernel/sched/core.c`:

Nice levels are multiplicative, with a gentle 10% change for every nice level changed. I.e. when a CPU-bound task goes from nice 0 to nice 1, it will get ~10% less CPU time than another CPU-bound task that remained on nice 0.

The "10% effect" is relative and cumulative: from _any_ nice level, if you go up 1 level, it's -10% CPU usage, if you go down 1 level it's +10% CPU usage. (to achieve that we use a multiplier of 1.25. If a task goes up by ~10% and another task goes down by ~10% then the relative distance between them is ~25%.)

Load Weights

- From `kernel/sched/core.c`:

```
const int sched_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,
};
```

- This array takes a value for each possible nice level in [-20, 19]

Some Examples

- Two tasks running at nice 0 (weight 1024)
  - Both get 50% of time: 1024/(1024+1024) = 0.5
- Task 1 is moved to nice -1 (priority boost):
  - T1: 1277/(1024+1277) ≈ 0.55
  - T2: 1024/(1024+1277) ≈ 0.45 (10% difference)
- Task 2 is then moved to nice 1 (priority drop):
  - T1: 1277/(820+1277) ≈ 0.61
  - T2: 820/(820+1277) ≈ 0.39 (22% difference)

Different Scheduling Classes

- `SCHED_FIFO`: Realtime FIFO scheduler, in which a process has to explicitly yield the CPU

- `SCHED_RR`: Realtime Round Robin Scheduler (might fallback to FIFO)

- `SCHED_OTHER`/`SCHED_NORMAL`: the common round-robin time-sharing scheduling policy

- `SCHED_DEADLINE` (since 3.14): Constant Bandwidth Server (CBS) algorithm on top of Earliest Deadline First queues

- `SCHED_DEADLINE` (since 4.13): CBS replaced with Greedy Reclamation of Unused Bandwidth (GRUB).

Scheduling Classes

```
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);


  void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int
                        flags);
  struct task_struct * (*pick_next_task) (struct rq *rq, struct
                        task_struct *prev, struct rq_flags *rf);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p);
  ...
  void (*set_curr_task) (struct rq *rq);
  int  (*select_task_rq)(struct task_struct *p, int task_cpu,
                        int sd_flag, int flags);
  ...
};
```

Scheduler Code Organization

- General code base and specific scheduler classes are found in `kernel/sched/`

- `core.c`: the common codebase

- `fair.c`: implementation of the basic scheduler (CFS: Completely Fair Scheduler)

- `rt.c`: the real-time scheduler

- `idle_task.c`: the idle-task class

Run Queues

```
struct rq {
    unsigned int nr_running;
    #define CPU_LOAD_IDX_MAX 5
    unsigned long cpu_load[CPU_LOAD_IDX_MAX];
    /* capture load from all tasks on this cpu */
    struct load_weight load;
    struct cfs_rq cfs;
    struct rt_rq rt;
    struct task_struct *curr, *idle, ...;
    u64 clock;
    /* cpu of this runqueue */
    int cpu;
}
```

Run Queues

- Added in 2.6
- Defined in `kernel/sched/sched.h`

```
DECLARE_PER_CPU_SHARED_ALIGNED(struct rq,
runqueues);
```

```
#define cpu_rq(cpu)    (&per_cpu(runqueues, (cpu)))
#define this_rq()      this_cpu_ptr(&runqueues)
#define task_rq(p)     cpu_rq(task_cpu(p))
#define cpu_curr(cpu) (cpu_rq(cpu)->curr)
```
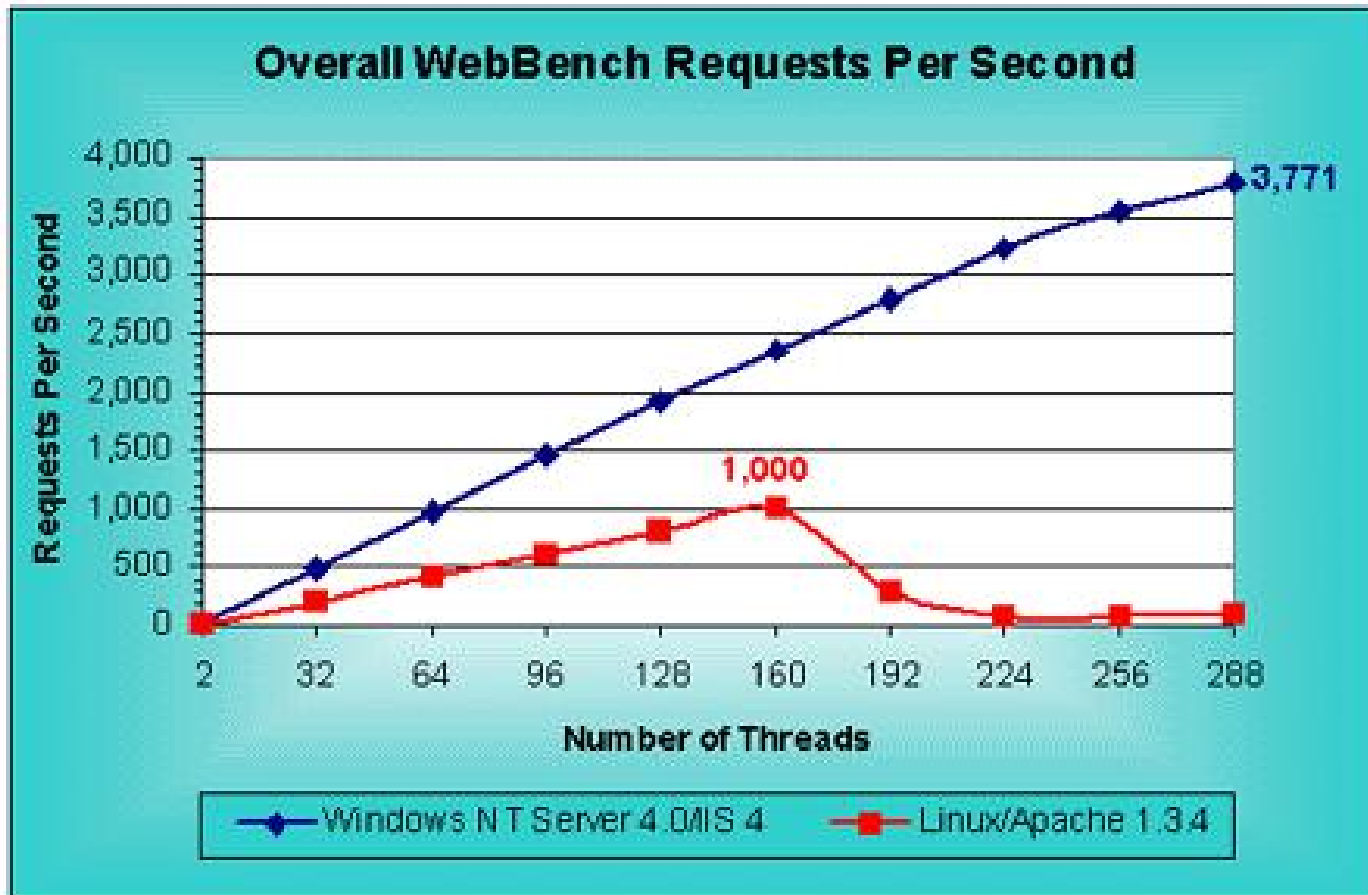
Wait Queues

- Defined in `include/linux/wait.h`
- This is a set of data structures to manage threads that are waiting for some condition to become true
- This is a way to put threads to sleep in kernel space
- It is a data structure which changed many times in the history of the kernel
- Suffered from the "Thundering Herd" performance problem

Thundering Herd Effect

Taken from 1999 Mindcraft study on Web and File Server Comparison

Wait Queues

```
#define WQ_FLAG_EXCLUSIVE           0x01

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;
```

Wait Queue API

- Implemented as macros in `include/linux/wait.h`

```
static inline void init_waitqueue_entry(struct
              wait_queue_entry *wq_entry,
          struct task_struct *p)
wait_event_interruptible(wq_head, condition)
wait_event_interruptible_timeout(wq_head,
              condition, timeout)
wait_event_hrtimeout(wq_head, condition,
              timeout)
wait_event_interruptible_hrtimeout(wq,
              condition, timeout)
```

Wait Queue API

```
void add_wait_queue(struct wait_queue_head *wq_head,
struct wait_queue_entry *wq_entry) {
    unsigned long flags;

    wq_entry->flags &= ~WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&wq_head->lock, flags);
    list_add(&wq_entry->entry, &wq_head->head);
    spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

Wait Queue API

```c
void add_wait_queue_exclusive(struct wait_queue_head
      *wq_head, struct wait_queue_entry *wq_entry)
{
      unsigned long flags;

      wq_entry->flags |= WQ_FLAG_EXCLUSIVE;
      spin_lock_irqsave(&wq_head->lock, flags);
      list_add_tail(&wq_entry->entry, &wq_head->head);
      spin_unlock_irqrestore(&wq_head->lock, flags);
}
```
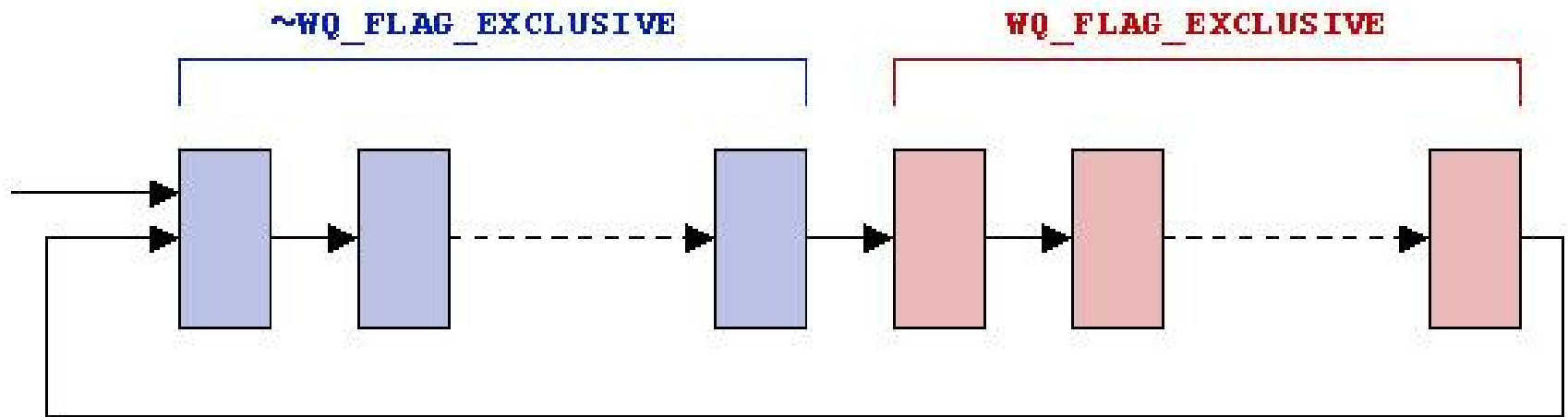
Wait Queue API

```c
void remove_wait_queue(struct wait_queue_head *wq_head,
                       struct wait_queue_entry *wq_entry)
{
        unsigned long flags;

        spin_lock_irqsave(&wq_head->lock, flags);
        list_del(&wq_entry->entry);
        spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

Wait Queue Exclusive

Wait Queue API

- Implemented as macros in `include/linux/wait.h`
- `wake_up(x)`
- `wake_up_nr(x, nr)`
- `wake_up_all(x)`
- `wake_up_locked(x)`
- `wake_up_all_locked(x)`

- `wake_up_interruptible(x)`
- `wake_up_interruptible_nr(x, nr)`
- `wake_up_interruptible_all(x)`
- `wake_up_interruptible_sync(x)`

Scheduler Entry Point

- The entry point for the scheduler is schedule(void) in `kernel/sched.c`

- This is called from several places in the kernel
  - *Direct Invocation*: an explicit call to `schedule()` is issued
  - *Lazy Invocation*: some hint is given to the kernel indicating that `schedule()` should be called soon (see `need_resched`)

- In general `schedule()` entails 3 distinct phases, which depend on the scheduler implementation:
  - Some checks on the current process (e.g., with respect to signal processing)
  - Selection of the process to be activated
  - Context switch

Periodic Scheduling

- `schedule_tick()` is called from `update_process_times()`

- This function has two goals:
  - Managing scheduling-specific statistics
  - Calling the scheduling method of the class

schedule_tick()

```c
/*
* 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;
    ...
    update_rq_clock(rq);
    curr->sched_class->task_tick(rq, curr, 0);
    update_cpu_load_active(rq);
    ...
}
```
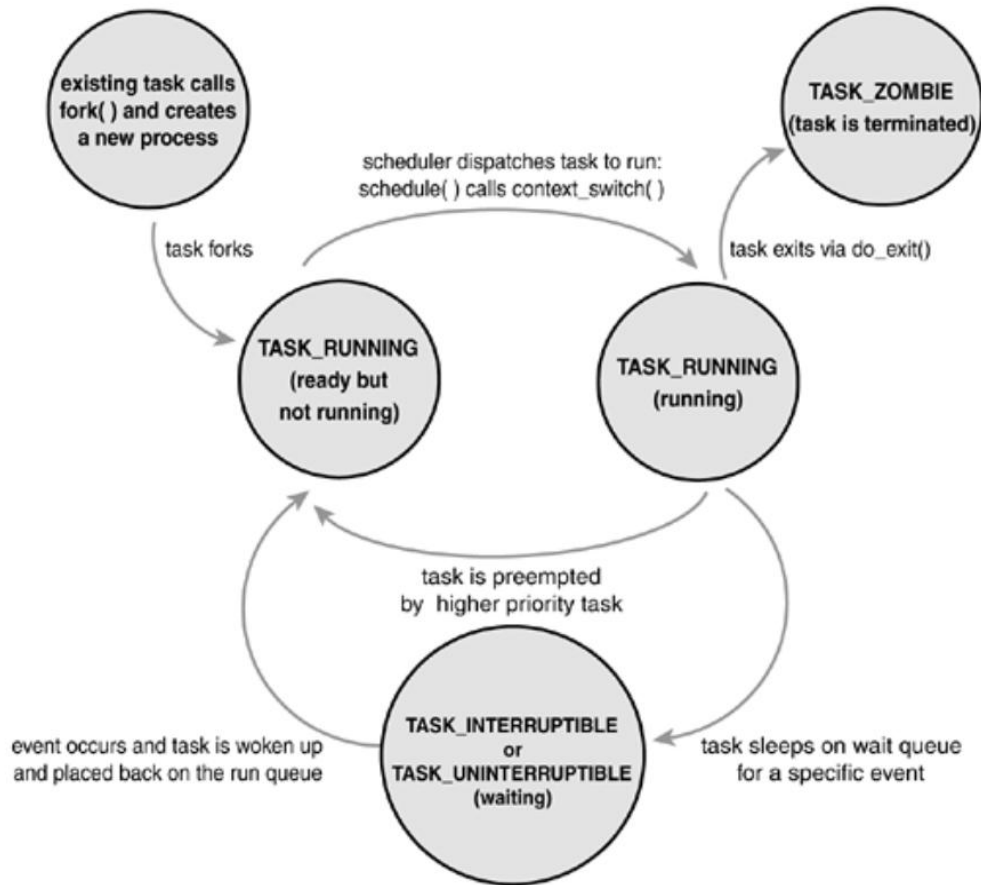
Task States

- The `state` field n the PCB tracks the current state of the process/thread

- Values are defined in `inlude/linux/sched.h`
  - `TASK_RUNNING`
  - `TASK_ZOMBIE`
  - `TASK_STOPPED`
  - `TASK_INTERRUPTIBLE`
  - `TASK_UNINTERRUPTIBLE`
  - `TASK_KILLABLE`

- All the PCBs registered in the runqueue are `TASK_RUNNING`

Task State Transition

Tasks Going to Sleep

- In case an operation cannot be completed immediately (think of a `read()`) the task goes to sleep in a wait queue

- While doing this, the task enters either the `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state

- At this point, the kernel thread calls `schedule()` to effectively put to sleep the currently-running one and pick the new one to be activated

TASK_*INTERRUPTIBLE

- Dealing with `TASK_INTERRUPTIBLE` can be difficult:
  - At kernel level, understand that the task has been resumed due to an interrupt
  - Clean up all the work that has been done so far
  - Return to userspace with `-EINTR`
  - Userspace has to understand that a syscall was interrupted (bugs here!)
- Conversely, a `TASK_UNINTERRUPTIBLE` might never be woken up again (the dreaded D state in ps)
- `TASK_KILLABLE` is handy for this (since 2.6.25)
  - Same as `TASK_UNINTERRUPTIBLE` except for fatal sigs.

Sleeping Task Wakes Up

- The event a task is waiting for calls one of the `wake_up*()` functions on the corresponding wait queue

- A task is set to runnable and put back on a runqueue

- It the woken up task has a higher priority than the other tasks on the runqueue, `TIF_NEED_RESCHED` is flagged

O(n) Scheduler (2.4)

- It has a linear complexity, as it iterates over all tasks

- Time is divided into *epochs*

- At the end of an epoch, every process has run once, using up its whole quantum if possible

- If processes did not use the whole quantum, they have half of the remaining timeslice added to the new timeslice

O(n) Scheduler (2.4)

```
asmlinkage void schedule(void) {
  int this_cpu, c; /* weight */
  ...
 repeat_schedule:
  /* Default process to select.. */
  next = idle_task(this_cpu);
  c = -1000; /* weight */
  list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
      int weight = goodness(p, this_cpu, prev->active_mm);
       if (weight > c)
         c = weight, next = p;
    }
  }
}
```

Computing the Goodness

goodness (p)= 20 - `p->nice`   (base time quantum)

+ `p->counter`   (ticks left in time quantum)

+1   (if page table is shared
             with the previous process)

+15   (in SMP, if p was last
             running on the same CPU)

Computing the Goodness

- Goodness values explained and special cases:
  - -1000: never select this process to run
  - 0: out of timeslice (`p->counter == 0`)
  - >0: the goodness value, the larger the better
  - +1000: a realtime process, select this

Epoch Management

```
……………
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
                p->counter = (p->counter >> 1) +
                                NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
}
……………
```

6 - p->nice/4

Analysis of the O(n) Scheduler

- Disadvantages:
  - A non-runnable task is also searched to determine its goodness
  - Mixture of runnable/non-runnable tasks into a single runqueue in any epoch
  - Performance problems on SMP, as the length of critical sections depends on system load

- Advantages:
  - Perfect Load Sharing
  - No CPU underutilization for any workload type
  - No (temporary) binding of threads to CPUs

Contention in the O(n) Scheduler on SMP

Core-0 calls `schedule()`

All other cores call `schedule()`

Core-0 returns
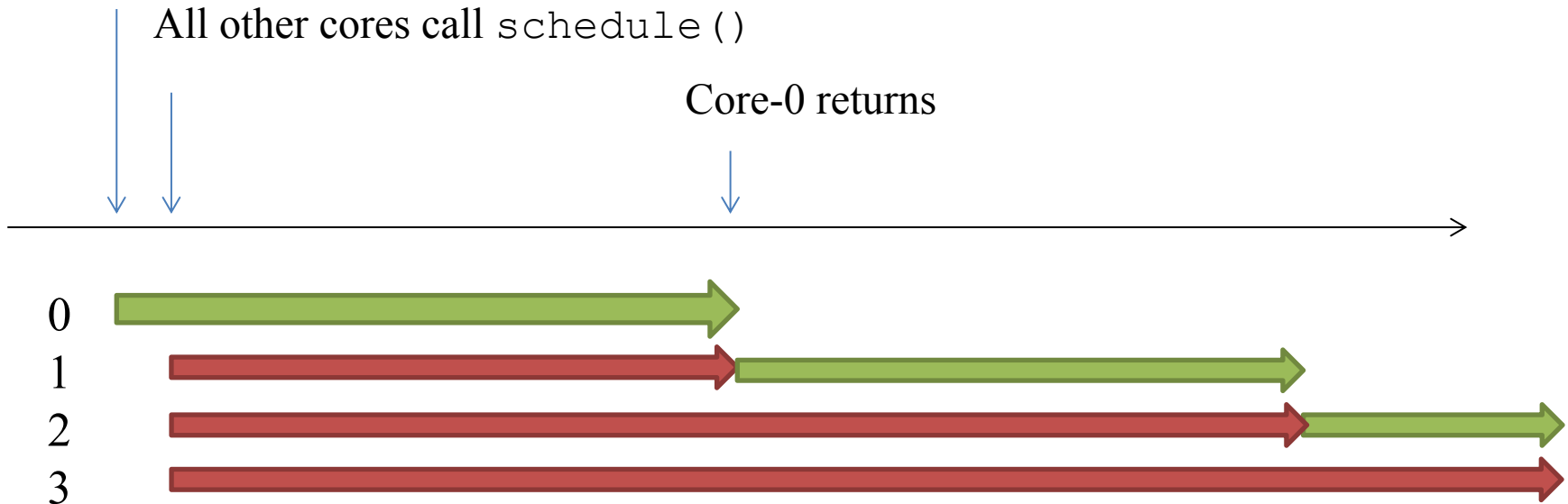
O(1) Scheduler (2.6.8)

- By Ingo Molnár
- Schedules tasks in constant time, indepentendly of the number of active processes
- Introduced the global priority scale which we discussed
- Early preëmption: if a task enters the `TASK_RUNNING` state its priority is checked to see whether to call `schedule()`
- Static priority for real-time tasks
- Dynamic priority for other tasks, recalculated at the end of their timeslice (increases interactivity)

Runqueue Revisited

```
struct runqueue {
    /* number of runnable tasks */
    unsigned long nr_running;
    ...
    struct prio_array *active;
    struct prio_array *expired;
    struct prio_array arrays[2];
}
```
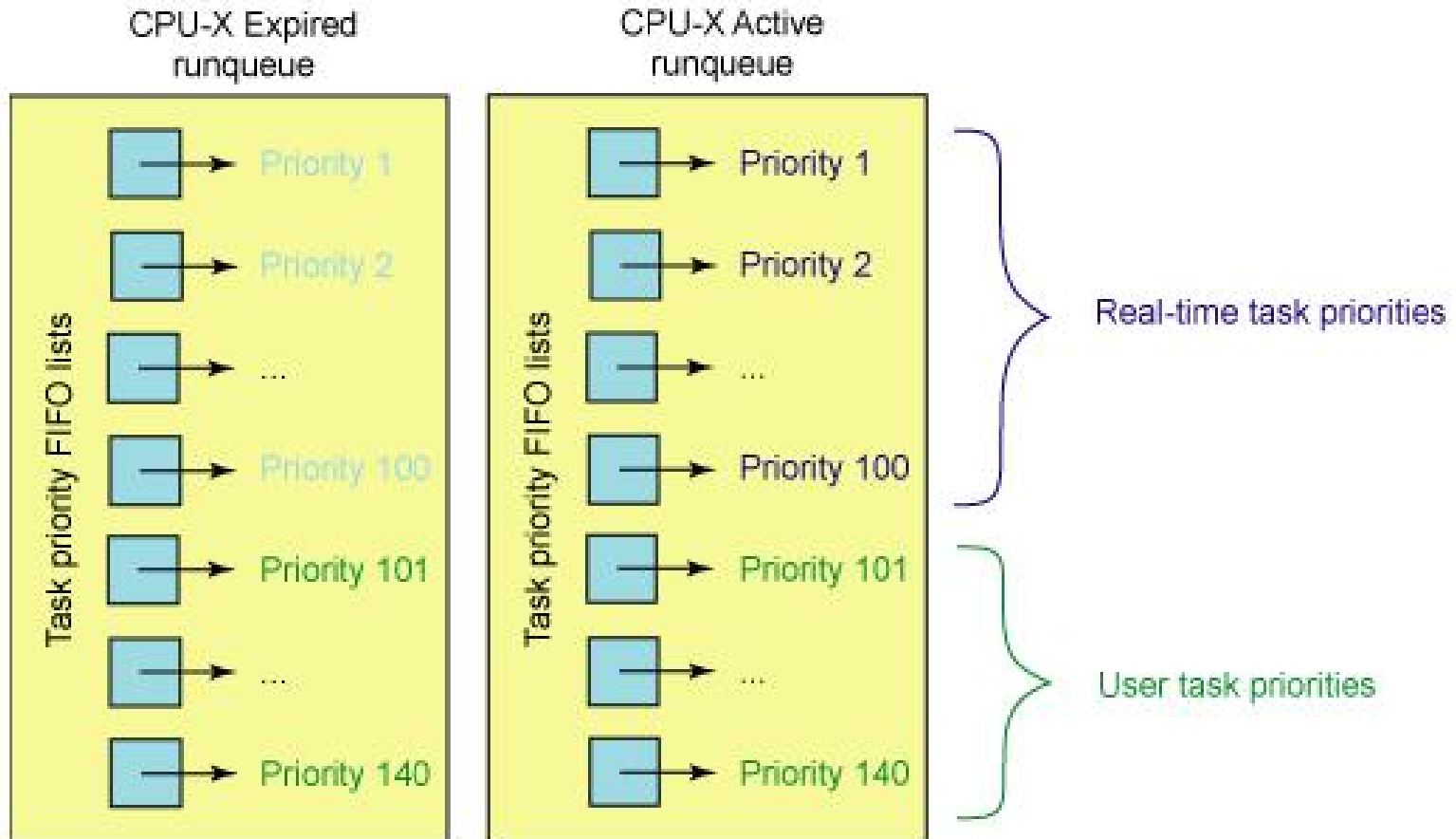
Runqueue Revisited

- Each runqueue has two `struct prio_array`:

```
struct prio_array {
    int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};
```

Runqueue Revisited

Runqueue Revisited

`schedule() → schedule_find_first_set()`

bit 0, priority 0

bit 10, priority 10 ⟵

										X		X	
				X									
								X					
						X							
							X						
							X			X	X	X	

bit 139
priority 139

⟵

Cross-CPU Scheduling

- Once a task lands on a CPU, it might use up its timeslice and get put back on a prioritized queue for rerunning—but how might it ever end up on another processor?

- If all the tasks on one CPU exit, might not one processor stand idle while another round-robins three, ten or several dozen other tasks?

- The 2.6 scheduler must, on occasion, see if cross-CPU balancing is needed.

- Every 200ms a CPU checks to see if any other CPU is out of balance and needs to be balanced with that processor. If the processor is idle, it checks every 1ms so as to get started on a real task earlier

Stack Variables Refresh

```
asmlinkage 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_qsctr_inc(cpu);
        prev = rq->curr;
        switch_count = &prev->nivcsw;
...
```

Stack Variables Refresh

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

prev->sched_class->put_prev_task(rq, prev);
next = pick_next_task(rq, prev);

if (likely(prev != next)) {
        sched_info_switch(prev, next);

        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        context_switch(rq, prev, next); /* unlocks the rq */
        /* the context switch might have flipped the stack from under
           us, hence refresh the local variables. */
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
} else spin_unlock_irq(&rq->lock);
...
```

Staircase Scheduler

- By Con Kolivar, 2004 (none of its schedulers in the official Kernel tree)

- The goal is to increase "responsiveness" and reduce the complexity of the O(1) Scheduler

- It is mostly based on dropping the priority recalculation, replacing it with a simpler rank-based scheme

- It is supposed to work better up to ~10 CPUs (tailored for desktop environments)

Staircase Scheduler

- The expired array is removed and the staircase data structure is used instead

		Priority rank									
Iteration	Base	-1	-2	-3	-4	-5	-6	-7	-8	-9	...
1	1	1	1	1	1	1	1	1	1	1	
2		2	1	1	1	1	1	1	1	1	
3			3	1	1	1	1	1	1		

- A process expiring its timeslice is moved to a lower priority

- At the end of the staircase, it gets to a MAX_PRIO-1 level with one more timeslice

- If a process sleeps (i.e., an interactive process) it gets back up in the staircase

- This approach favors interactive processes rather CPU-bound ones

Completely Fair Scheduler (2.6.23)

- Merged in October 2007
- This is since then the default Scheduler
- This models an "ideal, precise multitasking CPU" on real hardware
- It is based on a red-black tree, where nodes are ordered by process execution time in nanoseconds
- A maximum execution time is also calculated for each process

Completely Fair Scheduler (2.6.23)

Context switch (5.0)

- Context switch is implemented in the `switch_to()` macro in `/arch/x86/include/asm/switch_to.h`

- The macro is machine-dependent

```
#define switch_to(prev, next, last)                 \
do {                                                 \
        prepare_switch_to(next);                     \
        ((last) = __switch_to_asm((prev), (next)));  \
} while (0)
```

- `switch_to()` mainly executes the following two tasks:

  – TSS update

  – CPU control registers update

__switch_to_asm() (x86)

```
ENTRY(__switch_to_asm)
        /* Save callee-saved registers */
        pushq   %rbp
        pushq   %rbx
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %r15

        /* switch stack */
        movq    %rsp, TASK_threadsp(%rdi)
        movq    TASK_threadsp(%rsi), %rsp

#ifdef CONFIG_STACKPROTECTOR
        movq    TASK_stack_canary(%rsi), %rbx
        movq    %rbx, PER_CPU_VAR(irq_stack_union)+stack_canary_offset
#endif

        /* restore callee-saved registers */
        popq    %r15
        popq    %r14
        popq    %r13
        popq    %r12
        popq    %rbx
        popq    %rbp

        jmp     __switch_to
END(__switch_to_asm)
```