Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[11] Scheduling

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v2

Outline

1. **Introduction**
2. **Priorities and Weights**
3. **Scheduler Core**
    1. Wait Queues
    2. Scheduler Entry Point
    3. Scheduler Algorithms
4. **Context Switch**

11.1

11. Scheduling

Introduction

DIAG

Process Scheduling

Like every time sharing system, Linux achieves the effect of an apparent simultaneous execution of multiple processes by switching from one process to another in a very short time frame.

The scheduling policy is concerned with **when** to switch and **which** process to choose. The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives:

- fast process response time
- good throughput for background jobs
- no starvation
- reconciliation of the needs of low- and high- priority processes and so on

Linux Scheduling

Linux Scheduling is based on the time sharing technique: several processes run in "time multiplexing" because the CPU is divided into **slices**, one for each runnable process. Obviously one CPU can run only one process at a given instant, therefore when the currently running process is not terminated when its time slice or ***quantum*** expires, a process switch may take place.

Time sharing relies on timer interrupts and is thus transparent to process, no additional code needs to be inserted in the programs to ensure CPU time sharing.

The developing history of Linux has seen different scheduling algorithms.

11.2

11. Scheduling

Priorities and Weights

DIAG

Priority

In Linux, process priority is dynamic. The scheduler keeps track of what processes are doing and adjusts their priorities **periodically**. In this way, processes that have been denied the use of CPU for a long time interval are boosted by dynamically increasing their priority (and vice versa). Process in general are:

- CPU Bound, if require a lot of cpu time
- I/O Bound, if require a lot of I/O operations

Or, according to another classification:

- **interactive**, if they interact a lot with the user, therefore they spend much time waiting for keystrokes and mouse operations; these processes must be very responsive
- **batch**, if they do not need user interaction, since they run in background
- **real-time**, they have to follow strict scheduling requirements, they should be never blocked by a lower priority process (e.g. video, sound applications)

The Linux scheduler allows only to define real-time processes.

Nice and RT

Unix demands for priority based scheduling. To each process is associated a “nice” number 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

Priorities

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

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

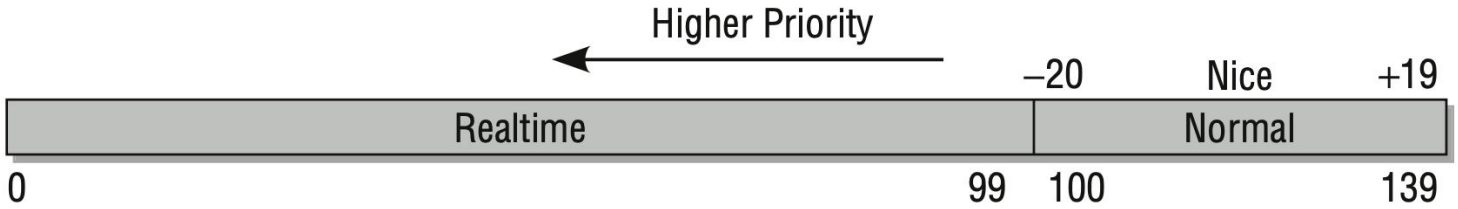


Figure 2-14: Kernel priority scale.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Priorities

Macros

```
#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)
```

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

Priorities

```
/*
* 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))
```

Priorities in `task_struct`

There are several fields for representing the priority in the task_struct:

- **`static_prio` (static)**: priority given "statically" by a user (and mapped into kernel's representation);
- **`normal_priority` (dynamic)**: 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)**: it is the priority considered by the scheduler, it changes over the process execution keeping track CPU bound processes to penalize, and I/O bound processes to boost;
- **`rt_priority` (static)**: the realtime priority for realtime tasks in [0, 99].

*__static__ means that is assigned at process startup but then it can be changed by issuing a system call, __dynamic__ means that the scheduler recomputes it during the process execution

Computing prio

V5.11

In kernel/sched/core.c

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

```
1655  /*
1656   * Calculate the current priority, i.e. the priority
1657   * taken into account by the scheduler. This value might
1658   * be boosted by RT tasks, or might be boosted by
1659   * interactivity modifiers. Will be RT if the task got
1660   * RT-boosted. If not then it returns p->normal_prio.
1661   */
1662  static int effective_prio(struct task_struct *p)
1663  {
1664          p->normal_prio = normal_prio(p);
1665          /*
1666           * If we are RT tasks or we were boosted to RT priority,
1667           * keep the priority unchanged. Otherwise, update priority
1668           * to the normal priority:
1669           */
1670          if (!rt_prio(p->prio))
1671                  return p->normal_prio;
1672          return p->prio;
1673  }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/core.c#L1662

Policy is SCHED_DEADLINE

Policy is SCHED_FIFO/RR

```
1635  /*
1636   * Calculate the expected normal priority: i.e. priority
1637   * without taking RT-inheritance into account. Might be
1638   * boosted by interactivity modifiers. Changes upon fork,
1639   * setprio syscalls, and whenever the interactivity
1640   * estimator recalculates.
1641   */
1642  static inline int normal_prio(struct task_struct *p)
1643  {
1644          int prio;
1645
1646          if (task_has_dl_policy(p))
1647                  prio = MAX_DL_PRIO-1;
1648          else if (task_has_rt_policy(p))
1649                  prio = MAX_RT_PRIO-1 - p->rt_priority;
1650          else
1651                  prio = __normal_prio(p);
1652          return prio;
1653  }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/core.c#L1642

Check if p->prio > 100

```
1627  /*
1628   * __normal_prio - return the priority that is based on the static prio
1629   */
1630  static inline int __normal_prio(struct task_struct *p)
1631  {
1632          return p->static_prio;
1633  }
```

Load weights

V5.11

The importance of a task is not only described by the priority, but also by a load weight, used to scale the time slice assigned to a scheduled process. The weight can be found in:

`task_struct->se->load` (of type **`struct load_weight`**)

```
327  struct load_weight {
328          unsigned long            weight;
329          u32                      inv_weight;
330  };
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/sched.h#L327

Load Weights

V5.11

```
9140  /*
9141   * Nice levels are multiplicative, with a gentle 10% change for every
9142   * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
9143   * nice 1, it will get ~10% less CPU time than another CPU-bound task
9144   * that remained on nice 0.
9145   *
9146   * The "10% effect" is relative and cumulative: from _any_ nice level,
9147   * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
9148   * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
9149   * If a task goes up by ~10% and another task goes down by ~10% then
9150   * the relative distance between them is ~25%.)
9151   */
9152  const int sched_prio_to_weight[40] = {
9153   /* -20 */     88761,     71755,     56483,     46273,     36291,
9154   /* -15 */     29154,     23254,     18705,     14949,     11916,
9155   /* -10 */      9548,      7620,      6100,      4904,      3906,
9156   /*  -5 */      3121,      2501,      1991,      1586,      1277,
9157   /*   0 */      1024,       820,       655,       526,       423,
9158   /*   5 */       335,       272,       215,       172,       137,
9159   /*  10 */       110,        87,        70,        56,        45,
9160   /*  15 */        36,        29,        23,        18,        15,
9161  };
```

nice = 19

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/core.c

Load weights

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)

Conclusions

Therefore for now we introduced that for a scheduling process we need:

- a set of dynamic and static priorities
- a load weight

11.3

11. Scheduling

Scheduler Core

DIAG

Scheduling Classes/Policies

Every Linux process is always scheduled according to one of the following scheduling classes. The policy is described in `task_struct->policy`:

- `SCHED_NORMAL` (also called `SCHED_OTHER`), conventional time-shared process, in general has a soft priority mechanism over the 'nice' range of -20 to +19 (static priority of 100-139) which decides according to the priority which task goes first, and how much timeslice it gets. This system dynamically alters the priority to allow interactive tasks to go first, and is designed to prevent starvation of lower priority tasks with an expiration policy
- `SCHED_RR`, is a fixed real time policy over the static range of 0-99 where a lower number (higher priority) task will repeatedly go ahead of _any_ tasks lower priority than itself. It is called RR because if multiple tasks are at the same priority it will Round Robin between those tasks
- `SCHED_FIFO`, is a fixed real time policy the static range of 0-99 where a lower number (higher priority) task will repeatedly go ahead of _any_ tasks with lower priority than itself. Unlike RR, if a task does not give up the cpu it will run indefinitely even if other tasks are the same static priority as itself.

Scheduling Classes/Policies

Every Linux process is always scheduled according to one of the following scheduling classes:

- `SCHED_BATCH`, does not preempt nearly as often as regular tasks would, thereby allowing tasks to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs.
- `SCHED_IDLE`, even weaker than `SCHED_BATCH`
- `SCHED_DEADLINE`, implementation of the Earliest Deadline First (EDF) scheduling algorithm, augmented with a mechanism (called Constant Bandwidth Server, CBS) that makes it possible to isolate the behavior of tasks between each other. CBS has been replaced with Greedy Reclamation of Unused Bandwidth (GRUB) from kernel 4.13.

Scheduling Classes

For each scheduling class/policy a set of standard function is defined as follows, in this way in order to schedule a task the scheduler core uses always the same set of APIs. We have:

- **enqueue_task(...)** Called when a task enters a runnable state. It puts the scheduling entity (task) into the red-black tree and increments the nr_running variable.
- **dequeue_task(...)** When a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree. It decrements the nr_running variable.
- **yield_task(...)** This function is basically just a dequeue followed by an enqueue, unless the compat_yield sysctl is turned on; in that case, it places the scheduling entity at the right-most end of the red-black tree.
- **check_preempt_curr(...)** This function checks if a task that entered the runnable state should preempt the currently running task.
- **pick_next_task(...)** This function chooses the most appropriate task eligible to run next.
- **set_curr_task(...)** This function is called when a task changes its scheduling class or changes its task group.
- **task_tick(...)** This function is mostly called from time tick functions; it might lead to process switch. This drives the running preemption.

Run-queue

```
1814  struct sched_class {
1815
1816  #ifdef CONFIG_UCLAMP_TASK
1817          int uclamp_enabled;
1818  #endif
1819
1820          void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
1821          void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
1822          void (*yield_task)   (struct rq *rq);
1823          bool (*yield_to_task)(struct rq *rq, struct task_struct *p);
1824
1825          void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);
1826
1827          struct task_struct *(*pick_next_task)(struct rq *rq);
1828
1829          void (*put_prev_task)(struct rq *rq, struct task_struct *p);
1830          void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);
1831
1832  #ifdef CONFIG_SMP
1833          int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
1834          int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);
1835          void (*migrate_task_rq)(struct task_struct *p, int new_cpu);
1836
1837          void (*task_woken)(struct rq *this_rq, struct task_struct *task);
1838
1839          void (*set_cpus_allowed)(struct task_struct *p,
1840                                   const struct cpumask *newmask,
1841                                   u32 flags);
1842
1843          void (*rq_online)(struct rq *rq);
1844          void (*rq_offline)(struct rq *rq);
1845
1846          struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
1847  #endif
1848
1849          void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
1850          void (*task_fork)(struct task_struct *p);
1851          void (*task_dead)(struct task_struct *p);
```

https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html?highlight=sched_normal#scheduling-classes - https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/sched.h#L1814

Overview

Scheduling can be activated in 2 ways: when a task goes to sleep (or yield the CPU) or by a periodic mechanism.

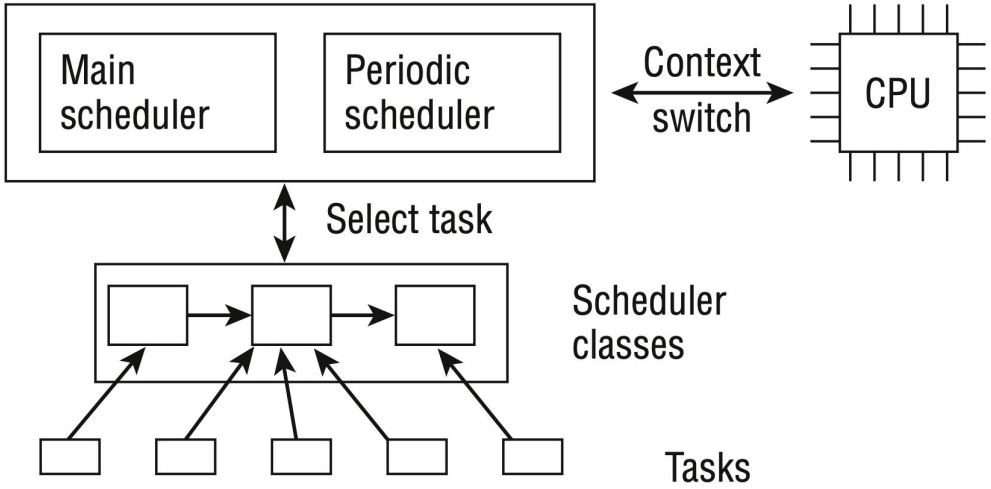


Figure 2-13: Overview of the components of the scheduling subsystem.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

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**), it implements `SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE`
- `rt.c`: the real-time scheduler implements `SCHED_FIFO` and `SCHED_RR`
- `idle_task.c`: generic entry points for the idle threads and implementation of the idle task scheduling class (not related to `SCHED_IDLE`) for scheduling the idle task (i.e. do_idle)

Run Queues

The central data structure of the core scheduler that is used to manage active processes is known as **run queue**. Each CPU has its own run queue, and each active process appears on just one run queue.

```
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;
}
```

Number of runnable processes in the queue

Past values of CPU load

Current load on the queue

Sub-runqueues for the cfs and the rt schedulers

The clock of the current run queue is updated each time the Periodic Scheduler is called

Pointers to the current running process and to the idle process to schedule when there is no other process to schedule

Run Queues

```
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)
```

11.3.1

11. Scheduling
3. Scheduler Core

Wait Queues

DIAG

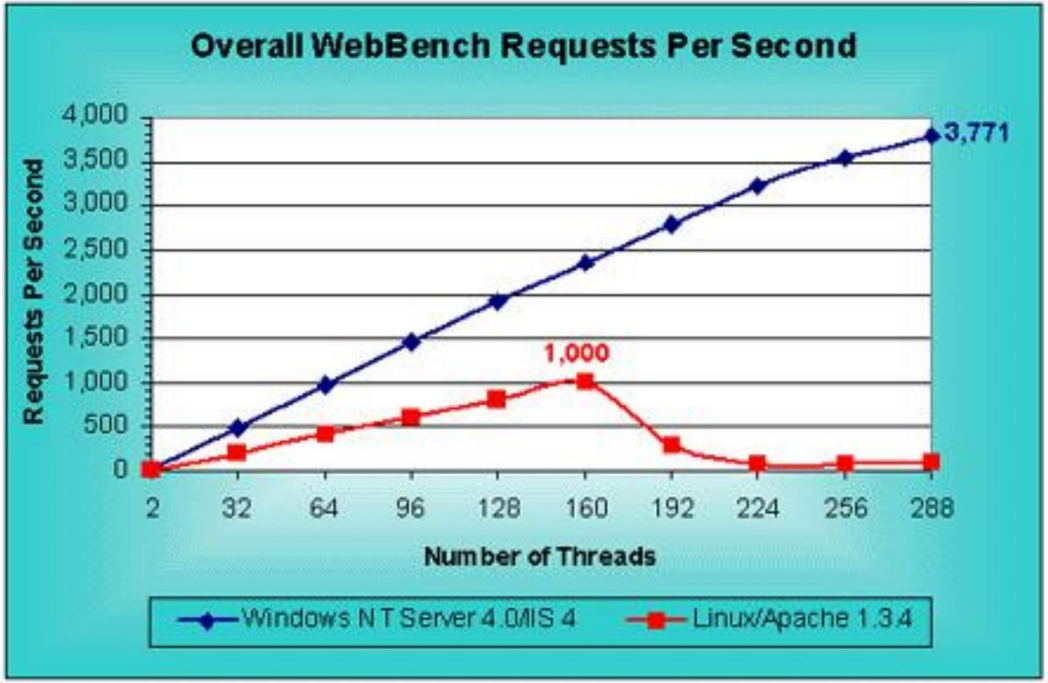
Wait Queues

Defined in `include/linux/wait.h` Wait Queues implement **conditional waits on events**: a process wishing to wait for a specific event places itself in the proper wait queue and relinquishes control. Therefore a wait queue represents a set of sleeping processes, which are woken up by the kernel when some condition becomes true.

Wait Queues changed many times in the history of the kernel. In the earlier version they suffered from the "Thundering Herd" performance problem.

Wait Queues

Thundering Herd Effect



Taken from 1999 Mindcraft study on Web and File Server Comparison

Wait Queues

```c
#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;
```

For solving the Thundering Herd problem the kernel defines two kinds of sleeping processes:

- ***exclusive***, are selectively woken up by the kernel
- ***non-exclusive***, are always woken up by the kernel

Specifies how waking up the sleeping process

https://elixir.bootlin.com/linux/v5.11/source/include/linux/wait.h

Wait Queues

APIs

- `static inline void init_waitqueue_entry(struct wait_queue_entry *wq_entry, struct task_struct *p)`
- `static inline void wait_event_interruptible(wq_head, condition)` - sleep until a condition gets true
- `static inline void wait_event_interruptible_timeout(wq_head, condition, timeout)` - sleep until a condition gets true or a timeout elapses
- `static inline void wait_event_hrtimeout(wq_head, condition, timeout)`
- `static inline void wait_event_interruptible_hrtimeout(wq, condition, timeout)`

https://elixir.bootlin.com/linux/v5.11/source/include/linux/wait.h

Wait Queues

Adding entry to a wait queue

```
18  void add_wait_queue(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry)
19  {
20          unsigned long flags;
21
22          wq_entry->flags &= ~WQ_FLAG_EXCLUSIVE;
23          spin_lock_irqsave(&wq_head->lock, flags);
24          __add_wait_queue(wq_head, wq_entry);
25          spin_unlock_irqrestore(&wq_head->lock, flags);
26  }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/wait.c#L18

```
29  void add_wait_queue_exclusive(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry)
30  {
31          unsigned long flags;
32
33          wq_entry->flags |= WQ_FLAG_EXCLUSIVE;
34          spin_lock_irqsave(&wq_head->lock, flags);
35          __add_wait_queue_entry_tail(wq_head, wq_entry);
36          spin_unlock_irqrestore(&wq_head->lock, flags);
37  }
38  EXPORT_SYMBOL(add_wait_queue_exclusive);
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/wait.c#L29

Wait Queues

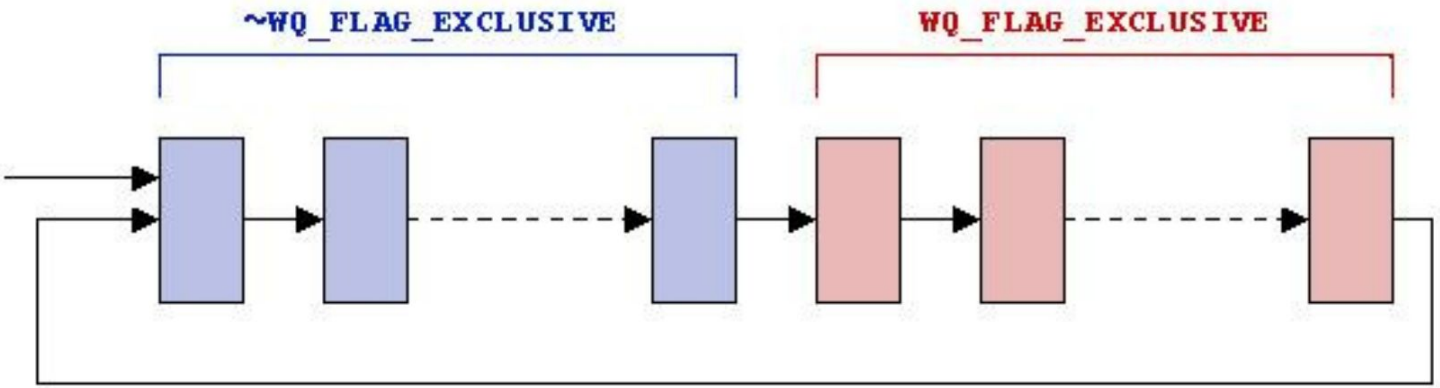
Removing entry from a wait queue

```
51  void remove_wait_queue(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry)
52  {
53          unsigned long flags;
54
55          spin_lock_irqsave(&wq_head->lock, flags);
56          __remove_wait_queue(wq_head, wq_entry);
57          spin_unlock_irqrestore(&wq_head->lock, flags);
58  }
59  EXPORT_SYMBOL(remove_wait_queue);
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/wait.c#L51

Wait Queues

Tasks organization



Wait Queues

APIs (2)

These functions wake up `TASK_NORMAL = TASK_INTERRUPTIBLE + TASK_UNINTERRUPTIBLE`

- `wake_up(x)`
- `wake_up_nr(x, nr)`
- `wake_up_all(x)`
- `wake_up_locked(x)`
- `wake_up_all_locked(x)`

These instead only `TASK_INTERRUPTIBLE`

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

11.3.2

11. Scheduling
3. Scheduler Core

Scheduler Entry Point

DIAG

Overview

Scheduling can be activated in 2 ways: when a task goes to sleep (or yield the CPU) or by a periodic mechanism.

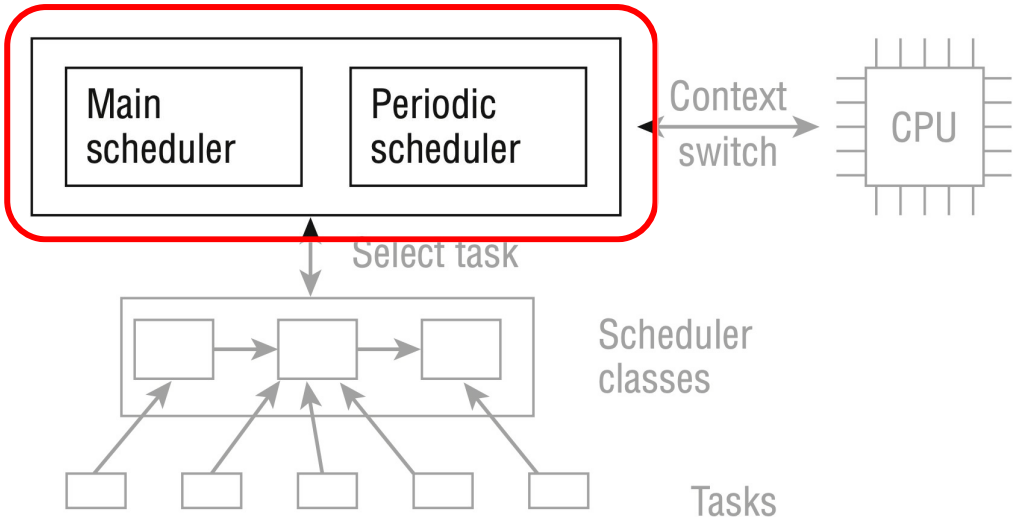


Figure 2-13: Overview of the components of the scheduling subsystem.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Scheduler Main Functions

The main functions used by the scheduler are:

`scheduler_tick()`
: Keeps the `time_slice` counter of `current` up-to-date

`try_to_wake_up()`
: Awakens a sleeping process

`recalc_task_prio()`
: Updates the dynamic priority of a process

`schedule()`
: Selects a new process to be executed

`load_balance()`
: Keeps the runqueues of a multiprocessor system balanced

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

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`)

These invocations can be triggered by the **Main Scheduler** or the **Periodic Scheduler**.

In general `schedule()` entails 3 distinct phases, which depend on the scheduler implementation:

1. some **checks** on the current process (e.g., with respect to signal processing)
2. **selection** of the process to be activated
3. context **switch**

Periodic Scheduler

The function `scheduler_tick()` is called from `update_process_times()`, called at every tick of the current CPU (remind the Time Management chapter).

This function has two goals:

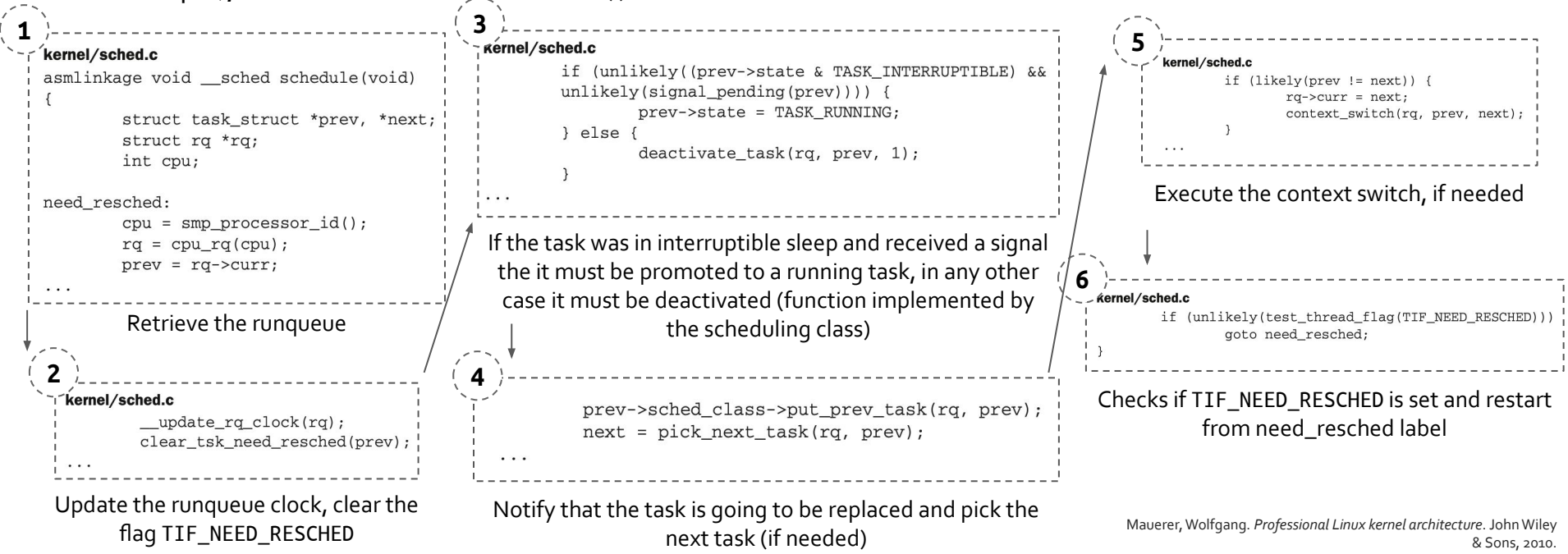
- managing scheduling-specific statistics
- calling the scheduling method of the class

```
4528  /*
4529   * This function gets called by the timer code, with HZ frequency.
4530   * We call it with interrupts disabled.
4531   */
4532  void scheduler_tick(void)
4533  {
4534          int cpu = smp_processor_id();
4535          struct rq *rq = cpu_rq(cpu);
4536          struct task_struct *curr = rq->curr;
4537          struct rq_flags rf;
4538          unsigned long thermal_pressure;
4539
4540          arch_scale_freq_tick();
4541          sched_clock_tick();
4542
4543          rq_lock(rq, &rf);
4544
4545          update_rq_clock(rq);
4546          thermal_pressure = arch_scale_thermal_pressure(cpu_of(rq));
4547          update_thermal_load_avg(rq_clock_thermal(rq), rq, thermal_pressure);
4548          curr->sched_class->task_tick(rq, curr, 0);
4549          calc_global_load_tick(rq);
4550          psi_task_tick(rq);
4551
4552          rq_unlock(rq, &rf);
4553
4554          perf_event_task_tick();
4555
4556  #ifdef CONFIG_SMP
4557          rq->idle_balance = idle_cpu(cpu);
4558          trigger_load_balance(rq);
4559  #endif
4560  }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/core.c#L4532

Main Scheduler

The main scheduler function (`schedule()`) is invoked directly in many points in the kernel to allocate the CPU to a process other than the currently active one. After returning from system calls the kernel also checks whether the flag TIF_NEED_RESCHED of the current process is set (by the Periodic Scheduler for example), and if it is checked `schedule()` is called.



Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Task States

The state field in the PCB tracks the current state of the process/thread. Values are defined in include/linux/sched.h:

- **TASK_RUNNING** the process is either executing on CPU or waiting to be executed
- **TASK_INTERRUPTIBLE** the process is sleeping until some condition becomes true
- **TASK_UNINTERRUPTIBLE** like TASK_INTERRUPTIBLE but they can be only woken up by the kernel, not by external signals
- **TASK_STOPPED** process has stopped (after signal SIGSTOP, SIGTSTP)
- TASK_PARKED
- TASK_DEAD
- **TASK_WAKEKILL** is designed to wake the process on receipt of fatal signals
- TASK_WAKING
- TASK_NOLOAD
- **TASK_NEW** the task has been just created
- TASK_STATE_MAX

Convenience macros for the sake of set_current_state:

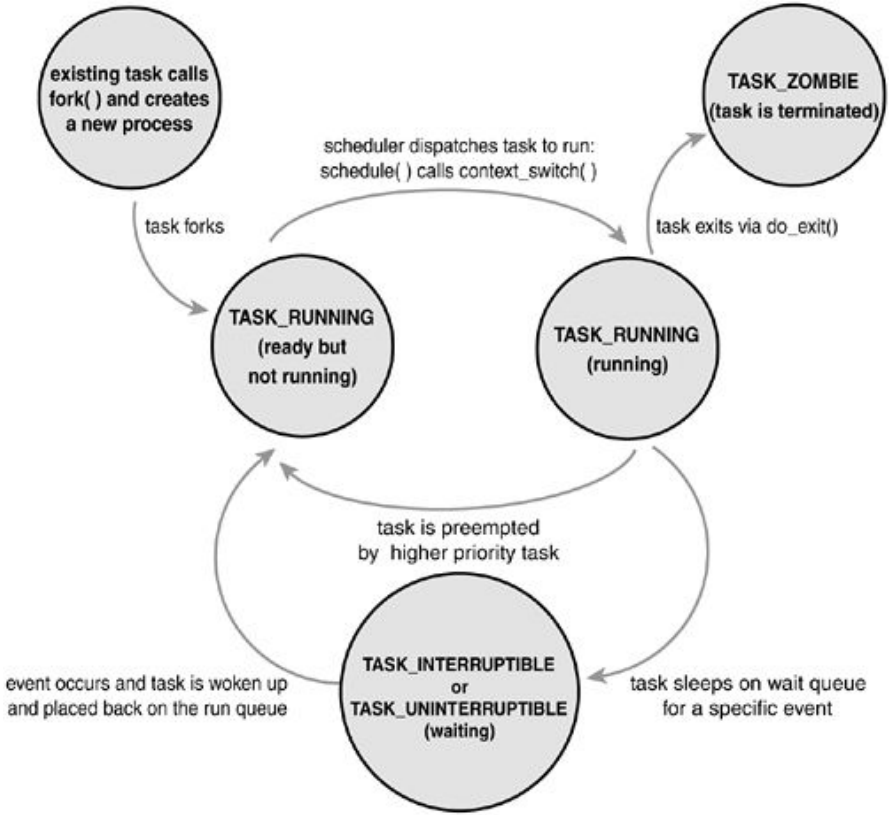
```
- #define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
- #define TASK_STOPPED  (TASK_WAKEKILL | __TASK_STOPPED)
- #define TASK_TRACED   (TASK_WAKEKILL | __TASK_TRACED)
- #define TASK_IDLE     (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
```

Convenience macros for the sake of wake_up():

```
- #define TASK_NORMAL   (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
```

All the PCBs registered in the runqueue are TASK_RUNNING.

Task State Transition



TASK_*INTERRUPTIBLE

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.

Dealing with `TASK_INTERRUPTIBLE` can be difficult when the syscall is interrupted for example:

- 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.

Waking up sleeping tasks

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.

11.3.3

11. Scheduling
3. Scheduler Core

Scheduling Algorithms

DIAG

Brief History

- **v1.2**: circular queue for runnable task management that operated with a round-robin scheduling policy. This scheduler was efficient for adding and removing processes (with a lock to protect the structure)
- **v2.2**: introduced the idea of scheduling classes, permitting scheduling policies for real-time tasks, non-preemptible tasks, and non-real-time tasks. The 2.2 scheduler also included support for symmetric multiprocessing (SMP)
- **v2.4**: relatively simple scheduler that operated in $O(N)$ time (as it iterated over every task during a scheduling event), time divided into epochs, inefficient for real-time tasks
- **v2.6**: $O(1)$ scheduler, was designed to solve many of the problems with the 2.4 scheduler—namely, the scheduler was not required to iterate the entire task list to identify the next task to schedule, very efficient but the code base was gigantic and obscure with magic constants, difficult to maintain. Due to the pressure of these problems and another proposal for a scheduler by Con Kolivas (the Rotating Staircase Deadline Scheduler), the $O(1)$ was replaced by the CFS (in v2.6.23), that is today in the stable branch of the kernel.

https://developer.ibm.com/technologies/linux/tutorials/l-completely-fair-scheduler

The O(n) scheduler

Up to kernel 2.6.7

The scheduling algorithm used in earlier versions of Linux was quite simple: at every process switch the kernel scanned the entire list of runnable processes (O(n)), computed their priorities and selected the "best" process to run.

For running the algorithm, the time is divided into epochs, at the end of an epoch, every process has run once, using its own quantum if possible. If a process did not use the whole quantum, they have half of the remaining time slice added to the new timeslice.

```
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;
        }
     }
}
```

The O(n) scheduler

The Goodness function

The goodness is computed as follows:

goodness (p)	= 20 - `p->nice`	(base time quantum)
	+ `p->counter`	(ticks left in the time quantum)
	+ 1	(if page table is shared with the previous process)
	+ 15	(if SMP and p was last running of the same CPU)

Possible values:

- -1000 never select this process to run
- 0 out of time slice (`p->counter == 0`)
- >0 the goodness value, the higher the better
- +1000 real-time process, always select

The O(n) scheduler

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
```

The O(n) scheduler

Analysis

Disadvantages:

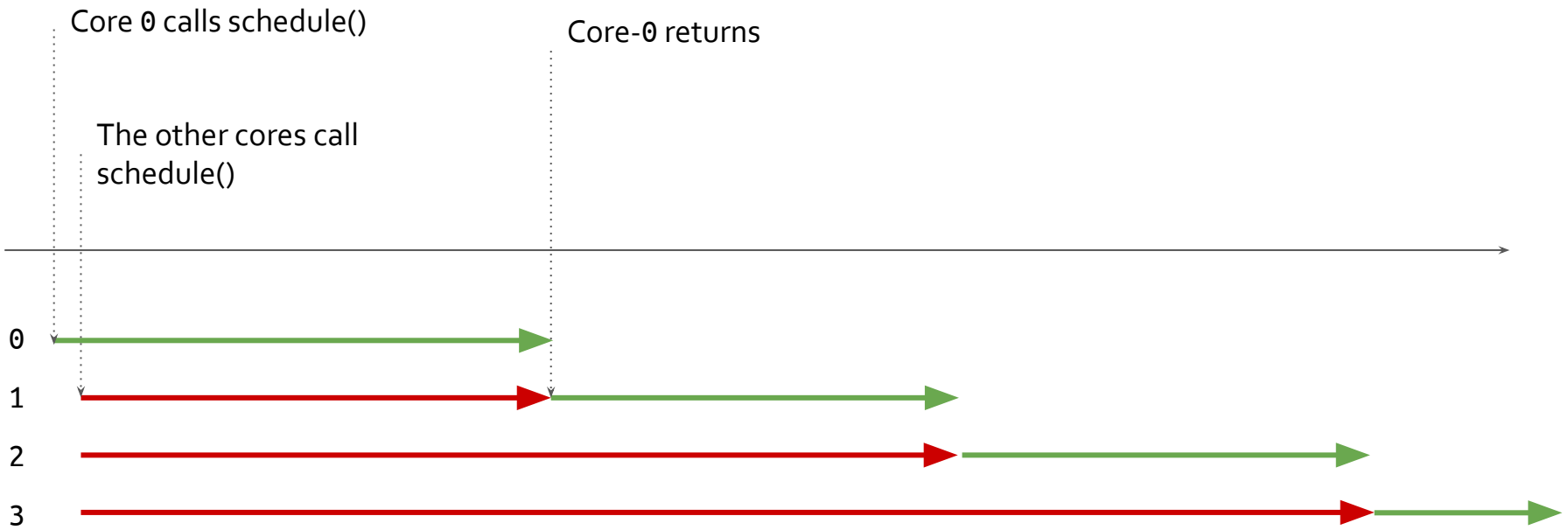
- a non-runnable task is also searched to determine its goodness
- mixture of runnable/non-runnable tasks into a single runqueue in any epoch
- Performance problems on SMP, as the length of critical sections depends on system load

Advantages:

- perfect Load Sharing
- no CPU underutilization for any workload type
- no (temporary) binding of threads to CPUs

The O(n) scheduler

Contention on SMP



The O(1) scheduler

From kernel 2.6.8

The O(1) scheduler has been introduced from version 2.6.8 by Ingo Molnàr. The principal characteristic of the algorithm is that schedules tasks in **constant time**, independently of the number of active processes.

It introduced:

- the global priority scale which we discussed;
- **early preemption**: 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).

The O(1) scheduler

Runqueues

As we already discussed, each CPU has its own **struct** runqueue, the concept has been introduced with this scheduler. However, this data structure keeps two pointers (to 2 sub-runqueues): one to the list of **active** processes and one to the list of **expired** processes. These are not just linked list, they are pointers to **struct** prio_array.

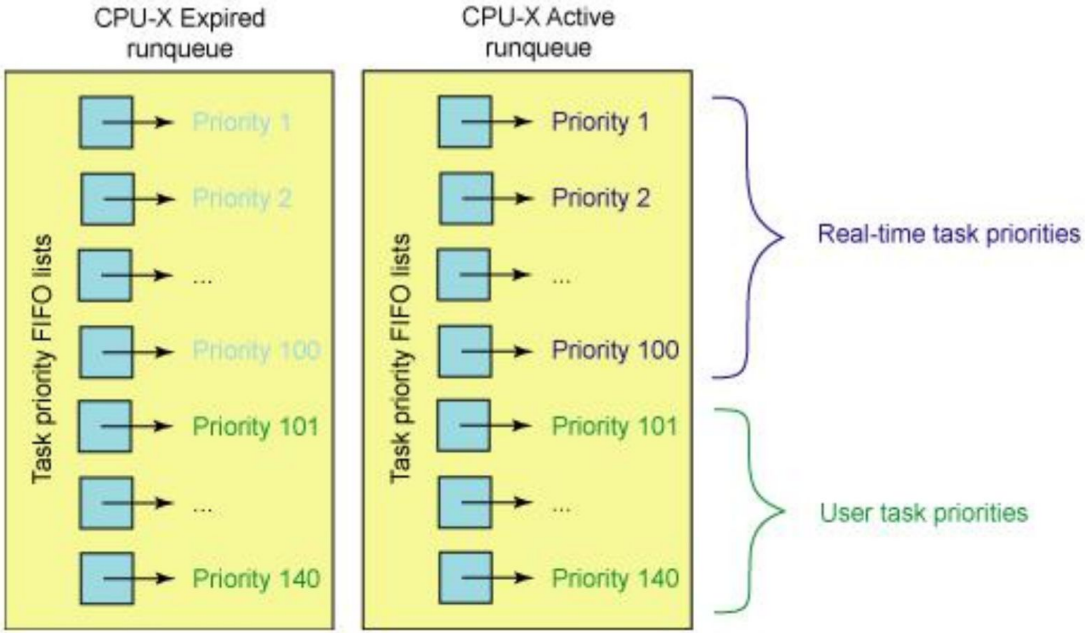
```
struct runqueue {
      /* number of runnable tasks */
      unsigned long nr_running;
      ...
      struct prio_array *active;
      struct prio_array *expired;
      struct prio_array arrays[2];
}
```

The **struct** prio_array maintains an array of list_head for each possible value of the priority, so 140 linked lists and a bitmap.

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

The O(1) scheduler

Runqueues



The O(1) scheduler

The job of the scheduler

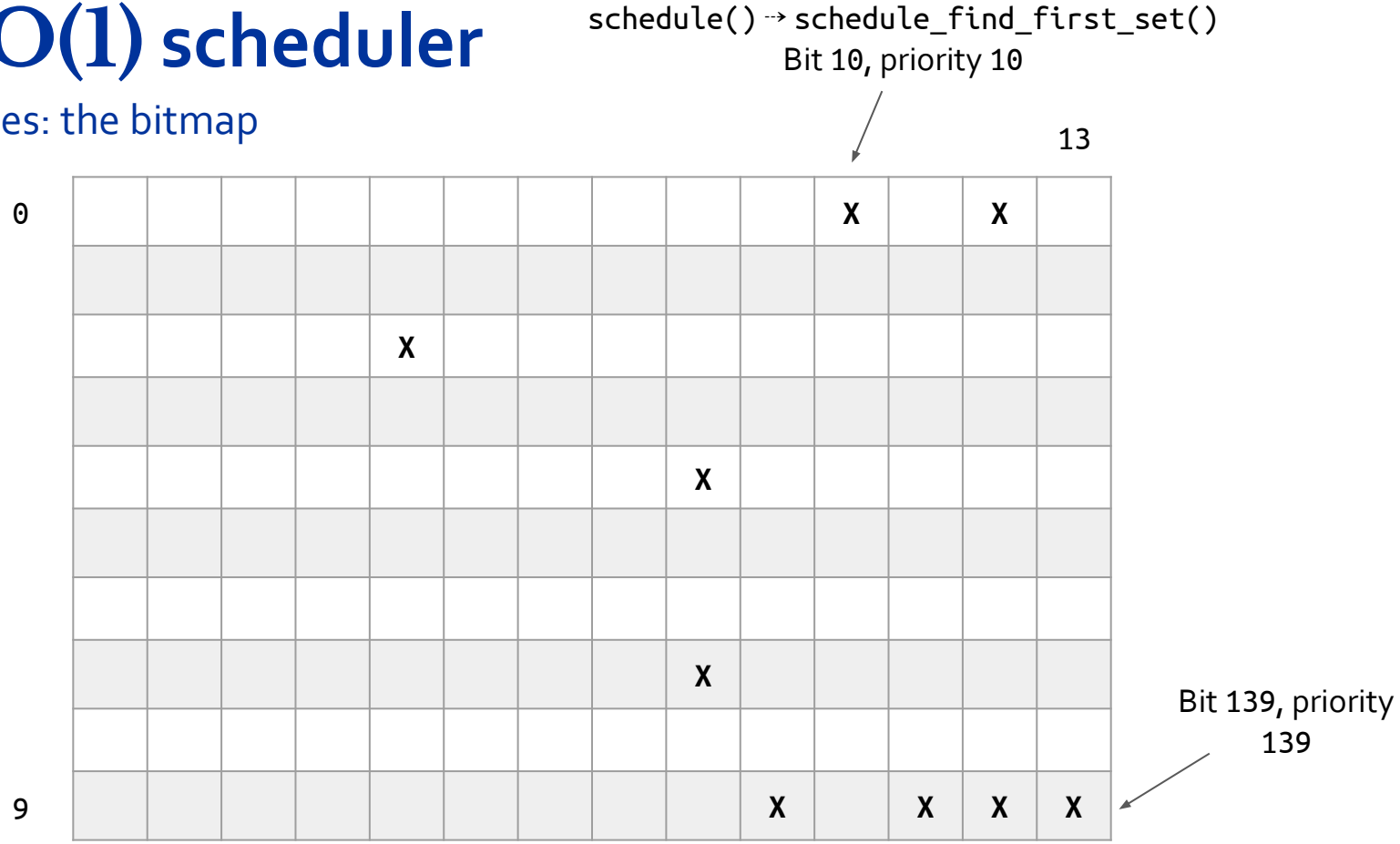
The idea behind the two sub-runqueues, is simple: when a task on the active runqueue uses all of its time slice it's moved to the expired runqueue. During this move the time slice is recalculated (and so its priority). If no task exists on the active runqueue the pointers for the active and the expired runqueues are swapped.

The scheduler always **chooses the task on the highest priority list to execute**. To make this process efficient, a bitmap is used to defined when tasks are on a given priority list. Therefore, on most architectures, the instruction `find_first_bit_set()` is used to find the highest priority bit set in one of five 32-bit words.

As a consequence, the time it takes to find a task to execute depends not on the number of active tasks but instead on the **fixed number of priorities**.

The O(1) scheduler

Runqueues: the bitmap



The O(1) scheduler

Prioritization

To prevent tasks from hogging the CPU and thus starving other tasks that need CPU access, the O(1) scheduler can dynamically alter a task's priority. It does so by penalizing tasks that are bound to a CPU and rewarding tasks that are I/O bound. I/O-bound tasks commonly use the CPU to set up an I/O and then sleep awaiting the completion of the I/O. This type of behavior gives other tasks access to the CPU.

Because I/O-bound tasks are viewed as altruistic for CPU access, their priority is decreased (a reward) by a maximum of five priority levels. CPU-bound tasks are punished by having their priority increased by up to five levels. Tasks are determined to be I/O-bound or CPU-bound based on an ***interactivity heuristic***. A task's interactiveness metric is calculated based on how much time the task executes compared to how much time it sleeps. Note that because I/O tasks schedule I/O and then wait, an I/O-bound task spends more time sleeping and waiting for I/O completion. This increases its interactive metric.

It's important to note that priority adjustments are performed only on user tasks, not on real-time tasks.

https://web.archive.org/web/20210224190439/https://www.ibm.com/developerworks/linux/library/l-scheduler

Cross-CPU Scheduling

v2.6.8

Once a task lands on a CPU, it might use up its time slice 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 CPU_i exit, the CPU_i stand idle while another CPU_j 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.

```
2243        cpu = smp_processor_id();
2244        if (unlikely(!rq->nr_running)) {
2245                idle_balance(cpu, rq);
2246                if (!rq->nr_running) {
2247                        next = rq->idle;
2248                        rq->expired_timestamp = 0;
2249                        wake_sleeping_dependent(cpu, rq);
2250                        goto switch_tasks;
2251                }
2252        }
```

https://elixir.bootlin.com/linux/v2.6.8/source/kernel/sched.c#L2245

```
1791 /*
1792  * idle_balance is called by schedule() if this_cpu is about to become
1793  * idle. Attempts to pull tasks from other CPUs.
1794  */
1795 static inline void idle_balance(int this_cpu, runqueue_t *this_rq)
1796 {
1797         struct sched_domain *sd;
1798
1799         for_each_domain(this_cpu, sd) {
1800                 if (sd->flags & SD_BALANCE_NEWIDLE) {
1801                         if (load_balance_newidle(this_cpu, this_rq, sd)) {
1802                                 /* We've pulled tasks over so stop searching */
1803                                 break;
1804                         }
1805                 }
1806         }
1807 }
```

https://elixir.bootlin.com/linux/v2.6.8/source/kernel/sched.c#L1795

Staircase Scheduler

The Staircase scheduler was proposed by Con Kolivar, 2004 but none of its schedulers have been merged in the Kernel tree.

The goal of the staircase scheduler 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. An expired process will be put back into the staircase, but at the next lower rank. It can, thus, continue to run, but at a lower priority. When it exhausts another time slice, it moves down again. And so on. The following little table shows how long the process spends at each priority level:

	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	1

When a process reaches the end of the staircase (iteration 2), it gets the previous base priority -1 but 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.

https://lwn.net/Articles/87729/

The Completely Fair Scheduler (CFS)

From v2.6.23

The Completely Fair Scheduler has been merged in October 2007. This is since then the default Scheduler. The CFS 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.

The main idea behind the CFS is to maintain balance (**fairness**) in providing processor time to tasks. This means processes should be given a fair amount of the processor. When the time for tasks is out of balance (meaning that one or more tasks are not given a fair amount of time relative to others), then those out-of-balance tasks should be given time to execute.

https://developer.ibm.com/technologies/linux/tutorials/l-completely-fair-scheduler

The Completely Fair Scheduler (CFS)

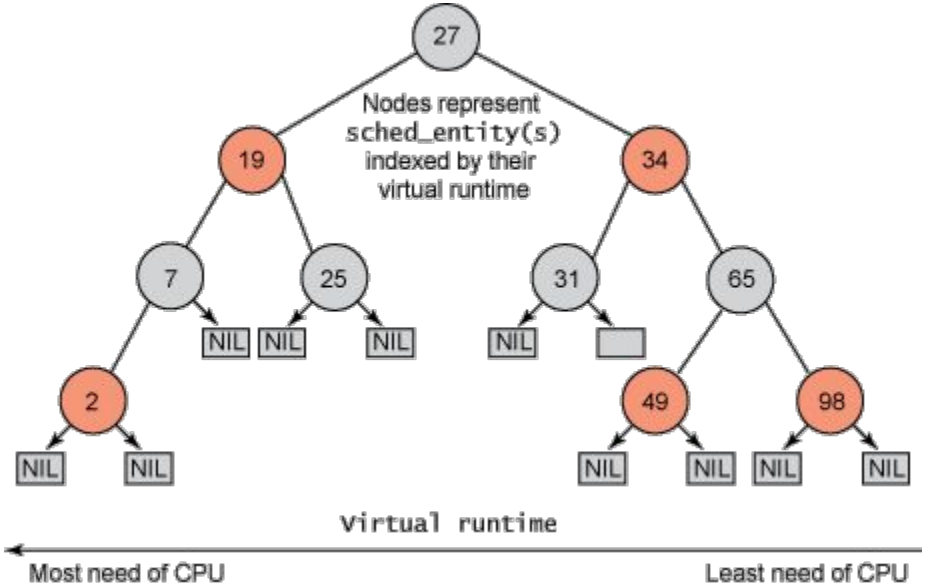
The Virtual Runtime

To determine the balance, the CFS maintains the amount of time provided to a given task in what's called the ***virtual runtime***. The smaller a task's virtual runtime - meaning the smaller amount of time a task has been permitted access to the processor - the higher its need for the processor. The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable (for example, waiting for I/O) receive a comparable share of the processor when they eventually need it.

But rather than maintain the tasks in a run queue, as has been done in prior Linux schedulers, the CFS **maintains a time-ordered red-black tree**. A red-black tree is a tree with a couple of interesting and useful properties. First, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other. Second, operations on the tree occur in $O(\log n)$ time (where n is the number of nodes in the tree). This means that you can insert or delete a task quickly and efficiently.

https://developer.ibm.com/technologies/linux/tutorials/l-completely-fair-scheduler

The Completely Fair Scheduler (CFS)



With tasks stored in the time-ordered red-black tree, tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the **left side** of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the **right side** of the tree. The scheduler then, to be fair, picks the left-most node of the red-black tree to schedule next to maintain fairness. The task accounts for its time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable. In this way, tasks on the left side of the tree are given time to execute, and the contents of the tree migrate from the right to the left to maintain fairness. Therefore, each runnable task chases the other to maintain a balance of execution across the set of runnable tasks.

https://developer.ibm.com/technologies/linux/tutorials/l-completely-fair-scheduler

The Completely Fair Scheduler (CFS)

Where have the priorities gone?

CFS doesn't use priorities directly but instead **uses them as a decay factor** for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay, where higher-priority tasks have lower factors of delay. This means that the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task. That's an elegant solution to avoid maintaining run queues per priority.

11.4

11. Scheduling

Context Switch

DIAG

Context Switch

V5.11

Context switch starts with the function in `kernel/sched/core.c`

```
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf)
```

The function does some checks on memory (*active_mm) and according to the situations performs some operation. Remember that a context switch can happen in the following cases.

```
4288          /*
4289           * kernel -> kernel   lazy + transfer active
4290           *   user -> kernel   lazy + mmgrab() active
4291           *
4292           * kernel ->   user   switch + mmdrop() active
4293           *   user ->   user   switch
4294           */
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/core.c#L4276

Context Switch

Then the function calls `switch_to()`.

```
4326            /* Here we just switch the register state and the stack. */
4327            switch_to(prev, next, prev);
4328            barrier();
4329
4330            return finish_task_switch(prev);
4331    }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/sched/core.c#L4276

The `switch_to()` is architecture-specific and mainly executes the following two tasks:

- TSS update
- CPU control registers update

```
47  #define switch_to(prev, next, last)                                     \
48  do {                                                                    \
49          ((last) = __switch_to_asm((prev), (next)));                     \
50  } while (0)
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/include/asm/switch_to.h#L47

Context Switch

switch_to()

As you can see:

- firstly the callee-saved register are pushed in the stack
- then the stack pointer is swapped
- in the end the callee-saved registers are popped from the stack (since they was pushed by the new process to be scheduled)

After these actions the control is passed to `__switch_to` that performs the switch of the FPU registers and of FS and GS registers.

But where the instruction pointer is set?

```
221  /*
222   * %rdi: prev task
223   * %rsi: next task
224   */
225  .pushsection .text, "ax"
226  SYM_FUNC_START(__switch_to_asm)
227          /*
228           * Save callee-saved registers
229           * This must match the order in inactive_task_frame
230           */
231          pushq   %rbp
232          pushq   %rbx
233          pushq   %r12
234          pushq   %r13
235          pushq   %r14
236          pushq   %r15
237
238          /* switch stack */
239          movq    %rsp, TASK_threadsp(%rdi)
240          movq    TASK_threadsp(%rsi), %rsp
241
242  #ifdef CONFIG_STACKPROTECTOR
243          movq    TASK_stack_canary(%rsi), %rbx
244          movq    %rbx, PER_CPU_VAR(fixed_percpu_data) + stack_canary_offset
245  #endif
246
247  #ifdef CONFIG_RETPOLINE
248          /*
249           * When switching from a shallower to a deeper call stack
250           * the RSB may either underflow or use entries populated
251           * with userspace addresses. On CPUs where those concerns
252           * exist, overwrite the RSB with entries which capture
253           * speculative execution to prevent attack.
254           */
255          FILL_RETURN_BUFFER %r12, RSB_CLEAR_LOOPS, X86_FEATURE_RSB_CTXSW
256  #endif
257
258          /* restore callee-saved registers */
259          popq    %r15
260          popq    %r14
261          popq    %r13
262          popq    %r12
263          popq    %rbx
264          popq    %rbp
265
266          jmp     __switch_to
267  SYM_FUNC_END(__switch_to_asm)
268  .popsection
```

Advanced Operating Systems and Virtualization

[11] Scheduling

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG