<Slides download>
http://www.pf.is.s.u-tokyo.ac.jp/class.html

Advanced Operating Systems

#5

Shinpei Kato
Associate Professor

Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

Course Plan

- Multi-core Resource Management
- Many-core Resource Management
- GPU Resource Management
- Virtual Machines
- Distributed File Systems
- High-performance Networking
- Memory Management
- Network on a Chip
- Embedded Real-time OS
- Device Drivers
- Linux Kernel

Schedule

1.	2018.9.28	Introduction + Linux Kernel (Kato)
2.	2018.10.5	Linux Kernel (Chishiro)
3.	2018.10.12	Linux Kernel (Kato)
4.	2018.10.19	Linux Kernel (Kato)
5.	2018.10.26	Linux Kernel (Kato)
6.	2018.11.2	Advanced Research (Chishiro)
7.	2018.11.9	Advanced Research (Chishiro)
8.	2018.11.16	(No Class)
9.	2018.11.23	(Holiday)
10.	2018.11.30	Advanced Research (Kato)
11.	2018.12.7	Advanced Research (Kato)
12.	2018.12.14	Advanced Research (Chishiro)
13.	2018.12.21	Linux Kernel
14.	2019.1.11	Linux Kernel
15.	2019.1.18	(No Class)
16.	2019.1.25	(No Class)

Process Scheduling

Abstracting Computing Resources

/* The case for Linux */

Acknowledgement:

Prof. Pierre Olivier, ECE 4984, Linux Kernel Programming, Virginia Tech

Outline

Outline

General information

Scheduling

- **Scheduler**: OS entity that decide which process should run, when, and for how long
- Multiplex processes in time on the processor: enables **multitasking**
  - Gives the user the illusion that processes are executing at the same time
- Scheduler is responsible for making the best use of the resource that is the CPU time
- Basic principle:
  - When in the system there are more ready-to-run processes than the number of cores
    - **The scheduler decides which process should run**

General information

Multitasking

- Single core: gives the illusion that multiple processes are running concurrently
- Multi-cores: enable true parallelism
- 2 types of multitasking OS:
  - **Cooperative multitasking**
    - A process does not stop running until it decides to do so (*yield* the CPU)
    - The operating system cannot enforce fair scheduling
      - For example in the case of a process that never yields
  - **Preemptive multitasking**
    - The OS can interrupt the execution of a process: *preemption*
    - Generally after the process expires its *timeslice*
    - And/or based on tasks priorities

General information

A bit of Linux scheduler history

- From v1.0 to v2.4: simple implementation
  - **But it did not scale to numerous processes and processors**
- V2.5 introduced the $O(1)$ scheduler
  - **Constant time scheduling decisions**
    - Scalability and execution time determinism
  - More info in [5]
  - Issues with latency-sensitive applications (Desktop computers)
- O(1) scheduler was replaced in 2.6.23 by what is still now the standard Linux scheduler:
  - **Completely Fair Scheduler (CFS)**
    - Evolution of the *Rotating Staircase Deadline* scheduler [2, 3]

General information

Scheduling policy - I/O vs compute-bound tasks

- Scheduling policy are the set of rules determining the choices made by a given model of scheduler
- **I/O-bound processes**:
  - Spend most of their time *waiting for I/O*: disk, network, but also keyboard, mouse, etc.
    - Filesystem, network intensive, GUI applications, etc.
    - Response time is important
  - Should run *often and for a small time frame*
- **Compute-bound processes**:
  - *Heavy use of the CPU*
    - SSH key generation, scientific computations, etc.
    - Caches stay hot when they run for a long time
  - Should *not run often, but for a long time*

General information

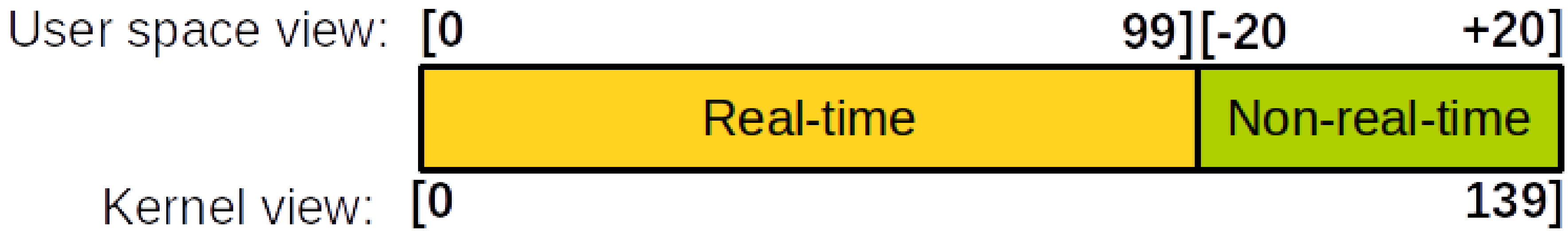
Scheduling policy - Priority

- **Priority**
  - Order process according to their ”importance” from the scheduler standpoint
  - A process with a higher priority will execute before a process with a lower one
- **Linux has 2 priority ranges**:
  - **Nice value**: ranges from -20 to +19, default is 0
    - High values of nice means lower priority
    - List process and their nice values with `ps ax -o pid,ni,cmd`
  - **Real-time priority**: range configurable (default 0 to 99)
    - Higher values mean higher priority
    - For processes labeled *real-time*
    - Real-time processes always executes before standard (nice) processes
    - List processes and their real-time priority using `ps ax -o pid,rtprio,cmd`

General information

Scheduling policy - Priority (2)

- User space to kernel priorities mapping:



General information

Scheduling policy - Timeslice

- **Timeslice** (quantum):
  - How much time a process should execute before being preempted
  - Defining the default timeslice in an absolute way is tricky:
    - Too long → bad interactive performance
    - Too short → high context switching overhead
- **Linux CFS does not use an absolute timeslice**
  - The timeslice a process receives is **function of the load of the system**
    - it is *a proportion* of the CPU
  - In addition, that timeslice is **weighted by the process priority**
- When a process *P* becomes runnable:
  - *P* will preempt the currently running process *C* is *P* consumed a smaller proportion of the CPU than *C*

General information

Scheduling policy - Policy application example

- 2 tasks in the system:
  - **Text editor**: I/O-bound, latency sensitive (interactive)
  - **Video encoder**: CPU-bound, background job
- Text editor:
  - A. *Needs a large amount of CPU time*
    - Does not need to run for long, but needs to have CPU time available whenever it needs to run
  - B. *When ready to run, needs to preempt the video encoder*
  - A + B = good interactive performance
- On a classical UNIX system, needs to set a correct combination of priority and timeslice
- Different with Linux: *the OS guarantee the text editor a specific proportion of the CPU time*

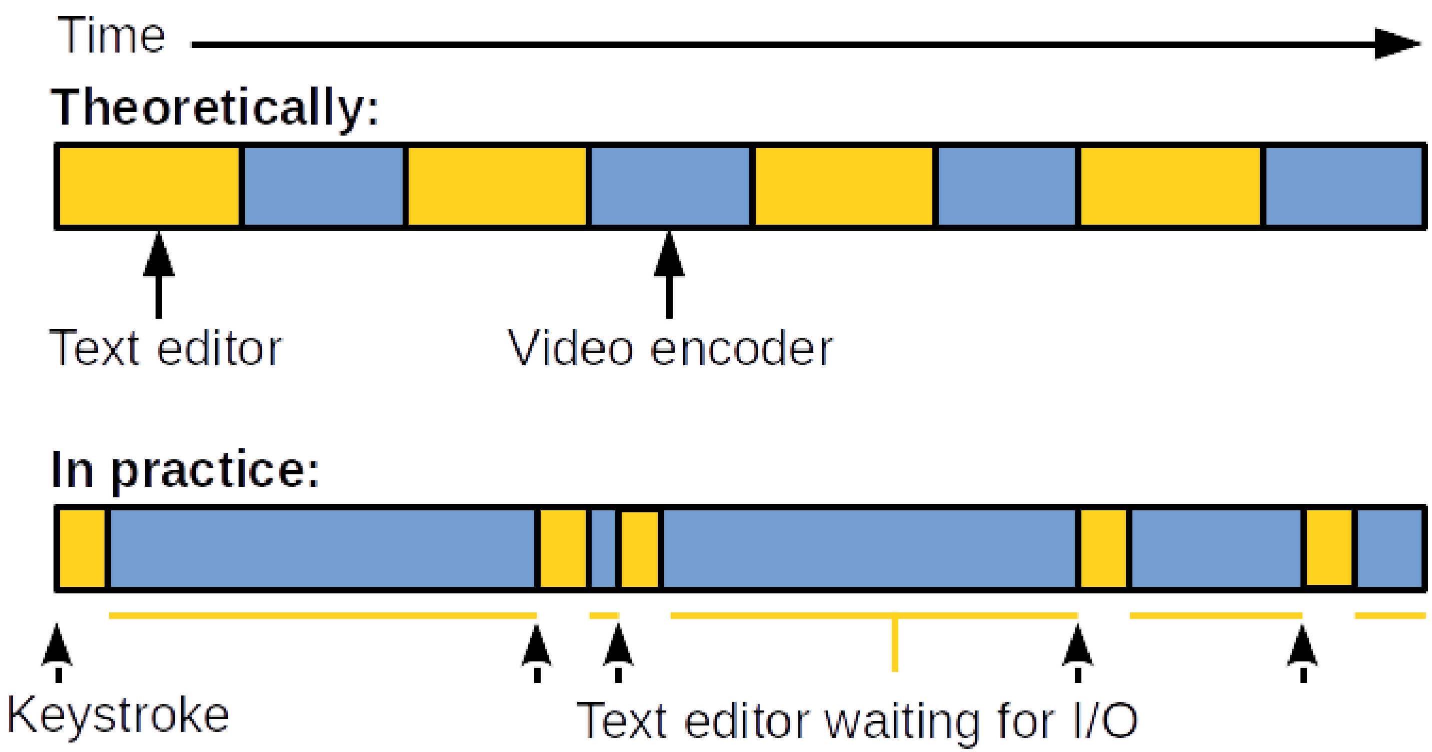
General information

Scheduling policy - Policy application example (2)

- Imagine only the two processes are present in the system and run at the same priority
  - Linux gives 50% of CPU time to each
- Considering an absolute timeframe:
  - **Text editor does not use fully its 50%** as it often blocks waiting for I/O
    - Keyboard key pressed
  - **CFS keeps track of the actual CPU time used by each program**
  - When the text editor wakes up:
    - CFS sees that **it actually used less CPU time than the video encoder**
    - Text editor preempts the video encoder

General information

Scheduling policy - Policy application example (3)



- Good interactive performance
- Good background, CPU-bound performance

Outline

Linux Completely Fair Scheduler

Scheduling classes

- **CPU classes**: coexisting CPU algorithms
  - Each task belongs to a class
- CFS: SCHED_OTHER, implemented in `kernel/sched/fair.c`
- Real-time classes: SCHED_RR, SCHED_FIFO, SCHED_DEADLINE
  - For predictable schedule
- `sched_class` datastructure:

```
struct sched_class {
  void (*enqueue_task) (/* ... */);
  void (*dequeue_task) (/* ... */);
  void (*yield_task) (/* ... */);
  void (*check_preempt_curr) (/* ... */);
  struct task_struct * (*pick_next_task) (/* ... */);
  void (*set_cur_task) (/* ...*/);
  void (*task_tick) (/* ... */);
  /* ... */
}
```

Linux Completely Fair Scheduler

sched_class hooks

- Functions descriptions:
  - enqueue_task(...)
    - Called when a task enters a runnable state
  - dequeue_task(...)
    - Called when a task becomes unrunnable
  - yield_task(...)
    - Yield the processor (dequeue then enqueue back immediatly)
  - check_preempt_curr(...)
    - Checks if a task that entered the runnable state should preempt the currently running task
  - pick_next_task(...)
    - Chooses the next task to run
  - set_curr_task(...)
    - Called when the currentluy running task changes its scheduling class or task group to the related scheduler
  - task_tick(...)
    - Called regularly (default: 10 ms) from the system timer tick handler, might lead to context switch

Linux Completely Fair Scheduler

Unix scheduling

- Classical UNIX systems *map priorities (nice values) to absolute timeslices*
- Leads to several issues:
  - **What is the absolute timeslice that should be mapped to a given nice value?**
    - Sub-optimal switching behavior for low priority processes (small timeslices)
  - **Relative nice values and their mapping to timeslices**
    - Nicing down a process by one can have very different effects according to the tasks priorities
  - **Timeslice must be some integer multiple of the timer tick**
    - Minimum timeslice and difference between two consecutive timeslices are bounded by the timer tick frequency

Linux Completely Fair Scheduler

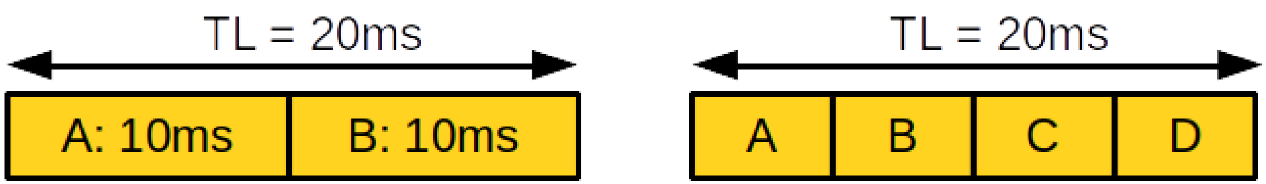
Fair scheduling

- **Perfect multitasking**:
  - From a single core standpoint
    - At each moment, each process of the same priority has received an exact amount of the CPU time
    - What we would get if we could *run n tasks in parallel on the CPU while giving them 1/n of the CPU processing power* → not possible in reality
    - Or if we could *schedule tasks for infinitely small amounts of time* → context switch overhead issue
- **3 main (high-level) CFS concepts:**
  1. CFS runs a process for some times, then swaps it for the runnable process that has run the least
  2. No default timeslice, CFS calculates how long a process should run according to the number of runnable processes
  3. That dynamic timeslice is weighted by the process priority (nice)

Linux Completely Fair Scheduler

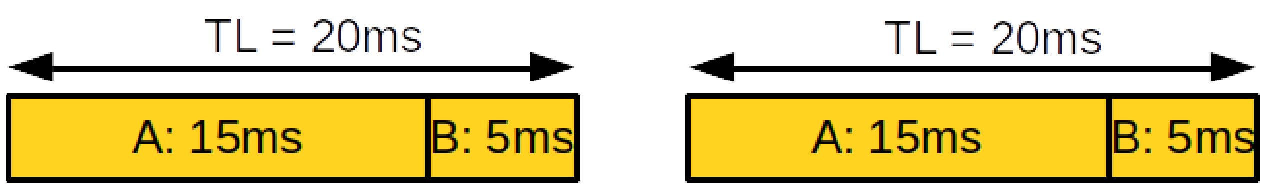
Fair scheduling (2)

- *Targeted latency*: period during which all runnable processes should be scheduled at least once
- Example: processes with the same priority



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

- Example: processes with different priorities



($A_{nice} = 0$; $B_{nice} = 5$) ($A_{nice} = 10$; $B_{nice} = 15$)

- *Minimum granularity*: floor at 1 ms (default)

Outline

CFS implementation

- **4 main components:**
  1. Time accounting
  2. Process selection
  3. Scheduler entry point (calling the scheduler)
  4. Sleeping & waking up

CFS implementation

Time accounting

- sched_entity structure in the task struct (se field)

```
struct sched_entity
{
  struct load_weight  load;
  struct rb_node      run_node;
  struct list_head  group_node;
  unsigned int      on_rq;

  u64      exec_start;
  u64      sum_exec_runtime;
  u64      vruntime;
  u64      prev_sum_exec_runtime;

  /* additional statistics not shown here */
}
```

- **Virtual runtime**
  - How much time a process has been executed (ns)

CFS implementation

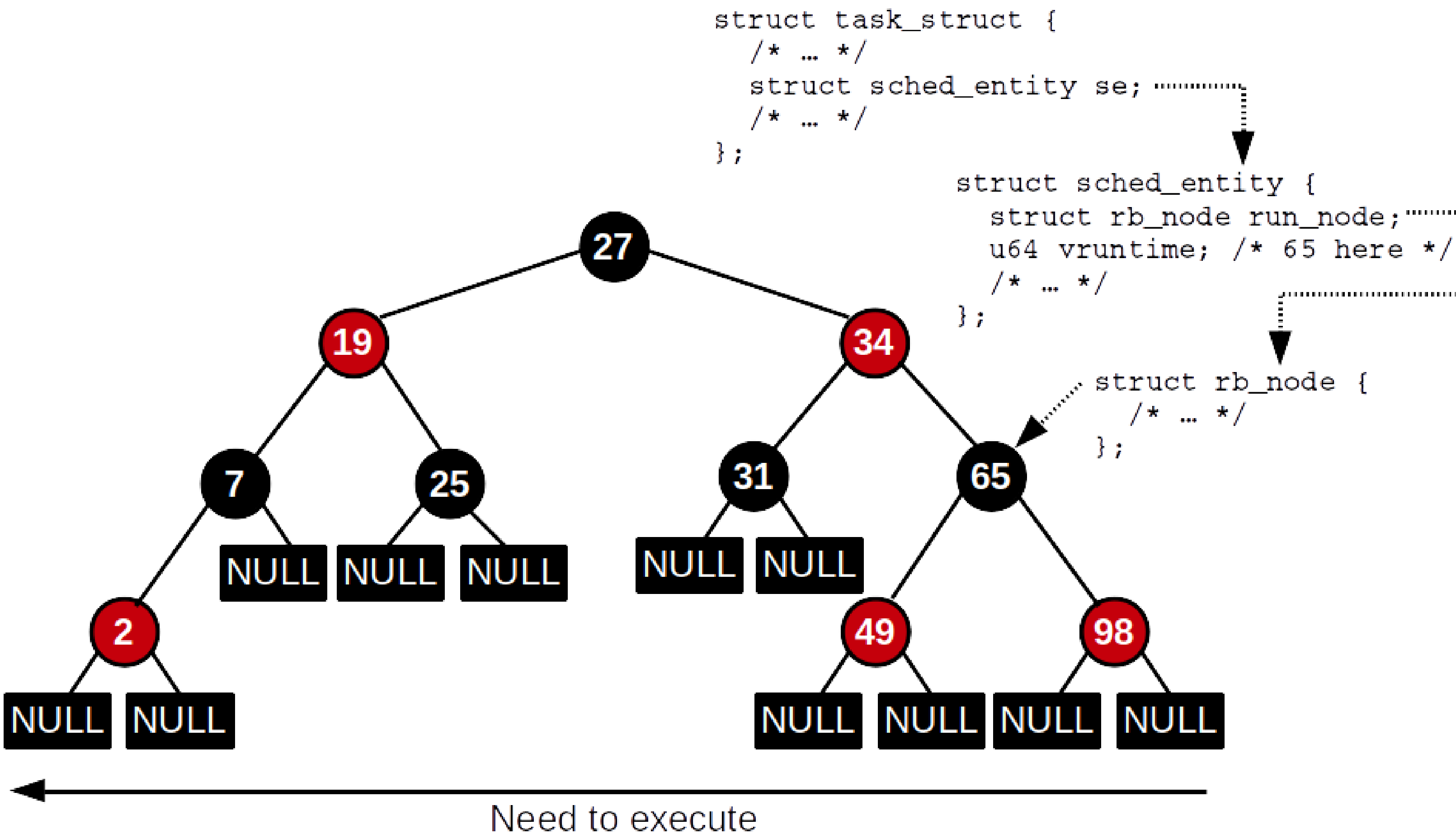
Time accounting (2)

```
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;
  if (unlikely((s64)delta_exec <= 0))
    return;

  curr->exec_start = now;

  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);

  curr->vruntime += calc_delta_fair(
      delta_exec, curr);
  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);
}
```

- Invoked regularly by the system timer, and when a process becomes runnable/unrunnable

CFS implementation

Process selection



- Adapted from [1]

CFS implementation

Process selection (2)

- When CFS needs to choose which runnable process to run next:
  - **The process with the smallest `vruntime` is selected**
  - It is the leftmost node in the tree

```
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
  struct rb_node *left = cfs_rq->rb_leftmost;

  if (!left)
    return NULL;

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

CFS implementation

Process selection: adding a process to the tree

- A process is added through `enqueue_entity`:

```
static 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;

  if (renorm && curr)
    se->vruntime += cfs_rq->min_vruntime;

  update_curr(cfs_rq);

  if (renorm && !curr)
    se->vruntime += cfs_rq->min_vruntime;

  update_load_avg(se, UPDATE_TG);
  enqueue_entity_load_avg(cfs_rq, se);
  account_entity_enqueue(cfs_rq, se);
  update_cfs_shares(cfs_rq);
```

```
  if (flags & ENQUEUE_WAKEUP)
    place_entity(cfs_rq, se, 0);

  check_schedstat_required();
  update_stats_enqueue(cfs_rq, se, flags);
  check_spread(cfs_rq, se);
  if (!curr)
    __enqueue_entity(cfs_rq, se);
  se->on_rq = 1;

  if (cfs_rq->nr_running == 1) {
    list_add_leaf_cfs_rq(cfs_rq);
    check_enqueue_throttle(cfs_rq);
  }
}
```

CFS implementation

Process selection: adding a process to the tree (2)

- __enqueue_entity:

```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);
    /*
     *Wedont care about collisions. Nodes with
     *the same key stay together.
     */
    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);
}
```

CFS implementation

Process selection: removing a process from the tree

- dequeue_entity:

```
static void
dequeue_entity(struct cfs_rq *cfs_rq,
      struct sched_entity *se, int flags)
{
    update_curr(cfs_rq);
    dequeue_entity_load_avg(cfs_rq, se);

    update_stats_dequeue(cfs_rq, se, flags
      );

    clear_buddies(cfs_rq, se);

    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);
```

```
    if (!(flags &DEQUEUE_SLEEP))
        se->vruntime -= cfs_rq->
      min_vruntime;

    return_cfs_rq_runtime(cfs_rq);

    update_cfs_shares(cfs_rq);

    if ((flags &(DEQUEUE_SAVE |
      DEQUEUE_MOVE)) == DEQUEUE_SAVE)
        update_min_vruntime(cfs_rq);
}
```

CFS implementation

Process selection: removing a process from the tree (2)

- __dequeue_entity:

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    if (cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;

        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }

    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

CFS implementation

Entry point: `schedule()`

- The kernel calls **`schedule()`** anytime it wants to invoke the scheduler
  - Calls `pick_next_task()`

```c
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct pin_cookie cookie)
{
  const struct sched_class *class = &fair_sched_class;
  struct task_struct *p;

  if (likely(prev->sched_class == class && rq->nr_running == rq->cfs.h_nr_running)) {
    p = fair_sched_class.pick_next_task(rq, prev, cookie);
    if (unlikely(p == RETRY_TASK))
      goto again;

    if (unlikely(!p))
      p = idle_sched_class.pick_next_task(rq, prev, cookie);
    return p;
  }
again:
  for_each_class(class) {
    p = class->pick_next_task(rq, prev, cookie);
    if (p) {
      if (unlikely(p == RETRY_TASK))
        goto again;
      return p;
    }
  }

  BUG(); /* the idle class will always have a runnable task */
}
```

CFS implementation

Sleeping and waking up

- Multiple reasons for a task to sleep:
  - Specified amount of time, waiting for I/O, blocking on a mutex, etc.
- Going to sleep - steps:
  1. Task marks itself as sleeping
  2. Task enters a *waitqueue*
  3. Task leaves the rbtree of runnable processes
  4. Task calls `schedule()` to select a new process to run
- Inverse steps for waking up
- Two states associated with sleeping:
  - TASK_INTERRUPTIBLE
    - Will be awaken on signal reception
  - TASK_UNINTERRUPTIBLE
    - Ignore signals

CFS implementation

Sleeping and waking up: wait queues

- **Wait queue**:
  - List of processes waiting for an event to occur

```
typedef struct _wait_queue_head
      wait_queue_head_t
struct  wait_queue_head {
  spinlock_t lock;
  struct  list_head task_list;}
```

- Some simple interfaces used to go to sleep have races:
  - It is possible to go to sleep *after* the event we are waiting for has occurred
  - Recommended way:

```
/* We assume the wait queue we want to wait on is accessible through a variable q */

DEFINE_WAIT(wait); /* initialize a wait queue entry */

add_wait_queue(q, &wait);
while (!condition) { /* event we are waiting for */
  prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
  if(signal_pending(current))
     /* handle signal */
  schedule();
}
finish_wait(&q, &wait);
```

CFS implementation

Sleeping and waking up: wait queues (2)

- **Steps for waiting on a waitqueue:**
  1. Create a waitqueue entry (DEFINE_WAIT())
  2. Add the calling process to a wait queue (add_wait_queue())
  3. Call prepare_to_wait() to change the process state
  4. If the state is TASK_INTERRUPTIBLE, a signal can wake the task up → need to check
  5. Executes another process with schedule()
  6. When the task awakens, check the condition
  7. When the condition is true, get out of the wait queue and set the state accordingly using finish_wait()

CFS implementation

Sleeping and waking up: wake_up()

- **Waking up** is taken care of by wake_up()
  - **Awakes all the processes on a waitqueue** by default

```
#define wake_up(x) __wake_up(x, TASK_NORMAL, 1, NULL)
/* type of x is wait_queue_head_t */
```

- __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) &&(flags
        & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
      break; /* wakes up only a subset of 'exclusive' tasks */
  }
}
```

- Exclusive tasks are added through prepare_to_wait_exclusive()

CFS implementation

Sleeping and waking up: `wake_up()` (2)

- A wait queue entry contains a pointer to a wake-up function
  - `include/linux/wait.h`:

```
typedefstruct_wait_queue wait_queue_t;
typedefint (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode, int flags, void *key);
int default_wake_function(*wait_queue_func_t)(wait_queue_t *wait, unsigned mode,
                                               int flags, void *key);

/* ... */

struct_wait_queue {
  /* ... */
  wait_queue_func_t func;
  /* ... */
}
```

- `default_wake_function()` calls `try_to_wake_up()` ...
  - ... which calls `ttwu_queue()` ...
  - ... which calls `ttwu_do_activate()` (put the task back on runqueue) ...
  - ... which calls `ttwu_do_wakeup` ...
  - ... which **sets the task state to `TASK_RUNNING`**

CFS implementation

CFS on multicores: brief, high-level overview

- **Per-CPU runqueues** (rbtrees)
  - To avoid costly accesses to shared data structures
- **Runqueues must be kept balanced**
  - Ex: dual-core with one large runqueue of high-priority processes, and a small one with low-priority processes
    - High-priority processes get less CPU time than low-priority ones
  - A load balancing algorithm is run periodically
    - Balances the queues based on processes priorities and their actual CPU usage
- More info: [6]

Outline

Preemption and context switching

Context switch

- A **context switch** is the action of swapping the process currently running on the CPU to another one
  - **Performed by the `context_switch()` function**
    - Called by `schedule()`

    1. Switch the address space through `switch_mm()`
    2. Switch the CPU state (registers) through `switch_to()`
- A task can voluntarily relinquish the CPU by calling `schedule()`
  - **But when does the kernel check if there is a need of preemption?**
    - `need_resched` flag (per-process, in the `thread_info` of `current`)
  - `need_resched` is set by:
    1. `scheduler_tick()` when the currently running task needs to be preempted
    2. `try_to_wake_up()` when a process with higher priority wakes up

Preemption and context switching

`need_resched`, user preemption

- The `need_resched` flag is checked:
  1. Upon returning to user space (from a syscall or an interrupt)
  2. Upon returning from an interrupt
- If the flag is set, `schedule()` is called
- **User preemption happens:**
  1. When returning to user space from a syscall
  2. When returning to user space from an interrupt
- With Linux, **the kernel is also subject to preemption**

Preemption and context switching

Kernel preemption

- In most of Unix-like, kernel code is non-preemptive:
  - It runs until it finishes
- **Linux kernel code is preemptive**
  - A task can be preempted in the kernel as long as execution is **in a safe state**
    - *Not holding any lock* (kernel is SMP safe)
- `preempt_count` in the `thread_info` structure
  - Indicates the current lock depth
- If `need_resched && !preempt_count` → safe to preempt
  - Checked when returning to the kernel from interrupt
  - `need_resched` is also checked when releasing a lock and `preempt_count` is 0
- Kernel code can also call directly `schedule()`

Preemption and context switching

Kernel preemption (2)

- **Kernel preemption can occur:**
  1. On return from interrupt to kernel space
  2. When kernel code becomes preemptible again
  3. If a task explicitly calls `schedule()` from the kernel
  4. If a task in the kernel blocks (ex: mutex, result in a call to `schedule()`)

Outline

Real-time scheduling policies

`SCHED_FIFO` and `SCHED_RR`

- *Soft real-time* scheduling classes:
  - Best effort, no guarantees
- **Real-time task of any scheduling class will always run before non-real time ones** (CFS, **`SCHED_OTHER`**)
  - `schedule()` → `pick_next_task()` → `for_each_class()`
- 2 ”classical” RT scheduling policies (`kernel/sched/rt.c`):
  - **`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

Other scheduling policies

- **`SCHED_DEADLINE`**:
  - Real-time policies mainlined in v3.14 enabling *predictable RT scheduling*
  - EDF implementation based on a period of activation and a worst case execution time (WCET) for each task
  - More info: `Documentation/sched-deadline.txt`, [4], etc.
- **`SCHED_BATCH`**: non-real-time, low priority background jobs
- **`SCHED_IDLE`**: non-real-time, *very* low priority background jobs

Outline

Scheduling-related syscalls

Scheduling syscalls list

- 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 syscalls

Usage example

```c
#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;
/* GetthePIDofthecallingprocess        */
  pid = getpid();
  printf("Mypidis:%d¥n", pid);

  /* Gettheschedulingclass    */
  ret = sched_getscheduler(pid);
  if(ret == -1)
    handle_err(ret,"sched_getscheduler");
  printf("sched_getschedulerreturns:"
    "%d¥n", ret);
  assert(ret == SCHED_OTHER);

  /* Getthepriority(nice/RT)    */
  sp.sched_priority = -1;
  ret = sched_getparam(pid, &sp);
  if(ret == -1)
    handle_err(ret,"sched_getparam");
  printf("Mypriorityis:%d¥n",
    sp.sched_priority);

  /* Setthepriority(nicevalue)      */
  ret = nice(1);
  if(ret == -1)
    handle_err(ret,"nice");
```

Scheduling-related syscalls

Usage example (2)

```c
/* Getthepriority  */
sp.sched_priority = -1;
ret = sched_getparam(pid, &sp);
if(ret == -1)
 handle_err(ret,"sched_getparam");
printf("Mypriorityis:%d¥n",
 sp.sched_priority);

/* SwitchschedulingclasstoFIFOand
 * thepriorityto99 */
sp.sched_priority = 99;
ret = sched_setscheduler(pid,
 SCHED_FIFO, &sp);
if(ret == -1)
    handle_err(ret,"sched_setscheduler");

/* Gettheschedulingclass     */
ret = sched_getscheduler(pid);
if(ret == -1)
    handle_err(ret,"sched_getscheduler");
printf("sched_getschedulerreturns:"
 "%d¥n", ret);
assert(ret == SCHED_FIFO);
```

```c
/* Getthepriority  */
sp.sched_priority = -1;
ret = sched_getparam(pid, &sp);
if(ret == -1)
   handle_err(ret,"sched_getparam");
printf("Mypriorityis:%d¥n",
 sp.sched_priority);

/* SettheRTpriority          */
sp.sched_priority = 42;
ret = sched_setparam(pid, &sp);
if(ret == -1)
   handle_err(ret,"sched_setparam");
printf("Prioritychangedto%d¥n",
 sp.sched_priority);

/* Getthepriority  */
sp.sched_priority = -1;
ret = sched_getparam(pid, &sp);
if(ret == -1)
   handle_err(ret,"sched_getparam");
printf("Mypriorityis:%d¥n",
 sp.sched_priority);
```

Scheduling-related syscalls

Usage example (2)

```c
/* GetthemaxpriorityvalueforSCHED_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("MaxRRprio:%d¥n", max_rr_prio);

/* GettheminpriorityvalueforSCHED_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("MinRRprio:%d¥n", min_rr_prio);

cpu_set_size = sizeof(cpu_set_t);
CPU_ZERO(&cs);  /* clearthemask   */
CPU_SET(0, &cs);
CPU_SET(1, &cs);
/* SettheaffinitytoCPUs0and1only        */
ret = sched_setaffinity(pid, cpu_set_size, &cs);
if(ret == -1)
  handle_err(ret,"sched_setaffinity");
```

```c
/* GettheCPUaffinity    */
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("AffinitytestsOK¥n");

/* YieldtheCPU   */
ret = sched_yield();
if(ret == -1)
  handle_err(ret,
    "sched_yield");

return EXIT_SUCCESS;
}
```

Additional documentation

- **CFS:**
  - http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/
  - http://elinux.org/images/d/dc/Elc2013_Na.pdf
- **Linux scheduling:**
  - https://www.cs.columbia.edu/smb/classes/s06-4118/l13.pdf

BibliographyI

1 Inside the linux 2.6 completely fair scheduler.
https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/.
Accessed: 2017-02-01.

2 The rotating staircase deadline scheduler.
https://lwn.net/Articles/224865/.
Accessed: 2017-01-29.

3 Rsdl completely fair starvation free interactive cpu scheduler.
https://lwn.net/Articles/224654/.
Accessed: 2017-01-29.

4 Sched deadline: a status update.
http://events.linuxfoundation.org/sites/events/files/slides/SCHED_DEADLINE-20160404.pdf.
Accessed: 2017-02-06.

5 Understanding the linux 2.6.8.1 cpu scheduler.
https://web.archive.org/web/20131231085709/http://joshaas.net/linux/linux_cpu_scheduler.pdf.
Accessed: 2017-01-28.

[6] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A.
The linux scheduler: A decade of wasted cores.
In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16, ACM, pp. 1:1–1:16.