Periodic scheduler for Linux OS

Mike Athanasakis – michath@csd.uoc.gr

Operating System TA – Winter 2015-2016

History

- Linux v1.2 – Round Robin
- Linux v2.2 – Scheduling Classes & Policies
- Linux v2.4 – Division in epochs, goodness of function
- Linux v2.6 – Runqueue O(1)
- Linux v2.6.21 – Completely Fair Scheduler (CFS)
  - Virtual time concept
  - Time-ordered red-black tree instead of queue
  - Maintains balance in providing processor time to tasks

Scheduling classes

- Linux scheduler at kernel/sched.c
  - It is modular, depending the type of task it changes scheduling algorithm.
  - It uses the idea of scheduling class.
  - Each task belongs to a scheduling class, that changes the way it gets scheduled.

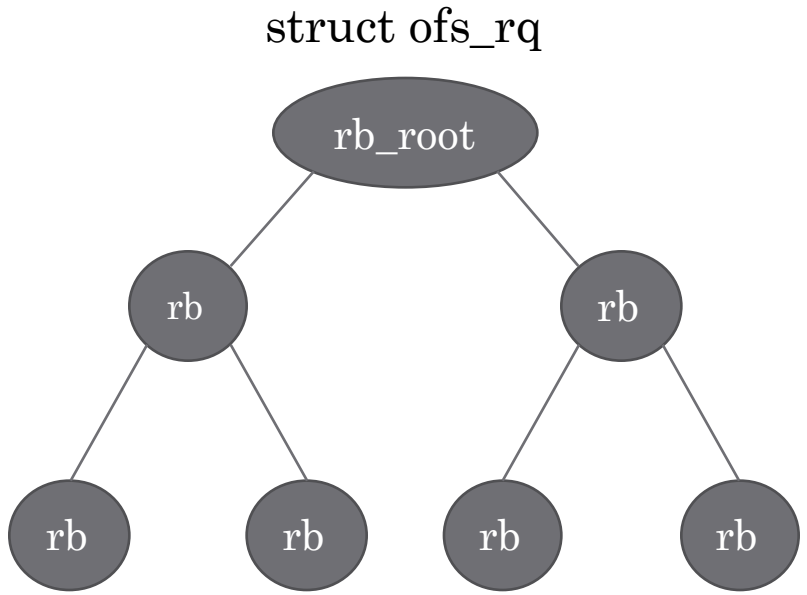
- sched.c calls an “overloaded” function that depending the scheduling class it calls different code

Task hierarchy in CFS

```
struct task_struct {
        …
        struct sched_entity se;
        …
}
struct sched_entity {
        …
        struct rb_node run_node;
        …
}

struct rb_node {
        …
        rb_node *right, left;
        …
}
```

Red-black tree to hold tasks



Scheduler and policies

- Scheduling policy is set by sched_setscheduler()
- Available scheduling policies
  - SCHED_FIFO – Special time-critical tasks
  - SCHED_RR – Round robin scheduling
  - SCHED_IDLE – Low priority tasks
  - SCHED_OTHER – Default linux task (normal)
  - SCHED_BATCH – CPU intensive tasks

Scheduling policies and their files

- Completely fair scheduler (SCHED_OTHER)
  - kernel/sched_fair.c
- Real time processes (SCHED_FIFO & SCHED_RR)
  - kernel/sched_rt.c
- Idle tasks (SCHED_IDLE)
  - kernel/sched_idle.c

Scheduling state of task

- Defined at /include/linux/sched.h
  - TASK_RUNNING 0
  - TASK_INTERRUPTIBLE 1
  - TASK_UNINTERRUPTIBLE 2
  - TASK_ZOMBIE 3
  - TASK_STOPPED 4

Maybe you can add a new task state?
Maybe TASK_PERIODIC?

Runqueue

the magic starts here

- Defined at kernel/sched.c is the main scheduling struct of Linux.

```
struct runqueue {
        …
        struct task_struct      *curr;          currently running task
        struct prio_array       *active;        active priority array
        struct prio_array       *expired;       expired priority array
        struct prio_array       arrays[2];      actual priority arrays
        …
}
```

Runqueue functions

- Called inside main schedule at kernel/sched.c
  - cpu_rq(processor) – returns CPU's runqueue
  - this_rq() – returns runqueue of current CPU
  - task_rq(task) – returns the runqueue where the task is in

void schedule(void); 1/2

- Located at kernel/sched.c it is the main scheduling function.

```
void asmlinkage __sched schedule (void) {
        struct task_struct *prev, *next;        previous & next task
        struct rq *rq;
        ...
        rq = cpu_rq(cpu);
        prev = rq->curr;                        current task will become the
                                                previous after the context switch


        pre_schedule (rq, prev);                depending the scheduling class
                                                the code to run changes
```

void schedule(void); 2/2

put_prev_task(rq, prev);	scheduling class dependent code
next = pick_next_task(rq);	the function that chooses the next task
...	
context_switch(rq, prev, next);	the actual context switch
…	
post_schedule(rq);	depending the scheduling class the code to run changes

pick_next_task(rq)

```
if (likely(rq->nr_running  == rq->cfs.nr_running)) {
        p = fair_sched_class.pick_next_task(rq);
        if (likely(p))
                return p;

}


for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
                return p;
}
```

struct sched_class

- Located at include/linux/sched.h
- How to handle enqueue/dequeue of a specific sched_class

```
void (*enqueue_task) (struct rq, struct task_struct, int flags);
void (*dequeue_task) (struct rq, struct task_struct, int flags);
```

- During the context switch how to handle the sched_class

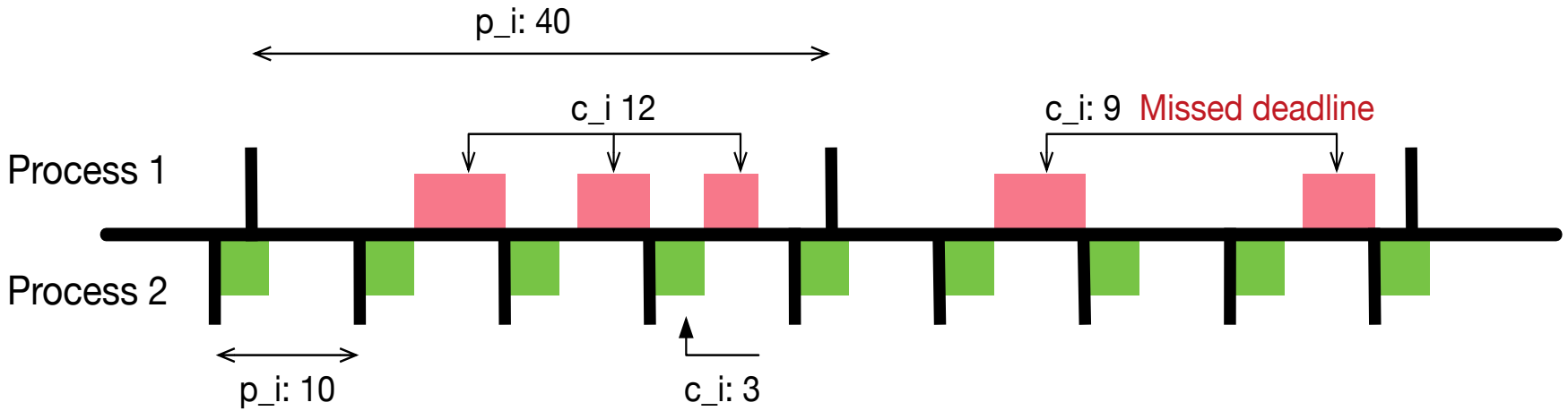
```
struct task_struct * (*pick_next_task) (struct rq *rq);
void (*put_prev_task) (struct rq *rq, struct task_struct *p);
```

Assignment 4
A periodic scheduler with a short period first

- Each process has a period_time (p_i) and a computation time (c_i) in milliseconds.
- Each task has to run **exactly c_i** time every **p_i** time.
- If a task doesn't run c_i time every p_i then we say it missed a deadline.
- We choose what periodic process to run first by choosing the one with the smallest period time (shortest period first).
- Remember that normal Linux schedule quantum is 100ms.

Periodic tasks example

Process 1: p_i: 40 seconds
c_i: 12 seconds
Process 2: p_i: 10 seconds
c_i: 3 seconds



How to test

- Create a simple test program that takes as argument the p_i and c_i
- Run a 1st task test instance with p_i/c_i: 1000 / 200
- Run a 2nd task test instance with p_i/c_i: 2000 / 500
- Run a 3rd task test instance with p_i/c_i: 1500 / 400
- And so on… the tasks should start miss deadlines!

- Get creative on how to test it, it will score you points!

More help? Info? Deliverables?

- Just check the assignment pdf. It has much more text than it shows.
- If you need more help read the links, they have a lot of info that can make this assignment much easier.

- **This task is like a real problem out there**
  - Study the problem and design the solution.
  - Implement your solution and test it as much as you can.
  - Submit even the smallest piece of code to show your effort!