Overview of the Linux Scheduler Framework

WORKSHOP ON REAL-TIME SCHEDULING IN THE LINUX KERNEL

Pisa, June 27th, 2014

Marco Cesati

University of Rome Tor Vergata

Goals of the Linux scheduler

Select the "right" task to run

- on each available CPU
- when a task terminates, blocks, or becomes runnable
- and when requested by the time-sharing policy

Goals of the Linux scheduler (2)

Select the “right” task to run in such a way to

- be “fair” to tasks and users
- respect relative priorities among tasks
- minimize task response times
- maximize system throughput (tasks completed per unit time)
- balance load among CPUs
- minimize power consumption
- have small overhead
- degrade gracefully on highest loads

Why scheduling is so difficult, anyway?

- We cannot optimize for any given goal
  - fairness vs priority
  - throughput vs response time
  - load balancing vs power consumption

Many scheduling algorithms rely on **heuristic rules**

Practical definition of heuristic rule

"This is black magic, please don't ask, it's not your business anyway!"

One Sched to rule them all!

The Linux scheduler must achieve good results in

- High-performance clusters, mainframes
- High-end servers
- Desktop computers
- Laptops, tablets, smartphones
- Embedded devices

Any change of the scheduler must not impair performance

- for each class of machines
- with any realistic usage pattern

Tasks

A ***task*** is an execution flow that can be scheduled on a CPU

In Linux:

- “Task” is equivalent to “Process”
- A multi-threaded program runs as a group of related light-weight processes
- Basic data structure for a task: `struct task_struct`
  - Its address is the main “task identifier” inside the kernel
  - Includes hundreds of fields and pointers to other task-related data structures
  - `current` identifies the task in execution

Unix tasks

- Unix “normal” tasks
  - Arbitrary execution times
  - Relative priorities
  - Starvation-free time-sharing scheduling algorithms
- POSIX “real-time” tasks
  - Arbitrary execution times
  - Absolute priorities
  - Round-robin or FIFO scheduling algorithms for tasks of equal priority

Real-time tasks

- Real-time tasks
  - Deadlines on completion times
  - Absolute priorities based on temporal parameters
  - Usually periodic or sporadic tasks
  - Dedicated priority-based scheduling algorithms (EDF, RM, ...)

Since version 3.14, Linux supports native real-time tasks

There is an on-going global effort to make Linux usable in real-time systems (see also RTAI, `CONFIG_PREEMPT_RT` patch, ...)

Current scheduler framework

- Introduced by Ingo Molnar in kernel 2.6.23 (2007)
- Based on different "scheduling policies" included in "scheduling classes"
- Original design assumptions:
  - Partitioned scheduler (decisions are local to each CPU)
  - If necessary, tasks can be migrated among the CPUs
  - Each scheduling class has a unique priority
  - On some CPU, no task of a given class can be run if tasks of classes with higher priorities are runnable
- Real-time policies break some of these assumptions

Scheduling classes and policies

Class	Policy
stop_sched_class	
dl_sched_class	SCHED_DEADLINE
rt_sched_class	SCHED_FIFO SCHED_RR
fair_sched_class	SCHED_NORMAL SCHED_BATCH SCHED_IDLE
idle_sched_class	

Class `stop_sched_class`

- Highest priority scheduling class
- Mechanism to force running a function on some CPU(s) by preempting and freezing any other task
- Not associated with a scheduling policy
  - just one kernel thread per CPU (`migration/`*N*)
- Used by task migration, RCU, CPU unplugging, ...

Class `dl_sched_class`

- By Dario Faggioli & Juri Lelli, kernel version 3.14 (Nov. 2013)
- Scheduling policy `SCHED_DEADLINE`
- Highest priority tasks in the system

Further details in another talk!

Class `rt_sched_class`

- POSIX “real-time” tasks
- 99 absolute priorities
- Scheduling policies for tasks of equal priority:
  - `SCHED_FIFO`: cooperative scheduling, First Come First Served
  - `SCHED_RR`: round-robin scheduling, 100 ms timeslice by default

Class `fair_sched_class`

- By Ingo Molnar, kernel version 2.6.23 (July 2007)
- Completely Fair Scheduler (CFS)
- Three scheduling policies:
  - `SCHED_NORMAL`: normal "Unix" tasks
  - `SCHED_BATCH`: batch (non-interactive) tasks
  - `SCHED_IDLE`: low-priority tasks

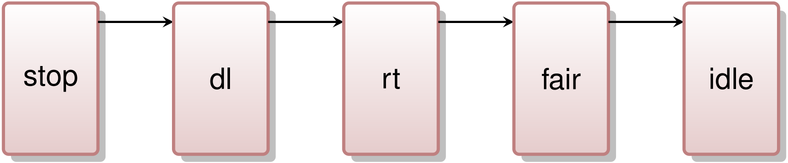
Further details in another talk!

Class `idle_sched_class`

- Lowest priority scheduling class
- Not associated with a scheduling policy
  - just one task per CPU ("idle" thread `swapper/`N)
- Executed only when no other task is runnable

The scheduling class

- Implemented by `struct sched_class`
- Basically a collection of function pointers (methods)
- Each structure is linked to the next lower class in the hierarchy by a `next` pointer
    - the `for_each_class` macro iterates over all classes



Main methods of the scheduling class

- `enqueue_task`: invoked when a task becomes runnable
- `dequeue_task`: invoked when a task is no longer runnable
- `pick_next_task`: select the best task to be run next (scheduling decision)
  - return `NULL` if no runnable task was found
  - return `RETRY_TASK` if a runnable task appeared in a higher-priority class
- `check_preempt_curr`: check if the new runnable task should preempt the current one
- `task_tick`: periodically invoked to update task parameters (for time sharing)

The schedule() function

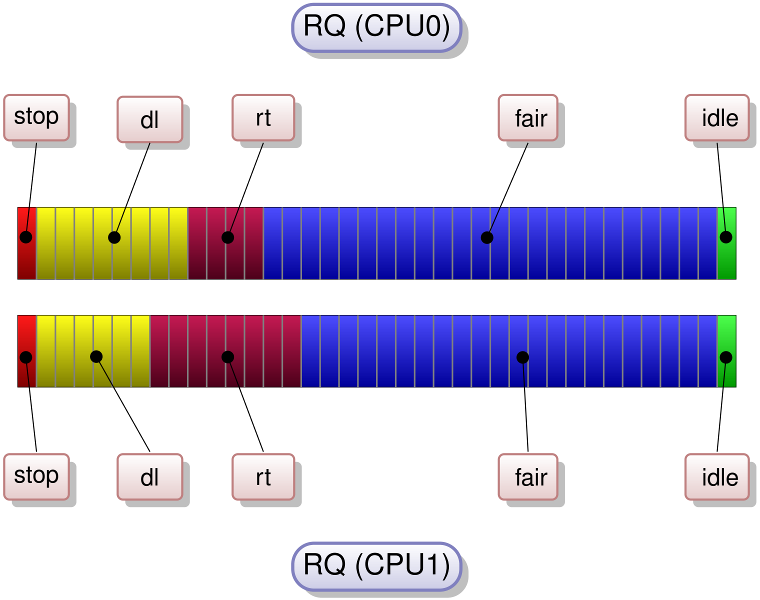
- Executed on a CPU to select the best task to be run next
- `schedule()` picks the best runnable task starting from the highest-priority scheduler class

```
again:
  for_each_class(class) {
    p = class->pick_next_task(rq, prev);
    if (p) {
      if (p == RETRY_TASK)
        goto again;
      return p;
    }
  }
```

Runqueues

- Runnable tasks are included in a runqueue: $\boxed{\texttt{struct rq}}$
- There is a runqueue for each CPU in the system
- Lots of information available here:
  - number of runnable tasks in queue
  - current and past CPU load
  - information specific to each scheduling class
    - also lists or trees collecting tasks
  - various statistics and accounting information

Example: two runqueues



Reserved CPU bandwidth

- `SCHED_DEADLINE`, `SCHED_FIFO`, and `SCHED_RR` tasks may easily starve "normal" tasks
- To make life easier for RT developers, some CPU bandwidth can be reserved to "normal" tasks:

> In an interval of time of length P μs, at least N μs are reserved for "normal" tasks

- $P \longleftarrow$ `/proc/sys/kernel/sched_rt_period_us`
- $P - N \longleftarrow$ `/proc/sys/kernel/sched_rt_runtime_us`
- The reserved CPU bandwidth mechanism can also be applied to real-time scheduling groups

Important scheduler topics deserving further discussions

- CFS and deadline schedulers (stay tuned!)
- Load balancing and scheduling domains
- Per-entity load tracking
- Group scheduling
- Power-aware CPU scheduling
- Linsched and other scheduler testing frameworks

Thank you!