The CPU Scheduler

Luca Abeni
luca.abeni@santannapisa.it

The Scheduler

- Scheduler: part of the OS kernel responsible for deciding how to assign resources to tasks
- CPU scheduler: decides which task(s) to execute
  - Implements the CPU scheduling algorithm
  - Responsible for building the schedule $\sigma : \mathcal{N} \rightarrow (\Gamma \cup \text{idle})^M$ (M is the number of CPUs)
    - Function $\sigma(t) = (\tau_1, ...\tau_M)$ mapping time in a set of scheduled tasks
- In Linux, function `schedule()` (defined in `kernel/sched/core.c`)
- Remember? To block a task:
  - Change its state (`set_task_state()`)
  - Invoke the scheduler (`schedule()`)

Single-Processor vs Multi-Processor Scheduling

- Single CPU: $\sigma(t) = \tau$ (where τ can be "`idle`")
  - Function mapping time in one single task (can be the idle task)
- M CPUs: $\sigma(t) = (\tau_1, ...\tau_M)$
  - Function mapping time in a tuple of M tasks
- How to implement this in practice?
- Various possibilities, including:
  - Partitioned scheduling
  - Global scheduling

Global Scheduling

- The scheduler is free to move tasks between different CPUs
  - Tasks are "migrated" to respect some kind of *global invariant*
- The m "best" (highest priority, earliest deadline, smallest virtual time, ...) tasks are scheduled on m CPUs / cores
  - $m = \min\{M, |\Gamma|\}$
- From the conceptual point of view, one single global queue
  - From the implementation point of view, various possibilities

Partitioned Scheduling

- Each task is associated to a CPU
  - The scheduler does not generally migrate tasks
- One ready task queue per CPU / core
  - Single-processor scheduling algorithms can be reused
- Appropriate task partitioning is fundamental
  - Can be performed by the programmer or by the kernel
  - Possible load-balancing - re-partitioning

Scheduling in Unix / POSIX

- Multiple *scheduling policies*
  - Policy == Scheduling Algorithm
  - Defined per-task
  - Handled on a priority basis
- `SCHED_OTHER`: for "regular" tasks; optimized for throughput
- `SCHED_RR` / `SCHED_FIFO`: priority based scheduling algorithm, provides more control to the user
- Other (non-standard) policies can be added by the OS kernel

The Linux CPU Scheduler

- Per-CPU ready task queues (`runqueue`s)
  - Note: this is an *implementation detail*
  - Does not mean that Linux uses partitioned scheduling only!
- From the algorithmic point of view:
  - Partitioned scheduling with periodic re-balancing for `SCHED_OTHER`
  - Global scheduling (or similar) for `SCHED_FIFO` / `SCHED_RR`
  - Additional scheduling policy (`SCHED_DEADLINE`) based on global scheduling
- The `schedule()` function works on a single runqueue

Migrations between CPUs

- Migrations: implemented by moving a task from a runqueue to a different one
  - WARNING: locking!
- Can happen periodically (load balancing) as in `SCHED_OTHER`
- Or can happen when needed to respect a global invariant!
  - When? Every time a task wakes up or blocks
  - Again, locking issues... Migration should happen only in “safe” instants $\Rightarrow$ callbacks!
    - “Safe instant”: when releasing the local runqueue lock is safe

Scheduling Classes

- Every scheduling policy is associated to a “scheduling class”
- Scheduling class: set of functions to be invoked
  - When a task changes its state
  - When a new task needs to be scheduled
  - When a task is preempted / dispatched
  - Periodically at every system tick
  - Plus some other migration-related callbacks
- The `schedule()` function asks all the scheduling classes (starting from the highest priority one) for a task to be executed
  - `pick_next_task()`

Scheduling Code in Linux

- Implementation of the scheduler: `kernel/sched`
  - Lot of code, because Linux provides a huge amount of advanced functionalities (cgroup scheduling, cpusets, autogroup, ...)
- `core.c`: main scheduler functionalities (including `schedule()` and friends)
- A compilation unit (`.c` file) for each scheduling class
- Additional code for advanced functionalities
- `kernel/sched/sched.h`: private definitions for the scheduler

Scheduler Internals

- Ready tasks queue: `runqueue` → `struct rq` (in `kernel/sched/sched.h`)
  - Actually, different policies have different queues (`struct cfs_rq`, `struct rt_rq`, `struct dl_rq`)
- Task descriptor: `struct task_struct` (in `include/linux/sched.h`)
  - "Shared" in all the kernel sources...
  - Contains some "scheduling entities" (different policies use different entities)
- Scheduling policies: defined by `kernel/sched/{rt,deadline,fair}.c` and used by `kernel/sched/core.c`

schedule(): Some Details

- Invoked when a task blocks or wakes up, to select the next task
  - This is an over-simplification; check the comments before `__schedule()`
- Scheduler: must not be interrupted (by interrupts, or others)
  - Avoid recursive scheduler invocations...
  - Disable preemption and invoke `__schedule()`
  - Use spinlocks, not mutexes!
- `__schedule()`: selects a new `current`
  - `prev` = `rq->curr` / `current`
  - `next` = task to be scheduled
  - `next == prev` ⇒ no context switch

__schedule(): Some Details

- First, check if `prev` is going to block
  - `prev->state` different from 0 (`TASK_RUNNING`)
  - Notice: only if no signal pending!!!
- Then, select new task:
  - `next = pick_next_task()`
    - Check all the scheduling classes (in priority order)
    - Some optimizations for common cases
- If `next` $\neq$ `prev`, context switch!!!
- Notice: the runqueue is locked, but can be unlocked for migrations

Implementation of Fixed Priorities

- Fixed priority schedulers can be implemented with an array of queues (one per priority level)
- Insertion into the queue (task wake-up) $\rightarrow O(1)$ operation
- Extraction of the highest priority task from the queue (scheduling decision)
  - Find the highest priority non-empty queue
  - $O(n)$ search!!! Too much overhead!!!
- Overhead due to naive implementation, not to an inherent problem

More Efficient Implementation

- The scheduler scalability can be improved by using a bitmap
  - Array of bits to mark the queues that are non-empty
- The highest priority queue can be found by finding the most significant bit in a word
  - Extraction becomes $O(1)$ if there is an Assembly instruction that returns the first 1 bit in a word (CLZ)
  - If not, table to implement the operation $\lceil \log w \rceil$

Implementation of fixed priority - I