Scheduling II

- Multilevel queue scheduling
- Multiprocessor scheduling issues
- Real-time scheduling
- Linux scheduling
Motivation

- No one-size-fits-all scheduler
  - Different workloads
  - Different environment

- Building a general scheduler that works well for all is difficult!

- Real scheduling algorithms are often more complex than the simple scheduling algorithms we’ve seen so far
Combining scheduling algorithms

- **Multilevel queue scheduling**: ready queue is partitioned into multiple queues

- Each queue has its own scheduling algorithm
  - Foreground processes: RR
  - Background processes: FCFS

- Must choose scheduling algorithm to schedule between queues. Possible algorithms
  - RR between queues
  - Fixed priority for each queue
Multiprocessor scheduling issues

- Shared-memory Multiprocessor

- How to allocate processes to CPU?
Symmetric multiprocessor

- **Architecture**

  ![Diagram of Symmetric Multiprocessor](image)

  - Small number of CPUs
  - Same access time to main memory
  - Private cache
Global queue of processes

- One ready queue shared across all CPUs

- Advantages
  - Good CPU utilization
  - Fair to all processes

- Disadvantages
  - Not scalable (contention for global queue lock)
  - Poor cache locality

- Linux 2.4 uses global queue
Per-CPU queue of processes

- Static partition of processes to CPUs

Advantages
- Easy to implement
- Scalable (no contention on ready queue)
- Better cache locality

Disadvantages
- Load-imbalance (some CPUs have more processes)
  - Unfair to processes and lower CPU utilization
Hybrid approach

- Use both global and per-CPU queues
- Balance jobs across queues

Processor Affinity
- Add process to a CPU’s queue if recently run on the CPU
  - Cache state may still present

Linux 2.6 uses a very similar approach
SMP: “gang” scheduling

- Multiple processes need coordination
- Should be scheduled simultaneously
- Scheduler on each CPU does not act independently
- **Coscheduling (gang scheduling):** run a set of processes simultaneously
- **Global context-switch** across all CPUs
Real-time scheduling

- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
  - E.g., gaming, video/music player, autopilot...

- **Hard real-time** systems – required to complete a critical task within a guaranteed amount of time

- **Soft real-time** computing – requires that critical processes receive priority over less fortunate ones

- Linux supports soft real-time
Linux scheduling overview

- **Multilevel Queue Scheduler**
  - Each queue associated with a priority
  - Some processes’ priorities may be adjusted dynamically

- **Two classes of processes**
  - Soft real-time processes: *always schedule highest priority processes*
    - FCFS (SCHED_FIFO) or RR (SCHED_RR) for processes with same priority
  - Normal processes: *priority with aging*
    - RR for processes with same priority (SCHED_NORMAL)
Linux scheduling priorities

- **Soft real-time scheduling policies**
  - `SCHED_FIFO` (FCFS)
  - `SCHED_RR` (round robin)
  - Priority over normal tasks
  - 100 static priority levels (1..99)

- **Normal scheduling policies**
  - `SCHED_NORMAL`: standard
    - `SCHED_OTHER` in POSIX
  - `SCHED_BATCH`: CPU bound
  - `SCHED_IDLE`: lower priority
  - Static priority is 0
    - 40 dynamic priority
    - “Nice” values

- `sched_setscheduler()`, `nice()`
  - See man page for detailed description
Linux scheduler implementations

- **Linux 2.4: global queue, O(N)**
  - Simple
  - Poor performance on multiprocessor/core
  - Poor performance when n is large

- **Linux 2.5: O(1) scheduler, per-CPU run queue**
  - Solves performance problems in the old scheduler
  - Complex, error prone logic to boost interactivity
  - No guarantee of fairness

- **Linux 2.6: completely fair scheduler (CFS)**
  - Fair
  - Naturally boosts interactivity
Problems with O(1) scheduler

- Priorities for interactive processes?
  - Higher priorities than CPU-bound processes
  - How to detect interactive processes?
  - Heuristics: more sleep/wait time $\Rightarrow$ more interactive $\Rightarrow$ higher dynamic priorities
  - Ad hoc, can be unfair

- Fairness for processes with diff. priorities?
  - Convert priority to time slice
  - Higher priorities get bigger time slices
  - Aging for low-priority processes
  - Ad hoc, can be unfair
Ideal fair scheduling

- Infinitesimally small time slice
- $n$ processes: each runs uniformly at $1/n^{th}$ rate

1 Process

3 Processes

1/3rd progress

- "Ideal multitasking CPU"
- Weighted fair scheduling
- Fair queuing [John Nagle 1985], stride scheduling [Carl A. Waldspurger, 1995]
Completely Fair Scheduler (CFS)

- Approximate fair scheduling
  - Run each process once per schedule latency period
    - `sysctl_sched_latency`
  - Time slice for process $P_i$: $T \times \frac{W_i}{\text{Sum of all } W_i}$
    - `sched_slice()`

- Too many processes?
  - Lower bound on smallest time slice
  - Schedule latency = lower bound $\times$ number of procs

- Introduced in Linux 2.6.23
Picking the next process

- Pick proc with weighted minimum runtime so far
  - Virtual runtime: \( \text{task->vruntime} += \frac{\text{executed time}}{\text{Wi}} \)

- Example
  - P1: 1 ms burst per 10 ms (schedule latency)
  - P2 and P3 are CPU-bound
  - All processes have the same weight (1)

![Ready and Slice Diagram]

- Ready: P1, P2, P3
- Slice: 3ms, 5ms
- P1, P2, P3
Finding proc with minimum runtime fast

- **Red-black tree**
  - Balanced binary search tree
  - Ordered by vruntime as key
  - $O(\log N)$ insertion, deletion, update, $O(1)$: find min

- Tasks move from left of tree to the right
- min_vruntime caches smallest value
- Update vruntime and min_vruntime
  - When task is added or removed
  - On every timer tick, context switch
Converting nice level to weight

- Table of nice level to weight
  - static const int prio_to_weight[40] (kernel/sched/sched.h)
- Nice level changes by 1 ➞ 10% weight

- Pre-computed to avoid
  - Floating point operations
  - Runtime overhead
Fsck all that...

Enter BFS

The scheduler that shall not be named
Hierarchical, modular scheduler

• Code from kernel/sched/core.c:

```c
class = sched_class_highest;
for ( ; ; ) {
    p = class->pick_next_task(rq);
    if (p)
        return p;
/*
 * Will never be NULL as the idle class always
 * returns a non-NULL p:
 */
    class = class->next;
}
```
sched_class Structure

```c
static const struct sched_class fair_sched_class = {
    .next                   = &idle_sched_class,
    .enqueue_task           = enqueue_task_fair,
    .dequeue_task           = dequeue_task_fair,
    .yield_task             = yield_task_fair,
    .check_preempt_curr     = check_preempt_wakeup,
    .pick_next_task         = pick_next_task_fair,
    .put_prev_task          = put_prev_task_fair,
    .select_task_rq         = select_task_rq_fair,
    .load_balance           = load_balance_fair,
    .move_one_task          = move_one_task_fair,
    .set_curr_task          = set_curr_task_fair,
    .task_tick              = task_tick_fair,
    .task_fork              = task_fork_fair,
    .prio_changed           = prio_changed_fair,
    .switched_to            = switched_to_fair,
};
```
The runqueue

- All run queues available in array runqueues, one per CPU
- struct rq (kernel/sched/sched.h)
  - Contains per-class run queues (RT, CFS) and params
    - E.g., CFS: a red-black tree of task_struct (struct rb_root tasks_timeline)
    - E.g., RT: array of active priorities
    - Data structure rt_rq, cfs_rq,
- struct sched_entity (include/linux/sched.h)
  - Member of task_struct, one per scheduler class
  - Maintains struct rb_node run_node, other per-task params
- Current scheduler for task is specified by task_struct.sched_class
  - Pointer to struct sched_class
  - Contains functions pertaining to class (object-oriented code)
Adding a new Scheduler Class

- The Scheduler is modular and extensible
  - New scheduler classes can be installed
  - Each scheduler class has priority within hierarchical scheduling hierarchy
    - Linked list of sched_class sched_class.next reflects priority
  - Core functions: kernel/sched/core.c, kernel/sched/sched.h, include/linux/sched.h
    - Additional classes: kernel/sched/fair.c, rt.c

- Process changes class via sched_setscheduler syscall

- Each class needs
  - New runqueue structure in main struct rq
  - New sched_class structure implementing scheduling functions
  - New sched_entity in the task_struct
Backup slides
Linux O(1) scheduler goals

- Avoid starvation
- Boost interactivity
  - Fast response to user despite high load
  - Achieved by inferring interactive processes and dynamically increasing their priorities
- Scale well with number of processes
  - O(1) scheduling overhead
- SMP goals
  - Scale well with number of processors
  - Load balance: no CPU should be idle if there is work
  - CPU affinity: no random bouncing of processes
- Reference: Linux/Documentation/sched-design.txt
runqueue data structure

- Two arrays of priority queues
  - active and expired
  - Total 140 priorities [0, 140)
  - Smaller integer = higher priority

```
<table>
<thead>
<tr>
<th>priority</th>
<th>active array</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[1]</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[140]</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
```

27
Scheduling algorithm for normal processes

1. Find highest priority non-empty queue in \( \text{rq->active} \); if none, simulate aging by swapping \( \text{active} \) and \( \text{expired} \)

2. \( \text{next} = \) first process on that queue

3. Adjust \( \text{next}'s \) priority

4. Context switch to \( \text{next} \)

5. When \( \text{next} \) used up its time slice, insert \( \text{next} \) to the right queue the \( \text{expired} \) array and call \( \text{schedule()} \) again
Aging: the traditional algorithm

for(pp = proc; pp < proc+NPROC; pp++) {
    if (pp->prio != MAX)
        pp->prio++;
    if (pp->prio > curproc->prio)
        reschedule();
}

Problem: $O(N)$. Every process is examined on each schedule() call!

This code is taken almost verbatim from 6th Edition Unix, circa 1976.
Simulate aging

- Swapping active and expired gives low priority processes a chance to run
  
  - Advantage: $O(1)$
    - Processes are touched only when they start or stop running
Find highest priority non-empty queue

- **Time complexity:** $O(1)$
  - Depends on the number of priority levels, not the number of processes

- **Implementation:** a bitmap for fast look up
  - 140 queues $\rightarrow$ 5 integers
  - A few compares to find the first non-zero bit
  - Hardware instruction to find the first 1-bit
    - `bsfl` on Intel
Real-time policies

- First-in, first-out: **SCHED_FIFO**
  - Static priority
  - Process is only preempted for a higher-priority process
  - No time quanta; it runs until it blocks or yields voluntarily
  - RR within same priority level

- Round-robin: **SCHED_RR**
  - As above but with a time quanta

- Normal processes have **SCHED_NORMAL** scheduling policy
Multiprocessor scheduling

- Per-CPU runqueue

- Possible for one processor to be idle while others have jobs waiting in their run queues

- Periodically, rebalance runqueues
  - Migration threads move processes from one runqueue to another

- The kernel always locks runqueues in the same order for deadlock prevention
Adjusting priority

- Goal: dynamically increase priority of interactive process

- How to determine interactive?
  - Sleep ratio
  - Mostly sleeping: I/O bound
  - Mostly running: CPU bound

- Implementation: per process sleep_avg
  - Before switching out a process, subtract from sleep_avg how many ticks a task ran
  - Before switching in a process, add to sleep_avg how many ticks it was blocked up to MAX_SLEEP_AVG (10 ms)
Calculating time slices

- Stored in field `time_slice` in struct `task_struct`
- Higher priority processes also get bigger time-slice

- `task_timeslice()` in `sched.c`
  - If `(static_priority < 120)` then `time_slice = (140-static_priority) * 20`
  - If `(static_priority >= 120)` then `time_slice = (140-static_priority) * 5`
Example time slices

<table>
<thead>
<tr>
<th>Priority:</th>
<th>Static Pri</th>
<th>Niceness</th>
<th>Quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>100</td>
<td>-20</td>
<td>800 ms</td>
</tr>
<tr>
<td>High</td>
<td>110</td>
<td>-10</td>
<td>600 ms</td>
</tr>
<tr>
<td>Normal</td>
<td>120</td>
<td>0</td>
<td>100 ms</td>
</tr>
<tr>
<td>Low</td>
<td>130</td>
<td>10</td>
<td>50 ms</td>
</tr>
<tr>
<td>Lowest</td>
<td>139</td>
<td>20</td>
<td>5 ms</td>
</tr>
</tbody>
</table>
Priority partition

- Total 140 priorities $[0, 140)$
  - Smaller integer = higher priority
  - Real-time: $[0, 100)$
  - Normal: $[100, 140)$

- MAX_PRIO and MAX_RT_PRIO
  - include/linux/sched.h
Priority related fields in *struct task_struct*

- **static_prio**: static priority set by administrator/users
  - Default: 120 (even for realtime processes)
  - Set use `sys_nice()` or `sys_setpriority()`
    - Both call `set_user_nice()`

- **prio**: dynamic priority
  - Index to `prio_array`

- **rt_priority**: real time priority
  - `prio = 99 - rt_priority`

- **include/linux/sched.h**
Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues
  - Real-time scheduling

- Scheduling in Linux
  - Scheduling algorithm
  - Setting priorities and time slices
  - Other implementation issues
Bookkeeping on each timer interrupt

- `scheduler_tick()`
  - Called on each tick
    - timer_interrupt ➔ do_timer_interrupt ➔ do_timer_interrupt_hook ➔ update_process_times

- If realtime and SCHED_FIFO, do nothing
  - SCHED_FIFO is non-preemptive

- If realtime and SCHED_RR and used up time slice, move to end of rq->active[prio]

- If SCHED_NORMAL and used up time slice
  - If not interactive or starving expired queue, move to end of rq->expired[prio]
  - Otherwise, move to end of rq->active[prio]
    - Boost interactive

- Else // SCHED_NORMAL, and not used up time slice
  - Break large time slice into pieces TIMESLICE_GRANULARITY
Processor affinity

- Each process has a bitmask saying what CPUs it can run on
  - By default, all CPUs
  - Processes can change the mask
  - Inherited by child processes (and threads), thus tending to keep them on the same CPU

- Rebalancing does not override affinity
Load balancing

- To keep all CPUs busy, load balancing pulls tasks from busy runqueues to idle runqueues.
- If *schedule* finds that a runqueue has no runnable tasks (other than the idle task), it calls *load_balance*.
- *load_balance* also called via timer:
  - *schedule_tick* calls *rebalance_tick*
  - Every tick when system is idle
  - Every 100 ms otherwise
load_balance looks for the busiest runqueue (most runnable tasks) and takes a task that is (in order of preference):

- inactive (likely to be cache cold)
- high priority

load_balance skips tasks that are:

- likely to be cache warm (hasn't run for cache_decay_ticks time)
- currently running on a CPU
- not allowed to run on the current CPU (as indicated by the cpus_allowed bitmask in the task_struct)
Optimizations

- If next is a kernel thread, borrow the MM mappings from prev
  - User-level MMs are unused.
  - Kernel-level MMs are the same for all kernel threads
- If prev == next
  - Don’t context switch
CFS: Scheduling Latency

- Equivalent to time slice across all processes
  - Approximation of infinitesimally small
  - To set/get type: `$ sysctl kernel.sched_latency_ns`

- Each process gets equal proportion of slice
  - $\text{Timeslice}(\text{task}) = \text{latency}/\text{nr\_tasks}$
  - Lower bound on smallest slice
  - To set/get: `$ sysctl kernel.sched_min_granularity_ns$
  - Too many tasks? $\text{sched\_latency} = \text{nr\_tasks} \times \text{min\_granularity}$

- Priority through proportional sharing
  - Task gets share of CPU proportional to relative priority
  - $\text{Timeslice}(\text{task}) = \frac{\text{Timeslice}(t) \times \text{prio}(t)}{\sum_{all\_t}(\text{prio}(t))}$

- Maximum wait time bounded by scheduling latency
CFS: Picking the Next Process

- Pick task with minimum runtime so far
  - Tracked by vruntime member variable
  - Every time process runs for $t$ ns, $\text{vruntime} += t$ (weighed by process priority)

- How does this impact I/O vs CPU bound tasks
  - Task A: needs 1 msec every 100 sec (I/O bound)
  - Task B, C: 80 msec every 100 msec (CPU bound)
  - After 10 times that A, B, and C have been scheduled
    - $\text{vruntime}(A) = 10$, $\text{vruntime}(B, C) = 800$
    - A gets priority, B and C get large time slices (10msec each)

- Problem: how to efficiently track min runtime?
  - Scheduler needs to be efficient
  - Finding min every time is an $O(N)$ operation