W4118: advanced scheduling

Instructor: Junfeng Yang

Outline

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

- Scheduler examples
  - xv6 scheduler
  - Linux O(1) scheduler
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
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
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
Multiprocessor scheduling issues

- Shared-memory Multiprocessor

  ![Processes Diagram](image)

- How to allocate processes to CPU?
Symmetric multiprocessor

- Architecture

  ![](chart)

  - 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
Outline

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

- **Scheduler examples**
  - xv6 scheduler
  - Linux $O(1)$ scheduler
xv6 scheduler

- One global queue across all CPUs
- Local scheduling algorithm: RR
- `scheduler()` in `proc.c`
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
Algorithm overview

- Multilevel Queue Scheduler
  - Each queue associated with a priority
  - A process’s priority may be adjusted dynamically

- Two classes of processes
  - 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)
    - Aging is implemented efficiently
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>task lists</th>
<th>priority</th>
<th>task lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
<td>[0]</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[140]</td>
<td></td>
<td>[140]</td>
<td></td>
</tr>
</tbody>
</table>
```
Scheduling algorithm for normal processes

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

```c
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 6\(^{th}\) 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 scheduling

- Linux has soft real-time scheduling
  - No hard real-time guarantees
- All real-time processes are higher priority than any conventional processes
- Processes with priorities $[0, 99]$ are real-time
- Process can be converted to real-time via `sched_setscheduler` system call
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) time_slice = (140-static_priority) * 20`
  - If `(static_priority >= 120) 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>