Linux Scheduling

CSE 2431: Introduction to Operating Systems
Instructor: Adam C. Champion, Ph.D.
Reading: Sections 6.7.1, 18.5, 18.6, [OSC]; Chap. 4, [LKD]
Contents

• Brief Review of Process Scheduler
• Case Study: Linux Scheduler (2.6+ \textit{vs.} 2.4)
  – Overview
  – Main Challenges
  – Process Selection: $O(1)$ vs. $O(n)$ algorithm
  – Timeslice and Priority Recalculation
  – Pre-emptive Opportunities
  – Others
• Summary
What Does A Scheduler Do?

• Deciding which process runs next among all runnable processes
Goals of Linux Scheduler

- Multitasking OS
- Maximize resource utilization
- Accommodate both I/O-bound and CPU-bound processes
- Favor I/O-bound over CPU-bound processes
- Not starve CPU-bound processes
- Serve Real-Time (RT) applications
Design Choice of Linux Scheduler

- Timeslice-based instead of job-based
- Priority-based scheduling algorithm instead of FCFS
  - Schedule higher priority process first
  - Round-robin for the same priority process
  - Real-Time processes have higher priorities than those of all normal processes
- Preemptive instead of non-preemptive
Main Challenges

- Scheduling time (red segment) as short as possible
- Timeslice length and process priority:
  - Interactive processes
  - CPU-bounded processes
  - Real-Time (RT) processes
**O(n) Scheduling Algorithm in 2.4.***

- Iterate through all runnable processes, find maximum return value of `goodness()`
- The function `goodness()`:
  - Normally based on `nice` value and previously unused timeslice
  - RT process: `rt_priority` plus 1000
  - Favor processes within the same address space as the current one by adding 1
Problems for $O(n)$ Scheduling

- $O(n)$ algorithm, where $n$ is the number of runnable processes
- Problem:
  - Not scalable $\Rightarrow$ bad for heavy workload
  - Unpredictable $\Rightarrow$ bad for RT applications
$O(1)$ Scheduling Algorithm in 2.6+ (1)

<table>
<thead>
<tr>
<th>priority bitmap</th>
<th>1 2 3 4 5 6  ........</th>
<th>140  ....</th>
</tr>
</thead>
<tbody>
<tr>
<td>active priority array</td>
<td>0 1 0 0 0 1 ● ● ● 0 1 0 0 0 0 0</td>
<td>0 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>
O(1) Scheduling Algorithm in 2.6+ (2)

- Each priority level corresponds to one bit in the bitmap array, one linked list in active priority array.
- If linked list is not empty, mark corresponding bit
- Search for the highest priority process is to search first set bit in a fix-length bitmap.
  - Most architectures have instructions to do so
  - E.g. x86 has bsfl, PPC has cntlzw
- It’s an O(1) algorithm.
Process Priority and Timeslice Recalculation

- Static priority
  - A nice value
  - Inherited from the parent process
  - Set up by user

- Dynamic priority
  - Based on static priority and application characteristics (interactive or CPU-bound)
  - Favor interactive applications over CPU-bound ones
  - Heuristic guess about the app characteristics (sleep_ave)

- Timeslice is mapped from priority
- Some differences on how to calculate between 2.6+ and 2.4
**When To Recalculate Priority, Timeslice?**

- 2.4: Recalculate priority, timeslice for all runnable processes when they used up their timeslices
  - $O(n)$ at certain point of scheduling
- 2.6+: Two priority arrays – one active, other expired
  - Calculate priority, timeslice for a process once it used up its timeslice; move it from active array to expired array
  - When active array is empty, swap pointers for active array, expired array
  - $O(1)$ at certain point of scheduling, no iteration, no chunk of time for recalculating priority and timeslice, although the aggregated time is still $O(n)$ for recalculation.
How and When to Preempt?

- Kernel sets `need_resched` flag (per-process var.) at 2 places:
  - `scheduler_tick()`, a process used up its timeslice
  - `try_to_wake_up()`, higher-priority process awaken
- Kernel checks `need_resched` at certain places; if safe, call `schedule()`
- User preemption: return to userspace from system call or interrupt handler
- Kernel preemption:
  - Return to kernel space from interrupt handler (2.6+ only)
  - When kernel code becomes preemptible again (`preempt_count` counts acquired lock #) (2.6+ only)
  - A task in the kernel explicitly calls `schedule()`
  - A task in the kernel blocks (which results in a call to `schedule()`)

13
Real-Time Scheduling

• Several scheduling policies:
  – `SCHED_FIFO` (RT)
  – `SCHED_RR` (RT)
  – `SCHED_NORMAL` (non-RT)

• If RT processes have not finished, non-RT processes are not scheduled unless all RT processes blocked

• `SCHED_FIFO`: no timeslice, execute until process finishes unless there’s a higher priority process or it is blocked

• `SCHED_RR`: has certain timeslice, but still RT process
Others

• Sleeping and Waking Up
  – Process sleeps in wait queue to wait for certain event
  – When the event occurs, kernel wakes up all processes in the wait queue

• Load Balancer (SMP)
  – Each CPU has a run queue and its own scheduling
  – Once load among different CPUs is unbalanced, migration thread helps migrate processes from busy CPUs to lightly-loaded ones
Summary

• \textit{O(1) scheduling algorithm} (including picking up process and recalculate priority) makes Linux 2.6.* more scalable and also good for RT processes.
• Linux scheduler \textit{favors interactive processes} over CPU-bound processes by adjusting process priority
• Linux 2.6+ makes \textit{kernel code preemptive}
• Hard to achieve one-size-fits-all scheduling algorithms, but \textit{Linux 2.6+ scheduler did pretty well}
References

• Linux Cross-Reference, http://lxr.linux.no/
• Josh Aas, “Understanding the Linux 2.6.8.1 CPU Scheduler,” http://joshaas.net/linux/linux_cpu_scheduler.pdf