Linux Scheduling

CSE 2431: Introduction to Operating Systems
Reading: §§6.7.1, 18.5, 18.6, [OSC]; Chap. 4, [LKD]
Contents

• Brief Review of Process Scheduler
• Case Study: Linux Scheduler (2.6.* vs. 2.4.*)
  – Overview
  – Main Challenges
  – Process Selection: $O(1)$ vs. $O(n)$ algorithm
  – Timeslice & 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 the resource utilization
- Good for both I/O-bound and CPU-bound processes
- Favor I/O-bound over CPU-bound processes
- Not starve CPU-bounded applications
- Reasonably good for 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

• The length of timeslice and process priority
  – Interactive processes
  – CPU-bounded processes
  – Real-Time (RT) processes
\(O(n)\) Scheduling Algorithm in 2.4.*

- Iterate all the runnable processes and find the maximum return value of \texttt{goodness()}.
- The function \texttt{goodness()}:
  - Normally, based on \texttt{nice} value and timeslice unused before.
  - RT process, \texttt{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 $\Rightarrow$ not good for heavy workload
  – Unpredictable $\Rightarrow$ not good for RT applications
O(1) Scheduling Algorithm in 2.6.* (1)

| priority bitmap | 1 | 2 | 3 | 4 | 5 | 6 | .......... | 140 | .....
|-----------------|---|---|---|---|---|---|----------|-----|-------
| 0 1 0 0 0 1     |   |   |   |   |   |   | ...      |     |       

| active priority array |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |
|                       |   |   |   |   |   |   |   |   |   |

|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |
$O(1)$ Scheduling Algorithm in 2.6.* (2)

- Each priority level corresponds to one bit in the bitmap array and a link list in the active priority array.
- If the link list is not empty, the corresponding bit is marked.
- 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 cnt1zw
- 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 and Timeslice?

• 2.4.*: Recalculate the priority and timeslice for all the 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 the priority and timeslice for a process once it used up its timeslice and move it from active array to expired array
  – When the active array is empty, switch the pointers for active array and expired array
  – $O(l)$ at certain point of scheduling, no iteration, no a chunk of time for recalculating priority and timeslice, although the aggregated time is still $O(n)$ for the recalculation.
How and When to Preempt?

- Kernel sets the `need_resched` flag (per-process variable) at various locations
  - `scheduler_tick()`, a process used up its timeslice
  - `try_to_wake_up()`, higher-priority process awaken
- Kernel checks `need_resched` at certain points; if safe, `schedule()` will be invoked
- User preemption: return to userspace from a system call or an interrupt handler
- Kernel preemption
  - Return to kernel space from an 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()`)

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 will not be scheduled unless all RT processes are blocked.

• For SCHED_FIFO, no timeslice, execute until the process finishes unless there is a higher priority process or it is blocked.

• For SCHED_RR, has certain timeslice, but still RT process
Others

• Sleeping & Waking Up
  – A process sleeps in a wait queue to wait for a certain event
  – When the event happens, kernel will wake up all the processes in the wait queue

• Load Balancer (SMP)
  – Each CPU has a run queue and its own scheduling
  – Once the load among different CPUs are not balanced, migration thread help for migrate processes from busiest CPUs to light-loaded CPUs.
Summary

- **$O(1)$ scheduling algorithm** (including picking up process and recalculate priority) makes Linux 2.6.* more scalable and also good for RT processes.
- Linux scheduler **favors interactive processes** over CPU-bound processes by adjusting process priority.
- Linux 2.6.* makes **kernel code preemptive**.
- It is difficult to have one-size-fits-all scheduling algorithms, but **Linux 2.6.* scheduler did a pretty good job**.
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