Scheduling in Linux (2.6)

Don Porter
Last time

• We went through the high-level theory of scheduling algorithms
  – One approach was a multi-level feedback queue

• Today: View into how Linux makes its scheduling decisions
  – Note: a bit dated – this is from v2.6, but I think still pedagogically useful and more accessible than the new approach
Lecture goals

• Understand low-level building blocks of a scheduler
• Understand competing policy goals
• Understand the O(1) scheduler
(Linux) Terminology Map

• task – a Linux PCB
  – Really represents a thread in the kernel
    • (more on threads next lecture)

• Quantum – CPU timeslice
  – “Quanta” is plural, for those whose Latin is dusty
Outline

• Policy goals (review)
• O(1) Scheduler
Policy goals

• Fairness – everything gets a fair share of the CPU
• Real-time deadlines
  – CPU time before a deadline more valuable than time after
• Latency vs. Throughput: Timeslice length matters!
  – GUI programs should feel responsive
  – CPU-bound jobs want long timeslices, better throughput
• User priorities
  – Virus scanning is nice, but I don’t want it slowing things down
No perfect solution

• Optimizing multiple variables
• Like memory allocation, this is best-effort
  – Some workloads prefer some scheduling strategies
• Nonetheless, some solutions are generally better than others
Outline

- Policy goals
- $O(1)$ Scheduler
O(1) scheduler

- Goal: decide who to run next, independent of number of processes in system
  - Still maintain ability to prioritize tasks, handle partially unused quanta, etc
O(1) Bookkeeping

• runqueue: a list of runnable tasks
  – Blocked processes are not on any runqueue
  – A runqueue belongs to a specific CPU
  – Each task is on exactly one runqueue
    • Task only scheduled on runqueue’s CPU unless migrated

• $2 \times 40 \times \#CPUs$ runqueues
  – 40 dynamic priority levels (more later)
  – 2 sets of runqueues – one active and one expired
O(1) Data Structures

Active

139 ➔ 138 ➔ 137 ➔ • ➔ • ➔ 101 ➔ 100

Expired

139 ➔ 138 ➔ 137 ➔ • ➔ • ➔ 101 ➔ 100
O(1) Intuition

• Take the first task off the lowest-numbered runqueue on active set
  – Confusingly: a lower priority value means higher priority
• When done, put it on appropriate runqueue on expired set
• Once active is completely empty, swap which set of runqueues is active and expired
• “Constant time”, since fixed number of queues to check; only take first item from non-empty queue
O(1) Example

Active

139  
138  
137  
.    
.    
101  
100

Pick first, highest priority task to run

Expired

139  
138  
137  
.    
.    
101  
100

Move to expired queue when quantum expires
What now?

Active

<table>
<thead>
<tr>
<th>139</th>
</tr>
</thead>
<tbody>
<tr>
<td>138</td>
</tr>
<tr>
<td>137</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

Expired

<table>
<thead>
<tr>
<th>139</th>
</tr>
</thead>
<tbody>
<tr>
<td>138</td>
</tr>
<tr>
<td>137</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>
Blocked Tasks

• What if a program blocks on I/O, say for the disk?
  – It still has part of its quantum left
  – Not runnable, so don’t waste time putting it on the active or expired runqueues

• We need a “wait queue” associated with each blockable event
  – Disk, lock, pipe, network socket, etc.
Blocking Example

Active

139
138
137
•
•
101
100

Expired

139
138
137
•
•
101
100

Disk

Process goes on disk wait queue

Block on disk!
Blocked Tasks, cont.

• A blocked task is moved to a wait queue until the expected event happens
  – No longer on any active or expired queue!

• Disk example:
  – After I/O completes, interrupt handler moves task back to active runqueue
Time slice tracking

• If a process blocks and then becomes runnable, how do we know how much time it had left?

• Each task tracks ticks left in ‘time_slice’ field
  – On each clock tick: current->time_slice--
  – If time slice goes to zero, move to expired queue
    • Refill time slice
    • Schedule someone else
  – An unblocked task can use balance of time slice
  – Forking halves time slice with child
More on priorities

• 100 = highest priority
• 139 = lowest priority
• 120 = base priority
  – “nice” value: user-specified adjustment to base priority
  – Selfish (not nice) = -20 (I want to go first)
  – Really nice = +19 (I will go last)
Base time slice

\[
time = \begin{cases} 
(140 - prio) \times 20 ms & \text{prio} < 120 \\
(140 - prio) \times 5 ms & \text{prio} \geq 120
\end{cases}
\]

• “Higher” priority tasks get longer time slices
  – And run first

Don’t worry about memorizing these formulae
Goal: Responsive UIs

- Most GUI programs are I/O bound on the user
  - Unlikely to use entire time slice
- Users get annoyed when they type a key and it takes a long time to appear
- Idea: give UI programs a priority boost
  - Go to front of line, run briefly, block on I/O again
- Which ones are the UI programs?
**Idea: Infer from sleep time**

- By definition, I/O bound applications spend most of their time waiting on I/O
- We can monitor I/O wait time and infer which programs are GUI (and disk intensive)
- Give these applications a priority boost
- Note that this behavior can be dynamic
  - Ex: GUI configures DVD ripping, then it is CPU-bound
  - Scheduling should match program phases
Dynamic priority

\[ \text{dynamic priority} = \max \left( 100, \min \left( \text{static priority} - \text{bonus} + 5, 139 \right) \right) \]

- Bonus is calculated based on sleep time
- Dynamic priority determines a tasks’ runqueue
- This is a heuristic to balance competing goals of CPU throughput and latency in dealing with infrequent I/O
  - May not be optimal
Dynamic Priority in O(1) Scheduler

- Important: The runqueue a process goes in is determined by the **dynamic** priority, not the static priority
  - Dynamic priority is mostly determined by time spent waiting, to boost UI responsiveness
- Nice values influence **static** priority
  - No matter how “nice” you are (or aren’t), you can’t boost your dynamic priority without blocking on a wait queue!
Rebalancing tasks

• As described, once a task ends up in one CPU’s runqueue, it stays on that CPU forever
CPU 0

CPU 1 Needs More Work!

CPU 1

Rebalancing
Rebalancing tasks

- As described, once a task ends up in one CPU’s runqueue, it stays on that CPU forever
- What if all the processes on CPU 0 exit, and all of the processes on CPU 1 fork more children?
- We need to periodically rebalance
- Balance overheads against benefits
  - Figuring out where to move tasks isn’t free
Idea: Idle CPUs rebalance

- If a CPU is out of runnable tasks, it should take load from busy CPUs
  - Busy CPUs shouldn’t lose time finding idle CPUs to take their work if possible

- There may not be any idle CPUs
  - Overhead to figure out whether other idle CPUs exist
  - Just have busy CPUs rebalance much less frequently
Average load

• How do we measure how busy a CPU is?
• Average number of runnable tasks over time
• Available in /proc/loadavg
Rebalancing strategy

• Read the loadavg of each CPU
• Find the one with the highest loadavg
• (Hand waving) Figure out how many tasks we could take
  – If worth it, lock the CPU’s runqueues and take them
  – If not, try again later
Editorial Note

• $O(1)$ scheduler is not constant time if you consider rebalancing costs
  – But whatevs: Execution time to pick next process is one of only several criteria for selecting a scheduling algorithm
  – $O(1)$ was later replaced by a logarithmic time algorithm (Completely Fair Scheduler), that was much simpler
    • More elegantly captured these policy goals
    • Amusingly, not “completely fair” in practice
Summary

• Understand competing scheduling goals
• Understand O(1) scheduler + rebalancing