CS153: Scheduling

Chengyu Song

Slides modified from Harsha Madhyvasta, Nael Abu-Ghazaleh, and Zhiyun Qian
Administrivia

- Lab
  - 3 groups pending confirmation
  - Lab1 was released
    - Walkthrough: April 23
    - All deliverables: April 30
Recall: scheduling

• Making the decision which process/thread runs next
• Goals: "good service"
  • Utilization, throughput, turnaround time, response time, fairness, etc
• Three basic scheduling algorithms
  • FIFO, SJF, RR
Problems of basic algorithms

- FIFO: good: fairness; bad: turnaround time, response time
- SJF: good: turnaround time; bad: fairness, response time
- RR: good: fairness, response time; bad: turnaround time
Research question and challenges

- Q: is there a scheduler that provides both good turnaround time and response time and prevents starvation?

- Challenges
  - Limited information about a process in the beginning
  - How to adapt?
  - How to prevent *gaming* the scheduler
MLFQ

- Multiple-Level Feedback Queues
  - Multiple queues representing different job types
  - Different queues have different priorities
  - One job can only belong to one queue
  - Jobs can move among queues based upon *feedback*
Priority scheduling

- Priority scheduling
  - Choose next job based on priority
  - How to decide a process’ priority?
    - For SJF, priority = 1/(expected CPU usage)
    - Also can be either preemptive or non-preemptive
      - How?
MLFQ: priority queues

- Rule 0 (optional): high priority queues have short time quantum and low priority queues have long time quantum
- **Rule 1**: if Priority(A) > Priority(B), A runs (B doesn’t)
- **Rule 2**: if Priority(A) == Priority(B), A and B runs RR
Example: Unix scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)
  - Priority scheduling across queues, RR within a queue
MLFQ: feedback

• Q: how to use feedback to change priority?

• **Rule 3:** when a job enters the system, it is placed at the highest priority
  • Assumes good behavior

• **Rule 4a:** if a job uses up an entire time slice while running, its priority is **reduced**

• **Rule 4b:** if a job gives up the CPU before the time slice is up, it stays at the **same** priority level
## Example: priority change

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time Slice (ms)</th>
<th>Round Robin Queues</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td><img src="new_or_i_o_bound_task.png" alt="New or I/O bound task" /></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td><img src="time_slice_expiration.png" alt="Time slice expiration" /></td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td><img src="time_slice_expiration.png" alt="Time slice expiration" /></td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td><img src="time_slice_expiration.png" alt="Time slice expiration" /></td>
</tr>
</tbody>
</table>
Example: mixed workload

1. I/O bound
   - Issues I/O request
   - I/O completes

2. CPU bound
   - CPU gets request
   - CPU completes

3. Time

"Example: mixed workload"
Problem: starvation

- Priority schedulers can cause starvation
  - Low priority jobs can wait indefinitely, like in SJF
- MLFQ: rule 4 allows downward move but no upward
- **Rule 5**: after some time period $S$, move all the jobs in the system to the topmost queue
- Alternatively: increase priority as a function of waiting time
Problem: gaming the scheduler

- Currently design only remember *one* slice
  - Jobs can deliberately relinquishing the CPU before slice expires
- **Rule 4:** once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced
Proportional-share scheduler

• Good service metrics
  • Instead of optimizing for turnaround or response time
  • Guarantee that each job obtain a certain percentage of CPU time

• Also known as **fair-share** scheduler
  • Linux: CFS (Complete Fair Scheduler)
Lottery scheduling

- An efficient implementation of fair-share scheduling using *uniform randomness*
  - Each has a number of *tickets*
  - The more tickets a process has, the higher its probability of getting executed
  - Uniform randomness guarantees fairness
Example

- Global tickets: 100 (0-99)
- Process A: 75% (0-74)
- Process B: 25% (75-99)

// winning tickets
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 49

// resulting scheduling
A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
Ticket currency

- Allows a user to allocate tickets among their own jobs in whatever currency they would like
  - User A: 100 global tickets
    - A1: 500 local currency (→ 50 global tickets)
    - A2: 500 local currency (→ 50 global tickets)
  - User B: 100 global tickets
    - B1: 10 local currency (→ 100 global tickets)
  - Linux: cgroup
Multiprocessor scheduling

- This is its own topic, we won't go into it in detail
  - Could come back to it towards the end of the quarter
- What would happen if we used MLFQ / FS on a multiprocessor?
  - Contention for scheduler spinlock
  - Multiple MFQ used – this optimization technique is called *distributed locking* and is common in concurrent programming
- A couple of other considerations
  - Co-scheduling for parallel programs, core affinity
New challenges

- Scheduling has been studied for a long time, but it's not done yet
- Heterogeneous scheduling
  - Different cores have different computation power and energy consumption
- Battery powered devices
  - Energy consumption is critical
Summary

- Good services metrics
  - Utilization, throughput, ATT, AWT, fairness, etc
- Two more realistic scheduler
  - MLFQ, FS
For next class ...

- Thread
- Textbook module 26, 27