CPU Scheduling

Lecture 13

Disclaimer: some slides are adopted from book authors’ and Dr. Kulkarni’s slides with permission
Recap

• **Deadlock prevention**
  – Break any of four deadlock conditions
    • Mutual exclusion, no preemption, hold & wait, circular dependency
  – Banker’s algorithm
    • If a request is granted, can it lead to a deadlock?

• **CPU Scheduling**
  – Decides: which thread, when, and how long?
  – Preemptive vs. non-preemptive
Recap

• FIFO
  – In the order of arrival
  – Non-preemptive

• SJF
  – Shortest job first.
  – Non preemptive

• SRTF
  – Preemptive version of SJF
Quiz: SRTF

- Average waiting time?

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

\[
(9 + 0 + 15 + 2) / 4 = 6.5
\]
Issues

• FIFO
  – Bad average turn-around time

• SJF/SRTF
  – Good average turn-around time
  – IF you know or can predict the future

• Time-sharing systems
  – Multiple users share a machine
  – Need high interactivity → low response time
Round-Robin (RR)

• FIFO with preemption
• Simple, fair, and easy to implement
• Algorithm
  – Each job executes for a fixed time slice: quantum
  – When quantum expires, the scheduler preempts the task
  – Schedule the next job and continue...
Round-Robin (RR)

• Example
  – Quantum size = 4
  – Gantt chart

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_1</th>
<th>P_1</th>
<th>P_1</th>
<th>P_1</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

– Response time (between ready to first schedule)
  • P1: 0, P2: 4, P3: 7. average response time = (0+4+7)/3 = 3.67

– Waiting time
  • P1: 6, P2: 4, P3: 7. average waiting time = (6+4+7)/3 = 5.67
How To Choose Quantum Size?

• Quantum length
  – Too short $\rightarrow$ high overhead (why?)
  – Too long $\rightarrow$ bad response time
    • Very long quantum $\rightarrow$ FIFO

![Diagram showing process time and quantum sizes]
Round-Robin (RR)

- Example
  - Quantum size = 2
  - Gantt chart

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

- Response time (between ready to first schedule)
  - P1: 0, P2: 2, P3: 4. average response time = (0+2+4)/3 = 2

- Waiting time
  - P1: 6, P2: 6, P3: 7. average waiting time = (6+6+7)/3 = 6.33
Discussion

• Comparison between FCFS, SRTF(SJF), and RR
  – What to choose for smallest average waiting time?
    • SRTF (SFJ) is the optimal
  – What to choose for better interactivity?
    • RR with small time quantum (or SRTF)
  – What to choose to minimize scheduling overhead?
    • FCFS
Example

- Task A and B
  - CPU bound, run an hour
- Task C
  - I/O bound, repeat (1ms CPU, 9ms disk I/O)
- FCFS?
  - If A or B is scheduled first, C can begins an hour later
- RR and SRTF?
Example Timeline

RR with 100ms time quantum

RR with 1ms time quantum

SRTF
Summary

• First-Come, First-Served (FCFS)
  – Run to completion in order of arrival
  – Pros: simple, low overhead, good for batch jobs
  – Cons: short jobs can stuck behind the long ones

• Round-Robin (RR)
  – FCFS with preemption. Cycle after a fixed time quantum
  – Pros: better interactivity (optimize response time)
  – Cons: performance is dependent on the quantum size

• Shortest Job First (SJF)/ Shortest Remaining Time First (SRTF)
  – Shorted job (or shortest remaining job) first
  – Pros: optimal average waiting time (turn-around time)
  – Cons: you need to know the future, long jobs can be starved by short jobs
Agenda

- Multi-level queue scheduling
- Fair scheduling
- Real-time scheduling
- Multicore scheduling
Multiple Scheduling Goals

- Optimize for interactive applications
  - Round-robin
- Optimize for batch jobs
  - FCFS

- Can we do both?
Multi-level Queue

• Ready queue is partitioned into separate queues
  – Foreground: interactive jobs
  – Background: batch jobs

• Each queue has its own scheduling algorithm
  – Foreground: RR
  – Background: FCFS

• Between the queue?
Multi-level Queue Scheduling

• Scheduling between the queues
  – Fixed priority
    • Foreground first; schedule background only when no tasks in foreground
    • Possible starvation
  – Time slicing
    • Assign fraction of CPU time for each queue
    • 80% time for foreground; 20% time for background
Multi-level Feedback Queue

• Each queue has a priority

• Tasks migrate across queues
  – Each job starts at the highest priority queue
  – If it uses up an entire quantum, drop one-level
  – If it finishes early, move up one-level (or stay at top)

• Benefits
  – Interactive jobs stay at high priority queues
  – Batch jobs will be at the low priority queue
  – Automatically!
Completely Fair Scheduler (CFS)

- Linux default scheduler, focusing on **fairness**
- Each task owns a fraction of CPU time share
  - E.g., \( A=10\% , B=30\% , C=60\% \)
- Scheduling algorithm
  - Each task maintains its virtual runtime
    - Virtual runtime = executed time \( \times \) weight
  - Pick the task with the **smallest virtual runtime**
    - Tasks are sorted according to their virtual times
CFS Example

Tasks are sorted according to their virtual times

Scheduled the “neediest” task

- 5
- 6
- 8
- 10
CFS Example

On a next scheduler event, re-sort the list
But list is inefficient.

- Tasks are sorted according to their virtual times
Red-black Tree

- Self-balancing binary search tree
- Insert: \( O(\log N) \), Remove: \( O(1) \)

Figure source: M. Tim Jones, "Inside the Linux 2.6 Completely Fair Scheduler", IBM developerWorks
Weighed Fair Sharing: Example

Weights: gcc = 2/3, bigsim=1/3
X-axis: mcu (tick), Y-axis: virtual time
Fair in the long run
Real-Time Scheduling

• Goal: meet the deadlines of important tasks
  – **Soft** deadline: game, video decoding, ...
  – **Hard** deadline: engine control, anti-lock break (ABS)
    • 100 ECUs (processors) in BMW i3 [*]

• Priority scheduling
  – A high priority task preempts lower priority tasks
  – Static priority scheduling
  – Dynamic priority scheduling

Rate Monotonic (RM)

• Priority is assigned based on **periods**
  – Shorter period -> higher priority
  – Longer period -> lower priority

• Optimal static-priority scheduling
Earliest Deadline First (EDF)

- Priority is assigned based on deadline
  - Shorter deadline $\rightarrow$ higher priority
  - Longer deadline $\rightarrow$ lower priority
- Optimal dynamic priority scheduling

(3,1) ➔ (4,1) ➔ (5,2) ➔
Real-Time Schedulers in Linux

- **SCHED_FIFO**
  - Static priority scheduler

- **SCHED_RR**
  - Same as SCHED_FIFO except using RR for tasks with the same priority

- **SCHED_DEADLINE**
  - EDF scheduler
  - Recently merged in the Linux mainline (v3.14)
Linux Scheduling Framework

• First, schedule real-time tasks
  – Real-time schedulers: (1) Priority based, (2) deadline based
• Then schedule normal tasks
  – Completely Fair Scheduler (CFS)
• Two-level queue scheduling
  – Between queues?
Multiprocessor Scheduling

• How many scheduling queues are needed?
  – Global shared queue: all tasks are placed in a single shared queue (global scheduling)
  – Per-core queue: each core has its own scheduling queue (partitioned scheduling)
Global Scheduling

OS

RunQueue

tasks

HW

CPU1  CPU2  CPU3  CPU4
Partitioned Scheduling

- Linux’s basic design. Why?
Load Balancing

- Undesirable situation
  - Core 1’s queue: 40 tasks
  - Core 2’s queue: 0 task

- Load balancing
  - Tries to balance load across all cores.
  - Not so simple, why?
    - Migration overhead: cache warmup
Load Balancing

• More considerations
  – What if certain cores are more powerful than others?
    • E.g., ARM bigLITTLE (4 big cores, 4 small cores)
  – What if certain cores share caches while others don’t?
  – Which tasks to migrate?
    • Some tasks may compete for limited shared resources
Summary

• Multi-level queue scheduling
  – Each queue has its own scheduler
  – Scheduling between the queues

• Fair scheduling (CFS)
  – Fairly allocate CPU time across all tasks
  – Pick the task with the smallest virtual time
  – Guarantee fairness and bounded response time

• Real-time scheduling
  – Static priority scheduling
  – Dynamic priority scheduling
Summary

• Multicore scheduling
  – Global queue vs. per-core queue
    • Mostly per-core queue due to scalability
  – Load balancing
    • Balance load across all cores
    • Is complicated due to
      – Migration overhead
      – Shared hardware resources (cache, dram, etc)
      – Core architecture heterogeneity (big cores vs. small cores)
      – …
Some Edge Cases

• How to set the virtual time of a new task?
  – Can’t set as zero. Why?
  – System virtual time (SVT)
    • The minimum virtual time among all active tasks
    • cfs_rq->min_vruntime
  – The new task can “catch-up” tasks by setting its virtual time with SVT
Weighed Fair Sharing: Example 2

Weights: gcc = 2/3, bigsim=1/3
X-axis: mcu (tick), Y-axis: virtual time
gcc slept 15 mcu