CPU Scheduling
Administrative

• Midterm
  – Mar. 11, 2016
  – Closed book, in-class
  – Review class: Mar. 9, 2016
Agenda

• Introduction to CPU scheduling
• Classical CPU scheduling algorithms
CPU Scheduling

• CPU scheduling is a **policy** to decide
  – **Which** thread to run next?
  – **When** to schedule the next thread?
  – **How long**?

• Context switching is a **mechanism**
  – To change the running thread
Assumption: CPU Bursts

Execution model

- Program uses the CPU for a while and then does some I/O, back to use CPU, ..., keep alternating
CPU Scheduler

• An OS component that determines which thread to run, at what time, and how long
  – Among threads in the **ready queue**
CPU Scheduler

• When the scheduler runs?

  – The running thread finishes
  – The running thread voluntarily gives up the CPU
    • yield, block on I/O, ...
  – The OS **preempts** the current running thread
    • quantum expire (timer interrupt)
Performance Metrics for CPU Scheduling

- **CPU utilization**
  - % of time the CPU is busy doing something

- **Throughput**
  - # of jobs done / unit time

- **Turnaround time**
  - Time to complete a task (ready -> complete)

- **Waiting time**
  - Time spent on waiting in the ready queue

- **Response time**
  - Time to schedule a task (ready -> first scheduled)
Example

• Assumption: A, B, C are released at time 0

[Diagram showing the times of Process A]

• The times of Process A
  – Turnaround time: 9
  – Wait time: 5
  – Response time: 0
Example

• Assumption: A, B, C are released at time 0

• The times of Process B
  – Turnaround time: 5
  – Wait time: 3
  – Response time: 1
Example

• Assumption: A, B, C are released at time 0

- Turnaround time
- Wait time
- Response time

• The times of Process C
  – Turnaround time: 10
  – Wait time: 6
  – Response time: 2
Workload Model and Gantt Chart

• Workload model

<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>1</td>
<td>10</td>
</tr>
<tr>
<td>P4</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

• Gantt chart
  – bar chart to illustrate a particular schedule
Scheduling Policy Goals

• Maximize throughput
  – High throughput (#of jobs done / time) is always good

• Minimize response/completion time
  – Important to interactive applications (games, editor, ...)

• Fairness
  – Make all threads progress equally

• Goals often conflicts
  – Frequent context switching may be good for reducing response time, but not so much for maximizing throughput
First-Come, First-Served (FCFS)

- FCFS
  - Assigns the CPU based on the order of the requests.
  - Implemented using a FIFO queue.
- Example

- Suppose that the processes arrive in the order: \( P_1, P_2, P_3 \)

- Waiting time?
  - \( P_1 = 0; P_2 = 24; P_3 = 27 \)

- Average waiting time
  - \( (0 + 24 + 27)/3 = 17 \)
Example 2

- Suppose that the processes arrive in the order: \( P_2, P_3, P_1 \)

<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>24</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

- Waiting time?
  - \( P1 = 6; P2 = 0; P3 = 3 \)
- Average waiting time
  - \( (6 + 0 + 3)/3 = 3 \)
- Much better than previous case \( \rightarrow \) performance varies greatly depending on the scheduling order
Shortest Job First (SJF)

- Can we always do the best FIFO?
  - Yes: if you know the tasks’ CPU burst times

- Shortest Job First (SJF)
  - Order jobs based on their burst lengths
  - Executes the job with the shortest CPU burst first
  - SJF is optimal
    - Achieves minimum average waiting time
Shortest Job First (SJF)

• Example

  – Gantt chart

<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>6</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

  – Average waiting time?
  • \(\frac{3 + 16 + 9 + 0}{4} = 7\)

• How to know the CPU burst time in advance?
Determining CPU Burst Length

• Can only estimate the length
  – Next CPU burst similar to previous CPU bursts?
  – Predict based on the past history

• Exponential weighted moving average (EWMA)
  – of past CPU bursts

1. $t_n =$ actual length of $n^{th}$ CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha$, $0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$. 
Recap

• CPU Scheduling
  – Decides which thread, when, and how long?

• Metrics
  – Turnaround time
    • Time to complete a task (ready -> complete)
  – Waiting time
    • Time spent on waiting in the ready queue
  – Response time
    • Time to schedule a task (ready -> first scheduled)

• FIFO
• SJF
Administrivia

• Project 1 grading policy is updated.
  – Deadline: Tonight

• Quiz1 is posted
  – Chapter 1-4
  – Due: Friday
Shortest Job First (SJF)

• What if jobs don’t arrive at the same 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>

– Average waiting time

• \( \frac{0+7+15+8}{4} = 7.5 \)
Shortest Remaining Time First (SRTF)

• Preemptive version of SJF
• New shorter job preempt longer running job

<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>

• Average waiting time
  \[ \frac{(9 + 0 + 15 + 2)}{4} = 6.5 \]
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
Summary

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

• SJF
  – Shortest job first.
  – Non preemptive

• SRTF
  – Preemptive version of SJF
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>

– 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 → high overhead (why?)
  – Too long → bad response time
    • Very long quantum → FIFO
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)/ Shorted 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!
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):

Suppose A is blocked on I/O
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):

Suppose A is blocked on I/O
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):

Suppose A is returned from I/O
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):

Time

A B C B A C

2 3

C

time = 8
Example of Multilevel Feedback Queues

- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 (x weight)
  – Pick the task with the smallest virtual runtime
    • Tasks are sorted according to their virtual times
  – Time slice varies depending on the #of tasks
    • Slice = target_latency / #of tasks
CFS Example

- Tasks are sorted according to their virtual times
- Scheduled the “neediest” task
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: $O(1)$
Weighed Fair Sharing: Example

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

• Multi-level queue scheduling
  – Multiple scheduling policies
    • RR for interactive tasks
    • FCFS for batch tasks
  – Multi-level feedback queue scheduling
    • Tasks automatically migrate among the queues

• CFS
  – Fair sharing of CPU time
  – Based on (virtual) runtime of each task
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
Real-Time Systems

• The correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

• **A correct value at the wrong time is a fault.**

• Processes attempt to control or react to events that take place in the outside world

• These events occur in “real time” and tasks must be able to keep up with them

• Processes are associated with timing constraints (deadlines)
Real-Time Spectrum

No RT  Soft RT  Hard RT

Computer simulation  User interface  Internet video, audio  Tele communication  Flight control
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

Periodic Task Model

\[ a_{i0} = 0 \]

\[ d_{i0} = 8 \]

\[ a_{i1} = 10 \]

\[ d_{i1} = 18 \]

\[ a_{i2} = 20 \]

\[ C_i = 3 \]

\[ D_i = 8 \]

\[ T_i = 10 \]

\[ \tau_{i0} \]

\[ \tau_{i1} \]

response time for job #1: 6
Rate Monotonic (RM)

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

• Optimal static-priority scheduling

![Diagram showing two sets of tasks with different period lengths and priorities]
Earliest Deadline First (EDF)

• Priority is assigned based on deadline
  – Shorter deadline $\rightarrow$ higher priority
  – Longer deadline $\rightarrow$ lower priority

• Optimal dynamic priority scheduling
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?
Linux Scheduling Framework

- Completely Fair Scheduler (CFS)
  - SCHED_OTHER, SCHED_BATCH
- Real-time Schedulers
  - SCHED_DEADLINE, SCHED_FIFO, SCHED_RR
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)
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)
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)
      – ...

Acknowledgements

• Some slides are adopted from the notes of
  – Dr. Kulkarni at KU
  – Dr. Pellizzoni at Univ. of Waterloo
  – The book authors