Recall: Example of RR with Time Quantum = 20

- Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>53</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>68</td>
</tr>
<tr>
<td>P4</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

- Waiting time for
  - P1 = (68-20) + (112-88) = 72
  - P2 = (20-0) = 20
  - P3 = (28-0) + (88-48) + (125-108) = 85
  - P4 = (48-0) + (108-68) = 88

- Average waiting time = (72 + 20 + 85 + 88) / 4 = 66
- Average completion time = (125 + 28 + 153 + 112) / 4 = 104.5

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (\(\infty\))?
    - Get back FIFO
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

- Assuming zero-time context-switching, is RR always better than FCFS?
- Simple example:

<table>
<thead>
<tr>
<th>Job #</th>
<th>FIFO</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>991</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>992</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>999</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  - Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
- Total time for RR longer even for zero-time switch!
Earlier Example with Different Time Quantum

<table>
<thead>
<tr>
<th>Quantum</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31½</td>
</tr>
<tr>
<td>Q = 1</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>Q = 5</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61⅔</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57⅓</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61⅔</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66⅔</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83⅔</td>
</tr>
<tr>
<td>Completion Time</td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td>69⅓</td>
</tr>
<tr>
<td>Q = 1</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
<td>100⅔</td>
</tr>
<tr>
<td>Q = 5</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
<td>99⅓</td>
</tr>
<tr>
<td>Q = 8</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
<td>95⅓</td>
</tr>
<tr>
<td>Q = 10</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
<td>99⅓</td>
</tr>
<tr>
<td>Q = 20</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
<td>104⅔</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121⅔</td>
</tr>
</tbody>
</table>

Handling Differences in Importance: Strict Priority Scheduling

- Execution Plan
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation:
    - Lower priority jobs don’t get to run because higher priority jobs
  - Deadlock: Priority Inversion
    - Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task
    - Usually involves third, intermediate priority task that keeps running even though high-priority task should be running

Handling Differences in Importance: Strict Priority Scheduling

- Priority 3
  - Job 1
  - Job 2
  - Job 3
- Priority 2
  - Job 4
- Priority 1
  - Job 5
- Priority 0
  - Job 6
  - Job 7

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - long running jobs may never get CPU
    - In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!
Scheduling Fairness

• How to implement fairness?
  – Could give each queue some fraction of the CPU
    » What if one long-running job and 100 short-running ones?
    » Like express lanes in a supermarket—sometimes express lines get so long, get better service by going into one of the “slower” lines
  – Could increase priority of jobs that don’t get service
    » What is done in some variants of UNIX
    » This is ad hoc—what rate should you increase priorities?
    » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority
      ⇒ Effectively no more priorities
      ⇒ Interactive jobs suffer

Lottery Scheduling

• Yet another alternative: Lottery Scheduling
  – Give each job some number of lottery tickets
  – On each time slice, randomly pick a winning ticket
  – On average, CPU time is proportional to number of tickets given to each job
• How to assign tickets?
  – Higher priority jobs get more tickets
  – To approximate SRTF, short running jobs get more, long running jobs get fewer
  – To avoid starvation, every job gets at least one ticket (everyone makes progress)
• Advantage over strict priority scheduling: behaves gracefully as load changes
  – Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

• Lottery Scheduling Example
  – Assume short jobs get 10 tickets, long jobs get 1 ticket

<table>
<thead>
<tr>
<th># short jobs/ # long jobs</th>
<th>% of CPU each short jobs gets</th>
<th>% of CPU each long jobs gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>0/2</td>
<td>N/A</td>
<td>50%</td>
</tr>
<tr>
<td>2/0</td>
<td>50%</td>
<td>N/A</td>
</tr>
<tr>
<td>10/1</td>
<td>9.9%</td>
<td>0.99%</td>
</tr>
<tr>
<td>1/10</td>
<td>50%</td>
<td>5%</td>
</tr>
</tbody>
</table>

– What if too many short jobs to give reasonable response time?
  » If load average is 100, hard to make progress
  » One approach: log some user out

How to Evaluate a Scheduling algorithm?

• Deterministic modeling
  – takes a predetermined workload and compute the performance of each algorithm for that workload
• Queueing models
  – Mathematical approach for handling stochastic workloads
• Implementation/Simulation:
  – Build system which allows actual algorithms to be run against actual data – most flexible/general
How to Handle Simultaneous Mix of Diff Types of Apps?

• Can we use Burst Time (observed) to decide which application gets CPU time?
• Consider mix of interactive and high throughput apps:
  – How to best schedule them?
  – How to recognize one from the other?
    » Do you trust app to say that it is “interactive”?
  – Should you schedule the set of apps identically on servers, workstations, tablets, and cellphones?

• Assumptions encoded into many schedulers:
  – Apps that sleep a lot and have short bursts must be interactive apps — they should get high priority
  – Apps that compute a lot should get low(er?) priority, since they won’t notice intermittent bursts from interactive apps
• Hard to characterize apps:
  – What about apps that sleep for a long time, but then compute for a long time?
  – Or, what about apps that must run under all circumstances (say periodically)

What if we Knew the Future?

• Could we always mirror best FCFS?
• Shortest Job First (SJF):
  – Run whatever job has least amount of computation to do
  – Sometimes called “Shortest Time to Completion First” (STCF)
• Shortest Remaining Time First (SRTF):
  – Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  – Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
• These can be applied to whole program or current CPU burst
  – Idea is to get short jobs out of the system
  – Big effect on short jobs, only small effect on long ones
  – Result is better average response time

Discussion

• SJF/SRTF are the best you can do at minimizing average response time
  – Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  – Since SRTF is always at least as good as SJF, focus on SRTF
• Comparison of SRTF with FCFS and RR
  – What if all jobs the same length?
    » SRTF becomes the same as FCFS
      (FCFS is optimal if all jobs the same length)
  – What if jobs have varying length?
    » SRTF (and RR): short jobs not stuck behind long ones
Example to illustrate benefits of SRTF

- Three jobs:
  - A, B: both CPU bound, run for week
  - C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - While C uses the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Challenges

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    » When you submit a job, have to say how long it will take
    » To stop cheating, system kills job if takes too long
  - But: hard to predict job’s runtime even for non-malicious users

SRTF Continued:

- RR 100ms time slice
- RR 1ms time slice

SRTF Discussion

- Bottom line, can’t really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies on a given trace
  - Optimal, so can’t do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)
**Administrivia**

- **Midterm 1 Review session**: 3-6pm, Saturday, 2/24
  - GPB 100 and/or Hearst Field Annex A1 (tentative)
- **Midterm 1**: 6:30-8:30pm, Wednesday, 2/28 (no class)
  - Li Ka Shing 245: IDs ending in 0,1,2,3
  - Hearst Field Annex A1: …4,5,6
  - VLSB 2060: …7,8
  - Barrows 20: …9
  - CS 189: Wurster 102 (no electronics)
  - DSP: Soda 465 & 606
- **Project 1**: code due Friday, 3/2
  - **Midterm includes project 1 material** – everyone in group should understand all parts of the project!

**SRTF Further discussion (Cont.)**

- Bottom line, can’t really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies on a given trace
  - Optimal, so can’t do any better

- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

**Predicting the Length of the Next CPU Burst**

- **Adaptive**: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, in CPUs, etc.
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn’t help

- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    - Let \( t_{n-1}, t_{n-2}, t_{n-3}, \ldots \) be previous CPU burst lengths.
    - Estimate next burst \( \tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \)
  - Function \( f \) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    - \( \tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1} \) with \( 0<\alpha\leq1 \)
Multi-Level Feedback Scheduling

- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    » Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    » e.g. foreground – RR, background – FCFS
    » Sometimes multiple RR priorities with quantum increasing exponentially (highest: 1ms, next: 2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    » serve all from highest priority, then next priority, etc.
  - Time slice:
    » each queue gets a certain amount of CPU time
    » e.g., 70% to highest, 20% next, 10% lowest

Real-Time Scheduling (RTS)

- Efficiency is important but predictability is essential:
  - We need to predict with confidence worst case response times for systems
  - In RTS, performance guarantees are:
    » Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    » System/throughput oriented with post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!
- Hard Real-Time
  - Attempt to meet all deadlines
  - EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)
- Soft Real-Time
  - Attempt to meet deadlines with high probability
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - Important for multimedia applications
  - CBS (Constant Bandwidth Server)
Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:

![Diagram showing workload characteristics with tasks T1, T2, T3, and T4 with their respective deadlines and computation times.]

Example: Round-Robin Scheduling Doesn’t Work

![Diagram illustrating Round-Robin Scheduling with missed deadlines.]  

Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P, C)
- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is
- The scheduler always schedules the active task with the closest absolute deadline

![Diagram showing Earliest Deadline First (EDF) scheduling with tasks T1, T2, T3, and T4.]

A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren’t enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - Assuming you’re paying for worse response time in reduced productivity, customer angst, etc…
    - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization \( \rightarrow 100\% \)
- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit “knee” of curve
<table>
<thead>
<tr>
<th>Summary (1 of 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Round-Robin Scheduling:</strong></td>
</tr>
<tr>
<td>– Give each thread a small amount of CPU time when it executes;</td>
</tr>
<tr>
<td>cycle between all ready threads</td>
</tr>
<tr>
<td>– Pros: Better for short jobs</td>
</tr>
<tr>
<td><strong>Shortest Job First (SJF) / Shortest Remaining Time First (SRTF):</strong></td>
</tr>
<tr>
<td>– Run whatever job has the least amount of computation to do/least</td>
</tr>
<tr>
<td>remaining amount of computation to do</td>
</tr>
<tr>
<td>– Pros: Optimal (average response time)</td>
</tr>
<tr>
<td>– Cons: Hard to predict future, Unfair</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Summary (2 of 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lottery Scheduling:</strong></td>
</tr>
<tr>
<td>– Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)</td>
</tr>
<tr>
<td><strong>Multi-Level Feedback Scheduling:</strong></td>
</tr>
<tr>
<td>– Multiple queues of different priorities and scheduling algorithms</td>
</tr>
<tr>
<td>– Automatic promotion/demotion of process priority in order to approximate</td>
</tr>
<tr>
<td>SJF/SRTF</td>
</tr>
<tr>
<td><strong>Real-time scheduling</strong></td>
</tr>
<tr>
<td>– Need to meet a deadline, predictability essential</td>
</tr>
<tr>
<td>– Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling</td>
</tr>
</tbody>
</table>