CS162
Operating Systems and Systems Programming
Lecture 10

Scheduling

February 22th, 2017
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
Recall: CPU Scheduling

- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
• Question: How does OS decide which thread to dequeue?
  – Obvious queue to worry about is ready queue
  – Others can be scheduled as well, however

• **Scheduling**: deciding which threads are given access to resources from moment to moment
Assumption – CPU Bursts

- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst
Scheduling Assumptions

• CPU scheduling big area of research in early 70’s
• Many implicit assumptions for CPU scheduling:
  – One program per user
  – One thread per program
  – Programs are independent
Scheduling Assumptions (Cont.)

• Clearly, unrealistic but they simplify the problem
  – For instance: is “fair” about fairness among users or programs?
    » If I run one compilation job and you run five, you get five times as much CPU on many operating systems

• The high-level goal: Dole out CPU time to optimize some desired parameters of system
Scheduling Policy Goals/Criteria

• Minimize Response Time
  – Minimize elapsed time to do an operation (or job)
  – Response time is what the user sees:
    » Time to echo a keystroke in editor
    » Time to compile a program
    » Real-time tasks: Must meet deadlines imposed by World
Scheduling Policy Goals/Criteria (Cont.)

• Maximize Throughput
  – Maximize operations (or jobs) per second
  – Throughput related to response time, but not identical:
    » Minimizing response time will lead to more context switching than if you only maximized throughput
  – Two parts to maximizing throughput
    » Minimize overhead (for example, context-switching)
    » Efficient use of resources (CPU, disk, memory, etc)
Scheduling Policy Goals/Criteria (Cont.)

- **Fairness**
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    » Better *average* response time by making system *less* fair
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also “First In, First Out” (FIFO) or “Run until done”
    » In early systems, FCFS meant one program scheduled until done (including I/O)
    » Now, means keep CPU until thread blocks

- Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

  - Suppose processes arrive in the order: $P_1, P_2, P_3$
  The Gantt Chart for the schedule is:
FCFS Scheduling (Cont.)

• Example continued:

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0</td>
<td>24</td>
<td>27</td>
</tr>
</tbody>
</table>

– Waiting time for $P₁ = 0$; $P₂ = 24$; $P₃ = 27$
– Average waiting time: $(0 + 24 + 27)/3 = 17$
– Average Completion time: $(24 + 27 + 30)/3 = 27$

• Convoy effect: short process behind long process
FCFS Scheduling (Cont.)

• Example continued:
  – Suppose that processes arrive in order: \(P_2, P_3, P_1\). Now, we have:

<table>
<thead>
<tr>
<th>(P_2)</th>
<th>(P_3)</th>
<th>(P_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  – Waiting time for \(P_1\) = 6; \(P_2\) = 0; \(P_3\) = 3
  – Average waiting time: \((6 + 0 + 3)/3 = 3\)
  – Average Completion time: \((3 + 6 + 30)/3 = 13\)

• In second case:
  – Average waiting time is much better (before it was 17)
  – Average completion time is better (before it was 27)

• FIFO Pros and Cons:
  – Simple (+)
  – Short jobs get stuck behind long ones (-)
    » Safeway: Getting milk, always stuck behind cart full of small items
Round Robin (RR) Scheduling

• FCFS Scheme: Potentially bad for short jobs!
  – Depends on submit order
  – If you are first in line at supermarket with milk, you don’t care who is behind you, on the other hand…

• Round Robin Scheme
  – Each process gets a small unit of CPU time \((time\ quantum)\), usually 10-100 milliseconds
  – After quantum expires, the process is preempted and added to the end of the ready queue.
  – \(n\) processes in ready queue and time quantum is \(q\) $\Rightarrow$
    » Each process gets \(1/n\) of the CPU time
    » In chunks of at most \(q\) time units
    » No process waits more than \((n-1)q\) time units
RR Scheduling (Cont.)

• Performance
  – $q$ large $\Rightarrow$ FCFS
  – $q$ small $\Rightarrow$ Interleaved (really small $\Rightarrow$ hyperthreading?)
  – $q$ must be large with respect to context switch, otherwise overhead is too high (all overhead)
Example of RR with Time Quantum = 20

- Example:
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>53</td>
</tr>
<tr>
<td>P_2</td>
<td>8</td>
</tr>
<tr>
<td>P_3</td>
<td>68</td>
</tr>
<tr>
<td>P_4</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

<table>
<thead>
<tr>
<th></th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>28</td>
<td>48</td>
<td>68</td>
<td>88</td>
<td>108</td>
<td>112</td>
<td>125</td>
<td>145</td>
<td>153</td>
</tr>
</tbody>
</table>

- Waiting time for
  - \( P_1 = (68-20) + (112-88) = 72 \)
  - \( P_2 = (20-0) = 20 \)
  - \( P_3 = (28-0) + (88-48) + (125-108) = 85 \)
  - \( P_4 = (48-0) + (108-68) = 88 \)

- Average waiting time = (72+20+85+88)/4 = 66¼
- Average completion time = (125+28+153+112)/4 = 104½

- 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 (∞)?
    » 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-cost context-switching time, is RR always better than FCFS?
- Simple example:
  - 10 jobs, each take 100s of CPU time
  - RR scheduler quantum of 1s
  - All jobs start at the same time

- Completion Times:

<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-cost switch!
### Earlier Example with Different Time Quantum

**Quantum Completion Time**

<table>
<thead>
<tr>
<th></th>
<th>P_2</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>32</td>
<td>85</td>
<td>8</td>
<td>153</td>
</tr>
</tbody>
</table>

#### Best FCFS:

<table>
<thead>
<tr>
<th>Quantum</th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</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>31 1/4</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>61 1/4</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57 1/4</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61 1/4</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66 1/4</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83 1/2</td>
</tr>
</tbody>
</table>

#### Worst FCFS:

<table>
<thead>
<tr>
<th>Quantum</th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td>69 1/2</td>
</tr>
<tr>
<td>Q = 1</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
<td>100 1/2</td>
</tr>
<tr>
<td>Q = 5</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
<td>99 1/2</td>
</tr>
<tr>
<td>Q = 8</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
<td>95 1/2</td>
</tr>
<tr>
<td>Q = 10</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
<td>99 1/2</td>
</tr>
<tr>
<td>Q = 20</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
<td>104 1/2</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121 3/4</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 (Cont.)

• How to fix problems?
  – Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…
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 lanes get so long, get better service by going into one of the other 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⇒Interactive jobs suffer
Administrivia

• Midterm on **Monday 2/27 6:30-8PM**
  – Last names **A-K** 1 LeConte
  – **Last names L-T** 245 Li Ka Shing
  – Last Names **U-Z** 3 Leconte

• Includes today’s lecture

• Closed book, no calculators, **one double-side letter-sized page of handwritten notes**

• Review – Saturday, 2/25 3-6pm 145 Dwinelle
BREAK
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?
  - 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 (Cont.)

- 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, pads, and cellphones?
How to Handle Simultaneous Mix of Diff Types of Apps?

• 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 (i.e. FCFS is best can do 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
  - If only one at a time, C uses 90% of 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 Example continued:

RR 100ms time slice

RR 1ms time slice

CABAB…

Disk Utilization:
~90% but lots of wakeups!

Disk Utilization:
9/201 ~ 4.5%

Disk Utilization:
90%
SRTF Further discussion

• 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
• Bottom line, can’t really know how long job will take
  – However, can use SRTF as a yardstick for measuring other policies
  – 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, 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 \( t_{n-1}, t_{n-2}, t_{n-3}, \) etc. 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 \((0 < \alpha \leq 1)\)
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)
• 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
• **Countermeasure**: user action that can foil intent of OS designers
  – For multilevel feedback, put in a bunch of meaningless I/O to keep job’s priority high
  – Of course, if everyone did this, wouldn’t work!

• Example of Othello program:
  – Playing against competitor, so key was to do computing at higher priority the competitors.
  > Put in `printf`'s, ran much faster!
Summary (1 of 2)

- **Round-Robin Scheduling:**
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs

- **Shortest Job First (SJF) / Shortest Remaining Time First (SRTF):**
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
• **Lottery Scheduling:**
  – Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)

• **Multi-Level Feedback Scheduling:**
  – Multiple queues of different priorities and scheduling algorithms
  – Automatic promotion/demotion of process priority in order to approximate SJF/SRTF