Outline

• Definitions
  • Response time, throughput, scheduling policy, …

• Uniprocessor policies
  • FCFS, SJF/SRTF, Round Robin, …
  • Real-time scheduling

• Multiprocessor policies
  • Oblivious scheduling, gang scheduling, …
Definitions

- **Task**
  - User request (e.g., mouse click, web request, shell command, etc.)

- **Workload**
  - Set of tasks for system to perform

- **Scheduling algorithm**
  - Takes workload as input, decides which tasks to do first

- **Overhead**
  - How much extra work is done by scheduler?

- **Preemptive scheduler**
  - If we can take resources away from a running task

- **Work-conserving**
  - Resources are used whenever there is task to run
  - For non-preemptive schedulers, work-conserving is not always better

**Only preemptive, work-conserving schedulers to be considered in this lecture!**
Recall: CPU Scheduling

• Earlier, we talked about life-cycle of threads
  • Threads work their way from ready 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 thread gets resource from moment to moment
Execution Model

- Programs alternate between bursts of CPU and I/O
  - Use CPU for some period, then do I/O, then use CPU again, etc.
- CPU scheduling is about choosing thread which gets CPU for its next CPU burst
- With preemption, thread may be forced to give up CPU before finishing its burst
CPU Scheduling Assumptions

- There are many implicit assumptions for CPU scheduling
  - *One* program per user
  - *One* thread per program
  - Programs are *independent*

- These may not hold in all systems, but they simplify the problem

- High-level goal is to divide CPU time to optimize some desired properties
CPU Scheduling Policy Goals/Criteria

• Minimize average response time
  • Minimize elapsed time to do an operation (or task)
  • Response time is what users see
    • Time to echo a keystroke in editor
    • Time to compile a program
    • Real-time tasks must meet deadlines imposed by “environment”
Maximize **throughput**

- Maximize operations (or tasks) per time unit (e.g., second)
- Throughput related to response time, but not identical
  - Minimizing response time could lead to more context switching which will then hurt throughput (more on this later!)

Two parts to maximizing throughput

- Minimize overhead (e.g., context-switching)
- Efficient use of resources (e.g., CPU, disk, memory, etc.)
CPU Scheduling Policy Goals/Criteria (cont.)

• Achieve **fairness**
  • Share CPU time among *users* in some *equitable* way
  • What does equitable mean?
    • Equal share of CPU time?
      • What if some tasks don’t need their full share?
    • Minimize variance in worst case performance?
      • What if some tasks were running when no one else was running?
  • Who are users? Actual users or programs?
    • If A runs one thread and B runs five, B could get five times as much CPU time on many OS’s
  • Fairness is not minimizing average response time
    • Improving average response time could make system less fair (more on this later!)
Outline

• Definitions
  • Response time, throughput, scheduling policy, …

• Uniprocessor policies
  • FCFS, SJF/SRTF, Round Robin, …
  • Real-time scheduling

• Multiprocessor policies
  • Oblivious scheduling, gang scheduling, …
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also “First In, First Out” (FIFO)
  - In early systems, FCFS meant one program scheduled until done (including its I/O activities)
  - Now, it means that program keeps CPU until the end of its CPU burst

- Example:

<table>
<thead>
<tr>
<th>Thread</th>
<th>CPU Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>24</td>
</tr>
<tr>
<td>$T_2$</td>
<td>3</td>
</tr>
<tr>
<td>$T_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose threads arrive in order: $T_1, T_2, T_3$

The Gantt Chart for FCFS scheduling is:

```
       | T1 | T2 | T3 |
---|----|----|----|
0  |    |    |    |
24 |    |    |    |
27 |    |    |    |
30 |    |    |    |
```
FCFS Scheduling (cont.)

- Example continued:

\[ \begin{array}{|c|c|c|}
\hline
 & T_1 & T_2 & T_3 \\
\hline
0 & 24 & 27 & 30 \\
\hline
\end{array} \]

- Waiting time for \( T_1 \) is 0, for \( T_2 \) is 24, and for \( T_3 \) is 27
- Average waiting time is \( \frac{0 + 24 + 27}{3} = 17 \)
- Average response time is \( \frac{24 + 27 + 30}{3} = 27 \)
- Convoy effect: Short threads get stuck behind long ones
  - At supermarket, you with milk get stuck behind cart full of small items
FCFS Scheduling (cont.)

• If threads arrive in order: \( T_2, T_3, T_1 \), then we have

<table>
<thead>
<tr>
<th></th>
<th>T_2</th>
<th>T_3</th>
<th>T_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

• Waiting time for \( T_1 \) is 6, for \( T_2 \) is 0, and for \( T_3 \) is 3
• Average waiting time is \( (6 + 0 + 3)/3 = 3 \)
• Average response time is \( (3 + 6 + 30)/3 = 13 \)
• Average waiting time is much better (before it was 17)
• Average response time is better (before it was 27)

• Pros and cons of FCFS
  • Simple (+)
  • Short tasks get stuck behind long ones (-)
Round Robin (RR) Scheduling

• FCFS is potentially bad for short tasks!
  • 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
  • Each thread gets small unit of CPU time, called *time quantum* (usually 10-100 milliseconds)
  • Once quantum expires, thread is preempted and added to end of ready queue
  • *N* threads in ready queue and time quantum is *q* ⇒
    • Each thread gets *l/N* of CPU time in chunks of at most *q* time units
    • No thread waits more than *(N-1)q* time units
Example: RR with Time Quantum of 20

- Example:

<table>
<thead>
<tr>
<th>Thread</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>53</td>
</tr>
<tr>
<td>$T_2$</td>
<td>8</td>
</tr>
<tr>
<td>$T_3$</td>
<td>68</td>
</tr>
<tr>
<td>$T_4$</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is

```
T1   T2   T3   T4   T1   T3   T4   T1   T3   T3
0    20   28   48   68   88   108  112  125  145  153
```

- Waiting time for
  - $T_1 = (68 - 20) + (112 - 88) = 72$
  - $T_2 = (20 - 0) = 20$
  - $T_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$
  - $T_4 = (48 - 0) + (108 - 68) = 88$

- Average waiting time is $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$
- Average response time is $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$
Round-Robin Discussion

• Pros and cons of RR
  • Better for short tasks, Fair (+)
  • Context-switching time adds up for long tasks (-)

• How does performance change with time quantum?
  • What if it’s too long?
    • Response time suffers!
  • What if it’s too short?
    • Throughput suffers!
  • What if it’s infinite (∞)?
    • RR ⇒ FCFS
  • Time quantum must be long compared to context switching time, otherwise overhead will be too high
Round-Robin Discussion (cont.)

- Actual choices of time quantum
  - Initially, UNIX time quantum was one second
    - Worked ok when UNIX was used by one or two users
    - What if you use text editor while there are three compilations going on?
      - It takes 3 seconds to echo each keystroke!
  - Need to balance short-task performance and long-task throughput
    - Typical time quantum today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching
FCFS vs. RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Suppose there are 10 tasks, each take 100s of CPU time, RR quantum is 1s

<table>
<thead>
<tr>
<th>Task #</th>
<th>FCFS</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>
FCFS vs. RR (cont.)

- Completion times

<table>
<thead>
<tr>
<th>Task #</th>
<th>FCFS</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 have the same length
- Also, cache must be shared between all tasks with RR but can be devoted to each task with FIFO
  - Total time for RR is longer even for zero-cost context switching!
Earlier Example: RR vs. FCFS, Effect of Different Time Quanta

<table>
<thead>
<tr>
<th>Best FCFS</th>
<th>T₂ (8)</th>
<th>T₄ (24)</th>
<th>T₁ (53)</th>
<th>T₃ (68)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>32</td>
<td>85</td>
<td>153</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>32</td>
<td>85</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Waiting Time</th>
<th>Quantum</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
<td>T₂ (8)</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31 ½</td>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>T₄ (24)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response Time</th>
<th>Quantum</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
<td>T₁ (53)</td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td>69 ½</td>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>T₃ (68)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Earlier Example: RR vs. FCFS, Effect of Different Time Quanta (cont.)

<table>
<thead>
<tr>
<th>Worst FCFS</th>
<th>T₃ (68)</th>
<th>T₁ (53)</th>
<th>T₄ (24)</th>
<th>T₂ (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>68</td>
<td>121</td>
<td>145</td>
<td>153</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Waiting Time</th>
<th>Quantum</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₅</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31¼</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td></td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td></td>
<td>83½</td>
</tr>
<tr>
<td>Best FCFS</td>
<td></td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td></td>
<td>69½</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td></td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td></td>
<td>121¾</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response Time</th>
<th>Quantum</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₅</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</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>5</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>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Earlier Example: RR vs. FCFS, Effect of Different Time Quanta (cont.)

<table>
<thead>
<tr>
<th></th>
<th>Quantum</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting Time</td>
<td>Best FCFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>T1</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31¼</td>
</tr>
<tr>
<td>8</td>
<td>T2</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>16</td>
<td>T3</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61¼</td>
</tr>
<tr>
<td>24</td>
<td>T4</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57½</td>
</tr>
<tr>
<td>32</td>
<td>T5</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61¼</td>
</tr>
<tr>
<td>40</td>
<td>T6</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66¼</td>
</tr>
<tr>
<td>48</td>
<td>T7</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83½</td>
</tr>
<tr>
<td>56</td>
<td>T8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>T9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>T10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>T11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>T12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>T13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>T14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>T15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>T16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>T17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>T18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>T19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>T20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>T21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>T22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>Best FCFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>T23</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121</td>
<td>121¾</td>
</tr>
</tbody>
</table>

Response Time

<table>
<thead>
<tr>
<th></th>
<th>Quantum</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T1</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31¼</td>
</tr>
<tr>
<td>8</td>
<td>T2</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>16</td>
<td>T3</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61¼</td>
</tr>
<tr>
<td>24</td>
<td>T4</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57½</td>
</tr>
<tr>
<td>32</td>
<td>T5</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61¼</td>
</tr>
<tr>
<td>40</td>
<td>T6</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66¼</td>
</tr>
<tr>
<td>48</td>
<td>T7</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83½</td>
</tr>
<tr>
<td>56</td>
<td>T8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>T9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>T10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>T11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>T12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>T13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>T14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>T15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>T16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>T17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>T18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>T19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>T20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>T21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>T22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>Best FCFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>T23</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121</td>
<td>121¾</td>
</tr>
</tbody>
</table>
Shortest Task First (SJF) Scheduling

• Could we always mirror best FCFS?

• **Shortest Task First (SJF)**
  - Run task that 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 task arrives and has shorter time to completion than remaining time on current task, immediately preempt current task
  - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)

• These can be applied to whole program or current CPU burst
  - Key idea: get short tasks out of system
  - Big effect on short tasks, only small effect on long ones
  - Better average response time
SJF/SRTF Optimality

• SJF/SRTF minimize average response time! Why?
  • Consider alternative policy P (not SJF/SRTF) that is optimal
  • At some point, P chooses to run task that is not the shortest
  • Keep order of tasks the same, but run the shorter task first
  • This reduces average response time ⇒ contradiction!
SJF/SRTF Discussion

• SJF/SRTF are best you can do to minimize average response time
  • Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  • Since SRTF is always at least as good as SJF, we can just focus on SRTF

• Comparison of SRTF with FCFS
  • What if all tasks are the same length?
    • SRTF ⇒ FCFS (i.e., FCFS is best we can do if all tasks have the same length)
  • What if tasks have varying length?
    • Unlike FCFS, with SRTF, short tasks do not get stuck behind long ones
Mix of CPU and I/O Bound Tasks: FCFS vs. RR vs. SRTF

- Example: Suppose there are three tasks
  - A and B are both CPU bound with CPU bursts that last for a week
  - C is I/O bound with iterations of 1ms CPU burst followed by 9ms I/O burst
  - If A or B run by themselves, CPU utilization is 100% and I/O utilization is 0%
  - If C runs by itself, CPU utilization is 10% and I/O utilization is 90%

- With happens under FCFS scheduling policy?
  - Once A or B get in, keep CPU for two weeks ⇒ poor avg. response time

- What about RR or SRTF?
  - Easier to see with a timeline
Mix of CPU and I/O Bound Tasks: FCFS vs. RR vs. SRTF (cont.)

RR with 40ms time quantum

CPU

A | B | A | B

I/O

I/O Utilization: ~11%

RR with 1ms time quantum

CPU

C, A, B, A, B, A, B, C, A, B, A, ...

I/O

I/O Utilization: ~82%

SRTF

CPU

A | A | A | A | A | A | A | A | A | A | A | A | A | A

I/O

I/O Utilization: ~90%
Downsides of SRTF

• **Starvation**: Large tasks may never run if short ones keep coming

• **Overhead**: Short tasks preempt long ones ⇒ too many context switches

• **Unfair**: Large tasks are penalized, there is high variance in response time

• **Impractical**: We need to somehow predict future (but how?)
  • Some systems ask users
    • When you submit your task, you have to say how long it will take
    • Users could maliciously misreport length of their task
    • E.g., would it work if a supermarket uses SJF?
      • Customers could game the system: come with one item at a time
    • To prevent cheating, systems may kill tasks if they take too long
  • It’s hard to predict task’s runtime even for non-malicious users
Predicting Length of Next CPU Burst

- **Adaptive**: Dynamically make predictions based on past behavior
  - Works because programs have predictable behavior
    - If program was I/O bound in past, it'll likely be I/O bound in future
    - If behavior were random, this approach wouldn’t help

- **Example**: Use estimator function on previous bursts
  - Let $t_{n-1}, t_{n-2}, t_{n-3}, \ldots, t_1$ 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 any time series estimator (e.g., Kalman filters, etc.)
  - For instance, exponential averaging $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ with $(0 < \alpha \leq 1)$
Aside: Application Types

• Can we use past burst times to identify application types?
• Consider mix of interactive and high-throughput programs
  • How to best schedule them?
  • How to recognize one from the other?
    • Do you trust applications to say that they are “interactive”?
  • Should you schedule the set of applications identically on servers, workstations, pads, and cellphones?
Aside: Application Types (cont.)

• Assumptions encoded into many schedulers
  • Applications that sleep a lot and have short bursts must be interactive
    • Give them high priority
  • Applications that compute a lot must be high-throughput apps
    • Give them lower priority, since they won’t notice intermittent bursts from interactive applications

• In general, it is hard to characterize applications
  • What about applications that sleep for a long time, and then compute for a long time?
  • What about applications that must run under all circumstances
SRTF Final Notes

• Bottom line, we can’t really know how long tasks will take
  • However, we can use SRTF as yardstick for measuring other policies
  • Optimal, so we can’t do any better

• Pros & cons of SRTF
  • Optimal (average response time) (+)
  • Hard to predict future (-)
  • Too many context switches (-)
  • Unfair (-)
Strict Priority Scheduling

- Execution plan
  - Always execute highest-priority runnable tasks to completion
  - Each queue can be threaded in RR with some time-quantum

- Notice any problems?
  - **Starvation**: Lower priority tasks don't get to run because higher priority tasks
  - **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
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…
• Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc.)
  • long running tasks may never get any CPU time
  • In Multics, shut down machine, found 10-year-old task
• One approach: Give each queue some fraction of CPU
  • What if there are 100 short tasks and only one long task?
    • Like express lanes in a supermarket, sometimes express lanes get so long, get better service by going into one of other lines
• Another approach: Increase priority of tasks 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 task gets CPU time, so everyone increases in priority ⇒ Interactive tasks suffer
• Tradeoff: Fairness is usually gained by hurting average response time!
Multi-Level Feedback Queue 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 task's priority as follows (details vary)
  - Task starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)
Multi-Level Feedback Queue Scheduling (cont.)

- Result approximates SRTF
  - CPU bound tasks drop like a rock
  - Short-running I/O bound tasks stay near top

- Scheduling must be done between queues
  - Fixed priority scheduling
    - Serve all from highest priority, then next priority, etc.
  - Time slicing
    - Each queue gets fraction of CPU time
    - E.g., 70% to highest, 20% next, 10% lowest
• **Countermeasure**: user action that foil intent of OS designers
  • For multilevel feedback, put simple I/O’s to keep task’s priority high
  • Example of MIT Othello Contest
    • Cheater put printf’s, ran much faster than competitors!
    • Of course, if everyone did this, wouldn’t work!
Lottery Scheduling

- Give each task some number of lottery tickets
- On each time slice, randomly pick a winning ticket
- On average, CPU time is proportional to # of tickets given to task
- How to assign tickets?
  - Give tasks tickets proportional to their priorities
  - To approximate SRTF, give short tasks more and long tasks fewer
  - To avoid starvation, give every task at least one ticket (everyone makes progress)
- Compared to strict priority scheduling, lottery scheduling behaves gracefully as load changes
  - Adding or deleting one task affects all tasks proportionally, independent of how many tickets each task possesses
Lottery Scheduling Example

• Assume short tasks get 10 tickets, long tasks get 1 ticket

<table>
<thead>
<tr>
<th># short tasks/ # long tasks</th>
<th>% of CPU each short tasks gets</th>
<th>% of CPU each long tasks 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 tasks to give reasonable response time?
  • If load average is 100, hard to make progress
  • One approach is to log some users out
Max-Min Fair (MMF) Scheduling

- Always choose task with lowest accumulated CPU time so far
  - If chosen task doesn’t have CPU burst, schedule second lowest …
  - Break ties randomly if multiple tasks equally have lowest CPU time
- Goal is to give each task equal share of CPU time
  - With $N$ runnable threads, each thread should get $1/N$th of CPU time
- At any time $t$ we want to have
• Strict MMF causes too many context switches
  • It effectively turns to running one instruction of each task

• Relaxed MMF runs task with lowest accumulated CPU time for fixed time quantum before choosing next task

• Notice any problem?
  • Fixed quantum leads to poor response time as # of tasks increases
MMF Scheduling (cont.)

- Solution: Dynamically change time quantum
- **Target latency:** Time interval during which all tasks should run at least once
- Time quantum = Target latency / N
  - E.g., with 20ms target latency and 4 threads, time quantum is 5ms
- Notice any problem?
  - With 20ms target latency and 200 threads, time quantum becomes 0.1ms
  - Recall RR: Large context switching overhead if time quantum gets too small
- **Minimum granularity:** Minimum length of any time quantum
  - E.g., with target latency 20ms, 1ms minimum granularity, and 200 processes, time quantum is 1ms
Weighted Max-Min Fair Scheduling

- What if we want to give more to some and less to others (proportional share)?
- **Key Idea**: Assign weight $w_i$ to each thread $i$
- MMF uses single time quantum for all tasks

$$Q = \frac{\text{Target latency}}{N}$$

- Weighted MMF uses different time quanta for different tasks

$$Q_i = \frac{w_i \times \text{Target latency}}{\sum_{j=1}^{N} w_j}$$

- E.g., with 20ms target latency, 1ms minimum granularity, and 2 threads: A with weight 1 and B with weight 4
  - Time quantum for A is 4 ms
  - Time quantum for B is 16 ms
Weighted MMF Scheduling (cont.)

- Also track threads’ *virtual runtime* rather than their true wall-clock runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly
- Linux *Completely Fair Scheduler* deploys very similar ideas
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-threading (… 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)
Real-Time Workload Characteristics

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

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

  T1
  T2
  T3
  T4

  Missed Deadline!

  Time
Earliest Deadline First (EDF)

- Tasks are periodic with period $P$ and computation $C$ in each period: $(P, C)$
- Preemptive priority-based dynamic scheduling
- Tasks’ (current) priority is based on how close their deadline is
- Scheduler always schedules active task with closest deadline

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$(4, 1)$</td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td>$(5, 2)$</td>
<td></td>
</tr>
<tr>
<td>$T_3$</td>
<td>$(7, 2)$</td>
<td></td>
</tr>
</tbody>
</table>

![Chart showing Earliest Deadline First (EDF)]
EDF: Feasibility Testing

- Even EDF won't work if you have too many tasks.
- For $n$ periodic tasks with computation time $C_i$ and deadline and period $D_i$, feasible schedule exists if

$$\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1$$
How to Evaluate Scheduling Algorithms?

- Deterministic modeling
  - Take predetermined workload and compute performance of each algorithm
- 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
Starvation and Sample Bias

• Suppose you want to compare scheduling policies
  • Create some infinite sequence of arriving tasks
  • Start measuring
  • Stop at some point
  • Compute ART for finished tasks between start and stop

• Is this valid or invalid?
  • SJF and FCFS would complete different sets of tasks
    • Their ARTs are not directly comparable
    • E.g., suppose you stopped at any point in FCFS vs. SJF slide
Solutions for Sample Bias

• For both systems, measure for long enough that 
  # of completed tasks >> # of uncompleted tasks

• Start and stop system in idle periods
  • Idle period: no work to do
  • If algorithms are work-conserving, both will complete the 
    same set of tasks
## Choosing the Right Scheduling Algorithm

### Table: Criteria and Algorithms

<table>
<thead>
<tr>
<th>I Care About:</th>
<th>Then Choose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Throughput</td>
<td>FCFS</td>
</tr>
<tr>
<td>Avg. Response Time</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>I/O Throughput</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>Fairness (CPU Time)</td>
<td>Linux CFS</td>
</tr>
<tr>
<td>Fairness – Wait Time to Get CPU</td>
<td>Round Robin</td>
</tr>
<tr>
<td>Meeting Deadlines</td>
<td>EDF</td>
</tr>
<tr>
<td>Favoring Important Tasks</td>
<td>Priority</td>
</tr>
</tbody>
</table>
Outline

• Definitions
  • Response time, throughput, scheduling policy, …

• Uniprocessor policies
  • FCFS, SJF/SRTF, Round Robin, …
  • Real-time scheduling

• Multiprocessor policies
  • Oblivious scheduling, gang scheduling, …
Multicore Processor Scheduling

• There could be one ready queue for all cores

• Notice any problems?
  • Single bottleneck: Contention for ready queue’s lock
  • Limited cache reuse: Lack of data locality as tasks get scheduled on different cores

• Solution: each core has its own private ready queue

• Notice any problems?
  • Load balancing: Some cores might be idle while tasks pile up on others ready queues

• One solution: Work stealing
  • Idle cores steal waiting task from busy ones
Processor Affinity

• When task run on core, cache contents of that core stores recent memory accesses by that task
• This is referred to as core affinity of tasks
• Load balancing may affect core affinity as task migrate between cores
• Performance of migrated task suffers because it loses contents of what it had in cache of the core it was moved off of
  • Migration is justified only if performance loss is less than waiting time
• Soft affinity: OS tries to keep tasks on same core, but no guarantees
• Hard affinity: OS allows tasks to specify set of cores they may run on
NUMA and CPU Scheduling

- Uniform memory access (UMA): Cores experience same, uniform access time to any memory module

- Non-uniform memory access (NUMA): Cores access their local memory modules faster than remote memory modules

- If OS is NUMA-aware, it will assign memory closes to core that task is running on
Scheduling Multithreaded Programs

• So far, we assumed that there is one thread per program
• Now, consider scheduling multithreaded programs on multicore processor
• At any given time, multiple threads from same program could be running

   Core 1        Core 2        Core 3
       P1.1        P2.1        P1.2
       P2.3        P3.2        P1.3
       P3.1        P2.4        P2.2

Px.y: thread y in process x

• **Oblivious scheduling**: Cores independently schedule threads in their queue
  • Each thread is treated as independent task

  • What happens if one thread gets time-sliced while others are still running?
    • Assuming program uses locks and condition variables, it will still be correct
    • Performance, however, could suffer if threads actually depend on one another
Problem with Oblivious Scheduling: Bulk Synchronous Delay

- Data parallelism is common programming design pattern (e.g., Google MapReduce)
  - Data is split into roughly equal sized chunks
  - Chunks are processed independently on different cores
  - Once all chunks are processed, cores synchronize and communicate their results to next stage of computation
Problem with Oblivious Scheduling: Bulk Synchronous Delay (cont.)

- At each step, computation is limited by the slowest task
- If task is preempted on one core, its work is delayed, stalling all other cores
Problem with Oblivious Scheduling: Producer-Consumer Delay

- Producer-consumer design pattern is also very common
- Preempting a thread on one core stalls all others in the chain
- Some other problems with oblivious scheduling
  - Preempting a thread on the critical path will slow down the entire process
  - Preempting lock holder stalls others until lock holder is re-scheduled
Gang Scheduling

- Time is divided into equal intervals
- Threads from same process are scheduled at beginning of each interval
- Notice any problems?
  - CPU cycles are waisted when threads have different lengths
  - Some cores remain idle when a process doesn’t have enough tasks for all cores
Space Sharing

- Each process is assigned a subset of cores
- Minimizes processor context switches
How Many Cores Does a Process Need?

- There are overheads
  - E.g., creating extra threads, synchronization, communication
- Overheads shift the curve down
Amdahl’s Law

[ G. Amdahl 1967 ]

- Architects use it to estimate upper bounds on speedups

\[
\text{Speedup}(x) = \frac{T_1}{T_x} = \frac{T_1}{(1 - F')T_1 + \frac{FT_1}{x}} = \frac{x}{x(1 - F') + F}
\]
Amdahl’s Law (cont.)

[G. Amdahl 1967]

\[
\text{Speedup}(x) = \frac{T_1}{T_x} = \frac{T_1}{(1 - F)T_1 + \frac{FT_1}{x}} = \frac{x}{x(1 - F') + F}
\]
What Portion of Code is Parallelizable?
[Allen Karp and Horace Flatt 1990]

• Expert programmers may not know!
• Fortunately, we can measure speedup

\[ s(x) = x(1 - F) + F \]

Karp-Flatt Metric

\[ F = \left( 1 - \frac{1}{s(x)} \right) \left( 1 - \frac{1}{x} \right) \]
A Final Word On Scheduling

- When do details of scheduling policy and fairness really matter?
  - When there aren’t enough resources to go around

- When should you simply buy faster cores?
  (Or network link, or expanded highway, or …)
  - 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 you need X fully utilized core, but usually you will have to buy more than X because response time goes to infinity as utilization approaches 100%

- Interesting implication of this curve
  - Most scheduling algorithms work fine in linear portion of curve, fail otherwise
  - Argues for buying faster resources when hit knee of curve
Summary (1 of 2)

- **First-Come, First-Served (FCFS)**
  - Threads are served in the order of their arrival

- **Round-Robin (RR)**
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads

- **Shortest Task First (SJT) / Shortest Remaining Time First (SRTF):**
  - Run whatever task that has the least amount of computation to do/least remaining amount of computation to do

- **Multi-level Feedback Queue (MFQ)**
  - Multiple queues of different priorities and scheduling algorithms

- **Lottery Scheduling**
  - Give each thread a priority-dependent number of tickets
Summary (2 of 2)

• **Max-Min Fair (MMF)**
  • Give each task equal share of CPU time

• **Real-Time Scheduling**
  • Need to meet a deadline, predictability essential

• **Oblivious Scheduling**
  • Each core schedules its own threads

• **Gang Scheduling**
  • Schedule tasks from same process at the same time

• **Space Sharing**
  • Give each process some number of cores
Questions?
Acknowledgment

• Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny