Question: dispatcher can choose any thread on the ready queue to run; how to decide and which to choose?

- Depends on scheduling policy goals
- **minimize response time**: elapsed time to do an operation (or job)
  - Response time is what the user sees: elapsed time to
    - echo a keystroke in editor
    - compile a program
    - run a large scientific problem

- **maximize throughput**: operations (jobs) per second
  - two parts to maximizing throughput
    - minimize overhead (for example, context switching)
    - efficient use of system resources (not only CPU, but disk, memory, etc.)

- **fair**: share CPU among users in some equitable way
First Come First Served

- Example:
  
<table>
<thead>
<tr>
<th>Process</th>
<th>Exec. Time</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>

- Suppose that the processes arrive in the order: P₁, P₂, P₃

  The schedule is:

  
<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

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

FCFS scheduling (cont’d)

- Suppose that the processes arrive in the order
  
  P₂, P₃, P₁

  The time chart for the schedule is:

  
<table>
<thead>
<tr>
<th></th>
<th>P₂</th>
<th>P₃</th>
<th>P₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

  - Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average response time: (30 + 3 + 6)/3 = 13

  - FCFS Pros: simple; Cons: short jobs get stuck behind long jobs
Shortest-Job-First (SJF)

- Associate with each process the length of its exec. time
  - Use these lengths to schedule the process with the shortest time

- Two schemes:
  - Non-preemptive – once given CPU it cannot be preempted until completes its quota.
  - Preemptive – if a new process arrives with less work than the remaining time of currently executing process, preempt.

- SJF is optimal but unfair
  - Pros: gives minimum average response time
  - Cons: long-running jobs may starve if too many short jobs;
  - Difficult to implement (how do you know how long job will take)

Non-preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>8</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

- Average waiting time = (0 + 6 + 3 + 7)/4 = 4
- Average response time = (7 + 10 + 4 + 11)/4 = 8
Example of preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

```

Example of preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

```

Average waiting time = (9 + 1 + 0 + 2)/4 = 3
Average response time = (16 + 5 + 1 + 6)/4 = 7

Alternating CPU and I/O Bursts

- CPU-I/O Burst Cycle
- CPU burst distribution

```

Alternating CPU and I/O Bursts

- CPU-I/O Burst Cycle
- CPU burst distribution

```
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum). After time slice, it is moved to the end of the ready queue.
  
  Time Quantum = 10 - 100 milliseconds on most OS

- \( n \) processes in the ready queue; time quantum is \( q \)
  
  - each process gets \( 1/n \) of the CPU time in \( q \) time units at once.
  
  - no process waits more than \((n-1)q\) time units.
  
  - each job gets equal shot at the CPU

- Performance
  
  - \( q \) large \(\Rightarrow\) FCFS
  
  - \( q \) too small \(\Rightarrow\) throughput suffers. Spend all your time context switching, not getting any real work done

RR with time quantum = 20

<table>
<thead>
<tr>
<th>Process</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>53</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>17</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>68</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>24</td>
</tr>
</tbody>
</table>

- The time 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>37</td>
<td>57</td>
<td>77</td>
<td>97</td>
<td>117</td>
<td>121</td>
<td>134</td>
<td>154</td>
<td>162</td>
</tr>
</tbody>
</table>
```

- Typically, higher average turnaround than SJF, but better fairness.
RR vs. FCFS vs. SJF

- Three tasks A, B, C
  - A and B both CPU bound (can run for a week)
  - C is I/O bound: loop 1 ms CPU followed by 10 ms disk I/O
    - running C by itself gets 90% disk utilization

- with FIFO?
- with RR (100 ms time slice):
  - What is the disk utilization? 10ms of disk operation every 200ms
  - How much does C have to wait after I/O completes? 190 ms
- with RR (1 ms time slice):
  - What is the disk utilization? 10ms of disk operation every 11-12 ms
  - How much does C have to wait after I/O completes? 0 or 1 ms

Knowledge of future

- Problem: SJF or STCF require knowledge of the future
- How do you know how long program will run for?

- Option 1: ask the user
  - When you submit the job, say how long it will take
  - If your job takes more than that, jobs gets killed. (Hard to predict usage in advance.)

- Option 2:
  - Use past to predict future
  - If program was I/O bound in the past, likely to remain so
  - Favor jobs that have been at CPU least amount of time
Multilevel queue

- Ready queue is partitioned into separate queues:
  - Each with different priority
- OS does RR at each priority level
  - Run highest priority jobs first
  - Once those finish, run next highest priority etc
  - Round robin time slice increases (exponentially) at lower priorities
- Adjust each job's priority as follows:
  - Job starts in highest priority queue
  - If time slice is fully used when process is run, drop one level
  - If it is not fully used, push up one level
- CPU bound jobs drop like a rock, while short-running I/O bound jobs stay near top
- Still unfair – long running jobs may never get the CPU
- Could try to strategize!

Handling dependencies

- Scheduling = deciding who should make progress
  - Obvious: a thread's importance should increase with the importance of those that depend on it.
  - Naïve priority schemes violate this ("Priority inversion")
- Example: T1 at high priority, T2 at low
  - T2 acquires lock L. T1 tries to acquire the same lock.
- “Priority donation”
  - Thread's priority scales w/ priority of dependent threads
  - Works well with explicit dependencies
Lottery Scheduling

- Problem: this whole priority thing is really ad hoc.
  - How to ensure that processes will be equally penalized under load?
  - How to deal with priority inversion?

- Lottery scheduling
  - give each process some number of tickets
  - each scheduling event, randomly pick ticket
  - run winning process
  - to give P n% of CPU, give it ntickets * n%

- How to use?
  - Approximate priority: low-priority, give few tickets, high-priority give many
  - Approximate SJF: give short jobs more tickets, long jobs fewer. If job has at least 1, will not starve

Lottery Scheduling Example

- Add or delete jobs (& their tickets) affects all jobs proportionately
  - short job: 10 tickets; long job: 1 ticket

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

- Easy priority inversion:
  - Donate tickets to process you’re waiting on.
  - Its CPU% scales with tickets of all waiters.
Other notes

- Client-server:
  - Server has no tickets of its own
  - Clients give server all of their tickets during RPC
  - Server’s priority is sum of its active clients
  - Server can use lottery scheduling to give preferential service

- Ticket inflation: dynamic changes in priorities between trusting programs

- Currency:
  - Set up an exchange rate across groups
  - Can print more money within a group
  - Allows independent scheduling properties

- Compensation tickets
  - What happens if a thread is I/O bound and regularly blocks before its quantum expires?
  - If you complete fraction f, your tickets are inflated by 1/f