Preemptive Scheduling Policies

- In *preemptive scheduling*, server can switch to next request before completing current one
  - Preempted request is put back into pending list
  - Its servicing is resumed when it is scheduled again
- A request may be scheduled many times before it is completed
  - Larger scheduling overhead than with nonpreemptive scheduling
- Used in multiprogramming and time-sharing OSs

---

Round-Robin Scheduling with Time-Slicing (RR)

<table>
<thead>
<tr>
<th>Time of scheduling</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>c</th>
<th>t_x</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position of P_1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processes in ready queue P_2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1 implies P_3) P_3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>6.00</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>Head of queue P_4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process scheduled P_5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ t_x = 7.4 \text{ seconds}, \bar{w} = 2.32 \]

*In this example, \( \delta = 1 \)*

Figure 7.5 Scheduling using the round-robin policy with time-slicing (RR).
Example: Variation of Response Time in RR Scheduling

- At small values of $\delta$, $rt$ for a request may be higher for smaller values of $\delta$

<table>
<thead>
<tr>
<th>Time slice</th>
<th>5 ms</th>
<th>10 ms</th>
<th>15 ms</th>
<th>20 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average $rt$ for first subrequest (ms)</td>
<td>248.5</td>
<td>186</td>
<td>206.5</td>
<td>121</td>
</tr>
<tr>
<td>Average $rt$ for subsequent subrequest (ms)</td>
<td>270</td>
<td>230</td>
<td>230</td>
<td>210</td>
</tr>
<tr>
<td>Number of scheduling decisions</td>
<td>600</td>
<td>300</td>
<td>300</td>
<td>150</td>
</tr>
<tr>
<td>Schedule length (ms)</td>
<td>4200</td>
<td>3600</td>
<td>3600</td>
<td>3300</td>
</tr>
<tr>
<td>Overhead (percent)</td>
<td>29</td>
<td>17</td>
<td>17</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 7.6 Performance of RR scheduling for different values of $\delta$.

Least Completed Next (LCN)

LCN policy schedules the process that has consumed the least amount of CPU time

<table>
<thead>
<tr>
<th>Process scheduled</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time consumed by processes</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$c$</td>
<td>8.8 seconds, $\overline{w} = 2.72$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Issues:
- Short processes will finish ahead of long processes
- Starves long processes of CPU attention
- Neglects existing processes if new processes keep arriving in the system

Figure 7.7 Scheduling using the least completed next (LCN) policy.
Shortest Time to Go (STG)

STG policy schedules a process whose remaining CPU time requirement is the smallest

<table>
<thead>
<tr>
<th>Time of scheduling</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remaining CPU time requirement of a process</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process scheduled</td>
<td>P₁</td>
<td>P₂</td>
<td>P₃</td>
<td>P₄</td>
<td>P₅</td>
<td>P₆</td>
<td>P₇</td>
<td>P₈</td>
<td>P₉</td>
<td>P₁₀</td>
<td>P₁₁</td>
<td>P₁₂</td>
<td>P₁₃</td>
<td>P₁₄</td>
<td>P₁₅</td>
<td>P₁₆</td>
</tr>
</tbody>
</table>

Since it is analogous to the SRN policy, long processes might face starvation.

Scheduling in Practice

- To provide a suitable combination of system performance and user service, OS has to adapt its operation to the nature and number of user requests and availability of resources
  - A single scheduler using a classical scheduling policy cannot address all these issues effectively
- Modern OSs employ several schedulers
  - Up to three schedulers
- Some of the schedulers may use a combination of different scheduling policies
Long-, Medium-, and Short-Term Schedulers

- These schedulers perform the following functions:
  - **Long-term**: Decides when to admit an arrived process for scheduling, depending on:
    - Nature (whether CPU-bound or I/O-bound)
    - Availability of resources
    - Kernel data structures, swapping space
  - **Medium-term**: Decides when to swap out a process from memory and when to load it back, so that a sufficient number of *ready* processes are in memory
  - **Short-term**: Decides which *ready* process to service next on the CPU and for how long
    - Also called the *process scheduler*, or *scheduler*

Example: Long, Medium-, and Short-Term Scheduling in a Time-Sharing OS

![Diagram showing the flow of processes through Long-, Medium-, and Short-term schedulers](image)
Scheduling Data Structures and Mechanisms

- Interrupt servicing routine invokes context save
- Dispatcher loads two PCB fields - PSW and GPRs - into CPU to resume operation of process

Priority-Based Scheduling

- Overhead depends on number of distinct priorities, not on the number of ready processes
- Can lead to starvation of low-priority processes
  - Aging can be used to overcome this problem → process priorities are dynamic
- Can lead to priority inversion
  - Addressed by using the priority inheritance protocol
Round-Robin Scheduling with Time-Slicing

- Can be implemented through a single list of PCBs of ready processes
  - List is organized as a queue
- Scheduler removes first PCB from queue and schedules process described by it
  - If time slice elapses, PCB is put at the end of queue
  - If process starts I/O operation, its PCB is added at end of queue when its I/O operation completes
- PCB of a ready process moves toward the head of the queue until the process is scheduled

Multilevel Scheduling

- Combines priority-based scheduling and round-robin scheduling
- Scheduler maintains a number of ready queues; a priority and a time slice are associated with each ready queue
  - RR scheduling with time slicing is performed within it
  - High priority queue has a small time slice
    - Good response times for processes
  - Low priority queue has a large time slice
    - Low process switching overhead
- A process at the head of a queue is scheduled only if the queues for all higher priority levels are empty
  - Scheduling is preemptive
Multilevel Scheduling (continued)

- Priorities are static
  - Cannot handle a change in the computational or I/O behavior of a process
  - Cannot prevent starvation of processes in low priority levels
  - Cannot employ the priority inheritance protocol to overcome priority inversion

Multilevel Adaptive Scheduling

- Also called multilevel feedback scheduling
- Scheduler varies priority of a process so it receives a time slice consistent with its CPU requirement
- Scheduler determines “correct” priority level for a process by observing its recent CPU and I/O usage
  - Moves the process to this level
- Example: CTSS, a time-sharing OS for the IBM 7094 in the 1960s
  - Eight-level priority structure (0 is highest-priority level)
  - Level n has a time slice of $0.5 \times 2^n$ CPU seconds
  - If a process completely used up the time slice at its current priority level, it was demoted to the next higher numbered level
  - If a process spent more than a minute in ready state in its current priority level without obtaining any CPU service, it was promoted to the next lower numbered level
Fair Share Scheduling

- Fair share: fraction of CPU time to be devoted to a group of processes from same user or application
- Ensures an equitable use of the CPU by processes belonging to different users or different applications
- *Lottery scheduling* is a technique for sharing a resource in a probabilistically fair manner
  - Tickets are issued to applications (or users) on the basis of their fair share of CPU time
  - To allocate a CPU time slice, the scheduler holds a lottery in which only tickets of ready processes participate
  - The process holding the winning ticket is scheduled