Module 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU burst distribution
Alternating Sequence of CPU And I/O Bursts

- load store
- add store
- read from file
- wait for I/O
- store increment
- index
- write to file
- wait for I/O
- load store
- add store
- read from file
- wait for I/O
Histogram of CPU-burst Times
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.
- Scheduling under 1 and 4 is nonpreemptive.
- All other scheduling is preemptive.
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running.
Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
First-Come, First-Served (FCFS) Scheduling

- 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 that the processes arrive in the order: $P_1, P_2, P_3$
  
  The Gantt Chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

\[ P_2, P_3, P_1. \]

• The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<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>
<td>30</td>
</tr>
</tbody>
</table>

• Waiting time for \( P_1 = 6; P_2 = 0; P_3 = 3 \)
• Average waiting time: \( (6 + 0 + 3)/3 = 3 \)
• Much better than previous case.
• *Convoy effect* short process behind long process
Shortest-Job-First (SJR) Scheduling

• Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

• Two schemes:
  – nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  – Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).

• SJF is optimal – gives minimum average waiting time for a given set of processes.
Example of Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

- Average waiting time = \((0 + 6 + 3 + 7)/4 - 4\)
### Example of Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

```
<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$P_2$</td>
<td>$P_3$</td>
<td>$P_2$</td>
<td>$P_4$</td>
<td>$P_1$</td>
<td></td>
</tr>
</tbody>
</table>
```

- Average waiting time = $(9 + 1 + 0 + 2)/4 - 3$
Determining Length of Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.
  1. $t_n =$ actual length of $n^{th}$ CPU burst
  2. $\tau_{n+1} =$ predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq 1$
  4. Define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$
Examples of Exponential Averaging

- \( \alpha = 0 \)
  - \( \tau_{n+1} = \tau_n \)
  - Recent history does not count.

- \( \alpha = 1 \)
  - \( \tau_{n+1} = t_n \)
  - Only the actual last CPU burst counts.

- If we expand the formula, we get:
  \[
  \tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_n - 1 + \ldots \\
  + (1 - \alpha)^j \alpha t_n - 1 + \ldots \\
  + (1 - \alpha)^{n-1} t_n \tau_0
  \]

- Since both \( \alpha \) and \( 1 - \alpha \) are less than or equal to 1, each successive term has less weight than its predecessor.
Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem ≡ Starvation – low priority processes may never execute.
- Solution ≡ Aging – as time progresses increase the priority of the process.
Round Robin (RR)

- Each process gets a small unit of CPU time \((\text{time quantum})\), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are \(n\) processes in the ready queue and the time quantum is \(q\), then each process gets \(1/n\) of the CPU time in chunks of at most \(q\) time units at once. No process waits more than \((n-1)q\) time units.

- Performance
  - \(q\) large \(\Rightarrow\) FIFO
  - \(q\) small \(\Rightarrow\) \(q\) must be large with respect to context switch, otherwise overhead is too high.
Example: RR with Time Quantum = 20

<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>17</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:

```
0 20 37 57 77 97 117 121 134 154 162
```

- Typically, higher average turnaround than SJF, but better response.
How a Smaller Time Quantum Increases Context Switches
Turnaround Time Varies With The Time Quantum

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>7</td>
</tr>
</tbody>
</table>

![Graph showing the relationship between average turnaround time and time quantum]
Multilevel Queue

- Ready queue is partitioned into separate queues: foreground (interactive) and background (batch).

- Each queue has its own scheduling algorithm:
  - Foreground – RR
  - Background – FCFS

- Scheduling must be done between the queues:
  - Fixed priority scheduling; i.e., serve all from foreground then from background. Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS
Multilevel Queue Scheduling

- highest priority
  - system processes
- interactive processes
- interactive editing processes
- batch processes
- student processes

lowest priority
Multilevel Feedback Queue

• A process can move between the various queues; aging can be implemented this way.

• Multilevel-feedback-queue scheduler defined by the following parameters:
  – number of queues
  – scheduling algorithms for each queue
  – method used to determine when to upgrade a process
  – method used to determine when to demote a process
  – method used to determine which queue a process will enter when that process needs service
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$ – time quantum 8 milliseconds
  - $Q_1$ – time quantum 16 milliseconds
  - $Q_2$ – FCFS

- Scheduling
  - A new job enters queue $Q_0$ which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q_1$.
  - At $Q_1$ job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q_2$. 
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- *Homogeneous processors* within a multiprocessor.
- *Load sharing* \[\text{VERSUS affinity scheduling used to preserve CPU cache locality!!!}\]
- *Symmetric Multiprocessing (SMP)* – each processor makes its own scheduling decisions.
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.
Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones.
Dispatch Latency
Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP.

- Global Scheduling – How the kernel decides which kernel thread to run next.
Solaris 2 Scheduling

- Global priority: highest to lowest
- Scheduling order: first to last
- Class specific priorities: real time, system, interactive & time sharing
- Scheduler classes: kernel threads of real time LWP's, kernel service threads, kernel threads of interactive & time sharing LWP's
- Run queue: kernel threads of real time LWP's, kernel service threads, kernel threads of interactive & time sharing LWP's
Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm.

- FIFO Queue is Used if There Are Multiple Threads With the Same Priority.
Java Thread Scheduling (cont)

JVM Schedules a Thread to Run When:

- The Currently Running Thread Exits the Runnable State.
- A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not.
Time-Slicing

- Since the JVM Doesn’t Ensure Time-Slicing, the yield() Method May Be Used:

  ```java
  while (true) {
    // perform CPU-intensive task
    ...
    Thread.yield();
  }
  ```

  This Yields Control to Another Thread of Equal Priority.
### Thread Priorities

- **Thread Priorities:**

<table>
<thead>
<tr>
<th>Priority</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread.MIN_PRIORITY</td>
<td>Minimum Thread Priority</td>
</tr>
<tr>
<td>Thread.MAX_PRIORITY</td>
<td>Maximum Thread Priority</td>
</tr>
<tr>
<td>Thread.NORM_PRIORITY</td>
<td>Default Thread Priority</td>
</tr>
</tbody>
</table>

Priorities May Be Set Using `setPriority()` method:

```java
def setPriority(Thread.NORM_PRIORITY + 2);```

Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queuing models
- Implementation
Evaluation of CPU Schedulers by Simulation