Chapter 5: CPU Scheduling

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

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
  - Including 2?
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

- 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>0</th>
<th>6</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_2 )</td>
<td></td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>( P_3 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Waiting time for \( P_1 \) = 0; \( P_2 \) = 3; \( P_3 \) = 3
- Average waiting time: \((0 + 3 + 3)/3 = 2\)
- Much better than previous case
- Convoy effect – short process behind long process

Shortest-Job-First (SJF) 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 known 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>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)
- 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>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 = \((9 + 1 + 0 + 2)/4 = 3\)

**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. \( t_1 \) - actual length of CPU burst
2. \( t_{\text{next}} \) - predicted next CPU burst
3. \( t_{\text{next}} = t_1 - t_0 = n \cdot \alpha \cdot t_0 \)

**Examples of Exponential Averaging**

- \( \alpha = 0 \)
- \( t_{\text{next}} = t_0 \)
- Recent history does not count
- \( \alpha = 1 \)
- \( t_{\text{next}} = t_n \)
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  
  \[ t_{\text{next}} = \alpha \cdot t_1 + (1 - \alpha) \cdot \alpha \cdot t_2 + \ldots \]
  
  \[ + (1 - \alpha)^2 \cdot \alpha \cdot t_3 + \ldots \]
  
- Since both \( \alpha \) and \((1 - \alpha)\) are less than or equal to 1, each successive term has less weight than its predecessor

**Prediction of the Length of the Next CPU Burst**

**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 (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 of 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:
- Typically, higher average turnaround than SJF, but better response

Time Quantum and Context Switch Time

![Gantt chart of Round Robin scheduling]

Turnaround Time Varies With The Time Quantum

![Graph showing turnaround time vs. time quantum]

Multilevel Queue

- Ready queue is partitioned into separate queues: foreground (interactive) 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

![Diagram of multilevel queue scheduling]
**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

**Example of Multilevel Feedback Queue**

- Three queues:
  - \(Q_0\) – RR with time quantum 8 milliseconds
  - \(Q_1\) – RR 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 \(Q_2\).

**Multilevel Feedback Queues**

```
quantum = 8
```

```
quantum = 16
```

`FCFS`

**Multiple-Processor Scheduling**

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
- Load sharing
- *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

**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
Pthread Scheduling API

Operating System Examples

Solaris scheduling

Windows XP scheduling

Linux scheduling

Solaris Scheduling

Windows XP Priorities

Linux Scheduling

Two algorithms: time-sharing and real-time

Time-sharing

- Prioritized credit-based – process with most credits is scheduled next
- Credit subtracted when timer interrupt occurs
- When credit = 0, another process chosen
- When all processes have credit = 0, recredits occurs
- Based on factors including priority and history

Real-time

- Soft real-time
- Posix.1b compliant – two classes
- FCFS and RR
- Highest priority process always runs first
The Relationship Between Priorities and Time-slice length

<table>
<thead>
<tr>
<th>numeric priority</th>
<th>relative priority</th>
<th>time quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>highest</td>
<td>200 ms</td>
</tr>
<tr>
<td>99</td>
<td>real-time tasks</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>other tasks</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
</tbody>
</table>

List of Tasks Indexed According to Priorities

<table>
<thead>
<tr>
<th>active array</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>task lists</td>
</tr>
<tr>
<td>[0]</td>
<td>[0]</td>
</tr>
<tr>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td></td>
</tr>
<tr>
<td>[7]</td>
<td></td>
</tr>
<tr>
<td>[8]</td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td></td>
</tr>
<tr>
<td>[110]</td>
<td></td>
</tr>
</tbody>
</table>

Java Scheduling

- Loosely-defined scheduling policy. A thread runs until:
  1. Its time quantum expires
  2. It blocks for I/O
  3. It exits its run() method

Some systems may support preemption

Java Scheduling

- Priorities - values range from 1-10
  - MIN_PRIORITY is 1
  - NORM_PRIORITY is 5
  - MAX_PRIORITY is 10

Java Scheduling

Changing priority using setPriority()

```java
generic class HighThread implements Runnable {
    public void run() {
        Thread.currentThread().setPriority(Thread.NORM_PRIORITY + 1);  // remainder of run() method
    }
}
```

Java Scheduling

Relationship between Java and Win32 Priorities

<table>
<thead>
<tr>
<th>Java priority</th>
<th>Win32 priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MIN_PRIORITY</td>
<td>LOWEST</td>
</tr>
<tr>
<td>2</td>
<td>LOWEST</td>
</tr>
<tr>
<td>3</td>
<td>BELOW_NORMAL</td>
</tr>
<tr>
<td>4</td>
<td>BELOW_NORMAL</td>
</tr>
<tr>
<td>5</td>
<td>NORM_PRIORITY</td>
</tr>
<tr>
<td>6</td>
<td>NORM_PRIORITY</td>
</tr>
<tr>
<td>7</td>
<td>ABOVE_NORMAL</td>
</tr>
<tr>
<td>8</td>
<td>ABOVE_NORMAL</td>
</tr>
<tr>
<td>9</td>
<td>HIGHEST</td>
</tr>
<tr>
<td>10</td>
<td>HIGHEST</td>
</tr>
<tr>
<td>11</td>
<td>MAX_PRIORITY</td>
</tr>
<tr>
<td>12</td>
<td>MAX_PRIORITY</td>
</tr>
</tbody>
</table>
Java Scheduling

Java thread scheduling on Solaris

Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
- Implementation

Algorithm Evaluation

Evaluation of CPU schedulers by simulation

End of Chapter 5