Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Thread Scheduling
- Operating Systems Examples
- Java Thread 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...determines best scheduling method
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

CPU burst

I/O burst

CPU burst

I/O burst

CPU burst

I/O burst

wait for I/O

wait for I/O
Histogram of CPU-burst Times

- x-axis: burst duration (milliseconds)
- y-axis: frequency

The chart shows the distribution of burst durations with a peak around 0 milliseconds and a long tail extending to 40 milliseconds with a frequency of 160.
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
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

- **Process**  |  **Burst Time**  
- $P_1$       | 24           
- $P_2$       | 3            
- $P_3$       | 3

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

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></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
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)

\[
\text{Average waiting time} = \frac{0 + 6 + 3 + 7}{4} = 4
\]
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
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
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 (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 = FIFO
  - $q$ small: $q$ 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>P₁</td>
<td>53</td>
</tr>
<tr>
<td>P₂</td>
<td>17</td>
</tr>
<tr>
<td>P₃</td>
<td>68</td>
</tr>
<tr>
<td>P₄</td>
<td>24</td>
</tr>
</tbody>
</table>

The Gantt chart is:

```
P₁  P₂  P₃  P₄  P₁  P₃  P₄  P₁  P₃  P₃
0  20  37  57  77  97  117  121  134  154  162
```

Typically, higher average turnaround than SJF, but better response
Time Quantum and Context Switch Time

(process time = 10)

quantum  |  context switches
--- | ---
12 | 0
6 | 1
1 | 9

Silberschatz, Galvin and Gagne ©2003
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
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor system
- Load sharing/balancing among multiple systems
- Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing – all processors are available to all processes
Real-Time Scheduling

- Hard real-time systems – required to complete a critical task within a guaranteed, predictable amount of time
- Soft real-time computing – requires that critical processes receive priority over less fortunate ones
## Solaris 2 Scheduling

<table>
<thead>
<tr>
<th>Global Priority</th>
<th>Scheduling Order</th>
<th>Class-Specific Priorities</th>
<th>Scheduler Classes</th>
<th>Run Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>First</td>
<td>Real Time</td>
<td>Kernel threads of real-time LWPs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td></td>
<td></td>
<td>Kernel service threads</td>
<td></td>
</tr>
<tr>
<td>Interactive and Time Sharing</td>
<td></td>
<td></td>
<td>Kernel threads of interactive and time-sharing LWPs</td>
<td></td>
</tr>
<tr>
<td>Lowest</td>
<td>Last</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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, recrating 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
Thread Scheduling

- **Local Scheduling** – How the threads library decides which thread to put onto an available LWP...programmer control
- **Global Scheduling** – How the kernel decides which kernel thread to run next
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