CPU scheduling

- Maximum CPU utilization is obtained by multiprogramming.
- CPU – I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait.
- CPU burst distribution can be seen in slide 6.3.

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.
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 all output (for time-sharing environment).

First-Come, First-Served (FCFS) Scheduling

- Example: 
  
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst 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>

  The Gantt Chart for the schedule is:

  - Waiting time for P1 = 0; P2 = 24; P3 = 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:

  - 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 (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).

- SRTF 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>

- Gantt Chart of SJF (non-preemptive):

- Average waiting time = \((0 + 6 + 3 + 7)/4 = 4\)

**Example of Preemptive SJF (SRTF)**

<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>

- Gantt Chart of SRTF (preemptive SJF):

- 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:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^n \tau_0$$

- 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 (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 $\Rightarrow$ FCFS
  - $q$ 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>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:

```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>P₂</td>
<td>P₃</td>
<td>P₄</td>
<td>P₁</td>
<td>P₂</td>
<td>P₃</td>
<td>P₃</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>20</td>
<td>37</td>
<td>57</td>
<td>77</td>
<td>97</td>
<td>117</td>
<td>134</td>
<td>154</td>
</tr>
</tbody>
</table>
```

- Typically, higher average turnaround than SJF, but better response.

How a Smaller Time Quantum Increases Context Switches

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; e.g., 80% to foreground in RR, 20% to background in FCFS

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

Multilevel Feedback Queues

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 RR. 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 RR 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.
- Symmetric multiprocessing – all of the processors run an identical copy of the same kernel.
  - Implemented on Homogeneous processors.
  - A mechanism for data locking should be implemented.
  - All of UNIX versions, WindowsNT and Windows2000 are Symmetric.
- Asymmetric multiprocessing – only one processor running the scheduler which accesses the system data structures.
  - Processors can be unequal.
  - alleviating the need for data sharing.
  - Used on large clusters or Supercomputers.
Backfilling

- Backfilling is an asymmetric multiprocessing scheduling.
- Finding free nodes for the first job in the queue. If there aren't enough free nodes, the job will have to wait.
- However, if there are enough free nodes for another job in the queue, the free nodes will be assigned to the other job.
- In order to prevent starvation of jobs which use a higher number of nodes, we will require that a job selected out of order will not delay the execution of jobs which have a higher location in the queue.

Gang Scheduling

- Gang Scheduling is also an asymmetric multiprocessing scheduling.
- Gang Scheduling gives a job all the processors at the same time (or none for a time slice).
- Gang Scheduling slices the time in a Round-Robin manner. The jobs are put into a matrix. Each column in the matrix represents a node and each row represents a time slot. The rows are executed one by one.
- This approach tries to minimize the rows as much as possible, so each job will get more time quanta.
- Very useful when most of the jobs have a lot of communication between their processes.

Gang Scheduling (Cont.)

- Ousterhout Matrix - The jobs in successive slots are scheduled in turn in a round-robin manner.

Algorithm Evaluation

- Deterministic modeling (simulation) – takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queuing models
- Implementation
Linux Scheduling

- Linux has a multi level queue contains:
  - FIFO Real-Time.
  - RR Real-Time.
  - Non-Real-Time scheduling as described below.
- Processes may migrate from one queue to another just by an explicit command.
- The Non-Real-Time Linux scheduler partitions time into epochs.
- In each epoch, every process has a time slice of how long it may run.
- When there are no ready processes with a time slice left, a new epoch is started.
- The order within the epoch is set according to the dynamic priority.
  - The dynamic priority is a number between 100 to 140 that is set according to the nice value plus/minus bonus of 5. The bonus reflects the CPU consuming of the processes.

Linux Scheduling (cont.)

- The calculation of the time slice is based on:
  - The "nice" value (The range is -20 to 19).
  - Processes that did not use up all their previous time slice transfer half of it to the new epoch.
    - In Linux 2.6 the half of the unused time is not added.
- Linux 2.6 tries to recognize interactive processes, Essentially, tasks get an interactive credit when they sleep for a long time, and lose an interactive credit when they run for a long time.
- Linux 2.6 will give another time slice in the same epoch to an interactive process, if the interactive process uses up its initial time slice.
- In Linux 2.4 the time slices were calculated in the beginning of the epoch. In Linux 2.6 the time slice is calculated when a process leaves the CPU.

Windows Scheduling

- Windows has a multi level queue contains:
  - Real-Time.
  - Non-Real-Time scheduling as described below.
- Processes may migrate from one queue to another just by an explicit command.
- The "nice" value of Windows is called "thread's type".
  - Can be: 1) high, 2) above_normal, 3) normal, 4) below_normal, 5) low.
- There are also some dynamic rules:
  - Threads associated with the focus window get a triple quantum.
    - Multi-Media threads are not always in the focus window.
  - After an I/O wait, the priority is boosted by a factor that is inversely proportional to the speed of the I/O device. This is then decremented by one at the end of each quantum, until the original priority is reached again.
    - Keyboard and mouse are very slow; hence contribute a big boost.