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

- 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
- wait for I/O
Histogram of CPU-burst Times

![Graph showing the distribution 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

<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>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: \( \frac{(0 + 24 + 27)}{3} = 17 \)
FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order \( P_2 , P_3 , P_1 \)

- The Gantt chart for the schedule is:

```
      P_2  |   P_3   |   P_1   |
     ---+--------+--------|
   0   |   3    |   6    |
     ---+--------+--------|
          |        |   30   |
```

- 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 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)

- Average waiting time = $\frac{(9 + 1 + 0 + 2)}{4} = 3$
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. \( t_n = \text{actual length of } n^{th} \text{ CPU burst} \)
2. \( \tau_{n+1} = \text{predicted value for the next CPU burst} \)
3. \( \alpha, 0 \leq \alpha \leq 1 \)
4. Define: \[ \tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n \]
Prediction of the Length of the Next CPU Burst

\[ \alpha = \frac{1}{2} \text{ and } \tau_0 = 10 \]

<table>
<thead>
<tr>
<th>CPU burst ((t_i))</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ((\tau_i))</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Examples of Exponential Averaging

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

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

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

- 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 algorithm 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:

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P1</th>
<th>P3</th>
<th>P4</th>
<th>P1</th>
<th>P3</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>37</td>
<td>57</td>
<td>77</td>
<td>97</td>
<td>117</td>
<td>121</td>
<td>134</td>
<td>154</td>
</tr>
</tbody>
</table>
```

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

- Process time = 10
  - Quantum: 12
  - Context switches: 0
  - Process from 0 to 6
  - Quantum: 6
  - Context switches: 1
    - Process from 6 to 10
  - Quantum: 1
  - Context switches: 9
    - Process from 10 to 12
Turnaround Time Varies With The Time Quantum

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
</tbody>
</table>
Multilevel Queue

- Ready queue is partitioned into separate queues: for example 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 for background processes.
  - 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
  - Preemption?
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 promote 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 is placed at the end of queue $Q_0$. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q_1$.
  - When $Q_1$ job gains the CPU, it receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q_2$.
  - If a FCFS job gains the CPU, it will be pre-empted if a job in either $Q_0$ or $Q_1$ becomes ready
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 critical ones
Thread Scheduling

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

- Global Scheduling – How the kernel decides which kernel thread to run next
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
Linux 2.6 Scheduler

- Previous versions of Linux suffered from three deficiencies:
  - $O(n)$ complexity, Single runqueue lock on SMPs, Preemption not possible
- 2.6 scheduler designed to address these three issues
- Changes to the scheduler
  - Still have active runqueue and expired runqueue
  - When task on active runqueue uses up its time slice, it is moved to expired runqueue
  - During the move, its timeslice and priority is recalculated
  - If no tasks exist on the active runqueue for a given priority, the pointers for the active and expired runqueues are swapped
  - Scheduler simply chooses the task on the highest priority list to execute
  - Also supports dynamic task prioritization
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></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></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>● ●</td>
</tr>
<tr>
<td>[1]</td>
<td>● ● ●</td>
</tr>
<tr>
<td></td>
<td>● ● ● ●</td>
</tr>
<tr>
<td>[140]</td>
<td>●</td>
</tr>
</tbody>
</table>
Algorithm Evaluation

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

actual process execution

trace tape

CPU 10
I/O 213
CPU 12
I/O 112
CPU 2
I/O 147
CPU 173

simulation

FCFS

simulation

SJF

simulation

RR (q = 14)

performance statistics for FCFS

performance statistics for SJF

performance statistics for RR (q = 14)