CPU Scheduling
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Thread 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

- CPU burst
- I/O burst
- CPU burst
- I/O burst
Histogram of CPU-burst Times
CPU Scheduler

- Selects from among the threads in memory that are ready to execute, and allocates the CPU to one of them.

- CPU scheduling decisions may take place when a thread:
  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.
Recall the process/thread lifecycle diagram

```
new  admitted  interrupt  exit  terminated

ready  scheduler dispatch  running

waiting

1. I/O or event wait
2. Interrupt
3. I/O or event completion
4. Exit
```
Dispatcher

- Dispatcher module gives control of the CPU to the thread 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 thread and start another running
Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of threads that complete their execution per time unit
- Turnaround time – amount of time to execute a particular thread CPU burst
- Waiting time – amount of time a thread has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until it starts to execute (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>Thread</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>24</td>
</tr>
<tr>
<td>$T_2$</td>
<td>3</td>
</tr>
<tr>
<td>$T_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that the threads arrive in the order: $T_1$, $T_2$, $T_3$ at time 0

The Gantt Chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>24</td>
<td>27</td>
</tr>
</tbody>
</table>

Waiting time for $T_1 = 0$; $T_2 = 24$; $T_3 = 27$

Average waiting time: $(0 + 24 + 27)/3 = 17$
Suppose that the threads arrive in the order $T_2, T_3, T_1$ at time 0

- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>T$_2$</th>
<th>T$_3$</th>
<th>T$_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for $T_1 = 6; T_2 = 0; T_3 = 3$
- Average waiting time: $\frac{(6 + 0 + 3)}{3} = 3$
- Much better than previous case
- *Convoy effect* - short thread behind long thread
Shortest-Job-First (SJF) Scheduling

- Associate with each thread the length of its next CPU burst. Use these lengths to schedule the thread with the shortest next CPU burst.
- Two schemes:
  - non-preemptive – once CPU given to the thread it cannot be preempted until it completes its CPU burst
  - preemptive – if a new thread arrives with CPU burst length less than remaining time of current executing thread, 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 threads
Example of Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Thread</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$T_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$T_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

- Average waiting time = \( \frac{0 + 6 + 3 + 7}{4} = 4 \)
Example of Preemptive SJF

<table>
<thead>
<tr>
<th>Thread</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$T_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$T_4$</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. $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
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

\[ \alpha = \frac{1}{2} \text{ and } \tau_0 = 10 \]
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:
  \[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0 \]
- 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 thread
- The CPU is allocated to the thread with the highest priority (smallest integer ≡ highest priority)
  - preemptive
  - non-preemptive
- SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority threads may never execute
- Solution ≡ Aging – as time progresses, increase the priority of a thread
- Base priority vs dynamic priority
Round Robin (RR)

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

- If there are \( n \) threads in the ready queue and the time quantum is \( q \), then each thread gets \( \frac{1}{n} \) of the CPU time in chunks of at most \( q \) time units at once. No thread waits more than \( (n-1)q \) time units before it begins to execute.

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

<table>
<thead>
<tr>
<th>Thread</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>53</td>
</tr>
<tr>
<td>$T_2$</td>
<td>17</td>
</tr>
<tr>
<td>$T_3$</td>
<td>68</td>
</tr>
<tr>
<td>$T_4$</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

```
    T1  T2  T3  T4  T1  T3  T4  T1  T3  T3
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: 12
- Context switches: 0
- Quantum: 6
- Context switches: 1
- Quantum: 1
- Context switches: 9
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>

![Diagram showing turnaround time varies with time quantum](image_url)
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 threads.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its threads; e.g.
    - 80% to foreground in RR
    - 20% to background in FCFS
  - Preemption?
Multilevel Queue Scheduling

-最高优先级：系统进程
-交互进程
-交互编辑进程
-批处理进程
-学生进程

最优级
Multilevel Feedback Queue

- A thread 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 algorithm for each queue
  - method used to determine when to promote a thread
  - method used to determine when to demote a thread
  - method used to determine which queue a thread will enter when that thread 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 thread is placed at the end of queue $Q_0$. When it gains CPU, thread receives 8 milliseconds. If it does not finish in 8 milliseconds, thread is moved to queue $Q_1$.
  - When $Q_1$ thread 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 thread gains the CPU, it will be pre-empted if a thread 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 critical sections around system data structures
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 threads 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 – thread with most credits is scheduled next
- Credit subtracted when timer interrupt occurs
- When credit = 0, another thread chosen
- When all threads have credit == 0, recrediting occurs
  - Based on factors including priority and history

Real-time
- Soft real-time
- Posix.1b compliant – two classes
  - FCFS and RR
  - Highest priority thread always runs first
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 thread 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 threads exist on the active runqueue for a given priority, the pointers for the active and expired runqueues are swapped
  - Scheduler simply chooses the thread on the highest priority list to execute
  - Also supports dynamic thread 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>

- real-time tasks
- other tasks
## List of Tasks Indexed According to Priorities

<table>
<thead>
<tr>
<th>priority</th>
<th>task lists</th>
<th>priority</th>
<th>task lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>![Task 1]</td>
<td>[0]</td>
<td>![Task 1]</td>
</tr>
<tr>
<td>[1]</td>
<td>![Task 2]</td>
<td>[1]</td>
<td>![Task 2]</td>
</tr>
<tr>
<td></td>
<td>![Task 3]</td>
<td></td>
<td>![Task 3]</td>
</tr>
<tr>
<td></td>
<td>![Task 4]</td>
<td></td>
<td>![Task 4]</td>
</tr>
<tr>
<td>[140]</td>
<td>![Task 5]</td>
<td>[140]</td>
<td>![Task 5]</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
  - CPU 10
  - I/O 213
  - CPU 12
  - I/O 112
  - CPU 2
  - I/O 147
  - CPU 173

- Trace tape

- Simulation
  - FCFS
  - SJF
  - RR (q = 14)

- Performance statistics for FCFS
- Performance statistics for SJF
- Performance statistics for RR (q = 14)