Chapter 5
CPU Scheduling

Multiprogramming Objective
- Maximize CPU Utilization
- Maximize I/O Resource Utilization
- Fairly Allocate Memory Usage

CPU-I/O Burst Cycle
- CPU-Bound Program
  o Long CPU Bursts
  o Few, Short I/O Bursts
- I/O-Bound Program
  o Few, Short CPU Bursts
  o Many I/O Bursts

CPU Scheduler
- Short-Term Scheduler
- STS: selects PCB, i.e., process for assignment to the CPU, i.e., change of state Ready Queue → CPU
- Ready Queue contains the PCB’s of the Processes
- All PCB’s in the Ready Queue represent Processes waiting to run on the CPU

Queues may be implemented as
- FIFO Queues
- Priority Queues
- Unordered Link Lists
- Any other Data Structure required by the operational criteria

PCB’s
- normally reside in Memory
- but may be swapped out to disk by the Medium-Term Scheduler

Preemptive Scheduling

Scheduling Decisions
Process Changes State as follows
1. Running State → Wait State
   - I/O Request
   - Invocation of wait( )
2. Running State → Ready State
   - Interrupt Occurs
3. Wait State → Ready State
   - Completion of I/O
4. Running State → Termination

Non-Preemptive Scheduling
Allows only the following changes of state
1. Running State → Wait State
   - I/O Request
   - Invocation of wait( )
2. Running State → Termination

I.e., once the process gets the CPU, it keeps until the process requests a wait state or terminates
Preemptive Scheduling Impacts

- **Shared Data**
  - Process A is updating data
  - Process A is preempted by Process B
  - Process B uses data which is in an inconsistent state

- **Kernel**
  - Is busy on behalf of Process A,
    - e.g., changing kernel data for Process A,
    - e.g., modifying I/O Queues
  - Process A is preempted by Device Driver
  - Device Driver uses inconsistent data due to corrupted I/O Queue data

Unix

- before initiating a Context Switch
  - Waits for a System Call to Complete
  - An I/O block to Occur

- **Interrupts**
  - can occur at any time
  - cannot be ignored by the kernel
  - must be guarded from simultaneous usage

---

**Interrupt Processing Routine**

- Disable Interrupt Stream, i.e.,
  - Place subsequent “Disabled” Interrupts on a Stack or in a Queue
- Process the Current Interrupt
- Enable Interrupt Stream, i.e., check the Interrupt Data Structure for Suspended Interrupt Notifications
- Continue Processing Interrupts

---

**Dispatcher**

- **Effects the change of state from Ready to Running, i.e., Ready Queue → CPU**

- **Performs Context Switch**
  - In Kernel Mode
    - Copy Running Process Information → PCB
    - Copy Selected Process PCB Information → CPU
  - In User Mode
    - Starts Selected Program Running

- **Dispatch Latency**
  - Time Interval required to perform Context Switch
Scheduling Criteria

- **CPU Utilization**
  - 40% - 90%

- **Throughput**
  - number of processes that reach completion during a unit of time

- **Turnaround Time**
  - time interval from submission time to completion time

- **Wait Time**
  - total time waiting in the ready queue

- **Response Time**
  - time interval from job submission until the job first starts responding

- **Goal**
  - Maximize
    - CPU Utilization
    - Throughput
  - Minimize
    - Turnaround Time
    - Wait Time
    - Response Time

- **Implementation**
  - Optimize Average Measure
  - Optimize Maximum or Minimum Values
  - Minimize the Maximum Response Time
  - Minimize the Variance in the Response Time (interactive systems)
Scheduling Algorithms

First-Come, First-Served (FCFS) Scheduling

- Non-Preemptive Scheduling Algorithm
- Ready Queue – FIFO queue
- Linked List of PCB Blocks
- Average Waiting Time – not minimal, varies considerable – depends upon arrival times
  e.g., one CPU-Bound Process & many I/O Bound Processes
  Convoy effect
  - CPU-Bound Process issues I/O Request
    -> sits in IO Device (Wait) Queue
    I/O-Bound processes quickly finish I/O Requests
    -> I/O-Bound processes move to Ready Queue
    -> CPU sets idle
  CPU-Bound Process finally completes I/O Request
  -> CPU-Bound Process moves to Ready Queue, placed in queue after all the I/O-Bound Processes
  I/O-Bound processes quickly finish I/O Requests
  -> I/O-Bound processes move to Ready Queue
  CPU-Bound Process regains control of the CPU
  -> I/O-Bound Process sit in Ready Queue
  -> I/O Devices sit idle
Shortest Job First Scheduling or Shortest-Next-CPU-Burst Algorithm

- the process with the shortest length of the next CPU Burst is chosen from the Ready Queue
- optimal scheduling algorithm $\Rightarrow$ minimum average waiting time
- batch processing – user estimate of length of job-completion time
  - low estimate $\Rightarrow$ faster completion time
  - too low an estimate $\Rightarrow$ time-limit-completion interrupt and job termination penalty
- short-term scheduler has no way to estimate the length of the next CPU Burst

predicting the shortest next CPU Burst
- exponential average of previous (recorded) CPU Bursts
- $t_n$: length of the $n^{th}$ CPU burst
- $\tau_{n+1}$: predicted value of the next CPU burst
- $\tau_{n+1} = \alpha \times \tau_n + (1 - \alpha) \times t_n$ for $0 \leq \alpha \leq 1$
- $t_n$ represents the most recent factual information
- $\tau_n$ stores the accumulated past history
- $\alpha$ controls the relative weight of recent and past history
- If $\alpha = 0$ then $\tau_{n+1} = \tau_n$ hence recent history has no effect
- If $\alpha = 1$ then $\tau_{n+1} = t_n$ hence only the most recent CPU burst matters
- $(\alpha = 1/2) \Rightarrow$ recent history and past history are equally valued, i.e., weighted
- Initial value $\tau_0$ defined as a constant or an overall system average
- $\tau_{n+1}$ yields a weighted average with an exponentially decaying history, i.e., more recent events are given greater weight than more distant events, i.e.,

$$\tau_{n+1} = \alpha \times t_n + (1 - \alpha) \times \alpha \times t_{n-1} + \cdots + (1 - \alpha)^j \times \alpha \times t_{n-j} + \cdots + (1 - \alpha)^n \times t_0$$

where $\tau_0 = \begin{cases} 1 \\
\text{heuristic estimation of past CPU bursts}
\end{cases}$

i.e.,

$$\tau_{n+1} = \sum_{j=0}^{n+1} (1 - \alpha)^j \times \alpha \times t_{n-j}$$

- Preemptive SJF – Shortest-Remaining-Time-First Scheduling
  - new job has shorter next CPU burst estimate than the remainder of the current job's current CPU burst

- Non-preemptive SJF
  - currently running process finishes it's CPU burst regardless of the next CPU burst estimate of the new job

See Silberschatz Slides & Textbook for a discussion of

- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling
- Thread Scheduling
- Multiprocessor Scheduling
Solaris Scheduling

Priority-based Thread Scheduling
— each process belongs to one of the following classes:

- **Interactive**
  - time slicing quantum
  - dynamically alter priorities
  - 60 priority levels
  - multilevel feedback queue
  - quantum length depends upon the queue
    - higher priority ➔ smaller time slice
    - lower priority ➔ larger priority
  - CPU-bound processes ➔ lower priorities
  - I/O-bound processes ➔ higher priorities
  - processes using a Window utility are given a high priority

- **Time-Sharing** — the default class for a process
  - time slicing quantum
  - dynamically alter priorities
  - 60 priority levels
  - multilevel feedback queue
  - quantum length depends upon the queue
    - higher priority ➔ smaller time slice
    - lower priority ➔ larger priority
  - CPU-bound processes ➔ lower priorities
  - I/O-bound processes ➔ higher priorities

- **Fixed-Priority**
  - 60 priority levels
  - priorities are not dynamically altered

- **Fair-Share**
  - uses CPU Shares to make scheduling decisions
  - CPU Shares indicate an entitlement to available CPU resources
  - CPU Shares are allocated to a set of processes, i.e., a project

- **System**
  - kernel threads, e.g., scheduler, page daemon
  - priority of a system thread does not change
  - user threads running in kernel mode are not in the system class

- **Real-Time**
  - real-time threads have a guaranteed response w/in a bounded time period

- **Interrupt Threads** — 10 threads reserved for interrupt processing

---

**Dispatch Table**
used by both the Interactive and the Time-Sharing Classes

See Silberfiszatz text page 207 for a fragment of the table.

Algorithm for promoting and demoting process threads via priorities and time-quantum lengths; priorities and time-quantum lengths have an inverse relationship;

- thread uses its entire time-quantum ➔ priority is lowered
- thread returning from a wait queue ➔ priority raised to between 50-59

---

**CPU-bound process threads**
- low priority
**I/O-bound process threads**
- high priority

---

**Scheduler**
- converts class-specific priorities into global priorities
- selects thread with highest global priority to run

**Selected thread runs until it**
- blocks
- uses its time slice
- is preempted by a higher priority thread
- terminates

---

If there are multiple threads with the same global priority the scheduler uses a round-robin queue
Windows XP Scheduling
- uses a priority-based, preemptive scheduling algorithm which schedules threads
- kernel unit called the dispatcher performs the scheduling activities

Selected thread runs until it
- blocks
- uses its time slice
- is preempted by a higher priority thread
- terminates

- 32 priority levels, divided into two classes
  - Variable Class – priorities between 1-15
  - Real-Time Class – priorities between 16-31
  - priority 0 thread – memory management
- 32 queues – one for each priority level
- dispatcher traverses the queues (high→low) until it locates an executable thread
- no thread is ready to execute → dispatcher launches a special idle thread

Win32 API Priority Classes

<table>
<thead>
<tr>
<th>Base Priorities</th>
<th>Win32 API Priority Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>REALTIME_PRIORITY_CLASS</td>
<td>24</td>
</tr>
<tr>
<td>HIGH_PRIORITY_CLASS</td>
<td>13</td>
</tr>
<tr>
<td>ABOVE_NORMAL_PRIORITY_CLASS</td>
<td>10</td>
</tr>
<tr>
<td>NORMAL_PRIORITY_CLASS</td>
<td>8</td>
</tr>
<tr>
<td>BELOW_NORMAL_PRIORITY_CLASS</td>
<td>6</td>
</tr>
<tr>
<td>IDLE_PRIORITY_CLASS</td>
<td>4</td>
</tr>
</tbody>
</table>

Variable-Priority Classes
- priority of a thread belonging any of these classes can change; it depends upon the
  - priority class to which a thread belongs
  - relative priority within that class

Win32 API Relative Priorities

- TIME_CRITICAL
- HIGHEST
- ABOVE_NORMAL
- NORMAL
- BELOW_NORMAL
- LOWEST
- IDLE

Processes typically are members of the NORMAL_PRIORITY_CLASS unless
- parent of process was a member of the IDLE_PRIORITY_CLASS
- process was assigned to another class upon creation
The initial priority of a thread is typically the base priority of the process to which it belongs

Priority Classes

<table>
<thead>
<tr>
<th>Time-Critical</th>
<th>High</th>
<th>Above-Normal</th>
<th>Normal</th>
<th>Below-Normal</th>
<th>Idle-Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-Time</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Above-Normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Below-Normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
variable-class thread exhausts time-quantum \( \Rightarrow \) priority is lowered; limited by base priority

- variable-priority thread released from wait queue \( \Rightarrow \) priority is raised
  - amount of boost depends upon type of wait queue
    - wait for keyboard I/O \( \Rightarrow \) big boost
    - wait for disk I/O \( \Rightarrow \) small boost

- Window threads receive an additional boost to enhance response time

**Results**

1. yields good response time for windows & mice
2. enables I/O-bound threads to keep the I/O devices busy
3. CPU-bound threads have their CPU consumption limited, but they are permitted to use the spare CPU cycles in the background

```plaintext
used by many O/S, Unix, et. al.
```

<table>
<thead>
<tr>
<th>NORMAL_PRIORITY_CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Special Subclasses</strong></td>
</tr>
<tr>
<td>Foreground Process -- selected on the screen</td>
</tr>
<tr>
<td>Background Process – currently not selected on the screen</td>
</tr>
</tbody>
</table>

**Special Scheduling Rule**

- Process is selected to be in the foreground \( \Rightarrow \) increase time-quantum by some factor, e.g., three, which allows the process to run three times longer before the time-sharing preemption occurs
Linux Scheduling

- **Traditional Unix Scheduling Algorithm Deficiencies**
  - does not provide adequate support for Symmetric Multiprocessing Systems (SMP)
  - does not **scale well** as the task load (number of tasks) increases

- **Version 2.5 Scheduler**
  - runs in \( O(1) \) time, i.e., constant amount of time regardless of the task
  - provides increased support for SMP
    - Processor Affinity
    - Load Balancing
  - provides support for Interactive tasks, e.g., fairness
  - preemptive, priority-based system
    - priority ranges
      - Real-Time Range 0-99
      - Nice Range – nice value 100-140
    - lower values \( \rightarrow \) higher priorities
    - higher priorities \( \rightarrow \) larger time-quanta, i.e., quantum’s
    - lower priorities \( \rightarrow \) smaller time-quanta

- **Runqueue Data Structure** – list of all runnable tasks (maintained by the kernel)
  - Active Priority Queue – list of all tasks with time remaining in their time-slices
  - Expired Priority Queue – list of all tasks that have exhausted their time-slices
    - hence, not eligible to run until all other tasks have exhausted their time-slices
  - both queues independently maintain a list of tasks indexed by priority
  - tasks with **equal priority** in either queue are maintained as a linked list, First-Come, First-Served

- **Scheduler** selects a task with highest priority from the Active Priority Queue

- **Active Priority Queue == Empty** \( \rightarrow \) the two queues undergo a name switch, i.e.,
  - Active Priority Queue becomes the Expired Priority Queue
  - Expired Priority Queue becomes the Active Priority Queue

- **SMP Architecture** – each processor
  - maintains its own Runqueue Structure and schedules itself independently
  - schedules the highest priority task from its own Runqueue Structures

- **Real-Time Scheduling – Pthreads (POSIX.1b)**
  - tasks are assigned static priorities

- **Nice Scheduling**
  - dynamic priorities are based on the nice value \( \pm 5 \)
  - length of sleep (waiting) time determines the dynamic priority adjustment
  - increased sleep time \( \rightarrow \) increase in priority, i.e., -1 to -5
  - decreased sleep time \( \rightarrow \) CPU-bound \( \rightarrow \) decrease in priority, i.e., +1 to +5

See Silberschatz page 212, figure 5.15
Algorithm Evaluation

- Establishing algorithm criteria
  - maximize CPU utilization under a constrained maximum response time
  - maximizing throughput such that the turnaround time is linearly proportional to the total execution time
  - &/or other criteria relevant to the performance of the projected O/S

- Analytic Evaluation – mathematical modeling
  - Deterministic Modeling
    - Predetermined Workload
    - Determine the performance of each candidate algorithm against the workload
    - Gantt Charts – time distribution of events
    - In situations where the workload is constant over time, and where the workloads tested are realistic, this analysis yields a first good view of the algorithms performance
  - Queueing Models
    - Workload varies over time there is no static set of processes or times to use for deterministic modeling

---

**Queueing-Network Analysis**

**Computer System ↔ Network of Servers**

- Each server has a queue of waiting processes
  - CPU server with ready queue
  - I/O Devices with the device queues
  - etc

- Determine the distribution of CPU-bursts and I/O-bursts
  - Measure
  - Approximate
  - Estimate

- Yields
  - Exponential distribution, i.e., probability, of a particular CPU-burst
  - Distribution of the arrival times, i.e., when processes arrive in the system

- Arrival Rates & Service Rates are used to compute the
  - Average Throughput
  - Utilization
  - Waiting times
  - Average Queue Lengths
  - etc
  - for most scheduling algorithms

---

*Quit often leads to intractable mathematics and thus is not often used in practice*
o Simulations
  ➢ Model of an O/S (implemented to run on a computer)
  ➢ Data Sources
    ★ Random Numbers
    ★ Mathematical Distributions
      ✓ Uniform
      ✓ Exponential
      ✓ Poisson
    ★ Empirically
      ✓ Trace tapes – monitor a real system, record the sequence of actual events
      ✓ Major expenditure to undertake an empirical study, e.g., equipment, people

o Implementation
  ➢ Cost
  ➢ Users Adverse Adjustments to a system under modification
    ★ beta-versions
    ★ gamma-versions