Chapter 4

CPU Scheduling
Chapter 4: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

**Observation**

- Learning Outcomes: L04 (*see TAP*)
Introduction

- In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled.

- A process is executed until it must wait, typically for the completion of some I/O request.

- In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished.
Multiprogramming Objective

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. With multiprogramming, we try to use this time productively.

- Several processes are kept in memory at one time.

- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.

- This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

- Scheduling of this kind is a fundamental operating-system function.

- Almost all computer resources are scheduled before use.

- The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.
Basic Concepts

Processes can be described as either:

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

- 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** followed by **I/O burst**
- CPU burst distribution is of main concern
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (such as for an I/O request or invocation of the wait() system call.)
  2. Switches from running to ready state (for example in response to an interrupt)
  3. Switches from waiting to ready (say at completion of I/O or a return from wait( ))
  4. Terminates

- For conditions 1 and 4: **there is no choice: A new process must be selected.**
- For conditions 2 and 3 there is a choice:  *either continue running the current process,* or select a different one.

- Scheduling under 1 and 4 is **nonpreemptive (cooperative)**
  *Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes*

- **Scheduling under 2 and 3** **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities
Dispatcher

- Dispatcher is another component involved in the CPU-scheduling function.
- 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
- (The time consumed by the dispatcher)
- The dispatcher needs to be as fast as possible, as it is run on every context switch.
CPU Scheduling (Cont.)

Lecture Objectives

- To introduce CPU scheduling
- To define scheduling and Optimization Criteria
- To describe various CPU-scheduling algorithms
- To be able to analyze and compare CPU-scheduling algorithms

Observation

- Course learning outcome: L04 (see TAP)
- Assessment of Outcomes: MID#2, lab works, course project, final exam
- A course project “CPU Scheduling Algorithms – Performance Analysis and Implementation”
CPU Scheduling Algorithms

- **CPU scheduling**
  
  CPU scheduling deals with the problem of deciding which of the processes in the *ready queue* is to be *allocated* to the CPU.

- **Reminder**
  
  - *Algorithm*?
  - *Algorithm Writing Methods*?
  - *Algorithm Characteristics*?

- **CPU scheduling Algorithms**
  
  - First Come First Served (FCFS) Scheduling
  - Shortest-Job-First (SJF) Scheduling
  - Priority Scheduling
  - Round Robin (RR) Scheduling
  - Multilevel Queue Scheduling
## Scheduling Algorithms (Cont.)

<table>
<thead>
<tr>
<th>Scheduling Criteria</th>
<th>Optimization Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU utilization</strong>: Keep the CPU as busy as possible</td>
<td></td>
</tr>
<tr>
<td><strong>Throughput</strong>: # of processes that complete their execution per time unit</td>
<td></td>
</tr>
<tr>
<td><strong>Turnaround time</strong>: Amount of time to execute a particular process (=periods spent waiting to get into memory + waiting in the ready queue + executing on the CPU + doing I/O)</td>
<td></td>
</tr>
<tr>
<td><strong>Waiting time</strong>: Amount of time a process has been waiting in the ready queue</td>
<td></td>
</tr>
<tr>
<td><strong>Response time</strong>: Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)</td>
<td></td>
</tr>
</tbody>
</table>

6.11
First- Come, First-Served (FCFS) Scheduling

- With this scheme, the process that requests the CPU first is allocated the CPU first.

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time (msec)</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:

  Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3= 27$ msec
  Average waiting time: $(0 + 24 + 27)/3 = 51/3 = 17$ msec

What type of queue do you propose for FCFS policy implementation?
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>P_2</th>
<th>P_3</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

- Waiting time for \( P_2 = 0 \), \( P_3 = 3 \), \( P_1 = 6 \);
- Average waiting time: \( (0 + 3 + 6)/3 = 3 \)
- Much better than previous case

- *Cite a common life FCFS example.*
Common Life Example of FCFS Scheduling
Shortest-Job-First (SJF) Scheduling

- This algorithm associates with each process the length of its next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
- There is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value.
Example #1 of SJF Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>8</td>
</tr>
<tr>
<td>P₃</td>
<td>7</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
</tr>
</tbody>
</table>

- Process order: P₄, P₁, P₃, P₂
- SJF scheduling Gantt chart

Waiting time for P₄ = 0, P₁ = 3, P₃ = 9; P₂ = 16;
- Average waiting time = \((0+3 + 9+16) / 4 = 28/4 = 7\)
Common Life Example of SJF Scheduling
SJF vs. FCFS Algorithms

The Gantt chart for Shortest-Job-First (SJF) Scheduling

<table>
<thead>
<tr>
<th>P4</th>
<th>P1</th>
<th>P3</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>9</td>
<td>16</td>
</tr>
</tbody>
</table>

Average waiting time = (3 + 16 + 9 + 0) / 4 = 28/4 = 7

The Gantt chart for First-Come, First-Served (FCFS) Scheduling

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>14</td>
<td>21</td>
</tr>
</tbody>
</table>

Average waiting time =(6+14+21)/3=10.25

What can you conclude?

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
</tr>
</tbody>
</table>
The SJF scheduling algorithm is provably *optimal*, in that it gives the minimum average waiting time for a given set of processes.

Moving a short process before a long one:
- decreases the waiting time of the short process more than it increases the waiting time of the long process.

Consequently, the *average* waiting time decreases.
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 (When a process switches from the running (waiting) state to the ready state)
  - Nonpreemptive (once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state)

- A **preemptive** approach will preempt the CPU if the priority of the newly-arrived process is higher than the priority of the currently running process
- A **non-preemptive** approach will simply put the new process (with the highest priority) at the head of the ready queue
**Example of Priority Scheduling**

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$P_5$</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Process Priority Order: $P_2, P_5, P_1, P_3, P_4$
- Priority scheduling Gantt Chart

- Average waiting time = \(\frac{0+1+6+16+18}{5} = \frac{41}{5} = 8.2\) msec
SJF algorithm vs. Priority Scheduling

- The SJF algorithm is a special case of the general priority scheduling algorithm.
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is:
  
  \[ p = \frac{1}{CPU \text{ Burst Time}} \]

- The larger the CPU burst, the lower the priority, and vice versa.
Priority Scheduling (Cont.)

Problem ≡ **Starvation** – low priority processes may never execute

Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Saturday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes.

**Rumor**
When they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

**Solution** ≡ **Aging** – as time progresses increase the priority of the process
Common Life Example of Priority
Round Robin (RR)

- Designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- Each process gets a small unit of CPU time (time quantum $q$), 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.
- Timer interrupts every quantum to schedule next process.
- 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 = 4

<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>

- The Gantt chart is:

<table>
<thead>
<tr>
<th></th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_1 )</th>
<th>( P_1 )</th>
<th>( P_1 )</th>
<th>( P_1 )</th>
<th>( P_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Waiting time for \( P_2 = 4 \), \( P_3 = 7 \), \( P_1 = 10-4 = 6 \)

- Average waiting time = \((6+4+7)/3 = 17/3 = 5.66 \text{ msec}\)
- Typically, higher average turnaround than SJF, but better \textit{response}
- \( q \) should be large compared to context switch time
- \( q \) usually 10ms to 100ms, context switch < 10 usec

Abdulrazaq Abdulrahim et al., A \textbf{New Improved Round Robin (NIRR) CPU Scheduling Algorithm}, \textit{International Journal of Computer Applications (0975 – 8887) Volume 90 – No 4, March 2014}
RR vs FCFS

- The RR Gantt chart with Time Quantum = 4 is:

Average waiting time = \((6+4+7)/3=17/3=5.66\) msec

- The FCFS Gantt chart:

Waiting time for \(P_1 = 0; P_2 = 24; P_3 = 27\)

Average waiting time: \((0 + 24 + 27)/3 = 17\) msec
CPU Scheduling Algorithm Evaluation

- How do we select a CPU scheduling algorithm for a particular system?
- There are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.
- The first problem is defining the criteria to be used in selecting an algorithm.
- Criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as:
  - Maximizing CPU utilization under the constraint that the maximum response time is 1 second
  - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time
- Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.
Analytic evaluation - Deterministic Modeling

- One major class of evaluation methods is **analytic evaluation**.
- Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.
- One type of analytic evaluation is **deterministic modeling**.
Deterministic Modeling (Cont.)

- **Deterministic modeling** takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Deterministic modeling is simple and fast.
- It gives us exact numbers, allowing us to compare the algorithms.
- However, it requires exact numbers for input, and its answers apply only to those cases.
- The main uses of deterministic modeling are in describing scheduling algorithms and providing examples.
- In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm.
- Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately.
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- 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
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 enters queue $Q_0$ which is served FCFS
    - 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 FCFS 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
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**
NUMA and CPU Scheduling

Note that memory-placement algorithms can also consider affinity.
Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another
Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time $t$, deadline $d$, period $p$
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$
Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- $P_1$ is assigned a higher priority than $P_2$. 

[Diagram showing deadlines and scheduled times for $P_1$ and $P_2$.]
Missed Deadlines with Rate Monotonic Scheduling
Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority

![Diagram showing theEarliest Deadline First Scheduling (EDF)](image-url)
Proportional Share Scheduling

- $T$ shares are allocated among all processes in the system
- An application receives $N$ shares where $N < T$
- This ensures each application will receive $N / T$ of the total processor time
Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling
Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

Version 2.5 moved to constant order $O(1)$ scheduling time

- Preemptive, priority based
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- Map into global priority with numerically lower values indicating higher priority
- Higher priority gets larger q
- Task run-able as long as time left in time slice (**active**)
- If no time left (**expired**), not run-able until all other tasks use their slices
- All run-able tasks tracked in per-CPU **runqueue** data structure
  - Two priority arrays (active, expired)
  - Tasks indexed by priority
  - When no more active, arrays are exchanged

- Worked well, but poor response times for interactive processes
**Completely Fair Scheduler (CFS)**

**Scheduling classes**
- Each has specific priority
- Scheduler picks highest priority task in highest scheduling class
- Rather than quantum based on fixed time allotments, based on proportion of CPU time
- 2 scheduling classes included, others can be added
  1. default
  2. real-time

Quantum calculated based on **nice value** from -20 to +19
- Lower value is higher priority
- Calculates **target latency** – interval of time during which task should run at least once
- Target latency can increase if say number of active tasks increases

CFS scheduler maintains per task **virtual run time** in variable **vruntime**
- Associated with decay factor based on priority of task – lower priority is higher decay rate
- Normal default priority yields virtual run time = actual run time

To decide next task to run, scheduler picks task with lowest virtual run time
CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

![Task Tree Diagram]

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\log N)$ operations (where $N$ is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.
Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139

![Priority Chart]

<table>
<thead>
<tr>
<th>Real-Time</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>139</td>
</tr>
</tbody>
</table>

Priority

Higher

Lower
Algorithm Evaluation

■ How to select CPU-scheduling algorithm for an OS?
■ Determine criteria, then evaluate algorithms
■ **Deterministic modeling**
  ● Type of **analytic evaluation**
  ● Takes a particular predetermined workload and defines the performance of each algorithm for that workload
■ Consider 5 processes arriving at time 0:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
</tr>
<tr>
<td>$P_2$</td>
<td>29</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
<tr>
<td>$P_5$</td>
<td>12</td>
</tr>
</tbody>
</table>
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:
  - Non-preemptive SFJ is 13ms:
  - RR is 23ms:
Preemption Disadvantages

- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.

- Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures. Chapter 5 examined this issue in greater detail.

- Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs. Most modern UNIXes deal with this problem by making the process wait until the system call has either completed or blocked before allowing the preemption. Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.

- Some critical sections of code protect themselves from concurrency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should only be done in rare situations, and only on very short pieces of code that will finish quickly, (usually just a few machine instructions.)
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- 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: \( \tau_{n+1} = \alpha \ t_n + (1-\alpha)\tau_n \).

- Commonly, \( \alpha \) set to \( \frac{1}{2} \)
- Preemptive version called **shortest-remaining-time-first**
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