Chapter 5: CPU Scheduling
Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor 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
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
Histogram of CPU-burst Times

- Frequency on the y-axis
- Burst duration (milliseconds) on the x-axis
Alternating Sequence of CPU And I/O Bursts

- load store
- add store
- read from file

\[ \text{wait for I/O} \]

- store increment index
- write to file

\[ \text{wait for I/O} \]

- load store
- add store
- read from file

\[ \text{wait for I/O} \]
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)
Scheduling Algorithm 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></td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

Average waiting time: $(0 + 24 + 27)/3 = 17$
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></th>
<th>P_2</th>
<th>P_3</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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.

- 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.
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF scheduling chart

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

<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 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 = 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:

```
  P1   P2   P3   P1   P1   P1   P1   P1
  0    4    7    10   14   18   22   26   30
```

- 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
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; 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 $Q_2$. 
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as *process-contention scope (PCS)* since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is *system-contention scope (SCS)* – competition among all threads in system
API allows specifying either PCS or SCS during thread creation

- PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
- PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
```c
/* now join on each thread */
for (i = 0; i < NUM THREADS; i++)
    pthread join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```
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

- **Processor affinity** – process has affinity for processor on which it is currently running
  - soft affinity
  - hard affinity
NUMA and CPU Scheduling

![Diagram showing NUMA and CPU scheduling]

- CPU with fast access to memory
- Slow access between two memory modules
- Computer structure with two CPUs and memory modules

Silberschatz, Galvin and Gagne ©2009
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
Multithreaded Multicore System

![Diagram showing the concept of compute cycles (C) and memory stall cycles (M) in a multithreaded multicore system.](image)
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling
## Solaris Dispatch Table

<table>
<thead>
<tr>
<th>priority</th>
<th>time quantum</th>
<th>time quantum expired</th>
<th>return from sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>15</td>
<td>160</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>10</td>
<td>52</td>
</tr>
<tr>
<td>25</td>
<td>120</td>
<td>15</td>
<td>52</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
<td>20</td>
<td>53</td>
</tr>
<tr>
<td>35</td>
<td>80</td>
<td>25</td>
<td>54</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>30</td>
<td>55</td>
</tr>
<tr>
<td>45</td>
<td>40</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>40</td>
<td>58</td>
</tr>
<tr>
<td>55</td>
<td>40</td>
<td>45</td>
<td>58</td>
</tr>
<tr>
<td>59</td>
<td>20</td>
<td>49</td>
<td>59</td>
</tr>
</tbody>
</table>
Solaris Scheduling

- Interrupt threads
- Realtime (RT) threads
- System (SYS) threads
- Fair share (FSS) threads
- Fixed priority (FX) threads
- Timeshare (TS) threads
- Interactive (IA) threads
## Windows XP Priorities

<table>
<thead>
<tr>
<th></th>
<th>real-time</th>
<th>high</th>
<th>above normal</th>
<th>normal</th>
<th>below normal</th>
<th>idle priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-critical</td>
<td>31</td>
<td>15</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>
<td>6</td>
</tr>
<tr>
<td>above normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>below normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Linux Scheduling

- Constant order $O(1)$ scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- (figure 5.15)
## 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>real-time tasks</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>other tasks</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>priority</th>
<th>task lists</th>
<th>priority</th>
<th>task lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
<td>[0]</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[140]</td>
<td></td>
</tr>
<tr>
<td>[140]</td>
<td></td>
<td></td>
<td>[140]</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)
End of Chapter 5
<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th></th>
<th>P2</th>
<th></th>
<th>P3</th>
<th></th>
<th>P4</th>
<th></th>
<th>P5</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>10</td>
<td></td>
<td>39</td>
<td></td>
<td>42</td>
<td></td>
<td>49</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In-5.7
In 5.8

<table>
<thead>
<tr>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_5</th>
<th>P_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>10</td>
<td>20</td>
<td>32</td>
</tr>
</tbody>
</table>

61
<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
<th>$P_2$</th>
<th>$P_5$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>20</td>
<td>23</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>52</td>
</tr>
</tbody>
</table>
Dispatch Latency
Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority
Java Thread Scheduling (Cont.)

- JVM Schedules a Thread to Run When:
  1. The Currently Running Thread Exits the Runnable State
  2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not
Time-Slicing

- Since the JVM Doesn’t Ensure Time-Slicing, the yield() Method May Be Used:

  ```java
  while (true) {
    // perform CPU-intensive task
    ...
    Thread.yield();
  }
  ```

- This Yields Control to Another Thread of Equal Priority
## Thread Priorities

<table>
<thead>
<tr>
<th>Priority</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread.MIN_PRIORITY</td>
<td>Minimum Thread Priority</td>
</tr>
<tr>
<td>Thread.MAX_PRIORITY</td>
<td>Maximum Thread Priority</td>
</tr>
<tr>
<td>Thread.NORM_PRIORITY</td>
<td>Default Thread Priority</td>
</tr>
</tbody>
</table>

Priorities May Be Set Using `setPriority()` method:

```java
setPriority(Thread.NORM_PRIORITY + 2);
```
Solaris 2 Scheduling

<table>
<thead>
<tr>
<th>Global Priority</th>
<th>Scheduling Order</th>
<th>Class-specific Priorities</th>
<th>Scheduler Classes</th>
<th>Run Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>First</td>
<td>Real Time</td>
<td>Kernel threads of real-time LWPs</td>
<td></td>
</tr>
<tr>
<td>Lowest</td>
<td>Last</td>
<td>System</td>
<td>Kernel service threads</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interactive &amp; time sharing</td>
<td>Kernel threads of interactive &amp; time-sharing LWPs</td>
<td></td>
</tr>
</tbody>
</table>