Chapter 5: 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

Histogram 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>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that the processes arrive in the order: P1, P2, P3

This Gantt Chart for the schedule is:

- Waiting time for P1 = 0; P2 = 24; P3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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>P1</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P4</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>P1</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)
- Average waiting time = \((0 + 1 + 0 + 2)/4 = 3\)

**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. \(t_p\) = actual length of \(n^{th}\) CPU burst
2. \(\tau_{n+1}\) = predicted value for the next CPU burst
3. \(0 \leq \alpha \leq 1\)
4. Define:

\[
\tau_n = \alpha \tau_{n-1} + (1 - \alpha) t_n
\]

**Prediction of the Length of the Next CPU Burst**

**Examples of Exponential Averaging**

- \(\alpha = 0\)
  - Recent history does not count
- \(\alpha = 1\)
  - Only the actual last CPU burst counts

If we expand the formula, we get:

\[
\tau_n = \alpha t_n + (1 - \alpha) \tau_{n-1} + \cdots + (1 - \alpha)^k \tau_{n-k} + \cdots + (1 - \alpha)^{n+1} \tau_0
\]

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
- Non-preemptive
- 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 = 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:

Typically, higher average turnaround than SJF, but better response.

Time Quantum and Context Switch Time

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
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:
  - Q0 – RR with time quantum 8 milliseconds
  - Q1 – RR time quantum 16 milliseconds
  - Q2 – FCFS
- Scheduling
  - A new job enters queue Q2 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 Q1.
  - At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2.

Multilevel Feedback Queues

- 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 fortunate ones

Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next
#include <stdio.h>
#include <pthread.h>

#define NUM THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread tid[NUM THREADS];
    pthread attr 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);

    /* 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);
}

/* Operating System Examples */
- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

## Solaris 2 Scheduling

<table>
<thead>
<tr>
<th>priority</th>
<th>nice</th>
<th>time quantum</th>
<th>nice priority limit</th>
<th>system priority limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>95</td>
<td>99</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>80</td>
<td>99</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>75</td>
<td>95</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>70</td>
<td>80</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>65</td>
<td>65</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>50</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>55</td>
<td>45</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>35</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

## Windows XP Priorities

<table>
<thead>
<tr>
<th>mode</th>
<th>high</th>
<th>normal</th>
<th>below normal</th>
<th>idle</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>time critical</td>
<td>20</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>highest</td>
<td>25</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>normal</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>idle</td>
<td>10</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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, recrediting 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

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>real-time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tasks</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>200 ms</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></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

Algorithm Evaluation

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

End of Chapter 5
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);
```