Chapter 5: Process Scheduling
Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Thread Scheduling
- Operating Systems Examples
- Algorithm Evaluation
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

- 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**
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 – the interval from the time of submission of a process to the time of completion
- 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>$P_1$</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:

```
    P1   P2   P3
    0    24   27  30
```

- 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:

```
  P2  P3  P1
0   3   6   30
```

- 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.

- 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>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

![Process Schedule Diagram]

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

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</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. $t_n$ = 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:
   \[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n. \]
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:

\[
\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \ldots \\
+ (1 - \alpha) \alpha t_{n-j} + \ldots \\
+ (1 - \alpha)^{n+1} \tau_0
\]

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 \(\equiv\) highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem: Starvation \(\equiv\) low priority processes may never execute
- Solution: Aging \(\equiv\) as time progresses increase the priority of the process
### 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>

- The Gantt chart for the schedule is:

```
<table>
<thead>
<tr>
<th>P2</th>
<th>P5</th>
<th>P1</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>
```

- Average waiting time = \( \frac{6 + 0 + 16 + 18 + 1}{5} = 8.2 \)
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 $\Rightarrow$ FCFS
  - $q$ small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high
  - Rule of thumb: 80% of the CPU bursts should be shorter than the time quantum
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:

```
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

<table>
<thead>
<tr>
<th>Process Time</th>
<th>Quantum</th>
<th>Context Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

The chart illustrates the process time, quantum, and context switches for a system. The process time is 10 units, with quantum and context switch counts shown for different time intervals.
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>

![Graph showing the relationship between time quantum and average turnaround time]

**Legend:**
- **X-axis:** Time quantum
- **Y-axis:** Average turnaround time
- **Graph line:** Represents the data points for different processes.
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, and 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.
- The idea is to separate processes according to the characteristics of their bursts:
  - If a process uses too much CPU time, it will be moved to a low-priority queue.
    - This leaves I/O-bound and interactive processes in the higher-priority queues.
  - If a process waits too long in a low-priority queue, it may be moved to a higher-priority queue.
    - This prevents starvation.
Multilevel Feedback Queue (Cont.)

- 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 when $Q_0$ is empty
  - $Q_2$ – FCFS but only $Q_0$ and $Q_1$ are empty

- Scheduling
  - A new job enters queue $Q_0$. 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 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
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor system
- *Load sharing*
- *Asymmetric multiprocessing* – a single processor makes all scheduling decisions; the others execute only user code
  - only one processor accesses the system data structures, alleviating the need for data sharing
Multiple-Processor Scheduling (Cont.)

- Symmetric multiprocessing (SMP) – each processor is self-scheduling
  - Each process has its own queue or shares a common queue
  - Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute
  - The scheduler must be programmed carefully
    - Ensure that two processors do not choose the same process and that processes are not lost from queue.

- Processor affinity
  - A process has an affinity for the processor on which it is currently running
  - To avoid migration of processes because of the high cost of invalidating and re-populating caches
Load Balancing

- To keep the workload evenly distributed across all processors typically with own private ready queue
  - Push migration
    - A specific task periodically checks the load on each processor and evenly distributes the load if an imbalance is found
    - Linux – runs load-balancing algorithm every 200 ms
  - Pull migration
    - An idle processor pulls a waiting task from a busy processor
    - Linux

- Load balancing often counteracts the benefits of processor affinity
  - No absolute rule concerning what policy is best
Symmetric Multithreading

- Symmetric multithreading, SMT, or hyperthreading technology (Intel) provides multiple logical rather than processors.
- Each logical processor has its own architecture state, including registers, and interrupt handling.
- Each logical processor shares the resources of its physical processor, such as cache and buses.
- From OS perspective, threads are scheduled on logical processors.
- Note that SMT is a feature provided in hardware, not software.
  - Hardware must provide the representation of the architecture state of each logical processor as well as interrupt handling.
  - OS need not be designed differently if they are to run on an SMT system, but may be optimized.
A Typical SMT Architecture

logical CPU  logical CPU

physical CPU

system bus

logical CPU  logical CPU

physical CPU
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
  - Process-contention scope (PCS)

- Global Scheduling – How the kernel decides which kernel thread to run next
  - System-contention scope (SCS)

- PCS is done via priority
  - PCS will typically preempt the running thread in favor of a higher-priority thread
  - No guarantee of time slicing among threads of equal priority
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling
Solaris 2 Scheduling

- Global priority: highest to lowest
- Scheduling order: first to last
- Class-specific priorities: real time, system, interactive & time sharing
- Scheduler classes: kernel threads of real-time LWPs, kernel service threads, kernel threads of interactive & time-sharing LWPs
- Run queue:
  - Kernel threads of real-time LWPs
  - Kernel service threads
  - Kernel threads of interactive & time-sharing LWPs
## 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>
## Windows XP Priorities

<table>
<thead>
<tr>
<th>Priority</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

- 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, recrating 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>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>
<tr>
<td></td>
<td></td>
<td></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>![task list]</td>
<td>[0]</td>
<td>![task list]</td>
</tr>
<tr>
<td>[1]</td>
<td>![task list]</td>
<td>[1]</td>
<td>![task list]</td>
</tr>
<tr>
<td></td>
<td>![task list]</td>
<td></td>
<td>![task list]</td>
</tr>
<tr>
<td>[140]</td>
<td>![task list]</td>
<td>[140]</td>
<td>![task list]</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
- Simulation
- Implementation
### In-5.7

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>39</td>
<td>42</td>
<td>49</td>
<td>61</td>
</tr>
<tr>
<td>P_3</td>
<td>P_4</td>
<td>P_1</td>
<td>P_5</td>
<td>P_2</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>10</td>
<td>20</td>
<td>32</td>
<td>61</td>
</tr>
</tbody>
</table>
### In-5.9

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
<th>P₂</th>
<th>P₅</th>
<th>P₂</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>
<td>61</td>
</tr>
</tbody>
</table>
Queueing Models

- Model the computer system as a network of servers
  - Each server has a queue of waiting processes
- Knowing arrival rates and service rates
  - Utilization, queue length, wait time, and so on can be obtained
- A lot of math
- Arrival and service distributions may not be accurate, which are often defined in mathematically tractable but unrealistic ways.
Simulation

actual process execution

trace tape

CPU 10
I/O 213
CPU 12
I/O 112
CPU 2
I/O 147
CPU 173

simulation

FCFS

performance statistics for FCFS

simulation

SJF

performance statistics for SJF

simulation

RR (q = 14)

performance statistics for RR (q = 14)
The only completely accurate way to evaluate a scheduling algorithm

- Code it, put in the system and see how it work
- Expensive
End of Chapter 5
#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);
Pthread Scheduling API

/* 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);
}

Dispatch Latency

- event
- response interval
- interrupt processing available
- process made available
- dispatch latency
- real-time process execution
- conflicts
- dispatch
- time
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
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);
```