Lecture 9: CPU Scheduling
Chapter 5 (cont)
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
In-class Problems (1)

- What advantages does a preemptive CPU scheduling algorithm have over a non-preemptive one?
- Why do different levels of a multi-level feedback queue CPU scheduler have different time quantum?
- Some would say that round robin CPU scheduling does poorly when faced with jobs of equal length. What is their reasoning?
In-class Problems (2)

Assume you are given a uniprocessor system with one gigabyte of memory and a 300 gigabyte disk. The OS on the machine has a demand paged virtual memory system with a local page replacement policy and a multi-level feedback queue (MLFQ) CPU scheduler. On the system there are two compute-intensive jobs running: Job-A and Job-B. Job-A has a working set of 50 gigabytes while Job-B has a working set of 100 megabytes. Assume you left the system to run for a while until it reached a steady state with both jobs running.

- Which job would you expect to have a higher CPU scheduling priority from the MLFQ scheduler?
- Assume you add a second CPU to system, how would this affect the priorities of the jobs?

Justify your answer and state any assumptions you make.
Thread Scheduling

- Distinction between user-level and kernel-level threads
- **Process-contention scope (PCS):** scheduling competition is within the process
  - Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system
PCS vs. SCS

Figure 4.6 User-Level and Kernel-Level Threads
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.
Pthread Scheduling API

```c
#include <stdio.h>
#include <pthread.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);
    /* 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);
}
```

5.12 Silberschatz, Galvin and Gagne ©2009
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 (e.g., to avoid repopulating caches) – see next slide for an example.
  - soft affinity
  - hard affinity

- **Load balancing**:
  - Push migration
  - Pull migration
  - Conflicts with processor affinity
  - Often combination of the pull and push migration approaches
Architecture can affect processor affinity.
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Each core has its own register set: seen by OS as a processor
- 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
  - “Hardware” threads: hardware support includes logic for thread switching, thus decreasing the context switch time.
2 different levels of scheduling:
- Mapping software thread onto hardware thread
  - traditional scheduling algorithms
- Which hardware thread a core will run next
  - Round Robin (Ultra Sparc1) or dynamic priority-based
    (Intel Itanium, dual-core processor with two hardware-managed threads per core)
Operating System Examples
Solaris Scheduling

- kernel threads
- 6 classes of scheduling
- Default: time-sharing based on a multi-level feedback queue
### 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></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>

Priority-based scheduler:
- on X axis: classes of priorities
- on Y axis: relative priorities within a class
Base priority for a process: threads cannot go lower
Priority varies based on:
- quantum used: lower priority
- interrupt from keyboard: larger increase than from disk
Quantum varies for foreground vs. background process.
Linux Scheduling

- Constant order $O(1)$ scheduling time
- Preemptive, priority-based, with two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value (time-sharing) from 100 to 140. (lower value is higher priority)
- Support for SMP: each processor has its own runqueue, with two priority arrays, active and expired.
- When a task has exhausted its quanta, is considered expired and cannot be run until all other tasks have also exhausted their time quanta.
### 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>
In-class Scheduling Problems

- Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?

- Assume a system with priority scheduling in which user processes run with the lowest priority and system processes run with the highest priority. Lowest priority has round-robin scheduling. Is it possible for processes in the lowest class to starve? Explain your answer.

- What advantage does First Come First Served scheduling over Round Robin scheduling in a uniprocessor batch system?

- Suppose a scheduling algorithm favors those processes that have used little processor time in the recent past.
  - Explain why this algorithm favors I/O-bound processes.
  - Explain why this algorithm does not permanently deny processor time to CPU-bound processes.
In-class Scheduling Problems

Computer scientists have studied process and thread scheduling for decades. One reason is that we cannot agree on what to optimize.

- Give three examples of goals for which the schedule can be optimized. For each goal, describe:
  - A workload where that goal is important; and
  - A scheduling algorithm that targets that goal.

- Pick two of the three scheduling algorithms from your answer to part a) and explain how best you can integrate them into a single system. Does this achieve both goals under any circumstances? If not, when?
Assume that 5 processes arrive at the ready queue at the times shown below. The estimated next burst times are also shown. Assume that an interrupt occurs at every arrival time.

- What is the waiting time for each process if preemptive “shortest remaining time first” scheduling is used?
- What is the waiting time for each process if non-preemptive “shortest job first” scheduling is used?

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>