CPU Scheduling

These slides are created by Dr. Huang of George Mason University. Students registered in Dr. Huang’s courses at GMU can make a single machine readable copy and print a single copy of each slide for their own reference as long as the slide contains the copyright statement, and the GMU facilities are not used to produce the paper copies. Permission for any other use, either in machine-readable or printed form, must be obtained form the author in writing.

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
  5. Time slices expires
- Scheduling under 1 and 4 is nonpreemptive.
- 5 is preemptive.
Processing and IO Cycles

- Process execution consists of a cycle of CPU execution and I/O wait.

Type of Applications
From the Viewpoint of Scheduling

- **CPU bound**
  - long CPU bursts and short IO bursts
  - Spend most times in computing
  - Number crunching applications

- **I/O bound**
  - Short CPU bursts and long IO bursts
  - Spend most times in IO
  - Interactive and database applications
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 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>

- Processes arrive in the order: \( P_1, P_2, P_3 \)
- The Gantt Chart for the schedule is:

```

```

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

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order \( P_2, P_3, P_1 \).

- We have

```

```

- 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.
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

- **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 = \((9 + 1 + 0 + 2)/4 - 3\)
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. $t_n = \text{actual length of } n^{th}\text{CPU burst}$
2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

If we expand the formula, we get:

$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_n - 1 + \ldots$

$\ldots + (1 - \alpha)^j\alpha t_n - 1 + \ldots$

$\ldots + (1 - \alpha)^{n=1} t_n \tau_0$

- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.
Round Robin (RR)

- Each process gets a small slice 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.

With Pentium Pro 200MHz, context switching time is about 15 msec.
Example of RR, 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:

```
P_1 P_2 P_3 P_4 P_1 P_3 P_4 P_1 P_3 P_3
0  20  37  57  77  97 117 121 134 154 162
```

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

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).
- Problem $\equiv$ Starvation – low priority processes may never execute.
- Solution $\equiv$ Aging – as time progresses increase the priority of the process.
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**

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:
  - $Q_0$ – time quantum 8 milliseconds
  - $Q_1$ – 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 $Q_2$.
- Notice: completion means reaching the end of the current CPU burst.
Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.
  - An important subject on its own
  - But we will not discuss it further
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones.
  - Important for interactive applications, gaming

Linux Process Scheduling

- The basic time unit is a *tick* = 10 msec.
- The *nice* level of a process ranges from -20 to 19 (the higher the value, the nicer the process is to others).
- The remaining time quantum of the process is recorded in *counter* (in ticks).
- A task is *runnable* if it is in the RUNNING state and its quantum has not run out (*counter* > 0).
- The scheduler finds the runnable process with the highest (*counter - nice*).
- When the counter of all runnable processes reaches zero, all processes are updated as follows.

\[
counter = \frac{counter}{2} + (20 - nice + 1);
\]

- For a runnable process (whose original \(\text{counter}=0\)), this restores its quantum to \((20 - nice + 1)\), ranging from 2 to 40.

- Notice that even waiting processes (whose original \(\text{counter}>0\)) are updated as above.

- This gives advantages to IO-bound processes in two ways:
  - They have higher priority when ready to run
  - They can run longer when ready to run.

- Notice that interactive applications are IO bound.

- Can you see that \textit{aging} is supported to avoid starvation?
**Discussions**

- Several problems of Linux scheduling are known.
  - Recompute counters of all processes take too much time in large scale systems.
  - With large numbers of processes, the events of quantum exhausted for all runnables occur seldom, leaving interactive applications not boosted often enough
  - Some people find Linux *not smooth* under load.

**Scheduling in Windows 2000**

- Support 32 levels of priority.
- Windows 2000 API provides 6 classes:
  - Real time, High priority, above normal, normal priority, Below normal, Idle.
- There are 7 relative priorities.
  - Time critical, highest, above normal, normal, below normal, lowest, idle
Windows 2000 Priorities

Also called variable class

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>highest</td>
<td>26</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>lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- When a task of the variable class uses up its quantum, its priority is lowered.
  - CPU bound applications receive low priorities
- When a task waits on an IO operation, its priority is raised.
- The amount of priority boost depends on what the task is waiting for.
  - small boosts to disk
  - large boosts to keyboard/mouse
- The result: interactive applications receive high priorities.