Operating System Examples - Scheduling
References

- Silberschatz et al, Chapter 5.6, Chapter 22.3.22
Note: Process/Thread

- A unit of execution is a process
- Processes can create a unit of execution called a thread
  - Threads share data with processes
  - Operating systems schedule threads in the same way processes are scheduled
- We will discuss threads in much greater detail later in the course
Solaris
Solaris

- Solaris is a Unix Operating systems originally developed by Sun
- Sun was acquired by Oracle in 2010 and so Solaris is sometimes called Oracle Solaris
- Solaris was developed as proprietary software
- 2005: Sun released most of its code base and thus Solaris became open source
- Oracle closed up Solaris
Solaris

- Platforms: SPARC, i86pc (x86 and x86-64)
- Solaris has support for desktop environments but is mainly aimed at the server market
- Lots of innovations with Solaris esp under SUN
Solaris

- Solaris uses priority-based scheduling
- Each process belongs to one of six classes:
  - Time sharing
  - Interactive
  - Real time
  - System
  - File share
  - Fixed priority

- Default scheduling class:
  - Time sharing

- Within each class:
  - Different priorities
  - Different scheduling algorithms

- Priority levels from 0 to 169
Priorities

- **Interrupt**
- **Real-time class**
  - Highest priority
  - Guaranteed response time
  - Kernel processes (very few) assigned to this class
- **System class**
  - Kernel processes are assigned to this class e.g., scheduler
  - Priorities are static
Priorities

- **Time-sharing**
  - Processes can have their priorities adjusted

- **Fixed-priority**
  - Introduced in Solaris 9
  - No dynamic adjustment of priorities

- **Fair-share**
  - Uses CPU shares instead of priorities
  - A share is an indicator of entitlement
Real-Time Scheduling

- Real-time priority queues can be configured to operate with a FIFO-strategy or a Round Robin strategy.
- Real-time processes can preempt running processes that have lower priority.
  - Tend to be few real-time processes.
Time Sharing Scheduling Algorithm

- Default priority for a user process is time sharing
- Priorities are dynamic
- Use of a multilevel feedback queue
  - Each queue represents a different priority level
  - Time slices (quantum) are associated with a priority queue
- Higher priorities $\rightarrow$ smaller time slice
Time Sharing Scheduling Algorithm

- Solaris uses a dispatch table
- This is used to assign a new priority to a process. Two cases:
  - Time quantum expired without blocking
    - Could be a CPU intensive process
  - Process blocked before time quantum expired
- This new priority will be used to determine the process in a priority queue when the process returns to a READY state
### Solaris Dispatch Table

For time-sharing & Interactive processes

<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 Dispatch Table

- **Priority**: A higher number indicates a higher priority
- **Time quantum**: There is an inverse relationship between priorities and time quanta
- **Time quantum expired**: The new priority of a process that has used its entire time quantum without blocking
- **Return from sleep**: The priority of a process that is returning from the blocked state (such as waiting for I/O)
Scheduling Algorithm

- Time sharing scheduling algorithm
  - If a priority has used its entire time quantum without blocking its priority is changed (with one exception)
  - The priority of a priority that is returning from sleeping (e.g. waiting for I/O) has its priority increased
    - This allows for better support of interactive processes
Interactive Processes

- Interactive processes (e.g., window managers) typically have higher priority (good response time)
- CPU-bound processes have lower priority (good throughput)
Windows 7
Windows

- Windows refers to a collection of graphical operating systems developed by Microsoft
- There are recent versions of Windows for PCs, server computers, smartphones and embedded devices
- There is a specialized version of Windows that runs the Xbox One game console
- Windows was originally designed for desktop machines
Priorities

- Similar to that used in Windows XP
- The scheduler is called a dispatcher
- 32 priorities
- Priorities are divided into two classes:
  - User class: priorities 1 to 15
  - Real-time class: priorities 16 to 31
- Priority 0 is used for memory management processes
- There is a queue for each priority
Selecting a Process

- The dispatcher traverses the set of queues from highest to lowest until it finds a process that is ready to run.
- If there are no processes ready to run, the dispatcher executes the idle process.
- Priority of a preempted process may be modified before being returned to a ready state.
- Round robin
Adjusting Priority

- If process was in user class
  - Time quantum expires:
    - If process is in the user class the priority is lowered
  - Process switches from blocked to running:
    - Priority is increased
    - The amount depends on what the process was doing
    - Keyboard I/O gets a large increase while disk I/O gets a moderate increase

- Some processes always have a low priority e.g., disk fragmenter
Adjusting Priority

- The priority of a process cannot be lowered passed the base priority (lower threshold value) of the process.

- Windows 7 distinguishes between the foreground process that is currently selected on the screen and the background processes that are not currently selected.
  - Tends to give good response times to interactive processes that are using the mouse and windows.
Linux
Linux

- Used on desktops, servers, smartphones, embedded devices
- As of Nov 2014, 97% of all HPC systems ran Linux
Scheduling through Version 2.5

- Prior to kernel version 2.5, Linux ran a variation of standard UNIX scheduling algorithm.
- Version 2.5 moved to constant order $O(1)$.
- Linux uses the term “task” for process.
Linux 2.5 Scheduling

Goals

- Implement scheduling algorithms in $O(1)$ time.
- Optimize for the common case of only one or two runnable processes, yet scale well to multiple processors, each with many processes.

Linux uses the term **task** to refer to a process/thread

Linux priority levels are on the next slide
### Linux 2.5 - Priority Levels

<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></td>
<td>real-time tasks</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>other tasks</td>
<td>10 ms</td>
</tr>
<tr>
<td>140</td>
<td>lowest</td>
<td></td>
</tr>
</tbody>
</table>

Priorities and Time-slice length
Priorities 0-99 for real-time processes

Priorities 100-139 for normal (user) processes
Linux 2.5 Scheduling

- A process is considered for execution on the CPU as long as it has time remaining in its time slice (quantum)

- When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have all exhausted their time slice
Linux 2.5 Data Structures

- The kernel maintains a list of all runnable tasks in a runqueue data structure.

- A runqueue consists of two priority arrays:
  - Active: Contains all tasks with time remaining in their time slices.
  - Expired: Contains all expired tasks.

- The active, expired queues are indexed according to priority.
List of Tasks Indexed According to Priorities
Linux 2.5 Scheduling

- How is a process selected for execution?
  - The scheduler finds the highest-priority queue with a runnable process
  - Finds the first process on the queue

- What happens to a running process that does not complete its time quantum?
  - When that process returns to the \texttt{ready} state it goes into the active array
Linux 2.5 Scheduling

- When there are no more processes in the active array, the expired array becomes the active array and the active array becomes the expired array.

- A process’s priority (queue) is recalculated when the process has exhausted its time quantum and is to be moved to the expired array.
  - Calculation considers time spent waiting for I/O.
The Completely Fair Scheduler

- CFS is a significant departure from the traditional UNIX process scheduler.
- All processes are allotted a proportion of the processor’s time in as “fair” of a way as possible
- CFS tries to model an “ideal, precise multitasking CPU” – one that could run multiple processes simultaneously, giving each equal processing power.
We can measure how much runtime each task has had and try and ensure that it gets their fair share of time.

Although user processes are assumed to be equal the `nice` command can be used by the user to indicate the user's priority of an application in order to tune the performance.
CFS – Nice Values

- A **nice value** is assigned to each task
  - Nice values range from -20 to +19
  - Lower nice value indicates a higher relative priority
  - Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values
  - The default nice value is 0
Example: Let’s say you have a Linux Workstation

- Your priority is the best performance for your favourite game e.g., Minecraft
- Minecraft runs much better with more CPU
- Give Minecraft a low nice value indicating to the scheduler that this is high priority
CFS - Nice Values

- Making a job nice by increasing its nice number is only useful for processes that use a lot of CPU time
  - Probably not a good idea to increase the nice value of interactive processes

- CFS considers the nice value in its scheduling decisions
CFS - Virtual Runtime

- The **virtual run time** (vruntime) of a task is the actual runtime weighted by its niceness.
- The value of vruntime is used by the scheduler to determine the next process to run.
  - Process with the smallest vruntime is selected to run next.
- Measured by nanoseconds.
CFS - Virtual Runtime

- High nice values should result in less CPU time allocated to a process

- This implies that $vruntime$ cannot be the same as the real runtime
Example: Assume a process runs for 200 milliseconds

- Nice value is 0: `vruntime` will be 200 milliseconds
- Nice value < 0: `vruntime` will be less than 200 milliseconds
- Nice value > 0: `vruntime` will be greater than 200 milliseconds
CFS - Calculating vruntime

- Let $t$ represent the amount of time spent using the CPU when a process has the CPU.
- Vruntime is incremented by
  - $t \times \frac{\text{weight}_0}{\text{weight}_i}$ where
    - $\text{weight}_0$ is the weight of nice value 0
    - $\text{weight}_i$ is the weight of nice value $i$
  - We refer to $\frac{\text{weight}_0}{\text{weight}_i}$ as the decay factor.
- Weights of nice values are precomputed to avoid runtime overhead.
# CFS Example Weights

<table>
<thead>
<tr>
<th>Nice Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>3121</td>
</tr>
<tr>
<td>-1</td>
<td>1277</td>
</tr>
<tr>
<td>0</td>
<td>1024</td>
</tr>
<tr>
<td>1</td>
<td>820</td>
</tr>
<tr>
<td>5</td>
<td>335</td>
</tr>
</tbody>
</table>
## CFS Example Weights

<table>
<thead>
<tr>
<th>Nice Value</th>
<th>Decay Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>$\frac{1024}{3121} = 0.33$</td>
</tr>
<tr>
<td>-1</td>
<td>$\frac{1024}{1277} = 0.80$</td>
</tr>
<tr>
<td>0</td>
<td>$\frac{1024}{1024} = 1$</td>
</tr>
<tr>
<td>1</td>
<td>$\frac{1024}{820} = 1.24$</td>
</tr>
<tr>
<td>5</td>
<td>$\frac{1024}{335} = 3.05$</td>
</tr>
</tbody>
</table>
Example

- Assume only two tasks, $t_1$ and $t_2$ with nice values of 0
- Decay factor is 1
- The table reflects real time and virtual time values at a specific point in time

<table>
<thead>
<tr>
<th>$t_1$ real time</th>
<th>$t_1$ vruntime</th>
<th>$t_2$ real time</th>
<th>$t_2$ vruntime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>
How do make sure that a process doesn't starve?

More discussion later in the slides on RB trees.
Example

- Assume only two tasks, $t_1$ and $t_2$ with nice values of 0
- Decay factor is 1
- The reflects real time and virtual time values at a specific point in time

<table>
<thead>
<tr>
<th>$t_1$ real time</th>
<th>$t_1$ vruntime</th>
<th>$t_2$ real time</th>
<th>$t_2$ vruntime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>
Could a process starve?
- No

Let’s say that a task $t_1$ doesn’t get the process while $t_2$ always does
  - At some point $t_1$ will have a smaller value for $v_{runtime}$ since it is never being incremented.
Example

Assume two tasks, $t_1$ and $t_2$ with nice values of -1 and 1 respectively and decay factors of .80 and 1.24 respectively.

The table reflects real time and virtual time values at a specific point in time.

<table>
<thead>
<tr>
<th>$t_1$ real time</th>
<th>$t_1$ vruntime</th>
<th>$t_2$ real time</th>
<th>$t_2$ vruntime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>5</td>
<td>6.2</td>
</tr>
<tr>
<td>30</td>
<td>24</td>
<td>10</td>
<td>12.4</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>15</td>
<td>18.6</td>
</tr>
<tr>
<td>60</td>
<td>48</td>
<td>20</td>
<td>24.8</td>
</tr>
<tr>
<td>80</td>
<td>64</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>
Example

Assume two tasks, $t_1$ and $t_2$ with nice values of -5 and 5 respectively and decay factors of .33 and 3.05 respectively.

The table reflects real time and virtual time values at a specific point in time.

<table>
<thead>
<tr>
<th>$t_1$ real time</th>
<th>$t_1$ vruntime</th>
<th>$t_2$ real time</th>
<th>$t_2$ vruntime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10</td>
<td>30.5</td>
</tr>
<tr>
<td>30</td>
<td>9.9</td>
<td>30</td>
<td>91.5</td>
</tr>
<tr>
<td>50</td>
<td>16.5</td>
<td>50</td>
<td>152.5</td>
</tr>
<tr>
<td>60</td>
<td>19.8</td>
<td>60</td>
<td>183</td>
</tr>
<tr>
<td>80</td>
<td>26.4</td>
<td>80</td>
<td>244</td>
</tr>
</tbody>
</table>
Example

Assume two tasks, \( t_1 \) and \( t_2 \) with nice values of 5 and -5 respectively; decay factors of 3.05 and .33 respectively.

The table reflects real time and virtual time values for each run.

<table>
<thead>
<tr>
<th>( t_1 ) real time</th>
<th>( t_1 ) vruntime</th>
<th>( t_2 ) real time</th>
<th>( t_2 ) vruntime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30.5</td>
<td>5</td>
<td>1.65</td>
</tr>
<tr>
<td>30</td>
<td>91.5</td>
<td>10</td>
<td>3.33</td>
</tr>
<tr>
<td>50</td>
<td>152.5</td>
<td>15</td>
<td>4.95</td>
</tr>
<tr>
<td>60</td>
<td>183</td>
<td>20</td>
<td>6.6</td>
</tr>
<tr>
<td>80</td>
<td>244</td>
<td>25</td>
<td>8.25</td>
</tr>
</tbody>
</table>
CFS – Example 7

Example

- Assume two tasks, \( t_1 \) and \( t_2 \) with nice values of -5 and 5 respectively; decay factors of .33 and 3.05 respectively
- The table reflects real time and virtual time values for each run

<table>
<thead>
<tr>
<th>( t_1 ) real time</th>
<th>( t_1 ) vruntime</th>
<th>( t_2 ) real time</th>
<th>( t_2 ) vruntime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>5</td>
<td>15.25</td>
</tr>
<tr>
<td>30</td>
<td>9.9</td>
<td>10</td>
<td>30.5</td>
</tr>
<tr>
<td>50</td>
<td>16.5</td>
<td>15</td>
<td>45.75</td>
</tr>
<tr>
<td>60</td>
<td>19.8</td>
<td>20</td>
<td>61</td>
</tr>
<tr>
<td>80</td>
<td>26.4</td>
<td>25</td>
<td>76.05</td>
</tr>
</tbody>
</table>
CFS - Process Selection

- CFS selects the process with the minimum virtual runtime
- Avoids having run queues per priority level
- What about a data structure that represents the collection of tasks?
  - A single queue would be slow
  - Multiple queues make sense if there are relatively small number of them
  - There are many values of virtual runtime
A red-black (RB) tree is a binary search tree, which means that for each node, the left subtree only contains keys less than the node's key, and the right subtree contains keys greater than or equal to it.

A red-black tree has further restrictions which guarantee that the longest root-leaf path is at most twice as long as the shortest root-leaf path. This bound on the height makes RB Trees more efficient than normal BSTs.

Operations are in $O(\log n)$ time.
CFS -- Task Selection

- The key for each node is the `vruntime` of the corresponding task.
- To pick the next task to run, simply take the leftmost node.
  - This node is pointed to by a variable to reduce traversals.

CFS Task Selection

- What if multiple tasks have the same vrun time value?
- You can store multiple values in a list at a node with a sequence number indicating its order in the list.
CFS- “Time Slice” Calculation

- CFS identifies a targeted latency (TL) which is an interval of time during which every runnable task should run at least once.

- Proportions of CPU time are allocated from the value of targeted latency.

- Targeted latency can increase if the number of active tasks grows beyond a threshold value.
CFS- "Time Slice" Calculation

- Proportions of CPU time is calculated based on a mapping of nice value to a weight value
  - Examples: -20 is mapped to 88761, 0 is mapped to 1024

- The general idea is that
  - Every process that changes nice value up by one level gets 10% less CPU power
  - Every process that changes nice value down by one gets 10% more CPU power
CFS- “Time Slice” Calculation

- The mapping from nice values to weights uses an array
- The array contains one entry for each nice level
- The multiplier between the entries is 1.25
CFS “Time Slice” Calculation

Example

- Targeted latency = 20 ms
- Two tasks, $t_1$ and $t_2$, with nice values of 0
- The weight value for nice 0 is 1,024
- The share for each task is $\frac{1024}{1024+1024} = 0.5$ and thus each task will have a “time slice” of 10 ms
CFS “Time Slice” Calculation

Example

- Targeted latency = 20 ms
- Two tasks, $t_1$ and $t_2$, with nice values of 0 and 5 respectively
- The weight value for nice 0 is 1,024 and the weight for nice 5 is 335
- The share for task $t_1$ is $\frac{1024}{1024+335} = 0.75$ and thus
  - task $t_1$ will have a “time slice” of $0.75 \times 20 = 15$ ms
- The share for task $t_2$ is $\frac{335}{1024+335} = 0.25$
  - task $t_2$ will have a “time slice” of $20 \times 0.25 = 5$ ms
CFS

- No static slices
  - The switching rate depends on the system load
- Each process receives a proportion of the processor’s time
  - Length depends on how many other processes are running
Other
MAC OS X

- Based on MACH and Unix BSD
- Priorities are categorized into priority bands
  - Normal: Applications
  - System high priority: Processes with higher priority than Normal
  - Kernel mode: Reserved for kernel processes
  - Real-time: For processes that must be guaranteed a slice of CPU time by a particular deadline
Priorities change dynamically
  - Based on wait time and amount of time that the process has had the processor
  - Stay within the same priority band
Reschedules every tenth of a second and recomputes priorities once every second
Process will relinquish CPU after time quantum or when it must wait for an I/O completion
Feedback prevents starvation
Android

- For mobile devices
- Today it is the most commonly used platform
- Uses Linux for device managers, memory management, process management
Summary

- We have examined scheduling in several contemporary operating systems