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 threads
- Realtime (RT) threads
- System (SYS) threads
- Fair share (FSS) threads
  - Fixed priority (FX) threads
  - Timeshare (TS) threads
  - Interactive (IA) threads
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

<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>

For time-sharing & Interactive processes
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

- As of Dec 2015
  - Webservers: W3Cook reports that 96.5% of web servers run Linux (1.5% run Windows)
  - Desktops/laptops: 1.5% use Linux
  - Mobile devices: Android (based on Linux kernel) is used in 80% of all mobile devices
  - Platform of choice for film industry

- Linux is free and open-source

- Please note that Linux uses the term “task”
History of Linux Scheduler

- **Linux version 1.2**
  - Used circular queue for runnable task management
  - Round-robin
  - Efficient for adding and removing processes
  - Fast and Simple
History of Linux Scheduler

- **Linux version 2.2**
  - Introduced the idea of scheduling class
  - Permitting scheduling policies for
    - Real-time tasks
    - Non-preemptible tasks
    - Non-real time task
History of Linux Scheduler

- Linux version 2.4
  - Divided time into epochs
  - Within each epoch, every task was allowed to execute up to its time slice
  - Applying goodness function to determine which task to execute next
  - Simple, $O(N)$, inefficient, lack of scalability, weak for real-time systems
History of Linux Scheduling

- Linux version 2.5
  - Implement scheduling algorithms in $O(1)$ time.
    - Scales well to multiple processors, each with many processes.
  - Problem: Not responsive to interactive applications
    - Complex, error prone logic
    - No guarantee of fairness

- Current: Linux version 2.6
Linux Scheduling

- Linux has these scheduling classes:
  - Real-time (RT) classes
  - Completely fair scheduler (CFS) class

- Tasks in RT have higher precedence than tasks in the CFS

- We will first start with a discussion of nice values which is related to CFS.
CFS - move

- The *Completely Fair Scheduler* (CFS) is a significant departure from the traditional UNIX process scheduler.
- Runs tasks with the “gravest need”
- Tries to guarantee fairness (CPU Usage)
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 Use

- Let’s say that you want to run a nightly backup script on a server system like a web hosting server.
- Backup scripts can produce high load and longer latency for other processes.
  - Backup scripts are both CPU and I/O intensive.
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.

- Time slice for a task $i$:
  - $TL \times W_i / (\text{Sum of all } W_i)$
CFS- Time Slice Calculation

- A nice value is mapped 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

- Calculating Schedule Latency
  - There is a lower bound on smallest time slice
  - Schedule latency is
    - Lower bound * number of processes
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
  - \( t_1 \) will have a “time slice” of \( 0.75 \times 20 = 15 \) ms
- The share for task \( t_2 \) is \( \frac{335}{1024+335} = 0.25 \)
  - \( t_2 \) will have a “time slice” of \( 20 \times 0.25 = 5 \) ms
CFS – Picking the next process

- Pick process with the weighted minimum runtime so far
- The virtual run time \((\text{vruntime})\) of a task is the actual runtime weighted by its niceness
- The value of \(\text{vruntime}\) is used by the scheduler to determine the next process to run
  - Process with the smallest \(\text{vruntime}\) is selected to run next
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: \texttt{vruntime} will be 200 milliseconds
- Nice value < 0: \texttt{vruntime} will be less than 200 milliseconds
- Nice value > 0: \texttt{vruntime} will be greater than 200 milliseconds

Smaller nice values result in values of \texttt{vruntime} that grow more slowly than higher nice values.
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>1024/3121 = .33</td>
</tr>
<tr>
<td>-1</td>
<td>1024/1277 = .80</td>
</tr>
<tr>
<td>0</td>
<td>1024/1024 = 1</td>
</tr>
<tr>
<td>1</td>
<td>1024/820 = 1.24</td>
</tr>
<tr>
<td>5</td>
<td>1024/335 = 3.05</td>
</tr>
</tbody>
</table>
CFS - Using vruntime

- CFS assigns each task a virtual runtime to account for how long a task has run.
- Example: Assume two tasks $t_1$ and $t_2$ with:
  - nice values of 0
  - Assume $t_1$ runs for 200 milliseconds and $t_2$ runs for 100 milliseconds which is followed by a lot of other tasks
  - $t_2$ will be selected before $t_1$ for execution
CFS - Using vruntime

- Example: Assume two tasks $t_1$ and $t_2$ with
  - nice values of 0 and 5 respectively
  - $\text{vruntime}_0$ and $\text{vruntime}_1$ are initially zero
  - Decay factors of 1 and 3.05 respectively
  - Assume $t_1$ runs for 200 milliseconds and $t_2$ runs for 100 milliseconds which is followed by a lot of other tasks
    - $\text{vruntime}_0$ is 200 milliseconds
    - $\text{vruntime}_1$ is 305 milliseconds
  - $t_1$ will be selected before $t_2$ for execution
  - Let’s say that $t_1$ runs again for 200 milliseconds; $\text{vruntime}_0$ is now 400 milliseconds
  - $t_2$ will now be selected before $t_1$ for execution
CFS Starvation

- Could a process starve?
  - No

- Let’s say that a task $t_1$ doesn’t get the processor while $t_2$ always does
  - At some point $t_1$ will have a smaller value for vruntime since it is never being incremented.
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
CFS - RB Trees

- 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.
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 `vruntime` value?

- You can store multiple values in a list at a node with a sequence number indicating its order in the list.
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
CFS

- As a user you can assign nice values greater than zero
- You need root to assign nice values less than zero
- Optimizing nice values for applications seems rather complex
  - Yes it can be
Linux Scheduler

- Linux has these scheduling classes:
  - Real-time (RT) class
  - Completely Fair Scheduler (CFS) class
- We discussed CFS in detail
- Tasks in the RT class have higher precedence than tasks in the CFS
- A task stays in a class unless the application requests a different class
Linux Scheduler

- 140 priority levels:
  - RT: There are 100 priority lists ranging from 0 to 99.
  - User tasks: 101-140 (nice values -20 to 19)
- Within RT there are are two scheduling policies
  - SCHED_RR: round-robin
  - SCHED_FIFO: FIFO
- Round-robin has precedence over FIFO
Linux Scheduler

- Each class has its own run queue
  - A run queue is a data structure for a collection of processes
RT Rules

- **FIFO**
  - Tasks that running with this policy are not preempted unless
    - A Higher priority process arrives
    - Blocked
  - However, 5% of the CPU is reserved for non-FIFO tasks.

- **RR**
  - Functions as described earlier
Scheduler Policies

- Scheduling policy is set by `sched_setscheduler()`

- Scheduling policies:
  - `SCHED_FIFO`: for special time-critical applications
  - `SCHED_RR`: round robin scheduling
  - `SCHED_BATCH`: for CPU-intensive tasks
  - `SCHED_IDLE`: for low prioritized tasks
  - `SCHED_OTHER`: default
## System Calls Related to Scheduling

<table>
<thead>
<tr>
<th>System Calls</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>nice</td>
<td>Change the priority of a CFS task</td>
</tr>
<tr>
<td>getpriority</td>
<td>Get the priority of a process</td>
</tr>
<tr>
<td>setpriority</td>
<td>Set priority of a process</td>
</tr>
<tr>
<td>sched_getscheduler</td>
<td>Get the scheduling policy of a process</td>
</tr>
<tr>
<td>sched_setscheduler</td>
<td>Set the scheduling policy and priority of a process</td>
</tr>
<tr>
<td>sched_getparam</td>
<td>Get the priority of a process</td>
</tr>
<tr>
<td>sched_setparam</td>
<td>Set the priority of a process</td>
</tr>
<tr>
<td>sched_yield</td>
<td>Relinquish the processor voluntarily</td>
</tr>
<tr>
<td>sched_get_priority_min</td>
<td>Get the minimum priority value for a policy</td>
</tr>
<tr>
<td>sched_get_priority_max</td>
<td>Get the maximum priority value for a policy</td>
</tr>
<tr>
<td>sched_rr_get_interval</td>
<td>Get the time quantum value for the RR policy</td>
</tr>
</tbody>
</table>
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
We have examined scheduling in several contemporary operating systems.