CS 370: OPERATING SYSTEMS
[CPU SCHEDULING]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALICKARA
Topics covered in this lecture

- Scheduling Algorithms
  - Priority Scheduling
  - Lottery scheduling
  - Round robin scheduling

- Scheduling Examples
  - Windows, Linux
Prediction of CPU bursts:
Make estimates based on past behavior

- $t_n$: Length of the $n^{th}$ CPU burst
- $\tau_n$: Estimate for the $n^{th}$ CPU burst
- $\alpha$: Controls weight of recent and past history
- $\tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n$

- Burst is predicted as an exponential average of the measured lengths of previous CPU bursts
\( \alpha \) controls the relative weight of recent and past history

\[ \tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n \]

- Value of \( t_n \) contains our most recent information, while \( \tau_n \) stores the past history

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

- \( \alpha \) is less than 1, \((1-\alpha)\) is also less than one
  - Each successive term has less weight than its predecessor
The choice of $\alpha$ in our predictive equation

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

- If $\alpha=0$, $\tau_{n+1} = \tau_n$
  - Current conditions are transient

- If $\alpha=1$, $\tau_{n+1} = t_n$
  - Only most recent bursts matter
  - History is assumed to be old and irrelevant
The choice of $\alpha$ in our predictive equation

- If $\alpha=1/2$
  - Recent history and past history are **equally weighted**

- With $\alpha = 1/2$; successive estimates of $\tau$
  
  $t_0/2 \quad t_0/4 + t_1/2 \quad t_0/8 + t_1/4 + t_2/2 \quad t_0/16 + t_1/8 + t_2/4 + t_3/2$

  - By the $3^{rd}$ estimate, weight of $t_0$ has dropped to $1/8$. 
PRIORITY SCHEDULING
Priority Scheduling

- **Priority** associated with each process
- CPU allocated to process with **highest** priority
- Can be preemptive or nonpreemptive
  - If preemptive: Preempt CPU from a lower priority process when a higher one is ready
Depiction of priority scheduling in action

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Here: Lower number means higher priority

Wait time = (6 + 0 + 16 + 18 + 1) / 5 = 8.2
How priorities are set

- Internally defined priorities based on:
  - Measured quantities
  - Time limits, memory requirements, # of open files, ratio (averages) of I/O to CPU burst

- External priorities
  - Criteria outside the purview of the OS
  - Importance of process, $ paid for usage, politics, etc.
Issue with priority scheduling

- Can leave lower priority processes waiting indefinitely

- Perhaps apocryphal tale:
  - MIT’s IBM 7094 shutdown (1973) found processes from 1967!
Coping with issues in priority scheduling: Aging

- **Gradually increase priority** of processes that wait for a long time

- **Example:**
  - Process with priority of 127 and increments every 15 minutes
  - Process priority becomes 0 in no more than 32 hours
Can SJF be thought as a priority algorithm?

- Priority is **inverse** of CPU burst
- The larger the burst, the lower the priority
ROUND ROBIN SCHEDULING
Round-Robin Scheduling

- Similar to FCFS scheduling
  - Preemption to enable switch between processes

- Ready queue is implemented as FIFO
  - Process Entry: PCB at tail of queue
  - Process chosen: From head of the queue

- CPU scheduler goes around ready queue
  - Allocates CPU to each process one after the other
    - CPU-bound up to a maximum of 1 quantum
Round Robin: Choosing the quantum

- Context switch is **time consuming**
  - Saving and loading registers and memory maps
  - Updating tables
  - Flushing and reloading memory cache

- What if quantum is 4 ms and context switch overhead is 1 ms?
  - 20% of CPU time thrown away in administrative overhead
Round Robin: Improving efficiency by increasing quantum

- Let’s say quantum is 100 ms and context-switch is 1 ms
  - Now wasted time is only 1%

- But what if 50 concurrent requests come in?
  - Each with widely varying CPU requirements
  - 1st one starts immediately, 2nd one 100 ms later, …
  - The last one may have to wait for 5 seconds!
  - A shorter quantum would have given them better service
If quantum is set longer than mean CPU burst?

- **Preemption will not happen very often**
- Most processes will perform a blocking operation before quantum runs out
- Switches happens only when process blocks and cannot continue
Quantum: Summarizing the possibilities

- Too short?
  - Too many context switches
  - Lowers CPU efficiency

- Too long?
  - Poor responses to interactive requests
LOTTERY SCHEDULING
Lottery scheduling

- Give processes **lottery tickets** for various system resources
  - E.g. CPU time

- When a scheduling decision has to be made
  - Lottery ticket is *chosen at random*
  - Process holding **ticket gets** the resource
All processes are equal, but some processes are more equal than others

- More important processes are given **extra tickets**
  - Increase their odds of winning

- Let’s say there are 100 outstanding tickets
  - 1 process holds 20 of these
  - Has 20% chance of winning each lottery

- A process holding a fraction $f$ of tickets
  - Will get about a fraction $f$ of the resource
Lottery Scheduling: Properties (1/2)

- Highly **responsive**
  - Chance of winning is proportional to tickets

- Cooperating processes may **exchange** tickets
  - Process A sends request to B, and then hands B all its tickets for a faster response

- Avoids starvation
  - Each process holds at least one ticket .... Is guaranteed to have a non-zero probability of being scheduled
Lottery Scheduling: Properties (2/2)

- Solves problems that are *difficult to handle* in other scheduling algorithms

- E.g. video server that is managing processes that feed video frames to clients
  - Clients need frames at 10, 20, and 25 frames/sec
  - Allocate processes 10, 20 and 25 tickets
    - CPU divided into approximately 10:20:25
MULTIPROCESSOR/CORE ENVIRONMENTS
Load balancing: Migration based approaches

- **Push migration**
  - Specific task periodically checks for *imbalance*
  - Balances load by **pushing** processes from overloaded to less-busy processors.

- **Pull migration**
  - Idle processor pulls a waiting task from busy processor

- **Schemes not mutually exclusive:** used in parallel
  - Linux: Runs a load-balancing algorithm
    - Every 200 ms (**PUSH** migration)
    - When processor run-queue is empty (**PULL** migration)
Multicore processors place multiple processor cores on same physical chip

- Each core has its own register set
  - Appears to the OS as a separate physical processor

- Recent designs implement 2 or more hardware threads per core
  - If there is a memory stall (due to cache miss) on one thread, **switch** to another hardware thread
Coping with memory stalls

Thread

Thread 1

Thread 0

Compute cycle

Memory cycle

C M C M C M C M C M C M C M C M C M C M C M C M
Multithreading a processor

- **Coarse** grained
  - Thread executes on processor till a memory stall
  - Switch to another thread

- Switching between threads
  - *Flush* the instruction pipeline
  - *Refill* pipeline as new thread executes

- **Finer** grained (or interleaved)
  - Switch between threads at the boundary of an instruction cycle
  - Design includes logic for thread switching: overheads are low
Tiered scheduling on multicore processors

- First-level: OS
  - OS chooses which software thread to run on each hardware thread

- Second-level: Core
  - Decides which hardware thread to run

- UltraSPARC T1
  - 8 cores, and 4 hardware threads/core
  - Round robin to schedule hardware threads on core
SCHEDULING EXAMPLES
Scheduling examples

- Solaris
- Windows
- Linux
Scheduling Example: Solaris

- Thread belongs to 1 of six classes

- **Inverse relationship** between priorities and time slices
  - Higher priority = smaller time slice
    - Interactive processes
    - Priority 59: 20 millisecond quantum
  - Lower priority = bigger time slice
    - CPU bound processes
    - Priority 0 = 200 millisecond quantum
Solaris scheduling

Global Priority

- highest
- lowest

Scheduling order
- first
- last

Priority levels:
- interrupt threads
- realtime threads
- system threads
- interactive threads
- timeshare threads
- fixed priority threads
- fair share threads
Windows XP Scheduling
Scheduling Example: Windows XP

- Priority-based, preemptive scheduling
  - Highest priority thread will always run

- 32-level priority scheme
  - Variable class: priorities 1-15
  - Realtime class: priorities 16-31
  - Memory management thread: priority 0
Dispatcher in Windows XP

- Use a **queue** for each scheduling priority

- **Traverse** the queues from highest to lowest
  - *Until* it finds a thread that is ready to run

- If no ready thread is found?
  - Dispatcher will execute a special thread: **idle thread**
Idle thread in Windows

- Primary purpose is to **eliminate a special case**
  - Cases when no threads are runnable or ready
  - Idle threads are always in a *ready* state
    - If not already running

- Scheduler can always find a thread to execute

- If there are other eligible threads?
  - Scheduler will never select the idle thread
Idle threads in Windows

- Windows thread priorities go from 0-31
  - Idle thread priority can be thought of as -1

- Threads in the system idle process can also implement CPU power saving
  - On x86 processors, run a loop of `halt` instructions
  - Causes CPU to **turn off internal components**
    - Until an interrupt request arrives
  - Recent versions also **reduce the CPU clock speed**
Time consumed by the idle process

- It may seem that the idle process is monopolizing the CPU
  - It is merely acting as a *placeholder during free time*
  - Proof that no other process wants that CPU time
Scheduling Example: Windows XP
Identifies 6 priority classes for threads

- Thread priorities for classes are variable
- Relative priority for thread within a class
Windows XP priorities: Threads within a priority class also have a relative priority

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

Base priority for each thread class
Windows XP: Managing the priority of variable priority threads

- **Lowering** the priority of a thread
  - When a thread’s quantum runs out
    - Lower priority BUT not below base priority
Windows XP: Boosting the priority of threads

- Upon release from a **wait** operation
  - Thread waiting for keyboard IO gets big boost
  - Thread waiting for disk IO gets **moderate** boost

- Window with which user is **interacting**
  - Gives good response for interactive thread

- When process moves to **foreground**
  - Scheduling quantum boosted by 3
LINUX SCHEDULING
Highlights of Linux scheduling (1)

- Scheduling algorithm runs in constant time
- Implements real-time scheduling (POSIX 1.b)
  - Real-time tasks have static priorities
  - Other tasks have dynamic priorities
- We look at the algorithm in kernel version 2.5
  - Revised again in version 2.6.23 of the kernel [called: Completely Fair Scheduler]
Highlights of Linux scheduling (2)

- Preemptive, priority-based algorithm

- Two separate priority ranges
  - **Real-time** range: 0-99
  - **Nice** value: 100-140

- Numerically *lower* values indicate *higher* priority
Highlights of Linux scheduling (3)

- **UNLIKE** Solaris and Windows
  - Higher priority tasks = higher quanta
  - Lower priority tasks = lower quanta

- Task’s **interactivity** determined by
  - *Sleeping times* waiting for I/O
Task execution in Linux

- Task eligible for execution as long as it has time remaining in its time slice

- When a task has exhausted its time slice?
  - Ineligible for execution again, until …
  - All other tasks have exhausted their time quanta
Each runqueue contains two priority arrays: Active and Expired

- **Active array**
  - All tasks with time remaining in their time slices

- **Expired array**
  - Contains all expired tasks

- Each priority array contains list of tasks **indexed** according to priority
Swapping the active and expired arrays

- When **all tasks have exhausted** their time slices?
  - Active array is empty

- The two priority arrays are **exchanged**
  - Expired array becomes the active array, and vice versa
Linux: Tasks indexed according to priority

<table>
<thead>
<tr>
<th>Priority</th>
<th>ACTIVE ARRAY</th>
<th>EXPIRED ARRAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>[1]</td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td>[140]</td>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Little’s formula

- $n$ be the average queue length
- $W$ average wait time in the queue
- $\lambda$ average arrival rate of processes

When a process waits for time $W$

\[ \lambda \times W \] processes arrives

Steady state: Processes leaving = Processes arriving

\[ n = \lambda \times W \]
The contents of this slide-set are based on the following references
