CS162
Operating Systems and
Systems Programming
Lecture 12

Scheduling 3: Starvation (Finished), Deadlock

March 1st, 2022
Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Real-Time Scheduling

• Goal: **Predictability** of Performance!
  – We need to predict with confidence worst case response times for systems!
  – In RTS, performance guarantees are:
    » Task- and/or class centric and often ensured a priori
  – In conventional systems, performance is:
    » System/throughput oriented with post-processing (… wait and see …)
  – Real-time is about enforcing predictability, and does not equal fast computing!!!

• Hard real-time: for time-critical safety-oriented systems
  – Meet all deadlines (if at all possible)
  – Ideally: determine in advance if this is possible
  – Earliest Deadline First (EDF), Least Laxity First (LLF),
    Rate-Monotonic Scheduling (RMS), Deadline Monotonic Scheduling (DM)

• Soft real-time: for multimedia
  – Attempt to meet deadlines with high probability
  – Constant Bandwidth Server (CBS)
Are SRTF and MLFQ Prone to Starvation?

- In SRTF, long jobs are starved in favor of short ones
  - Same fundamental problem as priority scheduling
- MLFQ is an approximation of SRTF, so it suffers from the same problem
Cause for Starvation: Priorities?

• The policies we’ve studied so far:
  – *Always prefer to give the CPU to a prioritized job*
  – Non-prioritized jobs may never get to run

• But priorities were a means, not an end
• Our end goal was to serve a mix of CPU-bound, I/O bound, and Interactive jobs effectively on common hardware
  – Give the I/O bound ones enough CPU to issue their next file operation and wait (on those slow discs)
  – Give the interactive ones enough CPU to respond to an input and wait (on those slow humans)
  – Let the CPU bound ones grind away without too much disturbance
Recall: Changing Landscape…

Bell’s Law: New computer class every 10 years

Number crunching, Data Storage, Massive Inet Services, ML, ...

Productivity, Interactive

Streaming from/to the physical world

The Internet of Things!

Laptop

PDA

Mainframe

Mini

Workstation

PC

Laptop

PDA

Cell

Mote
Changing Landscape of Scheduling

• Priority-based scheduling rooted in “time-sharing”
  – Allocating precious, limited resources across a diverse workload
    » CPU bound, vs interactive, vs I/O bound
• 80’s brought about personal computers, workstations, and servers on networks
  – Different machines of different types for different purposes
  – Shift to fairness and avoiding extremes (starvation)
• 90’s emergence of the web, rise of internet-based services, the data-center-is-the-computer
  – Server consolidation, massive clustered services, huge flashcrowds
  – It’s about predictability, 95th percentile performance guarantees
DOES PRIORITIZING SOME JOBS NECESSARILY STARVE THOSE THAT AREN’T PRIORITIZED?
Key Idea: Proportional-Share Scheduling

• The policies we’ve studied so far:
  – **Always prefer to give the CPU to a prioritized job**
  – Non-prioritized jobs may never get to run

• Instead, we can share the CPU *proportionally*
  – Give each job a share of the CPU according to its priority
  – Low-priority jobs get to run less often
  – But all jobs can at least make progress (no starvation)
Recall: Lottery Scheduling

- Given a set of jobs (the mix), provide each with a share of a resource
  - e.g., 50% of the CPU for Job A, 30% for Job B, and 20% for Job C
- Idea: Give out tickets according to the proportion each should receive,
- Every quantum (tick): draw one at random, schedule that job (thread) to run
Lottery Scheduling: Simple Mechanism

- \( N_{\text{ticket}} = \sum N_i \)
- Pick a number \( d \) in 1 .. \( N_{\text{ticket}} \) as the random “dart”
- Jobs record their \( N_i \) of allocated tickets
- Order them by \( N_i \)
- Select the first \( j \) such that \( \sum N_i \) up to \( j \) exceeds \( d \).
Unfairness

- E.g., Given two jobs A and B of same run time (# Qs) that are each supposed to receive 50%,
  \[ U = \frac{\text{finish time of first}}{\text{finish time of last}} \]
- As a function of run time
Stride Scheduling

• Achieve proportional share scheduling without resorting to randomness, and overcome the “law of small numbers” problem.

• “Stride” of each job is $\frac{\text{big} \cdot W}{N_i}$
  
  – The larger your share of tickets, the smaller your stride
  
  – Ex: $W = 10,000$, $A=100$ tickets, $B=50$, $C=250$
  
  – A stride: 100, B: 200, C: 40

• Each job has a “pass” counter

• Scheduler: pick job with lowest pass, runs it, add its stride to its pass

• Low-stride jobs (lots of tickets) run more often
  
  – Job with twice the tickets gets to run twice as often

• Some messiness of counter wrap-around, new jobs, …
Linux Completely Fair Scheduler (CFS)

- Goal: Each process gets an equal share of CPU
  - $N$ threads “simultaneously” execute on $\frac{1}{N}$ of CPU
  - The *model* is somewhat like simultaneous multithreading – each thread gets $\frac{1}{N}$ of the cycles

- In general, can’t do this with real hardware
  - OS needs to give out full CPU in time slices
  - Thus, we must use something to keep the threads roughly in sync with one another
Linux Completely Fair Scheduler (CFS)

- Basic Idea: track CPU time per thread and schedule threads to match up average rate of execution

- **Scheduling Decision:**
  - “Repair” illusion of complete fairness
  - Choose thread with minimum CPU time
  - Closely related to Fair Queueing

- Use a heap-like scheduling queue for this…
  - $O(\log N)$ to add/remove threads, where $N$ is number of threads

- Sleeping threads don’t advance their CPU time, so they get a boost when they wake up again…
  - Get interactivity automatically!

CFS: Average rate of execution $= \frac{1}{N}$
Linux CFS: Responsiveness/Starvation Freedom

• In addition to fairness, we want **low response time** and starvation freedom
  – Make sure that everyone gets to run at least a bit!

• Constraint 1: *Target Latency*
  – Period of time over which every process gets service
  – Quanta = Target Latency / n

• Target Latency: 20 ms, 4 Processes
  – Each process gets 5ms time slice

• Target Latency: 20 ms, 200 Processes
  – Each process gets 0.1ms time slice (!!!)
  – Recall Round-Robin: large context switching overhead if slice gets to small
Linux CFS: Throughput

- **Goal: Throughput**
  - Avoid excessive overhead
- **Constraint 2: Minimum Granularity**
  - Minimum length of any time slice

- **Target Latency 20 ms, Minimum Granularity 1 ms, 200 processes**
  - Each process gets 1 ms time slice
Aside: Priority in Unix – Being Nice

• The industrial operating systems of the 60s and 70's provided priority to enforced desired usage policies.
  – When it was being developed at Berkeley, instead it provided ways to “be nice”.
• nice values range from -20 to 19
  – Negative values are “not nice”
  – If you wanted to let your friends get more time, you would nice up your job
• Scheduler puts higher nice-value tasks (lower priority) to sleep more …
  – In O(1) scheduler, this translated fairly directly to priority (and time slice)
• How does this idea translate to CFS?
  – Change the rate of CPU cycles given to threads to change relative priority
Linux CFS: Proportional Shares

- What if we want to give more CPU to some and less to others in CFS (proportional share)?
  - Allow different threads to have different rates of execution (cycles/time)
- Use weights! Key Idea: Assign a weight $w_i$ to each process $i$ to compute the switching quanta $Q_i$
  - Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$
  - Weighted Share: $Q_i = \left(\frac{w_i}{\sum_{p} w_p}\right) \cdot \text{Target Latency}$
- Reuse nice value to reflect share, rather than priority,
  - Remember that lower nice value $\Rightarrow$ higher priority
  - CFS uses nice values to scale weights exponentially: Weight=$1024/(1.25)^\text{nice}$
    » Two CPU tasks separated by nice value of 5 $\Rightarrow$
      Task with lower nice value has 3 times the weight, since $(1.25)^5 \approx 3$
- So, we use “Virtual Runtime” instead of CPU time
Example: Linux CFS: Proportional Shares

- Target Latency = 20ms
- Minimum Granularity = 1ms
- Example: Two CPU-Bound Threads
  - Thread A has weight 1
  - Thread B has weight 4
- Time slice for A? 4 ms
- Time slice for B? 16 ms
Linux CFS: Proportional Shares

- Track a thread's *virtual* runtime rather than its true physical runtime
  - Higher weight: Virtual runtime increases more slowly
  - Lower weight: Virtual runtime increases more quickly
- Scheduler’s Decisions are based on Virtual CPU Time
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - $O(1)$ time to find next thread to run (top of heap!)
  - $O(\log N)$ time to perform insertions/deletions
    » Cache the item at far left (item with earliest vruntime)
  - When ready to schedule, grab version with smallest vruntime (which will be item at
Choosing the Right Scheduler

<table>
<thead>
<tr>
<th>I Care About:</th>
<th>Then Choose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Throughput</td>
<td>FCFS</td>
</tr>
<tr>
<td>Avg. Response Time</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>I/O Throughput</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>Fairness (CPU Time)</td>
<td>Linux CFS</td>
</tr>
<tr>
<td>Fairness – Wait Time to Get CPU</td>
<td>Round Robin</td>
</tr>
<tr>
<td>Meeting Deadlines</td>
<td>EDF</td>
</tr>
<tr>
<td>Favoring Important Tasks</td>
<td>Priority</td>
</tr>
</tbody>
</table>
A Final Word On Scheduling

• When do the details of the scheduling policy and fairness really matter?
  – When there aren’t enough resources to go around
• When should you simply buy a faster computer?
  – (Or network link, or expanded highway, or …)
  – One approach: Buy it when it will pay for itself in improved response time
    » Perhaps you’re paying for worse response time in reduced productivity, customer angst, etc…
    » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization→100%
• An interesting implication of this curve:
  – Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
  – Argues for buying a faster X when hit “knee” of curve
Administrivia

- Prof Joseph’s office hours: Tuesdays 1-2pm and Thursdays 12-1 (room TBD)

- Project 1 (code, report, evals) all due TODAY (Tuesday 3/1)

- Homework 2 is due this Thursday 3/3

- Midterm 2 conflict requests are due this Friday 3/4
Deadlock: A Deadly type of Starvation

• Starvation: thread waits indefinitely
  – Example, low-priority thread waiting for resources constantly in use by high-priority threads

• Deadlock: circular waiting for resources
  – Thread A owns Res 1 and is waiting for Res 2
  – Thread B owns Res 2 and is waiting for Res 1

• Deadlock \implies Starvation but not vice versa
  – Starvation can end (but doesn’t have to)
  – Deadlock can’t end without external intervention
Example: Single-Lane Bridge Crossing

CA 140 to Yosemite National Park
Bridge Crossing Example

• Each segment of road can be viewed as a resource
  – Car must own the segment under them
  – Must acquire segment that they are moving into

• For bridge: must acquire both halves
  – Traffic only in one direction at a time

• Deadlock: Shown above when two cars in opposite directions meet in middle
  – Each acquires one segment and needs next
  – Deadlock resolved if one car backs up (preempt resources and rollback)
    » Several cars may have to be backed up

• Starvation (not Deadlock): 
  – East-going traffic really fast ⇒ no one gets to go west
Deadlock with Locks

Thread A:  
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:  
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

- This lock pattern exhibits non-deterministic deadlock  
  - Sometimes it happens, sometimes it doesn’t!
- This is really hard to debug!
Deadlock with Locks: “Unlucky” Case

Thread A:
- x.Acquire();
- y.Acquire(); <stalled> <unreachable>
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire(); <stalled> <unreachable>
- ...
- x.Release();
- y.Release();

Neither thread will get to run ⇒ Deadlock
Deadlock with Locks: “Lucky” Case

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

Sometimes, schedule won’t trigger deadlock!
Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right, but is blocked by other trains
- Similar problem to multiprocessor networks
  - Wormhole-Routed Network: Messages trail through network like a “worm”
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  - Called “dimension ordering” (X then Y)
Other Types of Deadlock

• Threads often block waiting for resources
  – Locks
  – Terminals
  – Printers
  – CD drives
  – Memory

• Threads often block waiting for other threads
  – Pipes
  – Sockets

• You can deadlock on any of these!
Deadlock with Space

<table>
<thead>
<tr>
<th>Thread A:</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
</tbody>
</table>

If only 2 MB of space, we get same deadlock situation
Dining Lawyers Problem

- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat

- What if all grab at same time?
  - Deadlock!

- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat

- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards
  - Can we formalize this requirement somehow?
Four requirements for occurrence of Deadlock

• Mutual exclusion
  – Only one thread at a time can use a resource.

• Hold and wait
  – Thread holding at least one resource is waiting to acquire additional resources held by other threads

• No preemption
  – Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

• Circular wait
  – There exists a set \{T_1, \ldots, T_n\} of waiting threads
    » \(T_1\) is waiting for a resource that is held by \(T_2\)
    » \(T_2\) is waiting for a resource that is held by \(T_3\)
    » …
    » \(T_n\) is waiting for a resource that is held by \(T_1\)
Detecting Deadlock: Resource-Allocation Graph

• System Model
  – A set of Threads $T_1, T_2, \ldots, T_n$
  – Resource types $R_1, R_2, \ldots, R_m$
    CPU cycles, memory space, I/O devices
  – Each resource type $R_i$ has $W_i$ instances
  – Each thread utilizes a resource as follows:
    » Request() / Use() / Release()

• Resource-Allocation Graph:
  – $V$ is partitioned into two types:
    » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  – request edge – directed edge $T_1 \rightarrow R_j$
  – assignment edge – directed edge $R_j \rightarrow T_i$
Resource-Allocation Graph Examples

- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock
**Deadlock Detection Algorithm**

- Let $[X]$ represent an m-ary vector of non-negative integers (quantities of resources of each type):
  
  $[\text{FreeResources}]$: Current free resources each type  
  $[\text{Request}_X]$: Current requests from thread $X$  
  $[\text{Alloc}_X]$: Current resources held by thread $X$

- See if tasks can eventually terminate on their own

  $[\text{Avail}] = [\text{FreeResources}]$
  Add all nodes to UNFINISHED
  do {
    done = true
    Foreach node in UNFINISHED {
      if ($[\text{Request}_{\text{node}}] \leq [\text{Avail}]$) {
        remove node from UNFINISHED
        $[\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{\text{node}}]$
        done = false
      }
    }
  } until(done)

- Nodes left in UNFINISHED $\Rightarrow$ deadlocked
How should a system deal with deadlock?

- Four different approaches:
  1. **Deadlock prevention**: write your code in a way that it isn’t prone to deadlock
  2. **Deadlock recovery**: let deadlock happen, and then figure out how to recover from it
  3. **Deadlock avoidance**: dynamically delay resource requests so deadlock doesn’t happen
  4. **Deadlock denial**: ignore the possibility of deadlock

- Modern operating systems:
  - Make sure the *system* isn’t involved in any deadlock
  - Ignore deadlock in applications
    - “Ostrich Algorithm”
Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn’t have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    » Bay bridge with 12,000 lanes. Never wait!
    » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don’t allow waiting
  - How the phone company avoids deadlock
    » Call Mom in Toledo, works way through phone network, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    » Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    » Consider: driving to San Francisco; when hit traffic jam, suddenly you’re transported back home and told to retry!
(Virtually) Infinite Resources

Thread A
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)

Thread B
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)

• With virtual memory we have “infinite” space so everything will just succeed, thus above example won’t deadlock
  – Of course, it isn’t actually infinite, but certainly larger than 2MB!
Techniques for Preventing Deadlock

• Make all threads request everything they’ll need at the beginning.
  – Problem: Predicting future is hard, tend to over-estimate resources
  – Example:
    » If need 2 chopsticks, request both at same time
    » Don’t leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time

• Force all threads to request resources in a particular order preventing any cyclic use of resources
  – Thus, preventing deadlock
  – Example (\( x.\text{Acquire}(), y.\text{Acquire}(), z.\text{Acquire}(), \ldots \))
    » Make tasks request disk, then memory, then…
    » Keep from deadlock on freeways around SF by requiring everyone to go clockwise
Summary

• Four conditions for deadlocks
  – Mutual exclusion
  – Hold and wait
  – No preemption
  – Circular wait

• Techniques for addressing Deadlock
  – Deadlock prevention:
    » write your code in a way that it isn’t prone to deadlock
  – Deadlock recovery:
    » let deadlock happen, and then figure out how to recover from it
  – Deadlock avoidance:
    » dynamically delay resource requests so deadlock doesn’t happen
    » Banker’s Algorithm provides on algorithmic way to do this
  – Deadlock denial:
    » ignore the possibility of deadlock