CS162
Operating Systems and
Systems Programming
Lecture 12

Scheduling 3: Scheduling Cont’d & Deadlock

October 5th, 2021
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
DOES PRIORITIZING SOME JOBS *NECESSARILY* STARVE THOSE THAT AREN’T PRIORITIZED?
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)
Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower priority value ⇒ higher priority (for nice values)
  - Highest priority value ⇒ Lower priority (for realtime values)
  - All algorithms $O(1)$
    » Timeslices/priorities/interactivity credits all compute when job finishes time slice
    » 140-bit bit mask indicates presence or absence of job at given priority level

- Two separate priority queues: “active” and “expired”
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped

- Timeslice depends on priority – linearly mapped onto timeslice range
  - Like a multi-level queue (one queue per priority) with different timeslice at each level
  - Execution split into “Timeslice Granularity” chunks – round robin through priority
Linux O(1) Scheduler

- Lots of ad-hoc heuristics
  - Try to boost priority of I/O-bound tasks
  - Try to boost priority of starved tasks
O(1) Scheduler Continued

• Heuristics
  – User-task priority adjusted ±5 based on heuristics
    » P→sleep_avg = (sleep_time – run_time) x coefficient
    » Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)
  – Interactive Credit
    » Earned when a task sleeps for a “long” time
    » Spend when a task runs for a “long” time
    » IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
  – However, “interactive tasks” get special dispensation
    » To try to maintain interactivity
    » Placed back into active queue, unless some other task has been starved for too long…

• Real-Time Tasks
  – Always preempt non-RT tasks
  – No dynamic adjustment of priorities
  – Scheduling schemes:
    » SCHED_FIFO: preempts other tasks, no timeslice limit
    » SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority
Proportional-Share Scheduling

• Instead using priorities, 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 \( \ldots N_{\text{ticket}} \) 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 \( d \).
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 $= \frac{1}{N}$
• 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 (n: number of processes)
• 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, 100 processes
  - Each process gets 1 ms time slice
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: 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: \( \text{Weight}=1024/(1.25)^{\text{nice}} \)
    » Two CPU tasks separated by nice value of 5 \( \Rightarrow \)
      Task with lower nice value has 3 times the weight, since \( (1.25)^5 \approx 3 \)
• Note: in practice “Virtual Runtime” instead of CPU time (but won’t discuss this in the lecture)
# 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>
How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - takes a predetermined workload and compute the performance of each algorithm for that workload

- Queueing models
  - Mathematical approach for handling stochastic workloads

- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data
  - Most flexible/general
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 $\to 100\%$

- 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
Announcements

• Midterm 1: Planning to release the grades by next Monday, 10/11
• Project 1 due Wednesday, 10/7 (code, final report, and peer evals all due)
  – Will incur slip day if any of them are late
• Group evaluations coming up for Project 1
  – Every person gets 20 pts/partner which they hand out as they wish
  – No points to yourself!
  – Projects are a zero-sum game: you must participate in your group!
• OH:
  – TAs only spend 15 minutes per student
  – Must have detailed ticket filled out and GDB pulled up
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 $\Rightarrow$ 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:
    » \text{Request()} / \text{Use()} / \text{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 \( T_i \rightarrow R_j \)
  – assignment edge – directed edge \( R_j \rightarrow T_i \)
Resource-Allocation Graph Examples

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

<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>

• 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 ($x$.Acquire(), $y$.Acquire(), $z$.Acquire(),…)
    » Make tasks request disk, then memory, then…
    » Keep from deadlock on freeways around SF by requiring everyone to go clockwise
Request Resources Atomically (1)

Rather than:
Thread A:
  x.Acquire();
  y.Acquire();
  ...
  y.Release();
  x.Release();
Thread B:
  y.Acquire();
  x.Acquire();
  ...
  x.Release();
  y.Release();

Consider instead:
Thread A:
  Acquire_both(x, y);
  ...
  y.Release();
  x.Release();
Thread B:
  Acquire_both(y, x);
  ...
  x.Release();
  y.Release();
Request Resources Atomically (2)

Or consider this:

Thread A

z.Acquire();
x.Acquire();
y.Acquire();
z.Release();
...
y.Release();
x.Release();

Thread B

z.Acquire();
y.Acquire();
x.Acquire();
z.Release();
...
x.Release();
y.Release();
Acquire Resources in Consistent Order

Rather than:

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

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

Consider instead:

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

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

Does it matter in which order the locks are released?
Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- 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)
Techniques for Recovering from Deadlock

• Terminate thread, force it to give up resources
  – In Bridge example, Godzilla picks up a car, hurls it into the river.  Deadlock solved!
  – Hold dining lawyer in contempt and take away in handcuffs
  – But, not always possible – killing a thread holding a mutex leaves world inconsistent

• Preempt resources without killing off thread
  – Take away resources from thread temporarily
  – Doesn’t always fit with semantics of computation

• Roll back actions of deadlocked threads
  – Hit the rewind button on TiVo, pretend last few minutes never happened
  – For bridge example, make one car roll backwards (may require others behind him)
  – Common technique in databases (transactions)
  – Of course, if you restart in exactly the same way, may reenter deadlock once again

• Many operating systems use other options
Another view of virtual memory: Pre-empting Resources

<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>

• Before: 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!

• Alternative view: we are “pre-empting” memory when paging out to disk, and giving it back when paging back in
  – This works because thread can’t use memory when paged out
Techniques for Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in deadlock
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

**THIS DOES NOT WORK!!!!**

- Example:

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

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

  Blocks…

  Wait?

  But it’s already too late…
Deadlock Avoidance: Three States

- Safe state
  - System can delay resource acquisition to prevent deadlock

- Unsafe state
  - No deadlock yet…
  - But threads can request resources in a pattern that *unavoidably* leads to deadlock

- Deadlocked state
  - There exists a deadlock in the system
  - Also considered “unsafe”
Deadlock Avoidance

• Idea: When a thread requests a resource, OS checks if it would result in deadlock an unsafe state
  – If not, it grants the resource right away
  – If so, it waits for other threads to release resources

• Example:

Thread A:

\[
\begin{align*}
& \text{x.Acquire();} \\
& \text{y.Acquire();} \\
& \text{...} \\
& \text{y.Release();} \\
& \text{x.Release();}
\end{align*}
\]

Thread B:

\[
\begin{align*}
& \text{y.Acquire();} \\
& \text{x.Acquire();} \\
& \text{...} \\
& \text{x.Release();} \\
& \text{y.Release();}
\end{align*}
\]

Wait until Thread A releases mutex X
Banker’s Algorithm for Avoiding Deadlock

• Toward right idea:
  – State maximum (max) resource needs in advance
  – Allow particular thread to proceed if:
    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

• Banker’s algorithm (less conservative):
  – Allocate resources dynamically
    » Evaluate each request and grant if some
      ordering of threads is still deadlock free afterward
    » Technique: pretend each request is granted, then run deadlock detection
      algorithm, substituting:
        ([Max]node-[Alloc]node <= [Avail]) for ([Request]node <= [Avail])
      Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Avoiding Deadlock

\[ \text{[Avail]} = \text{[FreeResources]} \]

Add all nodes to UNFINISHED

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

» Evaluate each request and grant if some ordering of threads is still deadlock free afterward

» Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
\[
(\text{[Max}_\text{node}]-\text{[Alloc}_\text{node}] \leq \text{[Avail]}) \text{ for } (\text{[Request}_\text{node}] \leq \text{[Avail]})
\]

Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Avoiding Deadlock

<table>
<thead>
<tr>
<th>[Avail] = [FreeResources]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add all nodes to UNFINISHED</td>
</tr>
<tr>
<td>do {</td>
</tr>
<tr>
<td>done = true</td>
</tr>
<tr>
<td>Foreach node in UNFINISHED {</td>
</tr>
<tr>
<td>if ([Max_{node}]-[Alloc_{node}] &lt;= [Avail]) {</td>
</tr>
<tr>
<td>remove node from UNFINISHED</td>
</tr>
<tr>
<td>[Avail] = [Avail] + [Alloc_{node}]</td>
</tr>
<tr>
<td>done = false</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>} until(done)</td>
</tr>
</tbody>
</table>

» Evaluate each request and grant if some ordering of threads is still deadlock free afterward

» Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

$$([\text{Max}_{node}]-[\text{Alloc}_{node}] \leq [\text{Avail}])$$ for $$([\text{Request}_{node}] \leq [\text{Avail}])$$

Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:
    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

- Banker’s algorithm (less conservative):
  - Allocate resources dynamically
    » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
      \((\text{Max}_{\text{node}} - \text{Alloc}_{\text{node}}) \leq \text{Avail})\) for \((\text{Request}_{\text{node}} \leq \text{Avail})\)
      Grant request if result is deadlock free (conservative!)
  - Keeps system in a “SAFE” state: there exists a sequence \(\{T_1, T_2, \ldots, T_n\}\) with \(T_1\) requesting all remaining resources, finishing, then \(T_2\) requesting all remaining resources, etc..
Banker’s Algorithm Example

• Banker’s algorithm with dining lawyers
  – “Safe” (won’t cause deadlock) if when try to grab chopstick either:
    » Not last chopstick
    » Is last chopstick but someone will have two afterwards

  – What if k-handed lawyers? Don’t allow if:
    » It’s the last one, no one would have k
    » It’s 2\text{nd} to last, and no one would have k-1
    » It’s 3\text{rd} to last, and no one would have k-2
    » …
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 an algorithmic way to do this
  – **Deadlock denial:**
    » ignore the possibility of deadlock