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-Monitonic Scheduling (RMS), Deadline Monotonic Scheduling (DM)
- Soft real-time: for multimedia
  - Attempt to meet deadlines with high probability
  - Constant Bandwidth Server (CBS)

Recall: Changing Landscape…

- **Bell's Law**: New computer class every 10 years
- **Computers Per Person**
  - 1:10³
  - 1:10⁶
  - 1:1
  - 10³:1

- **Mainframe** → **Workstation** → **PC** → **Laptop** → **PDA** → **Cell** → **Mote** → **The Internet of Things!**
- Number crunching, Data Storage, Massive Inet Services, ML, …
- Productivity, Interactive
- Streaming from/to the physical world

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
**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
- But priorities were a means, not an end:
  - Give priority to interactive tasks or I/O tasks for responsiveness
  - Lower priority given to long running tasks
- Instead, we can share the CPU proportionally
  - Give each job a share of the CPU according to its priority
  - Low-priority jobs get smaller share of CPU
  - But all jobs can at least make progress (no starvation)
- This idea is closely related to fair queueing

**Lottery Scheduling**

- Simple Idea:
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

**Lottery Scheduling Example (Cont.)**

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

<table>
<thead>
<tr>
<th># short jobs</th>
<th># long jobs</th>
<th>% of CPU each short job gets</th>
<th>% of CPU each long job gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td></td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>0/2</td>
<td></td>
<td>N/A</td>
<td>50%</td>
</tr>
<tr>
<td>2/0</td>
<td></td>
<td>50%</td>
<td>N/A</td>
</tr>
<tr>
<td>10/1</td>
<td></td>
<td>9.9%</td>
<td>0.99%</td>
</tr>
<tr>
<td>1/10</td>
<td></td>
<td>50%</td>
<td>5%</td>
</tr>
</tbody>
</table>

- What if too many short jobs to give reasonable response time?
  - If load average is 100, hard to make progress
  - One approach: log some user out

**Lottery Scheduling: Simple Mechanism**

- $N_{ticket} = \sum N_i$
- Pick a number $d$ in $1 \ldots N_{ticket}$ 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%,
  - \[ U = \text{finish time of first} / \text{finish time of last} \]
- 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 \[ \frac{\text{big division}}{N} \]
  - The larger your share of tickets, the smaller your stride
  - Ex: \( W = 10,000 \), \( A=100 \) tickets, \( B=50 \), \( C=250 \)
  - \( A \text{ stride: 100, } B: 200, C: 40 \)
- 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

Model: "Perfectly" subdivided CPU:

<table>
<thead>
<tr>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
</tr>
</tbody>
</table>

\[ \frac{1}{N} \]

CFS: Average rate of execution = \( \frac{1}{N} \):

<table>
<thead>
<tr>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
</tr>
</tbody>
</table>
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 \( w \) to each process \( l \) to compute the switching quanta \( Q_i \)
  - Basic equal share: \( Q_i = \text{Target Latency} \times \frac{1}{n} \)
  - Weighted Share: \( Q_i = \left( \frac{w_i}{\sum_w w_p} \right) \times \text{Target Latency} \)
- 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)^nice
    - Two CPU tasks separated by nice value of 5 \( \Rightarrow \) Task with lower nice value has 3 times the weight, since \((1.25)^5 = 3\)
- So, we use “Virtual Runtime” instead of CPU time
  - Virtual Runtime = Real CPU Time / Weight
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(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 the far left).

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

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

![Utilization vs Response time graph]

Administrvia

- Welcome to Project 2
  - Please get started earlier than last time!
- Midterm 2
  - Coming up in 2 weeks! (3/14)
  - Everything up to the midterm is fair game (perhaps deemphasizing the lecture on the day before….)

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 ⇒ 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

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

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

Deadlock with Locks

- 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();
... y.Release(); x.Release();

Thread B:
y.Acquire();
x.Acquire();
... 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!
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 \( T_3 \)
    - \( \ldots \)
    - \( T_n \) 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 $T_i \rightarrow R_j$ 
  - 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 $R_j \rightarrow T_i$

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 actually 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

- 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!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread A</td>
<td>Thread B</td>
</tr>
<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>

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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread A</td>
<td>Thread B</td>
</tr>
<tr>
<td>x.Acquire();</td>
<td>y.Acquire();</td>
</tr>
<tr>
<td>y.Acquire();</td>
<td>x.Acquire();</td>
</tr>
</tbody>
</table>
| ... | ...
| y.Release(); | x.Release(); |
| x.Release(); | y.Release(); |

Consider instead:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread A</td>
<td>Thread B</td>
</tr>
<tr>
<td>Acquire_both(x, y);</td>
<td>Acquire_both(y, x);</td>
</tr>
</tbody>
</table>
| ... | ...
| y.Release(); | x.Release(); |
| 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.Acquire();
x.Release();

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

Consider instead:

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

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

Does it matter in which order the locks are released?

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)

Techniques for Recovering from Deadlock
- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurl 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

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

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)

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!!!!

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

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

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

Thread B:
y.Acquire();
x.Acquire();
...
y.Release();
x.Release();
Banker’s Algorithm for Avoiding Deadlock

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

- 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!)

\[
\text{[Avail]} = \text{[FreeResources]}
\]
Add all nodes to UNFINISHED
\[\text{do}\{\]
\[\text{done} = \text{true}\]
\[\text{Foreach node in UNFINISHED}\{\]
\[\text{if} (\text{[Max}_{\text{node}}]-\text{[Alloc}_{\text{node}}] \leq \text{[Avail]}))\{\]
\[\text{remove node from UNFINISHED}\]
\[\text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{\text{node}}]\]
\[\text{done} = \text{false}\]
\[\}\]
\[\}\text{until}(\text{done})\]
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 2nd to last, and no one would have k-1
    » It’s 3rd to last, and no one would have k-2
    » …

Conclusion

• Proportional Share Scheduling (Lottery Scheduling, Stride Scheduling CFS)
  – 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)

• 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