CS162
Operating Systems and Systems Programming
Lecture 11

Scheduling (finished), Deadlock

March 3rd, 2020

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes*. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

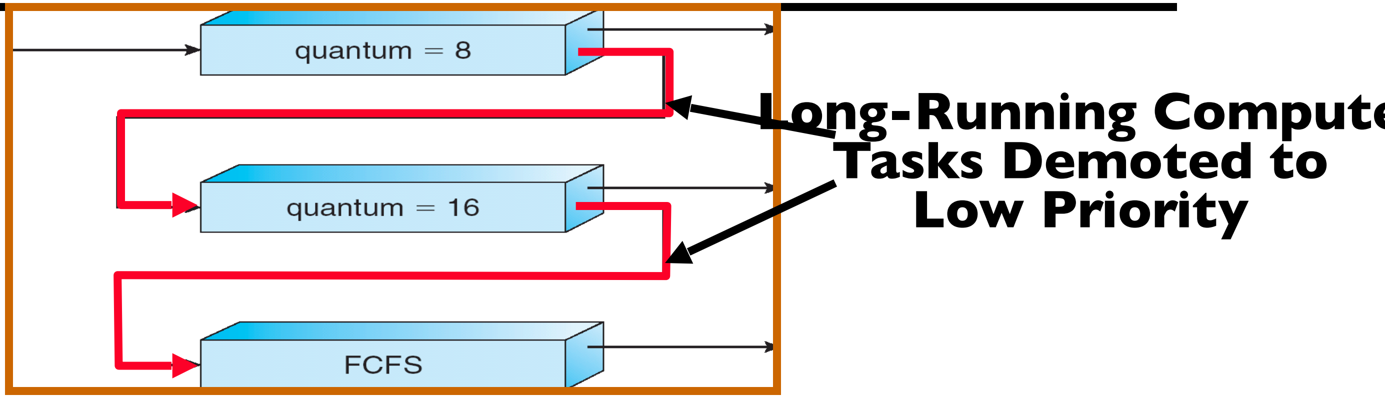
Recall: Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

Recall: What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Recall: Multi-Level Feedback Scheduling



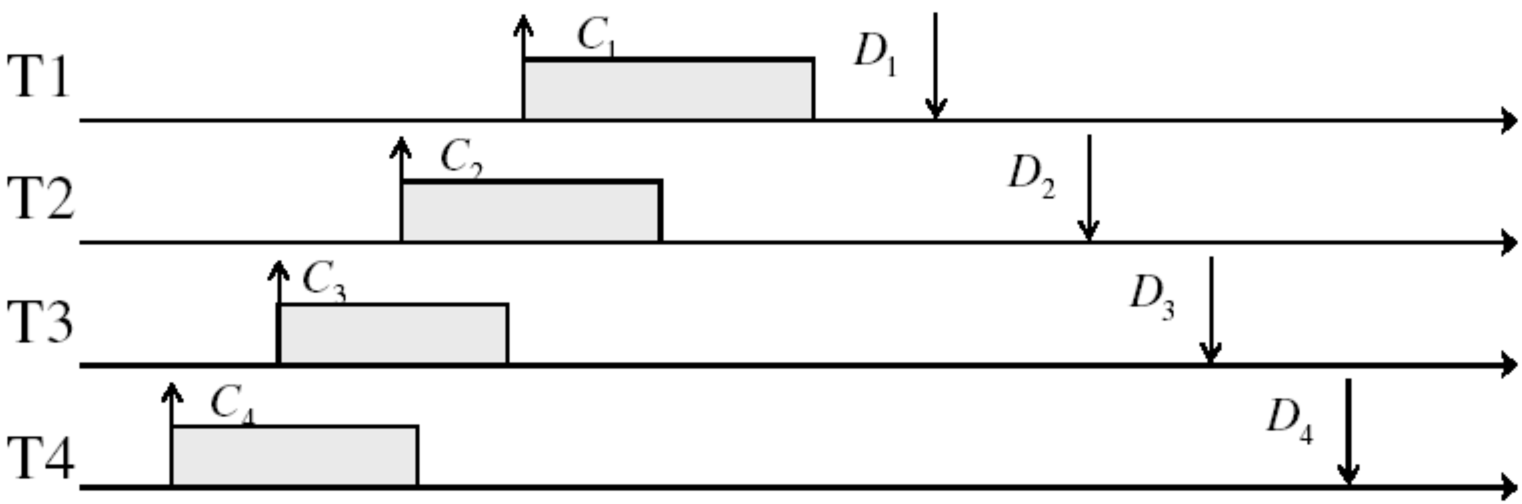
- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Real-Time Scheduling (RTS)

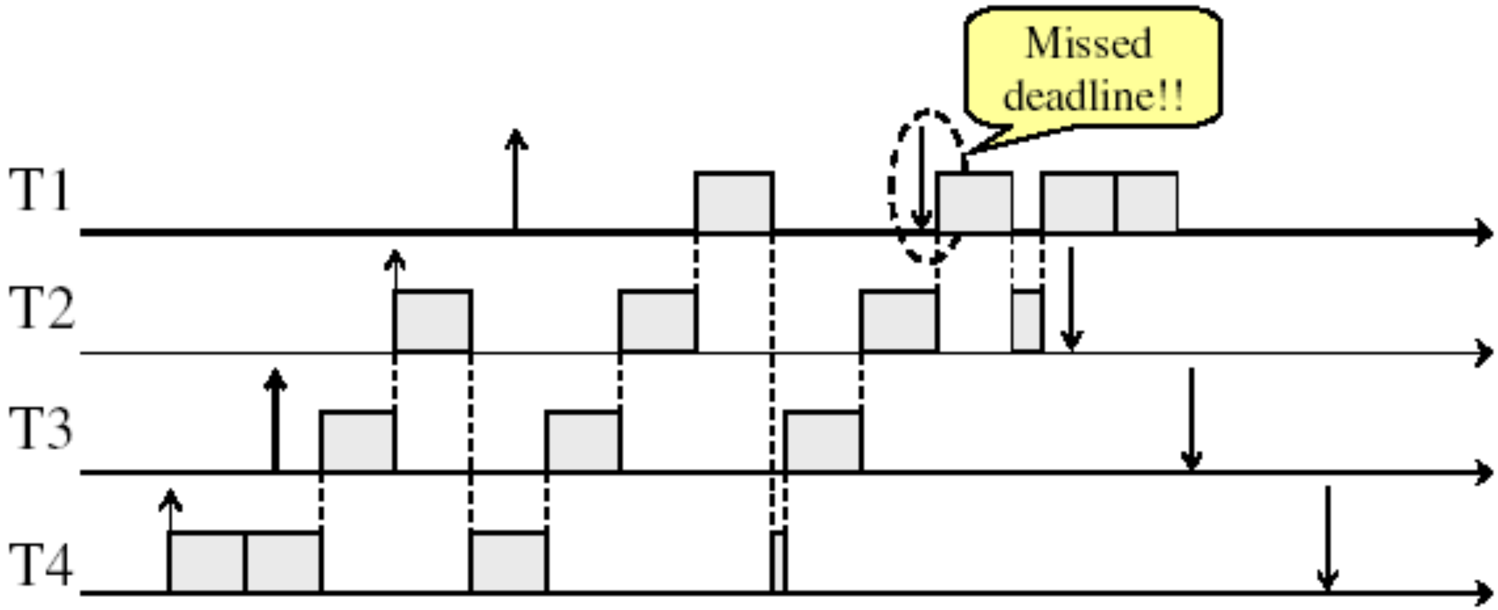
- Efficiency is important but predictability is essential:
  - We need to predict with confidence worst case response times for systems
  - In RTS, performance guarantees are:
    - » Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    - » System/throughput oriented with post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!
- Hard Real-Time
  - *Attempt to meet all deadlines*
  - EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)
- Soft Real-Time
  - *Attempt to meet deadlines with high probability*
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - Important for multimedia applications
  - CBS (Constant Bandwidth Server)

Recall: Realtime Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:



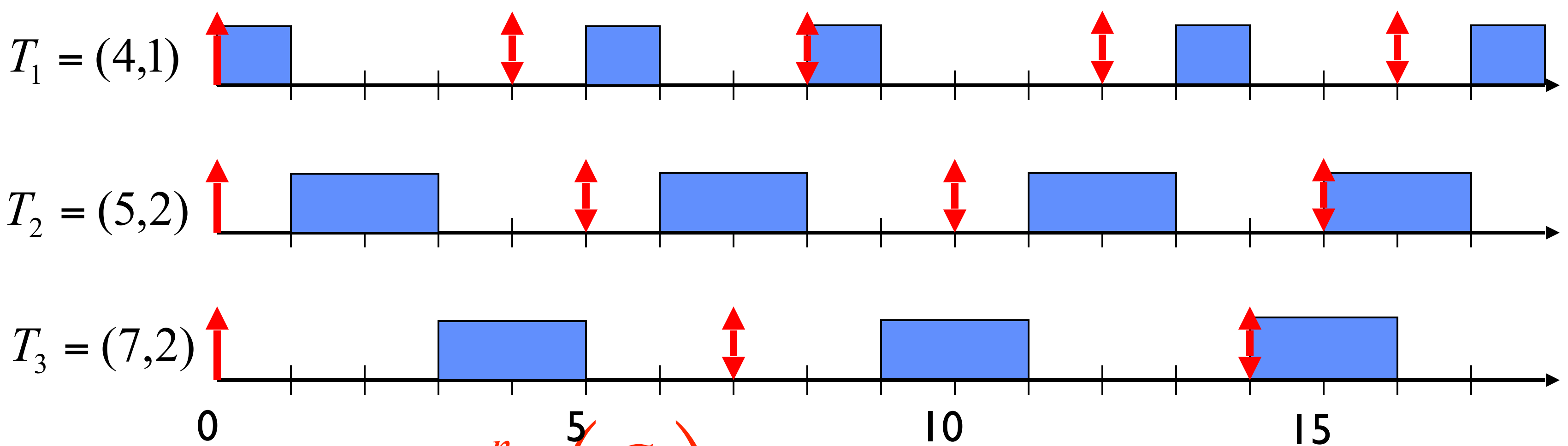
Recall: Round-Robin Scheduling Doesn't Work



Time →

Recall: Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P_i, C_i) for each task i
- Preemptive priority-based dynamic scheduling:
  - Each task is assigned a (current) priority based on how close the absolute deadline is (i.e. $D_i^{t+1} = D_i^t + P_i$ for each task!)
  - The scheduler always schedules the active task with the closest absolute deadline



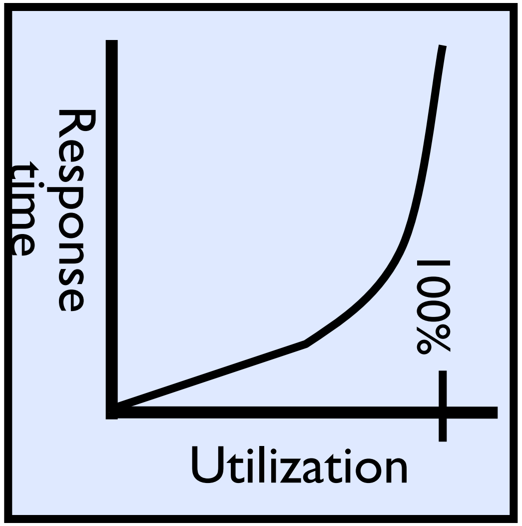
- Schedulable when $\sum_{i=1}^{n} \left(\frac{C_i}{P_i} \right) \leq 1$

Choosing the Right Scheduler

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness – Wait Time to Get CPU	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

A Final Word On Scheduling

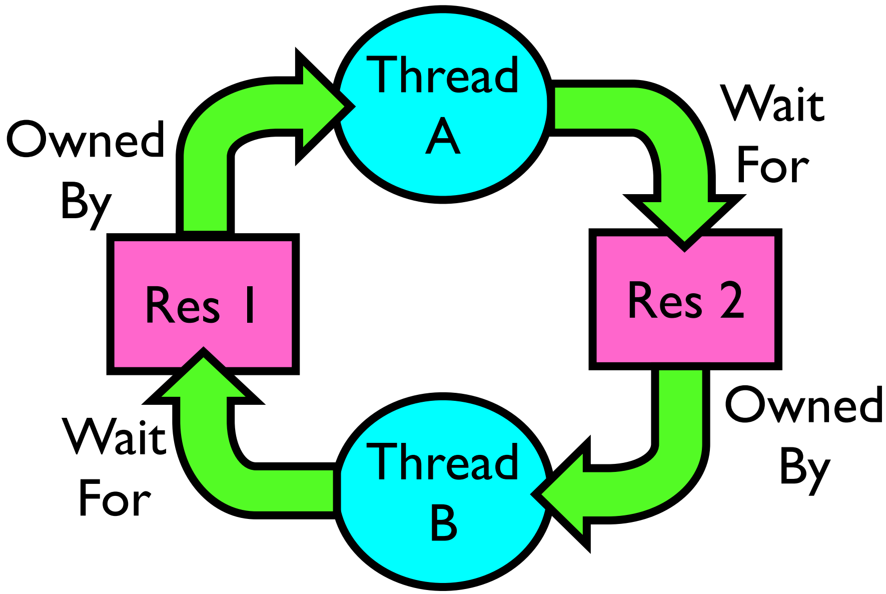
- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - » 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

Starvation vs Deadlock

- 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

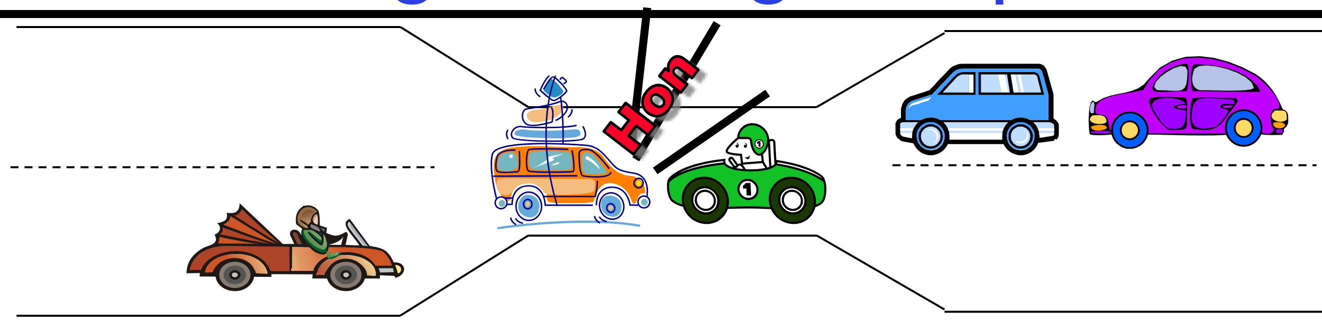


- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

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
  – Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  – Several cars may have to be backed up
- Starvation is possible
  – East-going traffic really fast ⇒ no one goes west

One Lane Bridge Revisited: Deadlock with Locks

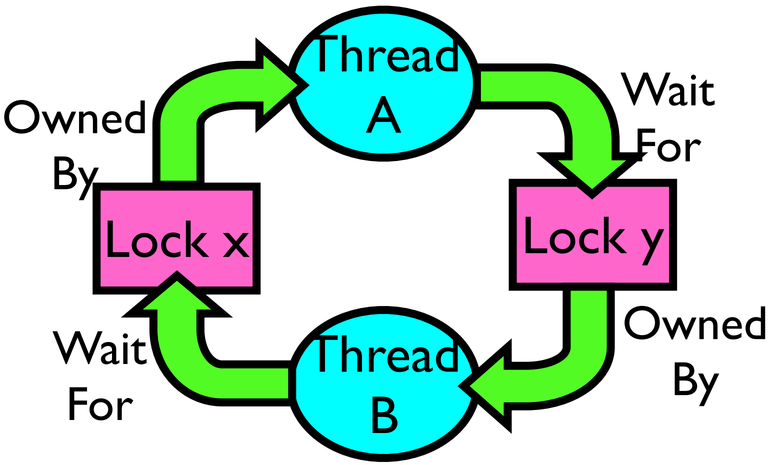
Thread A

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

Nondeterministic Deadlock



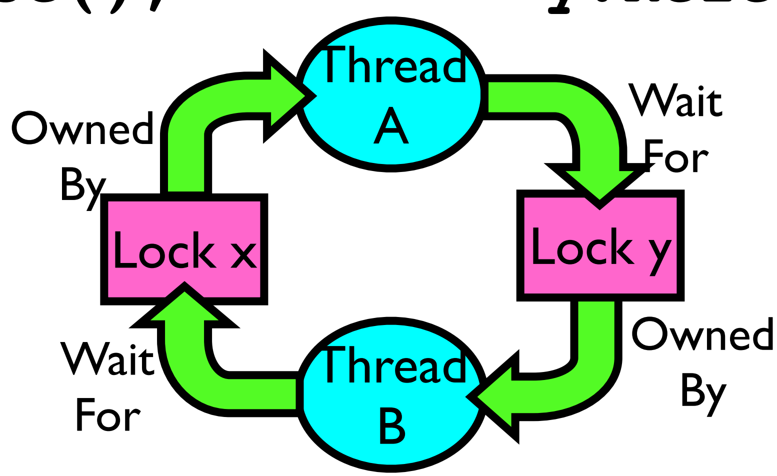
Deadlock with Locks: Unlucky Case

Thread A

```
x.Acquire();

y.Acquire();
<stalled>
<unreachable>
…
y.Release();
x.Release();
```

Thread B

```

y.Acquire();

x.Acquire(); <stalled>
<unreachable>
…
x.Release();
y.Release();
```



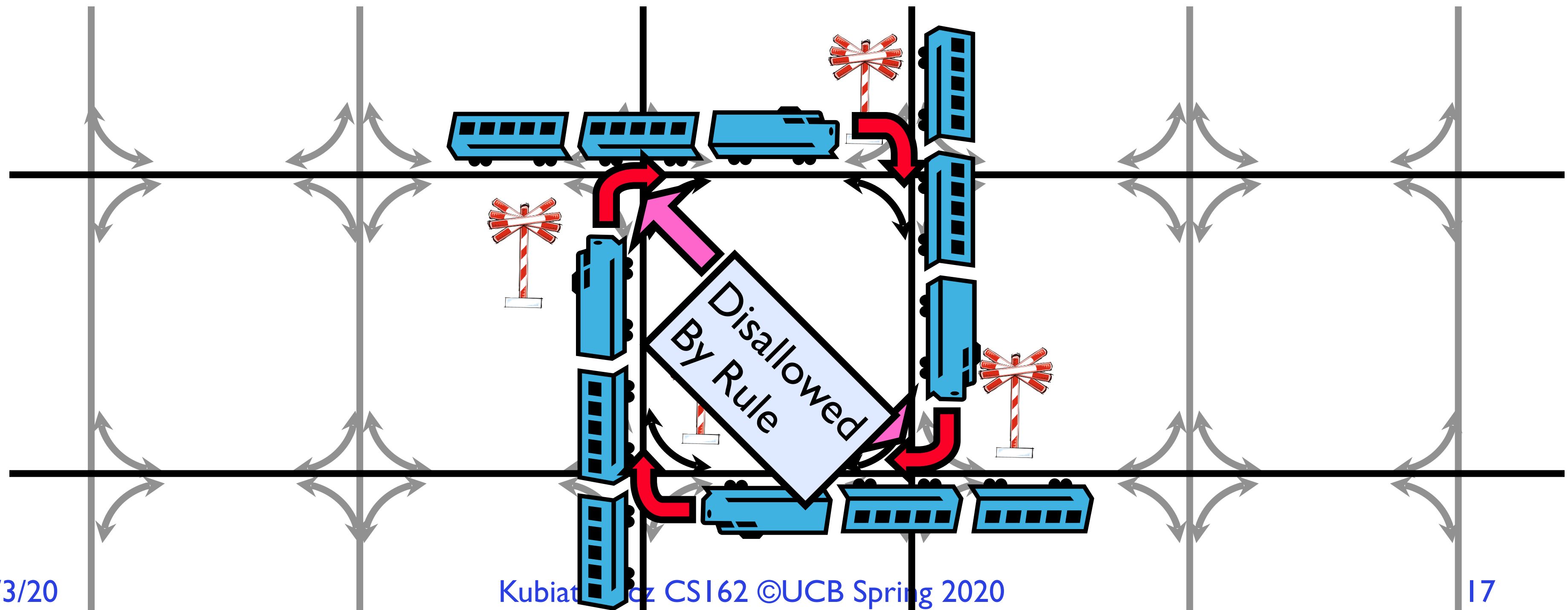
Deadlock with Locks: "Lucky" Case

Thread A	Thread B
`x.Acquire();`	
`y.Acquire();`	
…	`y.Acquire();`
`y.Release();`	
`x.Release();`	
	`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
  - 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)



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

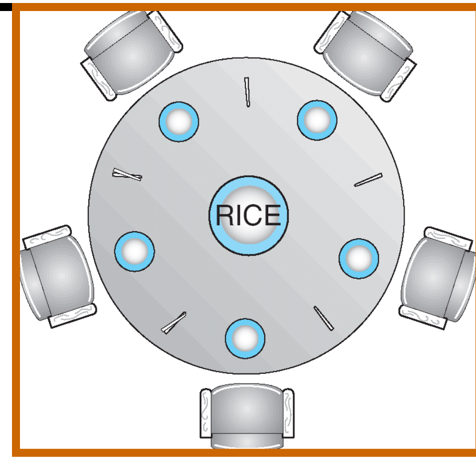
- You can deadlock on any of these!

Deadlock with Space

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

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

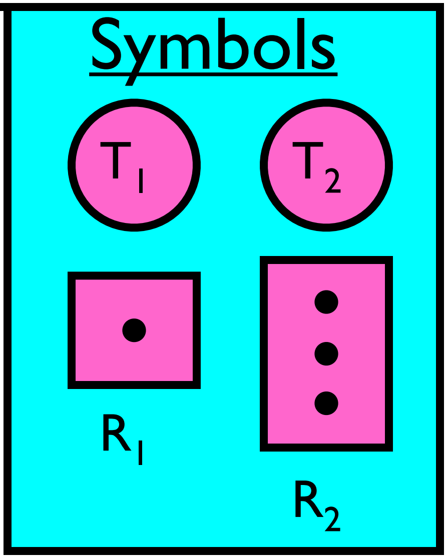
Four requirements for occurrence of Deadlock

- 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 $R_1, R_2, \ldots, R_m$

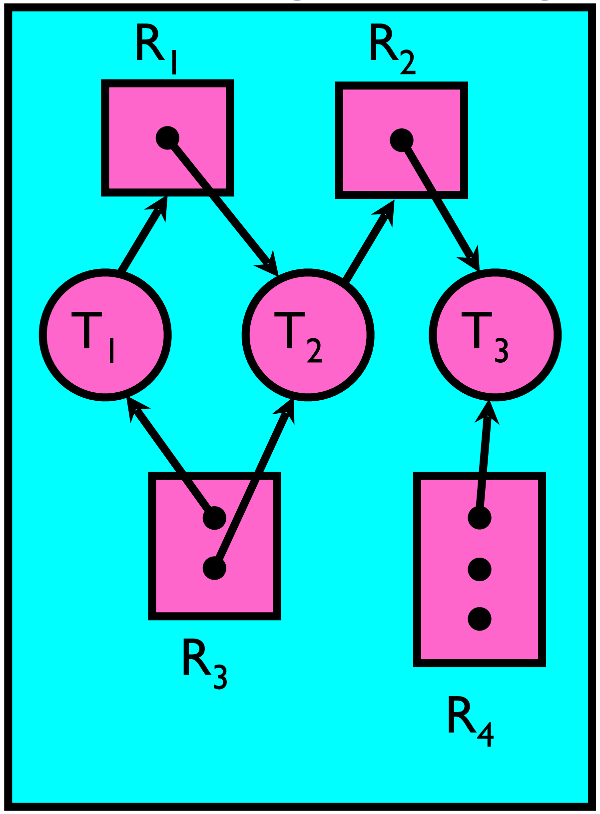
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`



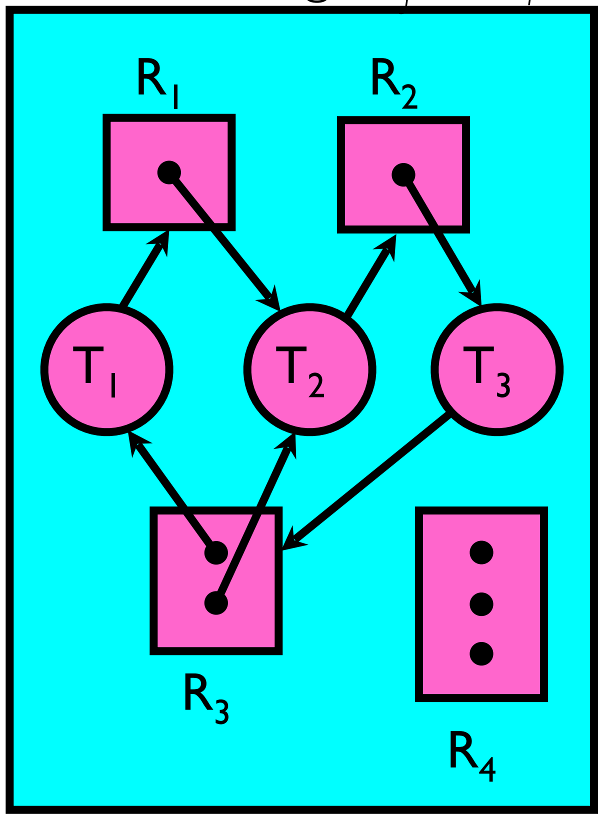
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed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

Resource-Allocation Graph Examples

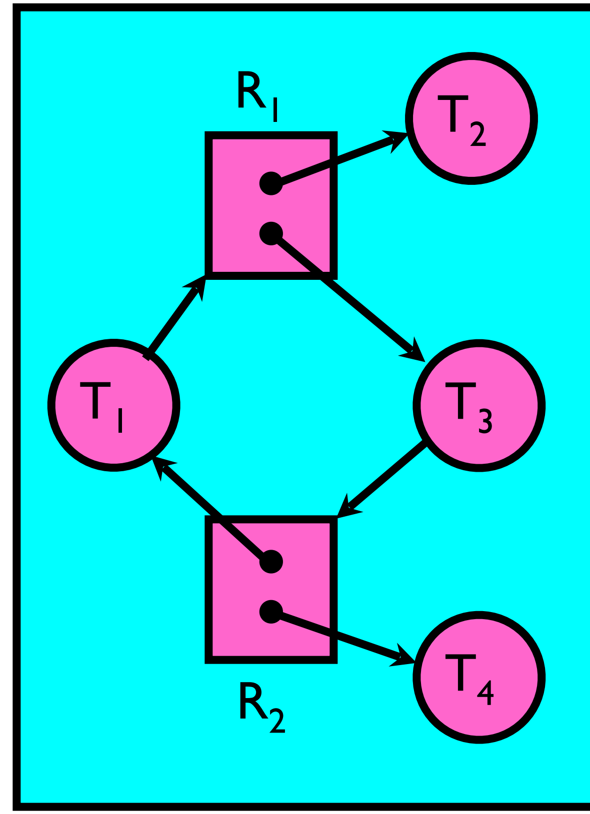
- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

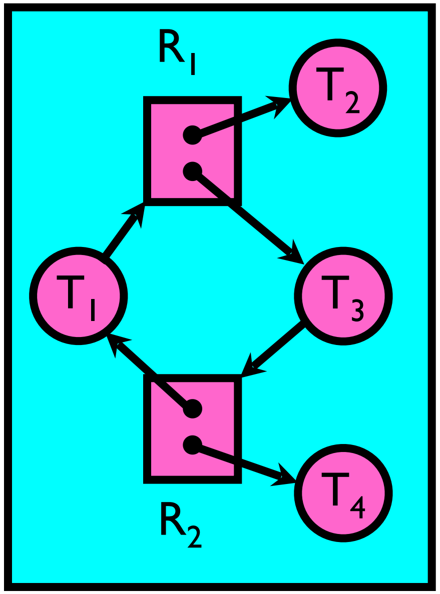
Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-array vector of non-negative integers (quantities of resources of each type):

`[FreeResources]:`	Current free resources each type
`[Request`$_X$`]:`	Current requests from thread X
`[Alloc`$_X$`]:`	Current resources held by thread X

  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in **UNFINISHED** ⇒ 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 to your Mom in Toledo, works its way through the phone lines, 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                    Thread B
AllocateOrWait(1 MB)        AllocateOrWait(1 MB)
AllocateOrWait(1 MB)        AllocateOrWait(1 MB)
Free(1 MB)                  Free(1 MB)
Free(1 MB)                  Free(1 MB)
```

With virtual memory we have "infinite" space so everything will just succeed.

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)

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

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();
```

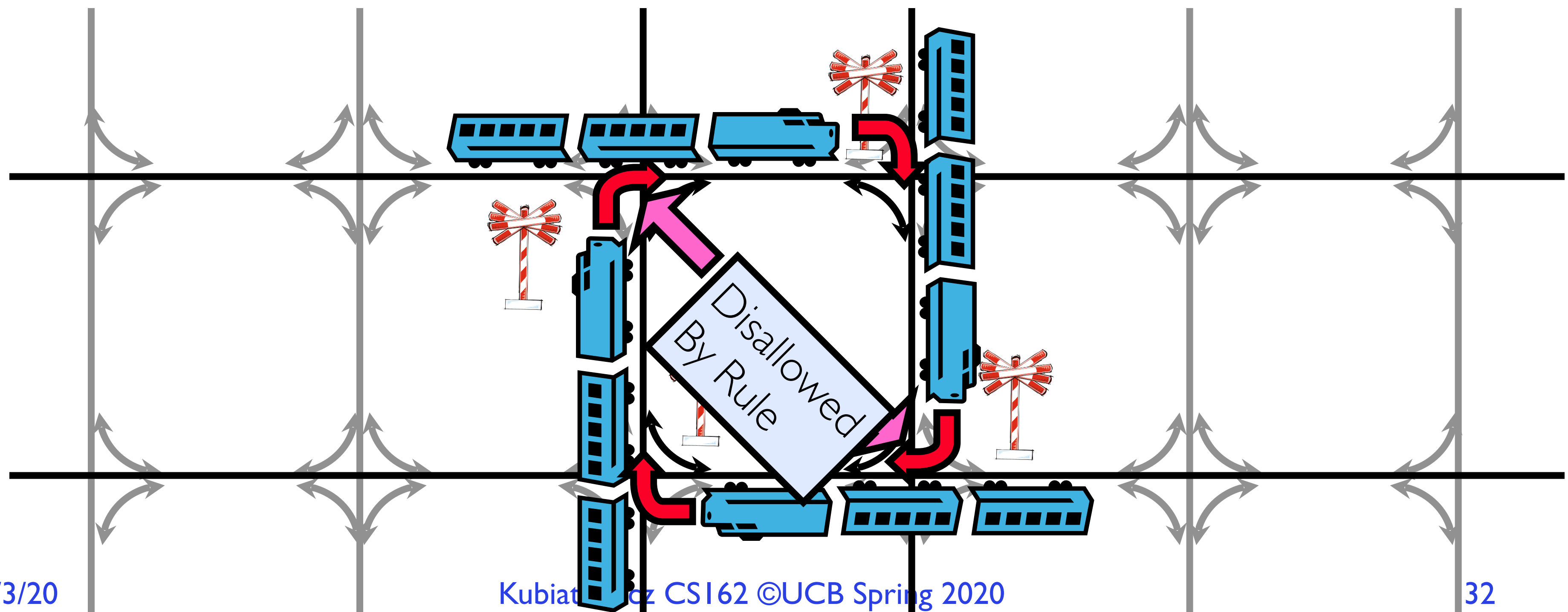
Thread B

```
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

Pre-empting Resources

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

With virtual memory we have "infinite" space so everything will just succeed.

Alternative view: we are "pre-empting" memory when paging out to disk, and giving it back when paging back in

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		Thread B	
	`x.Acquire();`	`y.Acquire();`	Wait…
Blocks…	`y.Acquire();`	`x.Acquire();`	But it's too late…
	`…`	`…`	
	`y.Release();`	`x.Release();`	
	`x.Release();`	`y.Release();`	

Deadlock Avoidance: Three States

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

Deadlock avoidance: prevent system from reaching an unsafe state

- 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	Thread B	
`x.Acquire();`	`y.Acquire();`	
`y.Acquire();`	`x.Acquire();`	Wait until Thread A releases the mutex
…	…	
`y.Release();`	`x.Release();`	
`x.Release();`	`y.Release();`	

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

```
[Avail] = [FreeResources]
     Add all nodes to UNFINISHED
     do {
          done = true
        Foreach node in UNFINISHED {
          if ([Request_node] <= [Avail]) {
             remove node from UNFINISHED
             [Avail] = [Avail] + [Alloc_node]
             done = false
          }
        }
     } until(done)
```

» 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

```
[Avail] = [FreeResources]
     Add all nodes to UNFINISHED
     do {
          done = true
        Foreach node in UNFINISHED {
          if ([Max_node]-[Alloc_node] <= [Avail]) {
             remove node from UNFINISHED
             [Avail] = [Avail] + [Alloc_node]
             done = false
          }
        }
     } until(done)
```

» 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!)

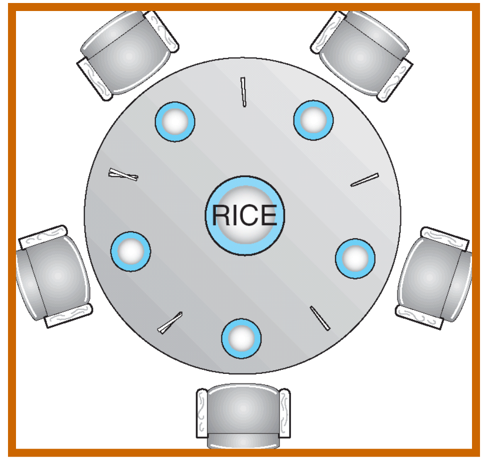
Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max remaining that might be needed by any thread

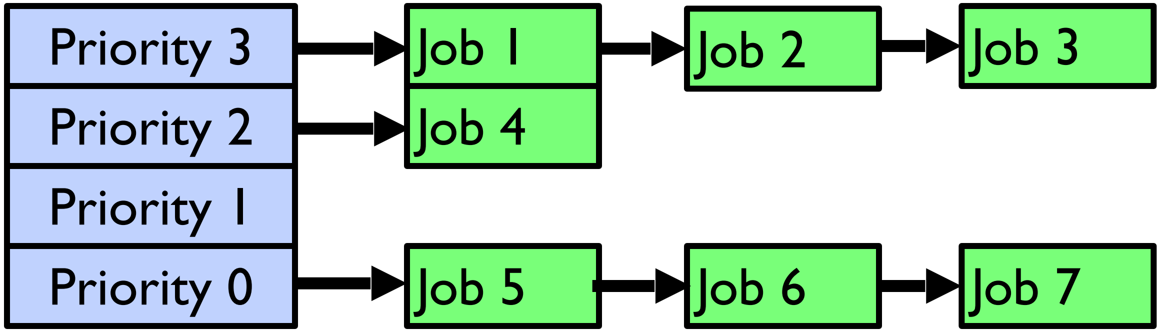
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
      ($[Max_{node}]$-$[Alloc_{node}]$ <= [Avail]) for ($[Request_{node}]$ <= [Avail])
      Grant request if result is deadlock free (conservative!)
    - » Keeps system in a "SAFE" state, i.e. 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..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

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
    - » …

Recall: Priority Scheduler



- Execution Plan
  - Always execute highest-priority runable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation:
    - » Lower priority jobs don't get to run because of higher priority jobs
  - Priority Inversion:
    - » Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task
    - » Usually involves third, intermediate priority task that keeps running even though high-priority task should be running
  - Are either of these problems examples of DEADLOCK?

Priority Donation as a remedy to Priority Inversion

- Does Priority Inversion cause Deadlock? Not usually.
- Consider:
  - 3 threads, T1, T2, T3 in priority order (T3 highest)
  - T1 grabs lock, T3 tries to acquire, then sleeps, T2 running
  - Will this make progress?
    - » No, as long as T2 is running
    - » But T2 could stop at any time and the problem would resolve itself…
    - » So, this is *not* a deadlock (it is a livelock). But is could last a long time…
  - Why is this a priority inversion?
    - » T3 is prevented from running by T2
- What is *priority donation*?
  - When high priority Thread TB is about to sleep while waiting for a lock held by lower priority Thread TA, it may *temporarily donate* its priority to the holder of the lock if that lock holder has a lower priority
    - » So, Priority(TB) => TA until lock is released
  - So, now, TA runs with high priority until it releases its lock, at which time its priority is restored to its original priority
- How does *priority donation* help both above priority inversion scenario?
  - Briefly raising T1 to the same priority as T3⇒T1 can run and release lock, allowing T3 to run
  - Does priority donation involve taking lock away from T1?
    - » NO! That would break semantics of the lock and potentially corrupt any information protected by lock!

Summary

- Real-time scheduling
  - Need to meet a deadline, predictability essential
  - Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- 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