Operating Systems
(Honor Track)

Scheduling 3: Scheduling & Deadlock

Xin Jin

Spring 2024

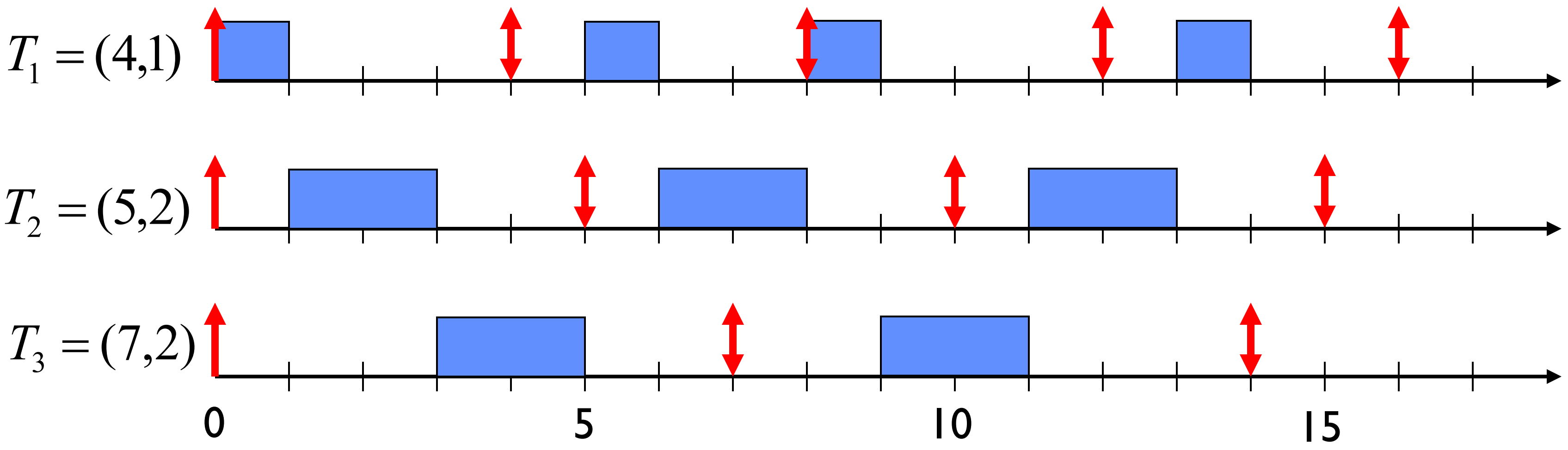
Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: 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*; 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 (admission control)
  - Earliest Deadline First (EDF), Rate-Monitonic Scheduling (RMS), Deadline Monotonic Scheduling (DM)
- Soft real-time: for multimedia
  - Attempt to meet deadlines with high probability
  - Constant Bandwidth Server (CBS)

Recap: Earliest Deadline First (EDF)

- Tasks i is periodic with period P_i and computation C_i 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



Recap: Ensuring Progress

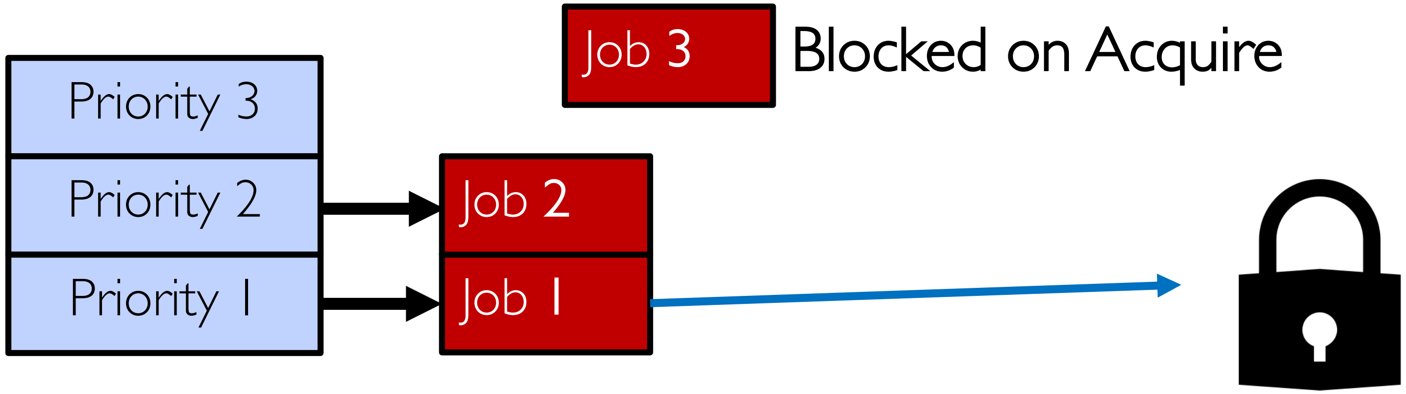
- Starvation: thread fails to make progress for an indefinite period of time

- Starvation ≠ Deadlock
  - Deadlock: cyclic requests for resources

- Let’s explore what sorts of problems we might encounter and how to avoid them…

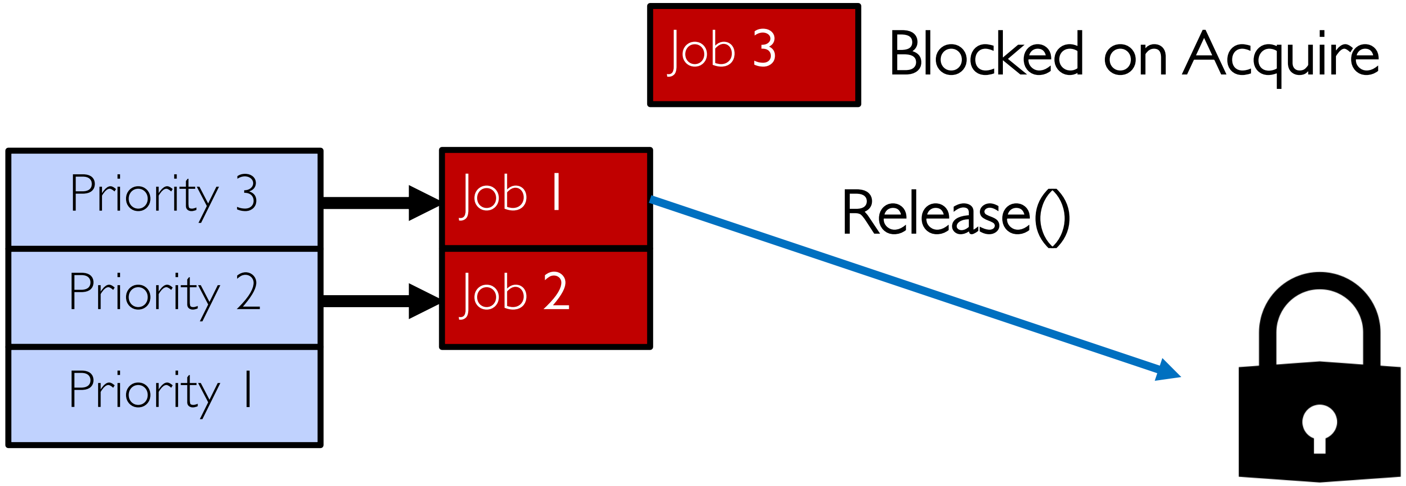
- Whether various scheduling policies can lead to starvation
  - LCFS
  - FCFS
  - Round robin
  - Priority scheduling
  - SRTF
  - MLFQ

Recap: Priority Inversion



- At this point, which job does the scheduler choose?
- Job 2 (Medium Priority)
- Priority Inversion

Recap: One Solution: Priority Donation/Inheritance



- Job 3 temporarily grants Job 1 its "high priority" to run on its behalf

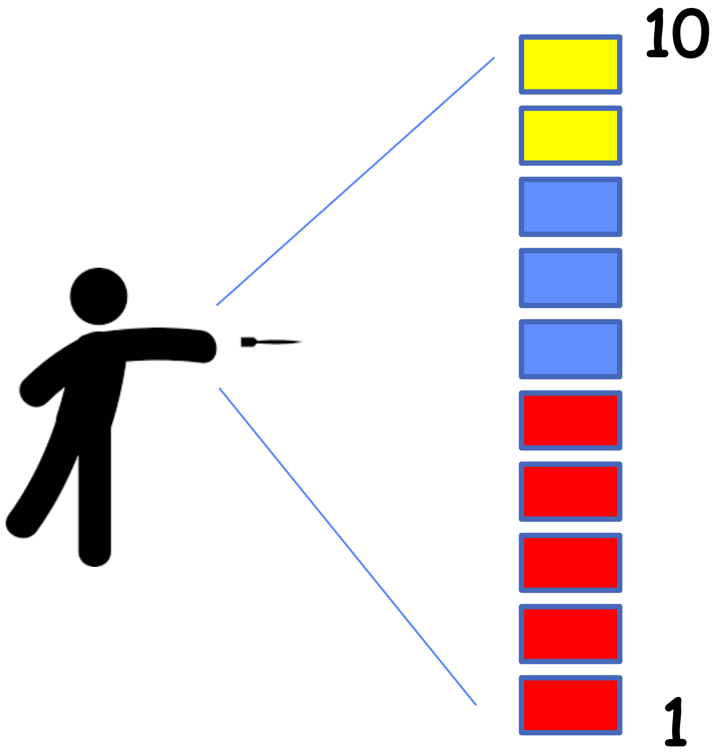
Recap: Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower nice value $\Rightarrow$ higher priority
  - Higher nice value $\Rightarrow$ lower priority
  - 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

Recap: 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)

Recap: Lottery Scheduling: Simple Mechanism

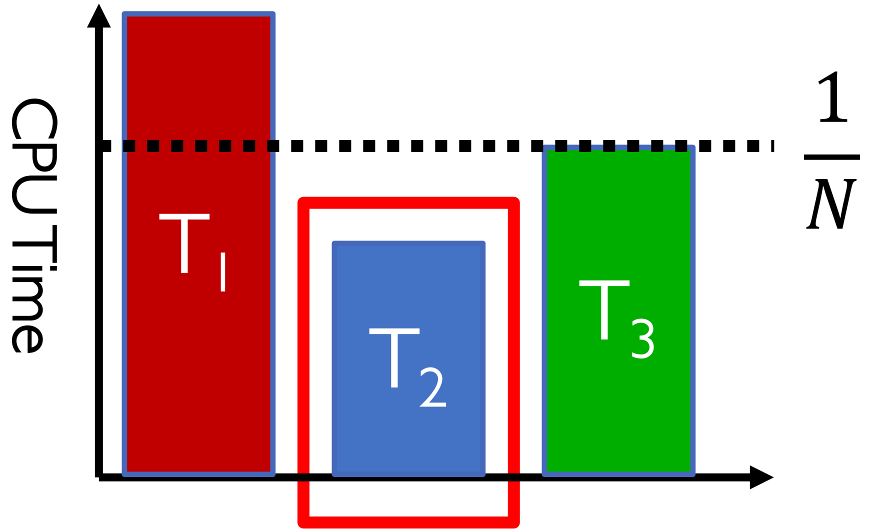


- $N_{ticket} = \sum \mathrm{N_i}$
- Pick a number d in $1 \;..\; N_{ticket}$ as the random "dart"
- Jobs record their $\mathrm{N_i}$ of allocated tickets
- Order them by $\mathrm{N_i}$
- Select the first j such that $\sum \mathrm{N_i}$ up to j exceeds *d*.

Recap: 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 $= \frac{1}{N}$:

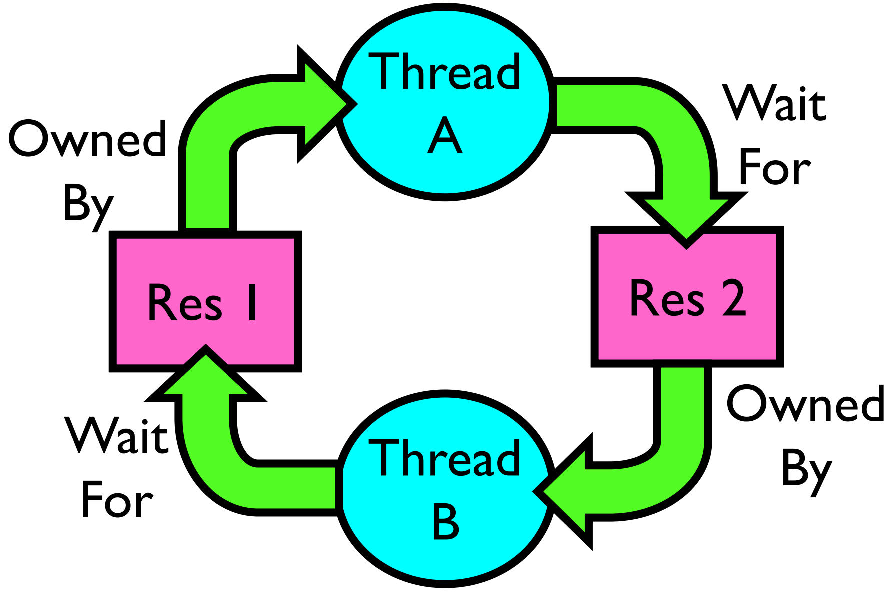


Recap: Choosing the Right Scheduler

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Completion 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

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

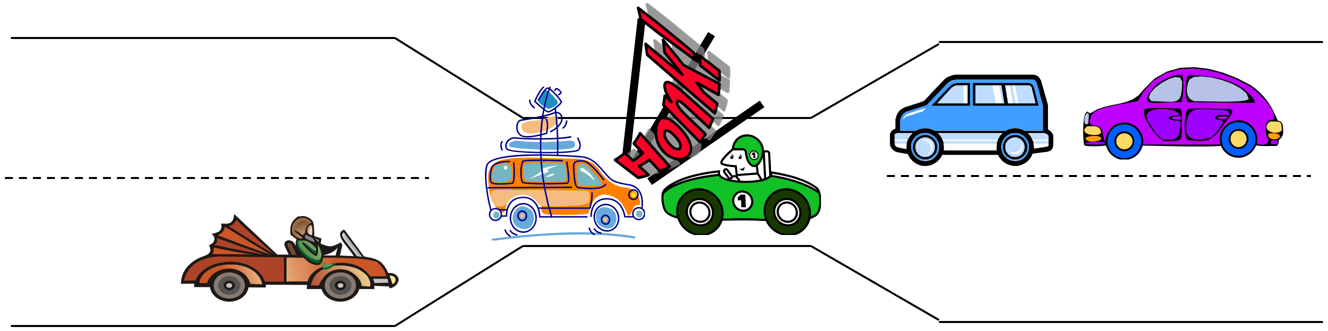


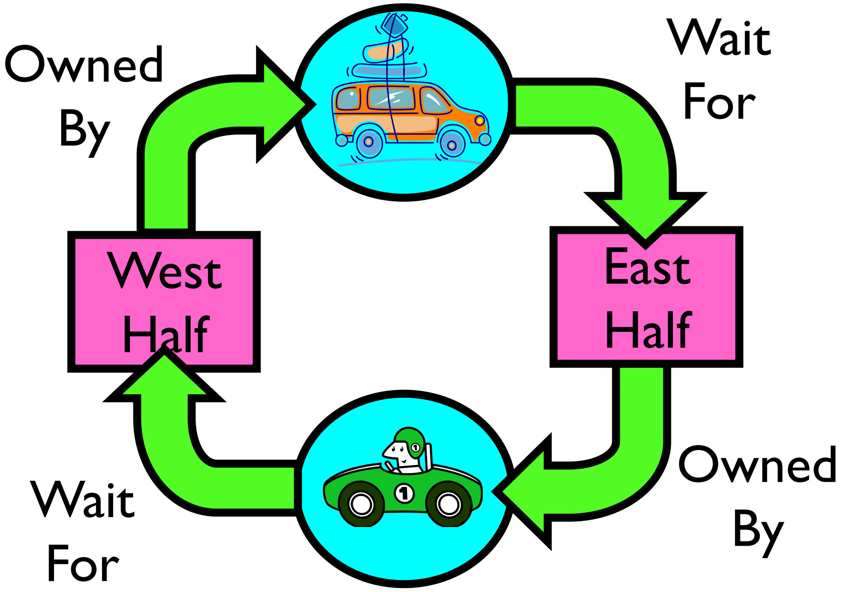
Example: Single-Lane Bridge Crossing

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



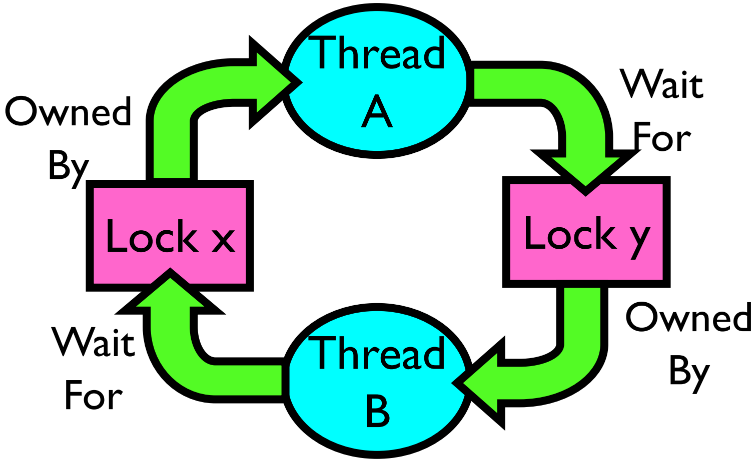


- Deadlock: Shown above when two cars in opposite directions meet in middle
  - Each acquires one segment and needs next
  - Deadlock resolved if one car backs up (preempt resources and rollback)
    - » Several cars may have to be backed up
- Starvation (not Deadlock):
  - East-going traffic really fast $\Rightarrow$ 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:

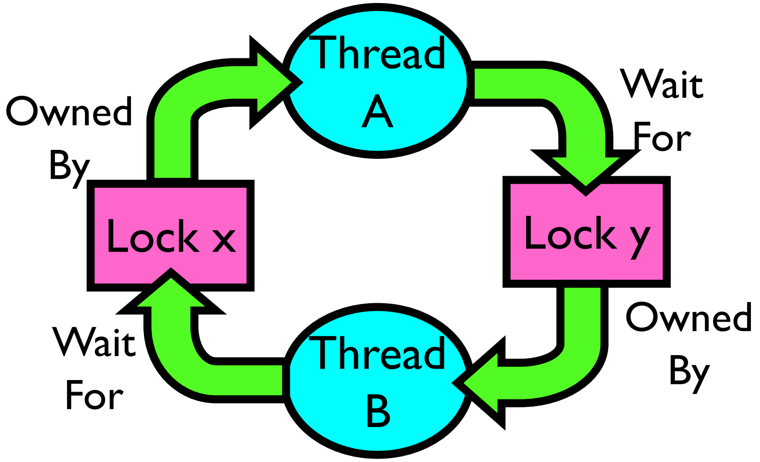
```
x.Acquire();

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!

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:

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

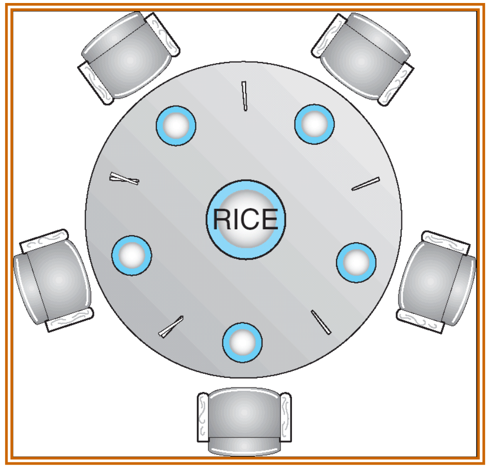
Thread B

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

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?

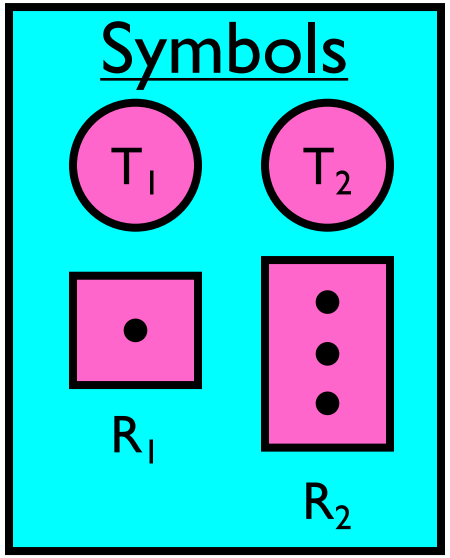
Four requirements for occurrence of Deadlock

- 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 $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



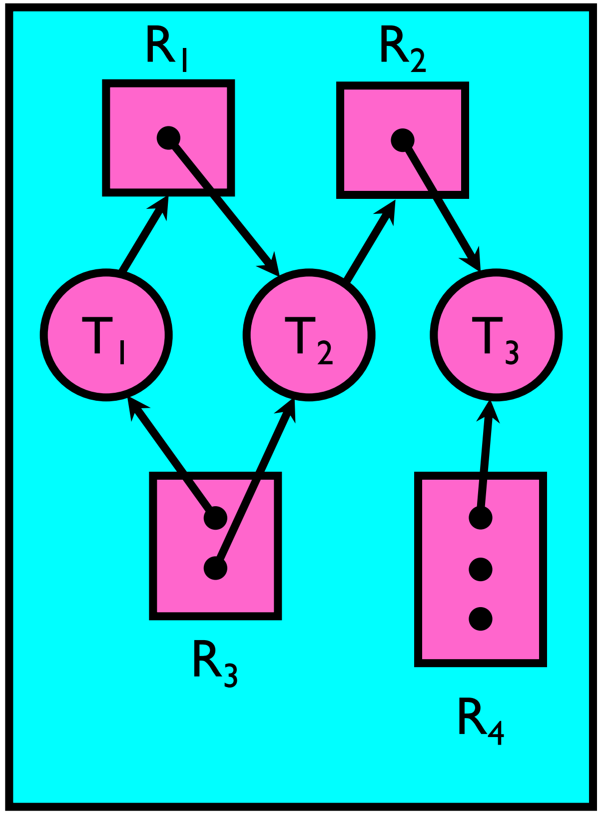
Group Discussion

- Topic: resource allocation graph
  - How to detect deadlocks?
  - Does a circle in a resource allocation graph mean a deadlock?

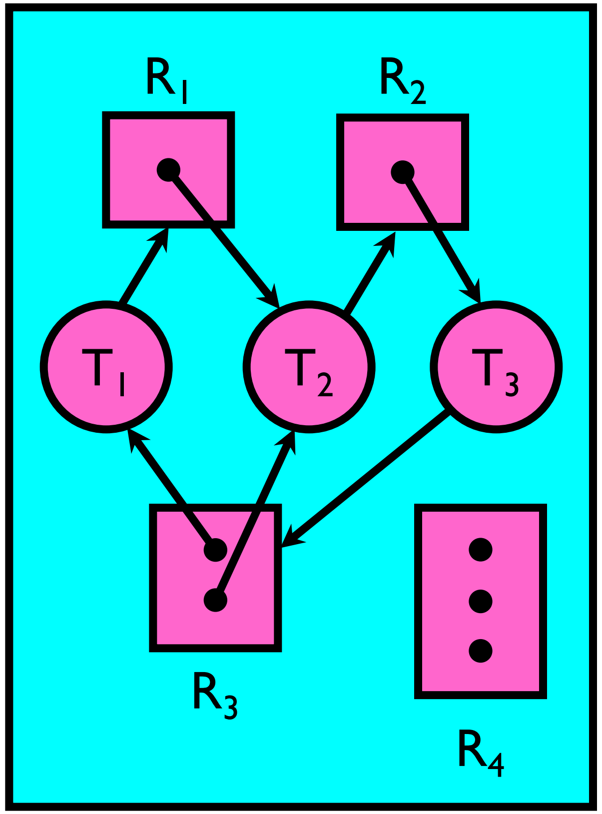
- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Resource-Allocation Graph Examples

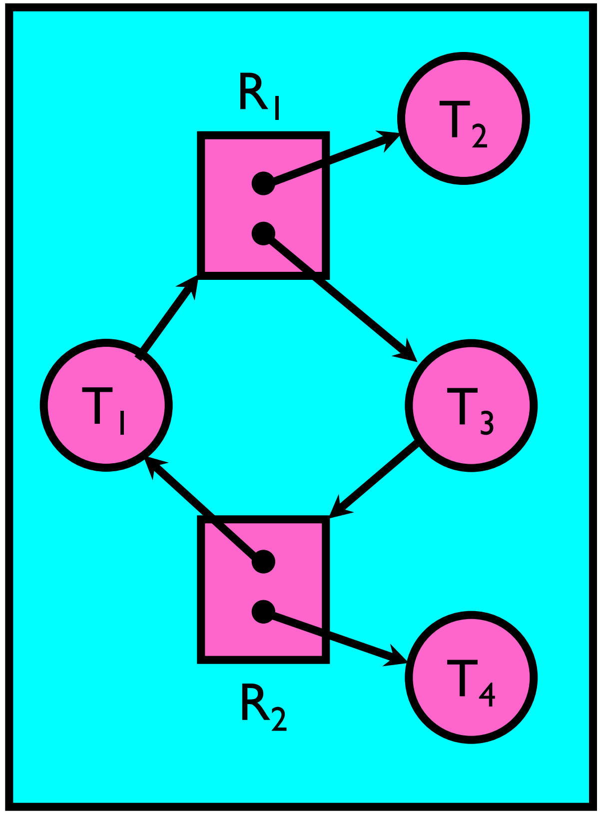
- Model:
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource
Allocation Graph



Allocation Graph
With Deadlock



Allocation Graph
With Cycle, but
No Deadlock

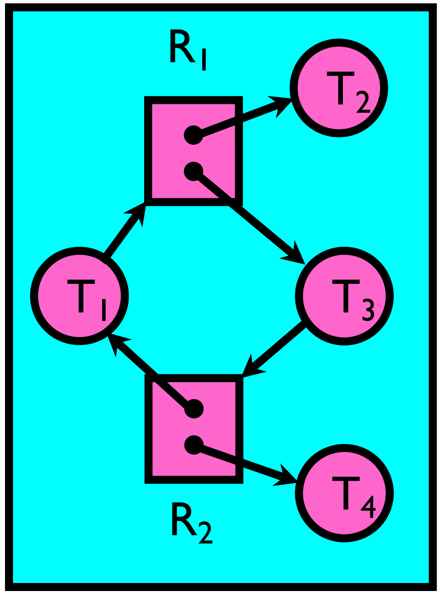
Deadlock Detection Algorithm

- Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

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

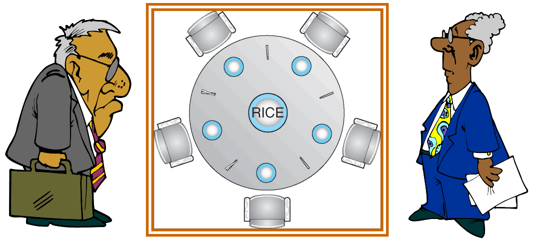
- See if tasks can eventually terminate on their own

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



- Nodes left in UNFINISHED $\Rightarrow$ deadlocked

Group Discussion



- Topic: deadlock detection algorithm
  - How to apply the algorithm to the dining lawyer's problem?
  - Case 1: resources are represented as [5], and each lawyer can use any two chopsticks
  - Case 2: resources are represented as [1, 1, 1, 1, 1], and each lawyer can only use nearby chopsticks

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

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

<u>Thread A</u>	**<u>Thread B</u>**
`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, 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:	Thread B:
`x.Acquire();`	`y.Acquire();`
`y.Acquire();`	`x.Acquire();`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`

Consider instead:

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

Does it matter in which order the locks are released?

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

<u>Thread A</u>:

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

<u>Thread B</u>:

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

- 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();   Blocks…
…
y.Release();
x.Release();
```

Thread B:

```
y.Acquire();
x.Acquire();   Wait?
…              But it's already too late…
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:

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

Thread B:

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

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) $\geq$ max
    remaining that might be needed by any thread

- Banker's algorithm:
  - 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

Banker's Algorithm for Avoiding Deadlock

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

$$([Max_{node}]-[Alloc_{node}] <= [Avail]) \text{ for } ([Request_{node}] <= [Avail])$$

Grant request if result is deadlock free

Banker's Algorithm for Avoiding Deadlock

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

  ([Max_{node}]-[$\text{Alloc}_{\text{node}}$] <= [Avail]) for ([$\text{Request}_{\text{node}}$] <= [Avail])

  Grant request if result is deadlock free

Banker's Algorithm for Avoiding Deadlock

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

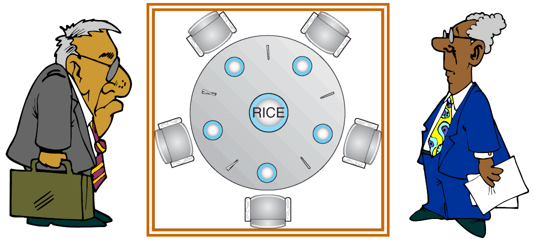
    (available resources - #requested) $\geq$ max
    remaining that might be needed by any thread

- Banker's algorithm:
  - 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
  - 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..

Group Discussion

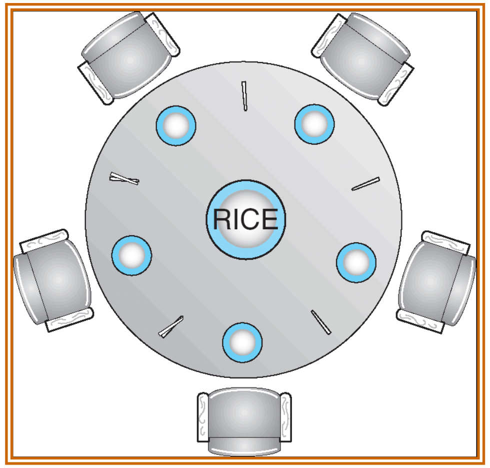


- Topic: Banker's algorithm
  - How to apply Banker's algorithm to the dining lawyer's problem?
  - Case 1: resources are represented as [5], and each lawyer can use any two chopsticks
  - Case 2: resources are represented as [1, 1, 1, 1, 1], and each lawyer can only use nearby chopsticks

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

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
    » …

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