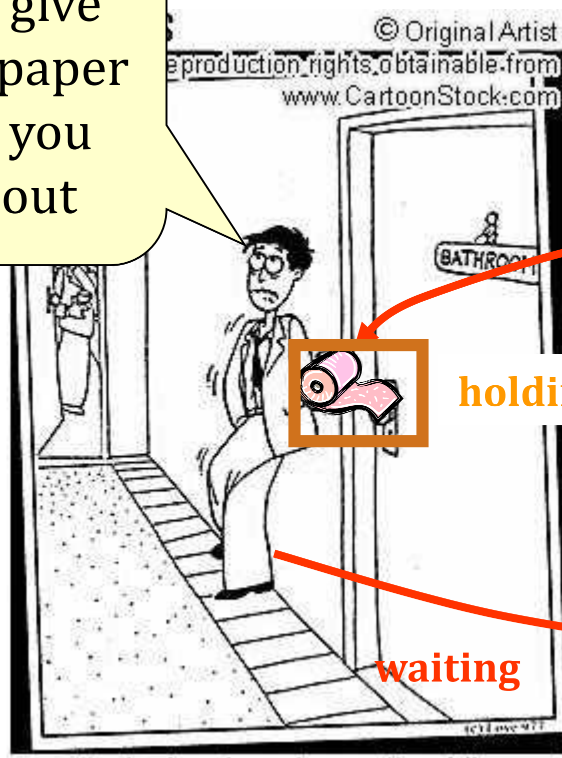
Lecture 7: Deadlock

Bo Tang @ 2020, Spring

Deadlock

I won't come out unless you give me the paper

I won't give you the paper unless you come out



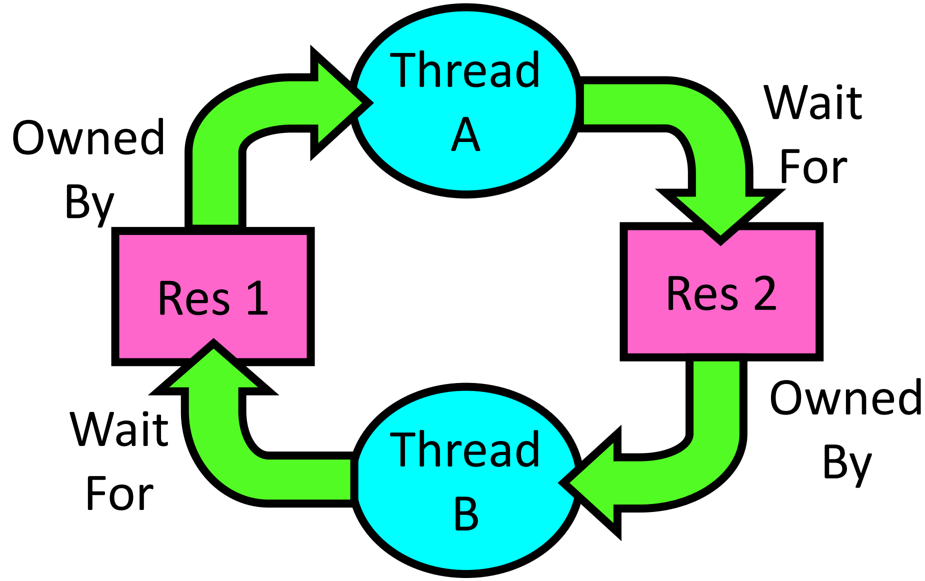
waiting

holding

waiting

Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - 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 does not have to)
    - Deadlock canot end without external intervention

Conditions for Deadlock

- Deadlock not always deterministic

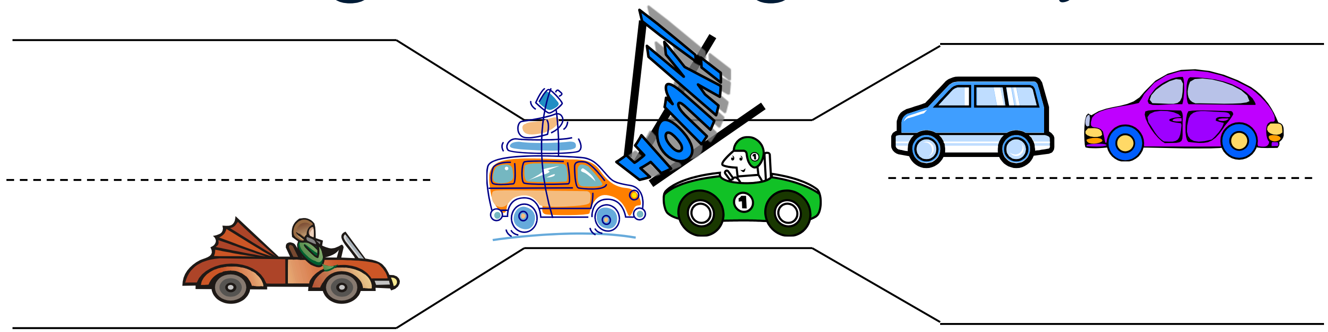
```
Thread A            Thread B

sem_wait(x);        sem_wait(y);
sem_wait(y);        sem_wait(x);

sem_post(y);        sem_wait(x);
sem_post(x);        sem_wait(y);
```

  - Deadlock will not always happen with this code
    - Have to have exactly the right timing
    - So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…
- Deadlocks occur with multiple resources
  - Means you cannot decompose the problem
  - Cannot solve deadlock for each resource independently
  - System with 2 disk drives and two threads
    - Each thread needs 2 disk drives to function
    - Each thread gets one disk and waits for another one

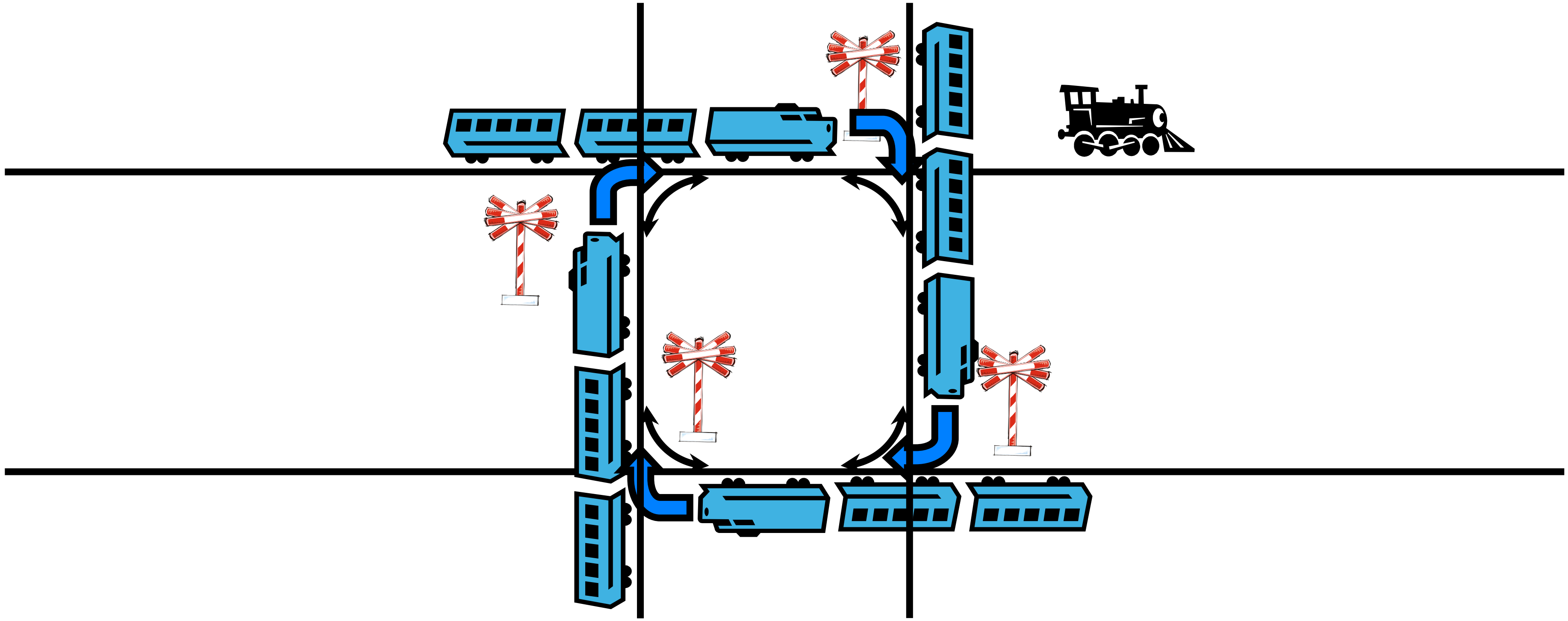
Bridge Crossing Example



- 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 $\Rightarrow$ no one goes west

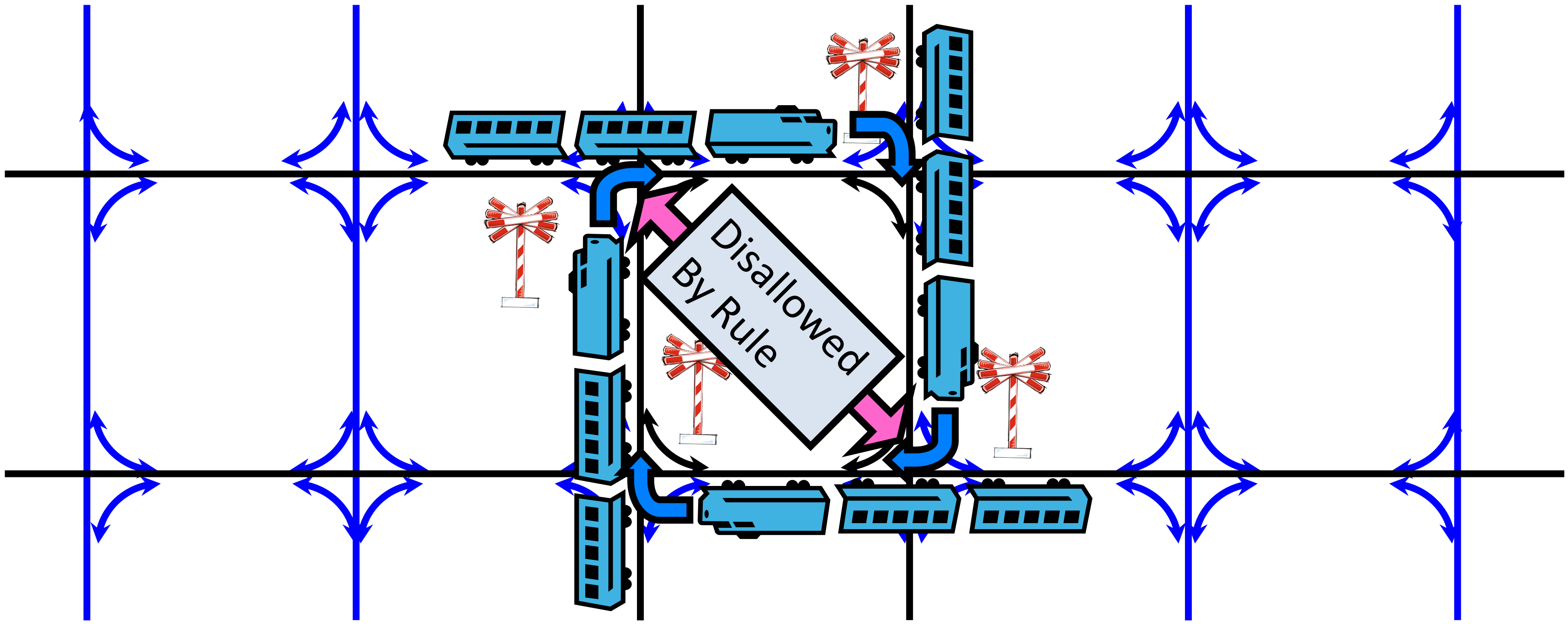
Train Example

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



Train Example

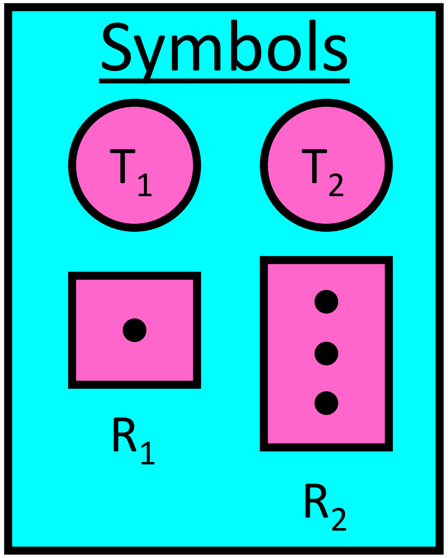
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Four requirements for 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

Resource-Allocation Graph

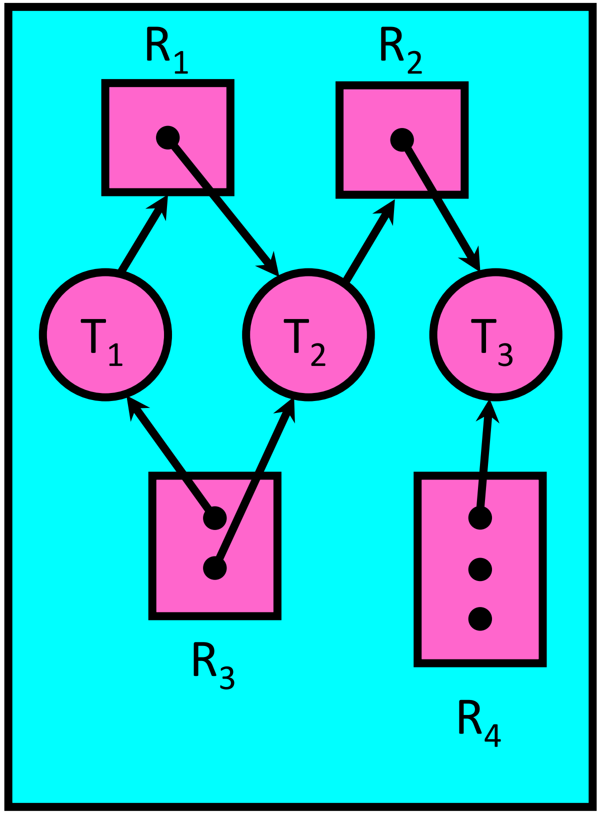


- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

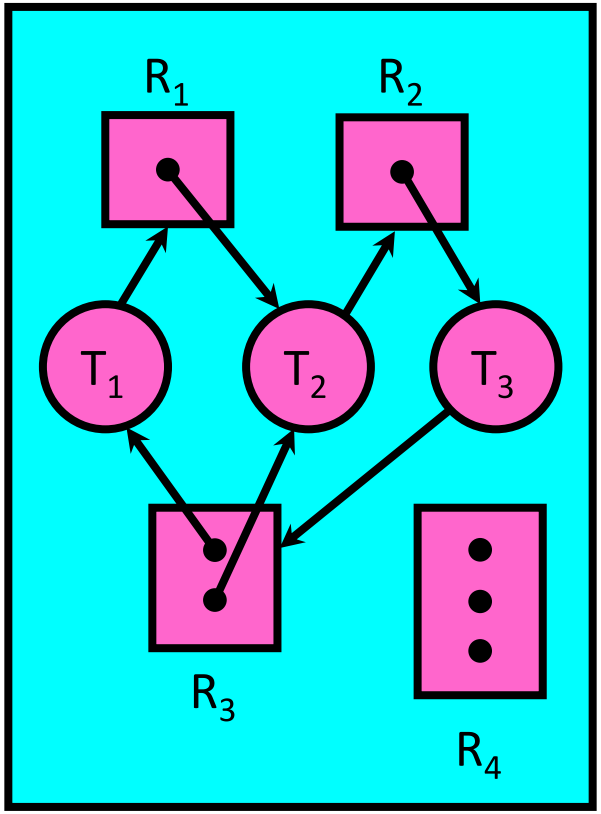
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, 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

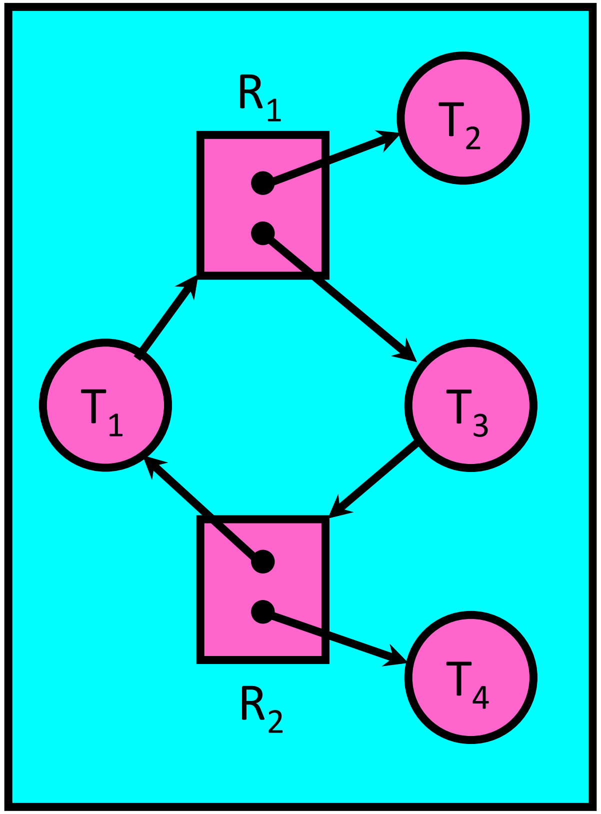
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph with Cycle, but No Deadlock

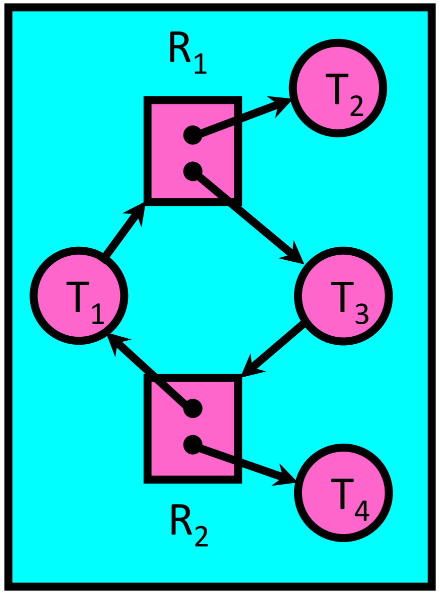
Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

Deadlock Detection Algorithm

- Only one of each type of resource $\Rightarrow$ look for loops
- More general 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
    Foreach 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

What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining philosopher
  - 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
  - Does not always fit with semantics of computation
- Roll back actions of deadlocked threads
  - 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

Techniques for Preventing Deadlock

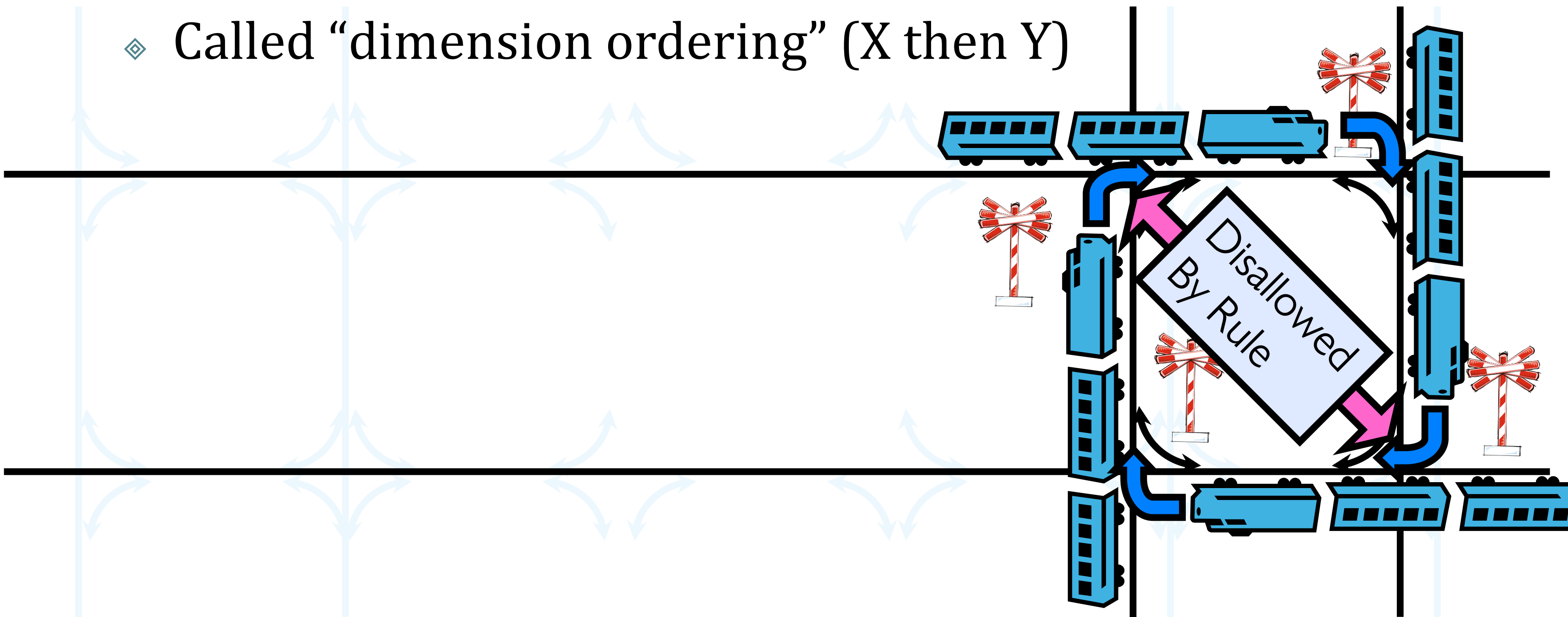
- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Examples:
    - Bay bridge with 12,000 lanes. Never wait!
    - Infinite disk space (not realistic yet?)
- No sharing of resources (totally independent threads)
  - Not very realistic
- Do not allow waiting
  - 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 SUSTech; when hit traffic jam, suddenly you are transported back home and told to retry!

Techniques for Preventing Deadlock

- Make all threads request everything they will 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 not leave home until we know no one is using any intersection between home and SUSTech; 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.P, y.P, z.P,…)
    - Make tasks request disk, then memory, then…
    - Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Review: Train Example

- 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)



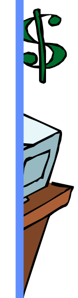
Banker's Algorithm for Preventing Deadlock

- 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
      ([Maxnode]-[Allocnode] ≤ [Avail]) for ([Requestnode] ≤ [Avail])
      Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Preventing Deadlock

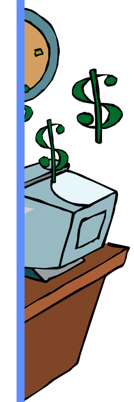
```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
      done = true
      Foreach node in UNFINISHED {
         if ([Request_node] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
         }
      }
} until(done)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



- 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 Preventing 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)
```



ordering of threads is still deadlock free afterward

```
[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)
```

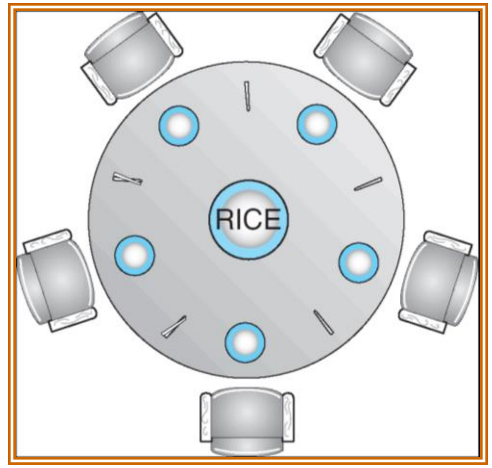
Banker's Algorithm for Preventing Deadlock

- 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 philosophers
  - "Safe" (will not 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 philosophers? Do not allow if:
    - It is the last one, no one would have k
    - It is 2nd to last, and no one would have k-1
    - It is 3rd to last, and no one would have k-2
    - ...

Thank You!