Starvation and Deadlock

9 January 2025
Lecture 10

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

First Come First Serve

Round Robin (RR)

Scheduling quantum

Priority scheduling

- Priority inversion

Starvation

Measurements

- Average wait time
- Average completion time

Shortest Job First

Shortest Remaining Time First

Lottery Scheduling

Multi-level feedback

O(1)

CFS

- Virtual Runtime

Topics for Today

- Starvation and Deadlock

Today’s concepts

Starvation vs DEADLOCK

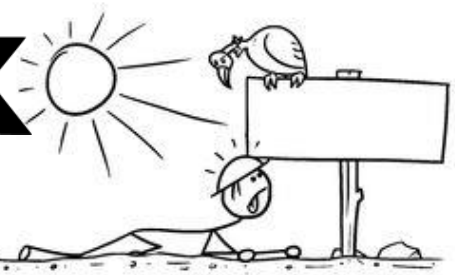
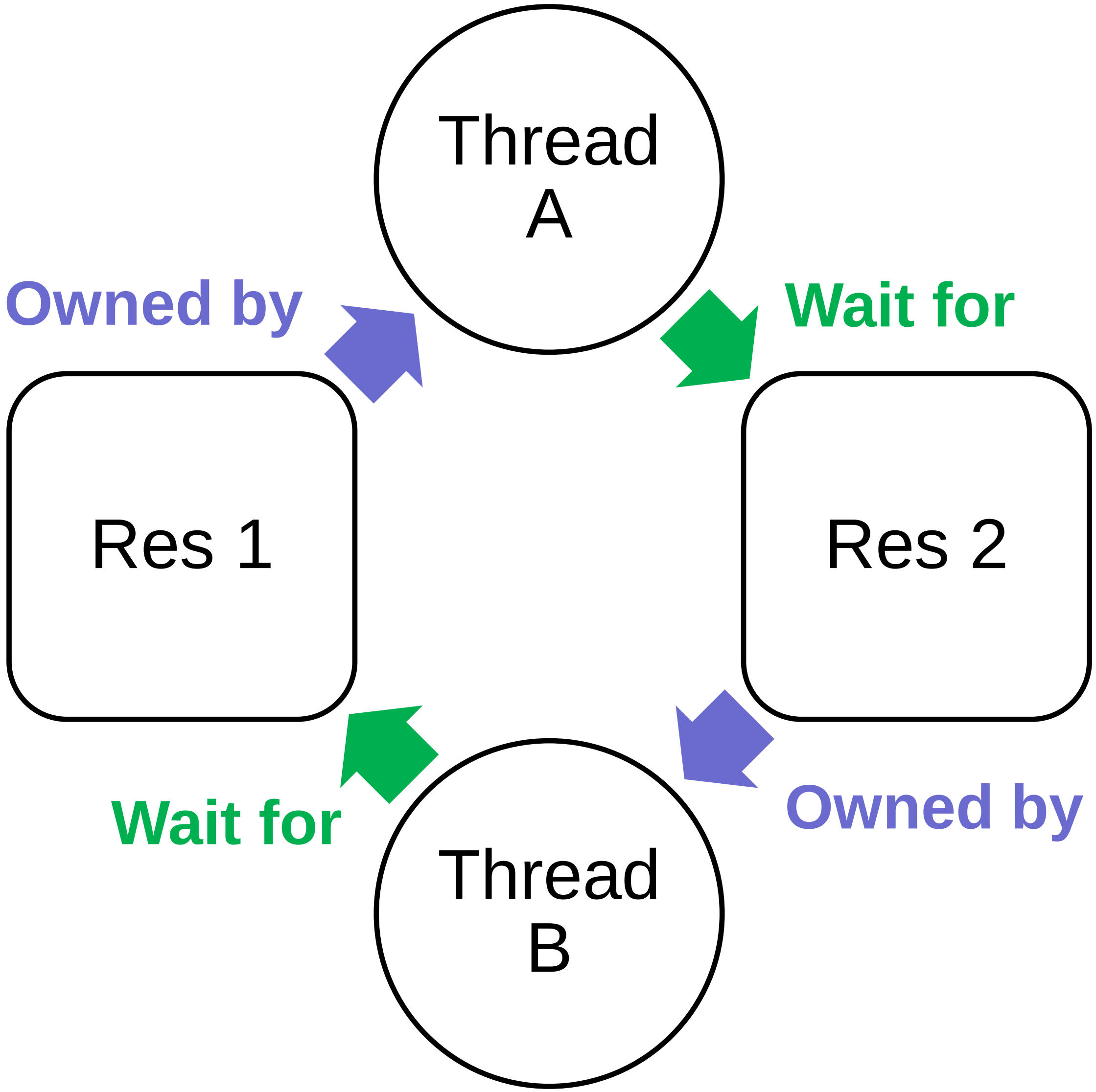


Image Source: https://thumbs.dreamstime.com/t/vector-cartoon-man-crawling-desert-found-empty-text-doodle-stickman-thirsty-middle-meet-sign-87969853.jpg

- 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

Image Credit: http://mcs109.bu.edu/site/files/deadlock/citydeadlock.jpg

DEADLOCK Graph



Four requirements for Deadlock

Mutual exclusion	Hold and wait	No preemption	Circular wait
Only one thread at a time can use a resource.	Thread holding at least one resource is waiting to acquire additional resources held by other threads	Resources are released only voluntarily by the thread holding the resource, after thread is finished with it	There exists a set $\{T_1, \dots, T_n\}$ 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

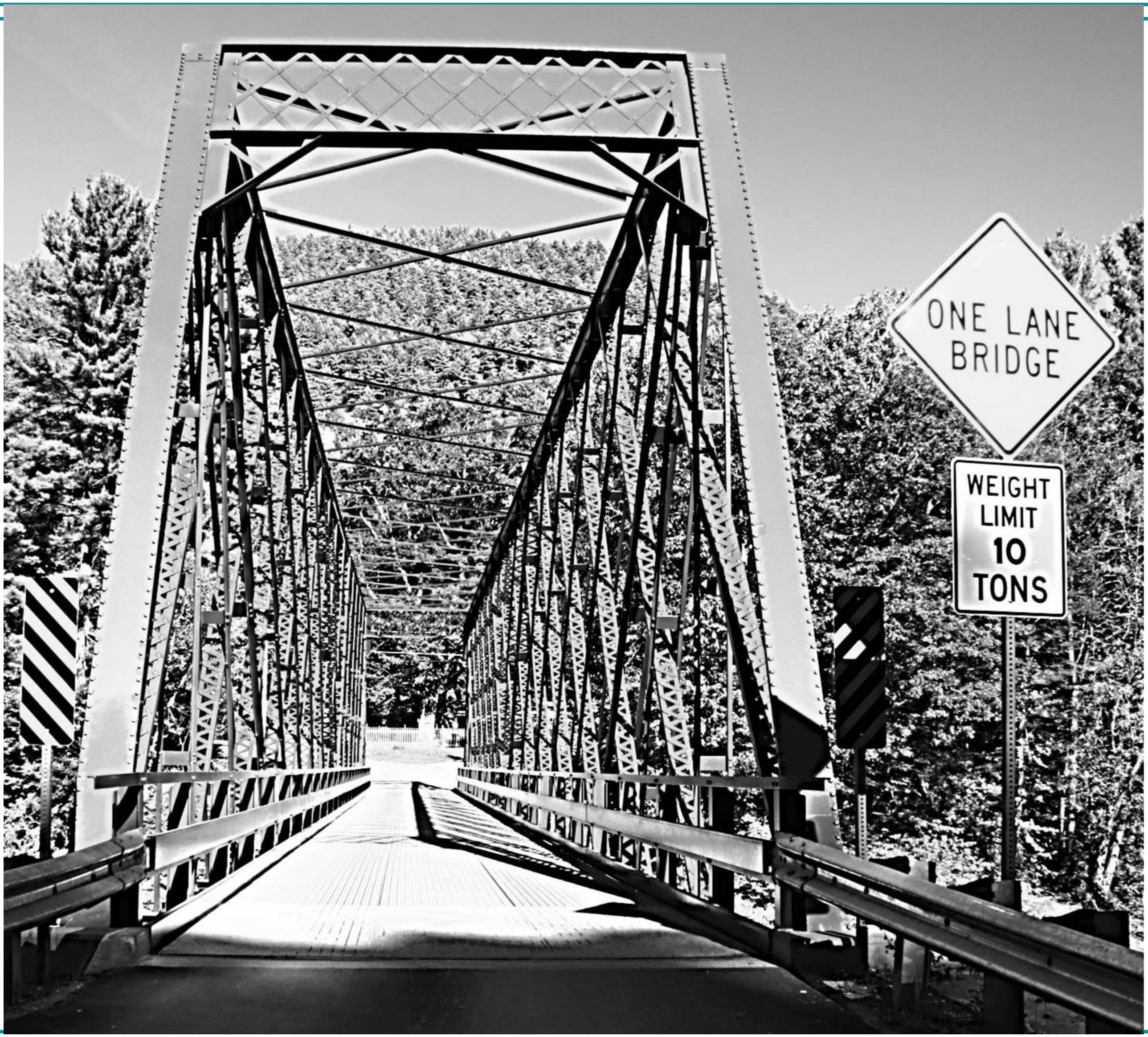
Occasional Deadlock

- Deadlock not always deterministic: Example of two mutexes:

Thread A	Thread B
`x.P();`	`y.P();`
`y.P();`	`x.P();`
`y.V();`	`x.V();`
`x.V();`	`y.V();`

- Deadlock won't always happen with this code
  - Have to have exactly the right timing ("wrong" timing?)
  - So you release a piece of software, and you tested it, and there it is, controlling a humus making machine…
- Deadlocks occur with multiple resources
  - Means you can't decompose the problem
  - Can't solve deadlock for each resource independently
- Example: System with two disk drives and two threads
  - Each thread needs two disk drives to function
  - Each thread gets one disk and waits for another one

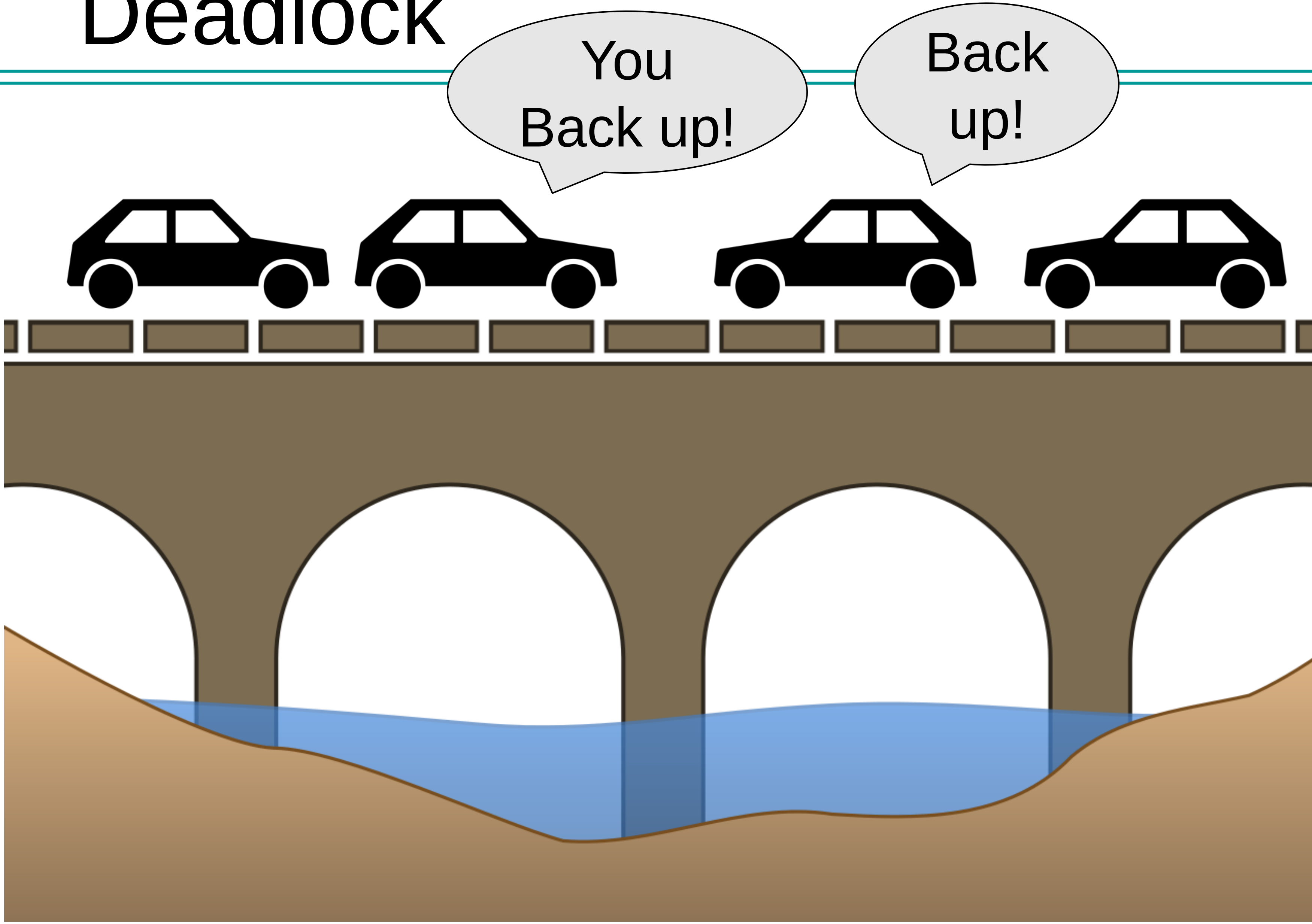
One Lane Bridge Crossing



One Lane Bridge Crossing

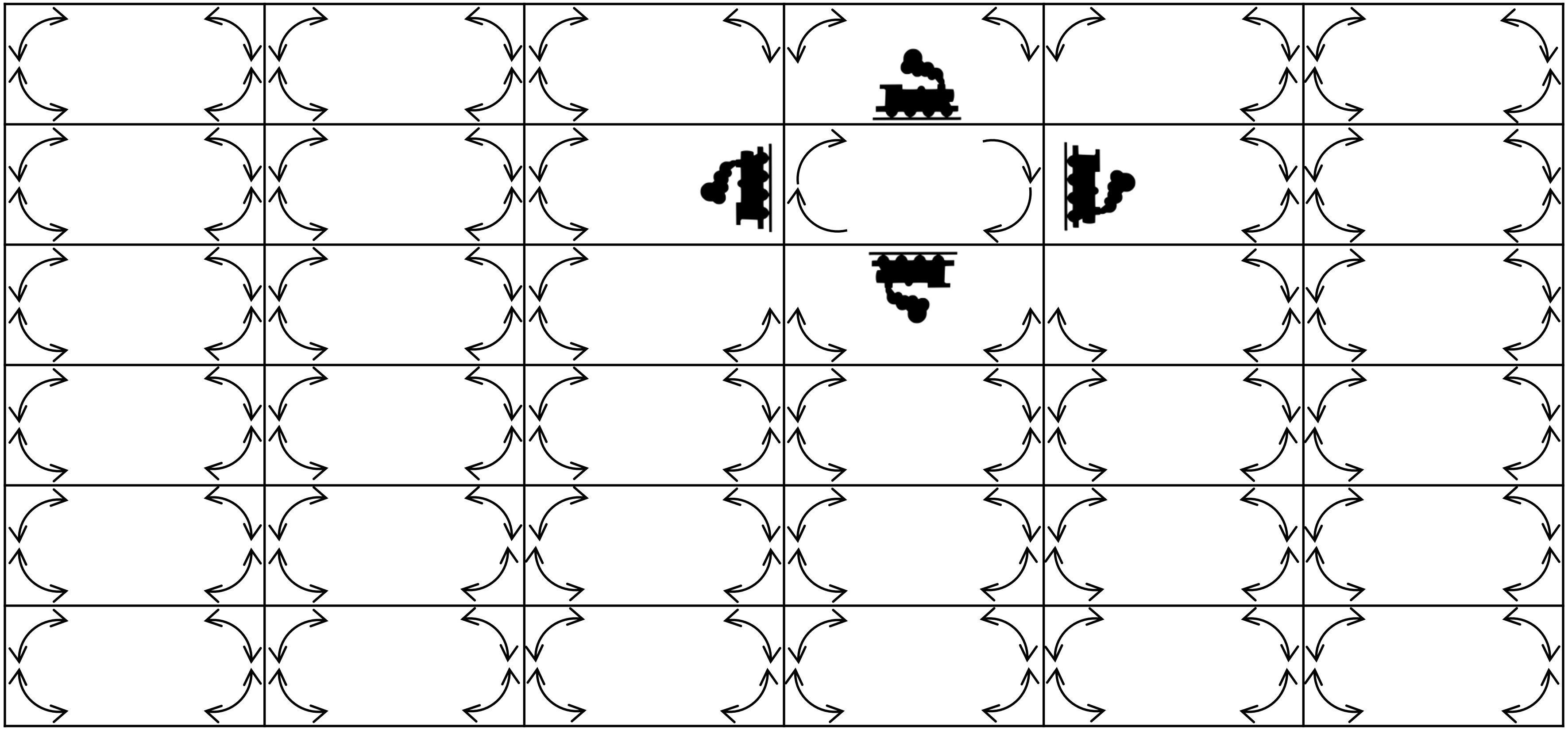
- Each segment of road is a resource
  - Car must own the segment under it
  - Must acquire segment that it is moving into
- For bridge: Must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and 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

Deadlock



Trains (Wormhole-Routed Network)

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



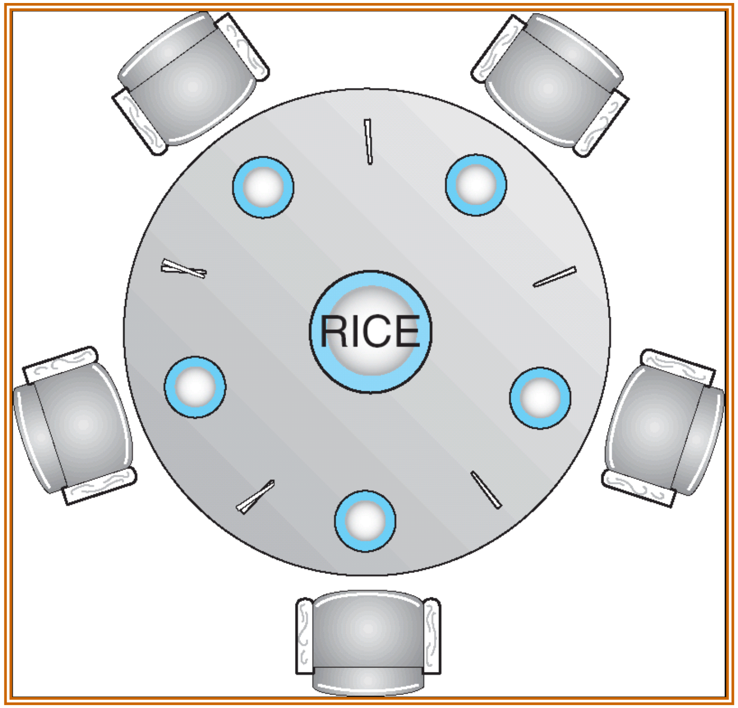
Trains (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Similar problem to multiprocessor networks

- Fix? Imagine grid extends in all four directions
  - How can we prevent circular dependency?

Dining Students Problem

- Five chopsticks/Five students (really cheap restaurant)
  - Free-for all: Student 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 (!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?

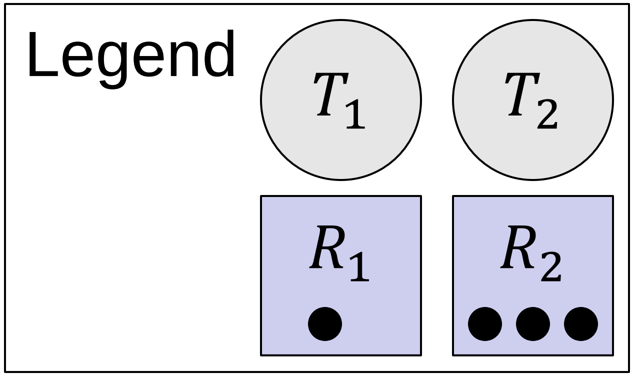


Picturing deadlock can help us understand it better…

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()`

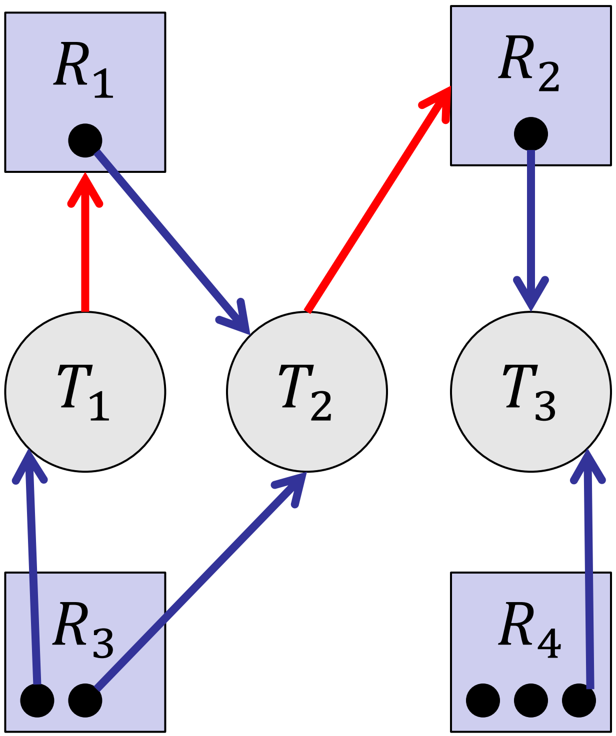


- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$ the set of threads in the system.
    - $R = \{R_1, R_2, \ldots, R_n\}$ the set of resource types in system
  - Request Edge – Directed edge $T_i \rightarrow R_j$
  - Assignment Edge – Directed edge $R_j \rightarrow T_i$

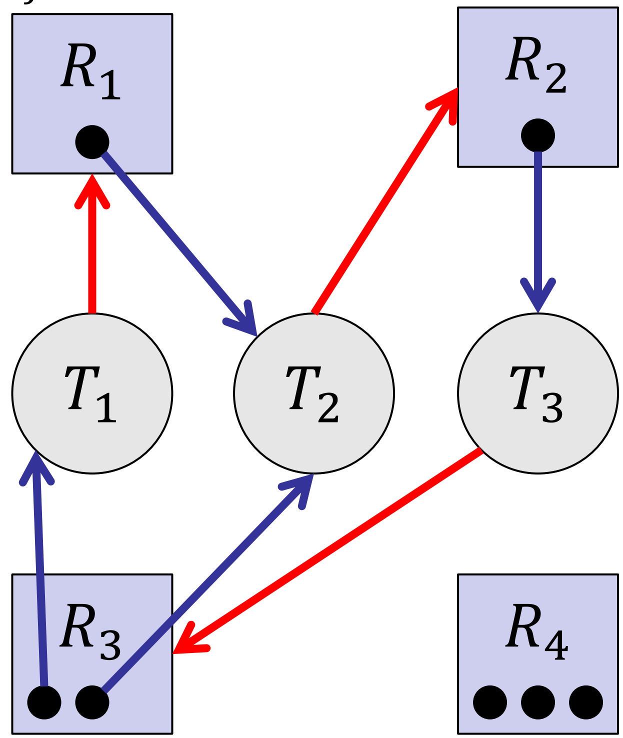
Resource Allocation Graph Examples

Request Edge – Directed edge $T_i \rightarrow R_j$

Assignment Edge – Directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph

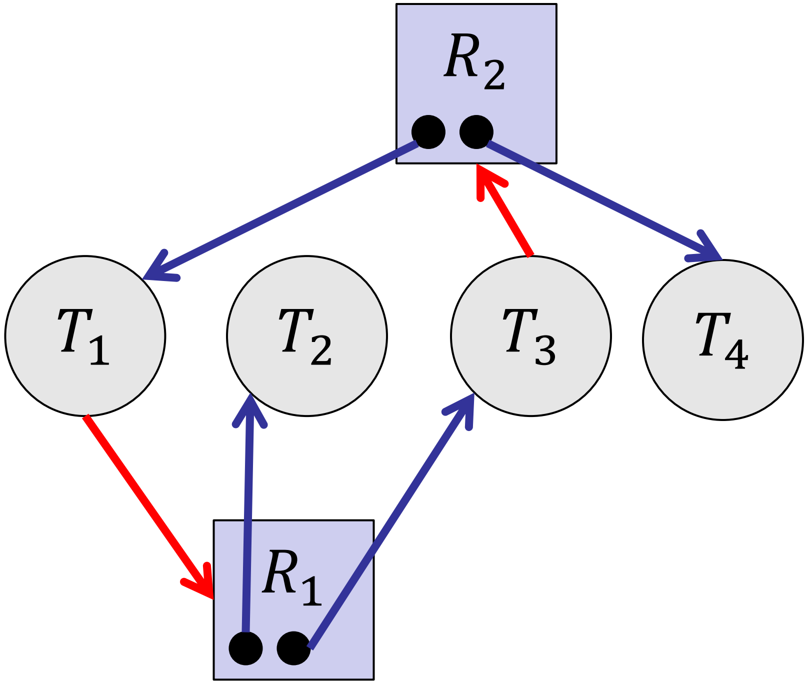


Allocation Graph with Deadlock

Resource Allocation Graph Examples

Request Edge – Directed edge $T_1 \rightarrow R_j$

Assignment Edge – Directed edge $R_j \rightarrow T_i$



Allocation Graph with Cycle
But No Deadlock

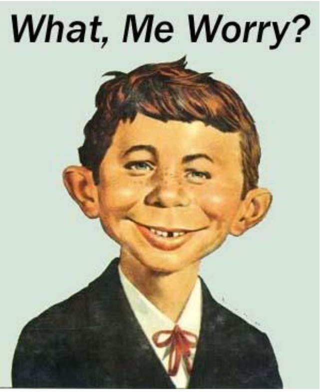
Methods for Handling Deadlock

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

Image sources: http://www.besthackingtricks.com/wp-content/uploads/2015/08/Recover.jpeg,
https://s-media-cache-ak0.pinimg.com/originals/d4/c6/ea/d4c6ea0e2d387adadf284ac846b05090.jpg

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

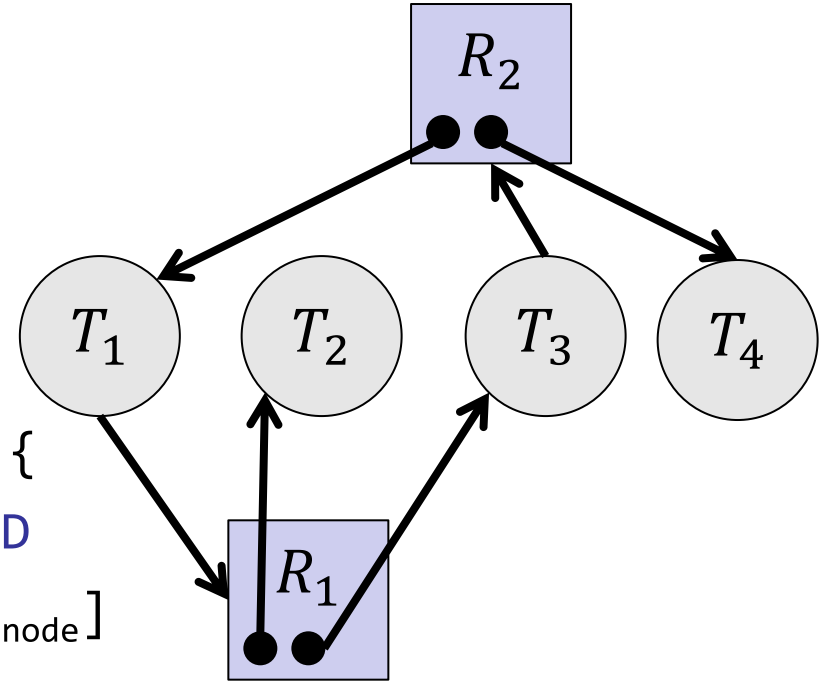
    [$\texttt{Request}_X$]: Current requests from thread X

    [$\texttt{Alloc}_X$]: Current resources held by thread X

- See if tasks can eventually terminate on their own

Deadlock Detection Algorithm

```
[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)
```



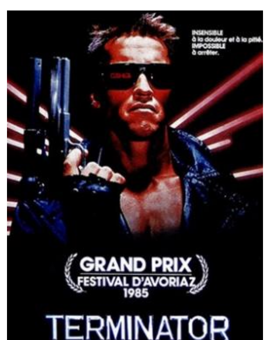
- Nodes left in UNFINISHED $\Rightarrow$ deadlocked

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!
- Expel a dining student (or tell him there's free beer elsewhere)
- Not always possible – killing a thread holding a mutex leaves the world inconsistent



Preempt resources

Without killing off thread

- Take away resources from thread temporarily
- Doesn't always fit with semantics of computation

Image source: http://www.aperfectworld.org/05202.html

What to do when detect deadlock?

Roll back

Undo 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)
- If you restart in exactly the same way, may reenter deadlock once again

None of the above

- Many operating systems use other options

Image source: Walmart

Techniques for Preventing Deadlock

- 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 (ex. virtual memory)
- Examples:
  - Ayalon Fwy with 12,000 lanes. Never wait!
  - Infinite disk space (not realistic yet?)

No Sharing of resources

- Totally independent threads
- Not very realistic

Techniques for Preventing Deadlock

Don't allow waiting

- How the phone company avoids deadlock
  - Call to your Mom in Haifa, works its way through the phone lines, but if blocked get busy signal.
- Technique used in Ethernet and some multiprocessor nets
  - Everyone speaks at once. On collision, back off and retry
- Inefficient, since need to retry
  - Consider: Driving to Tel Aviv; when hit traffic jam, suddenly you're transported back home and told to retry!

Image source: http://images.myparkingsign.com/img/lg/K/Fire-Zone-No-Parking-Sign-K-4442.gif

Request Ahead

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 Ayalon Fwy at a time

Image source: http://ralphies.com/assets/images/order-before.jpg

Techniques for Preventing Deadlock

Ordering

Force all threads to request resources in a particular order preventing any cyclic use of resources

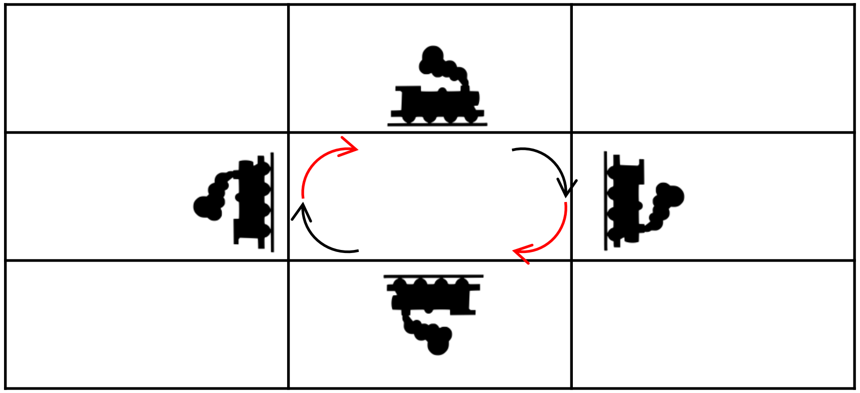
- Thus, preventing deadlock

Example `(x.P, y.P, z.P,…)`

- Make tasks request disk, then memory, then…
- Keep from deadlock on freeways around Tel Aviv by requiring everyone to go clockwise

Image source: http://www.drivingtesttips.biz/wp-content/uploads/2014/05/mini-roundabout-sign.jpg

Recall: Dimension Ordering



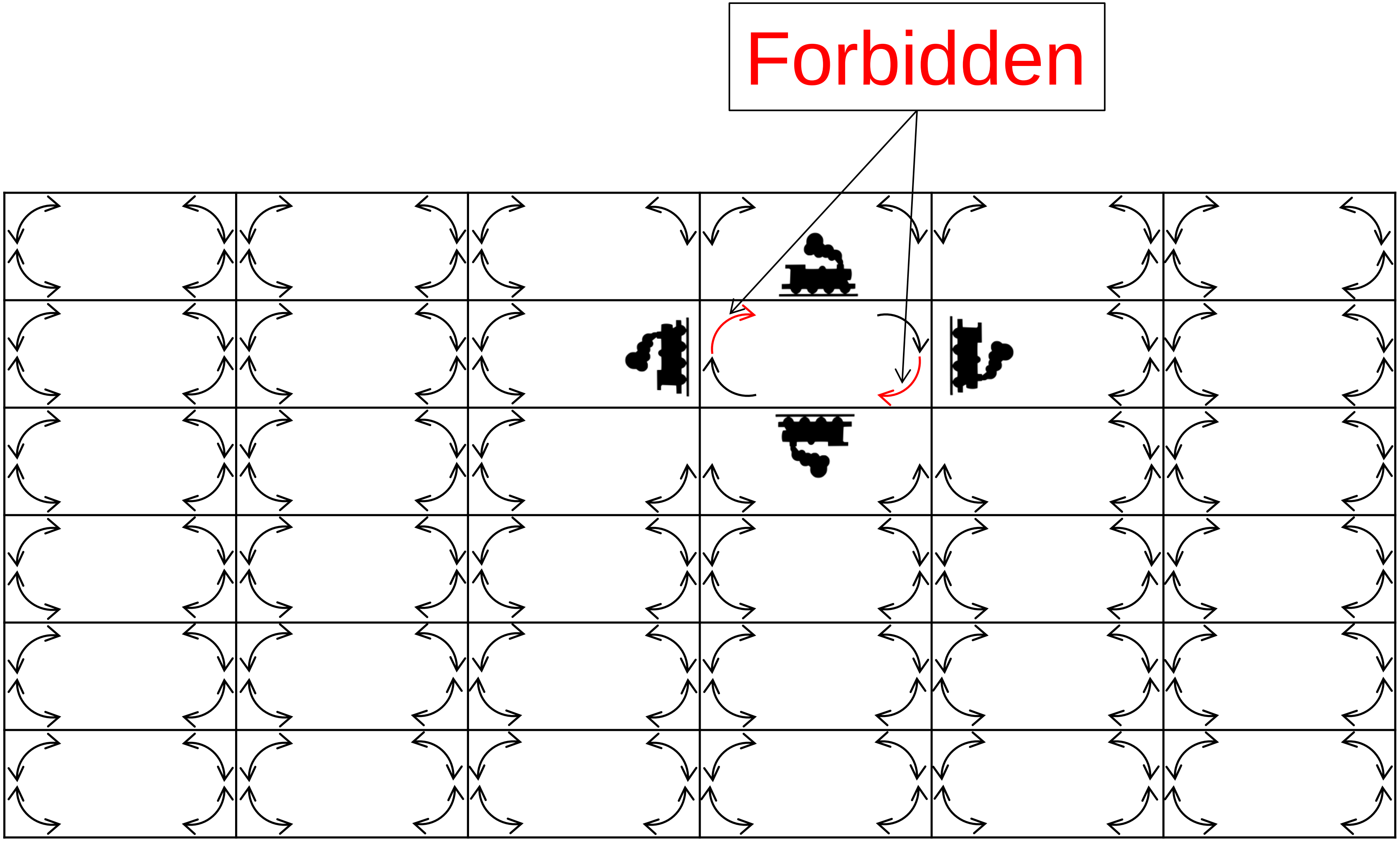
- Force ordering of channels (tracks)
- Protocol: Always go east-west first, then north-south
  - X then Y

Trains (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Similar problem to multiprocessor networks

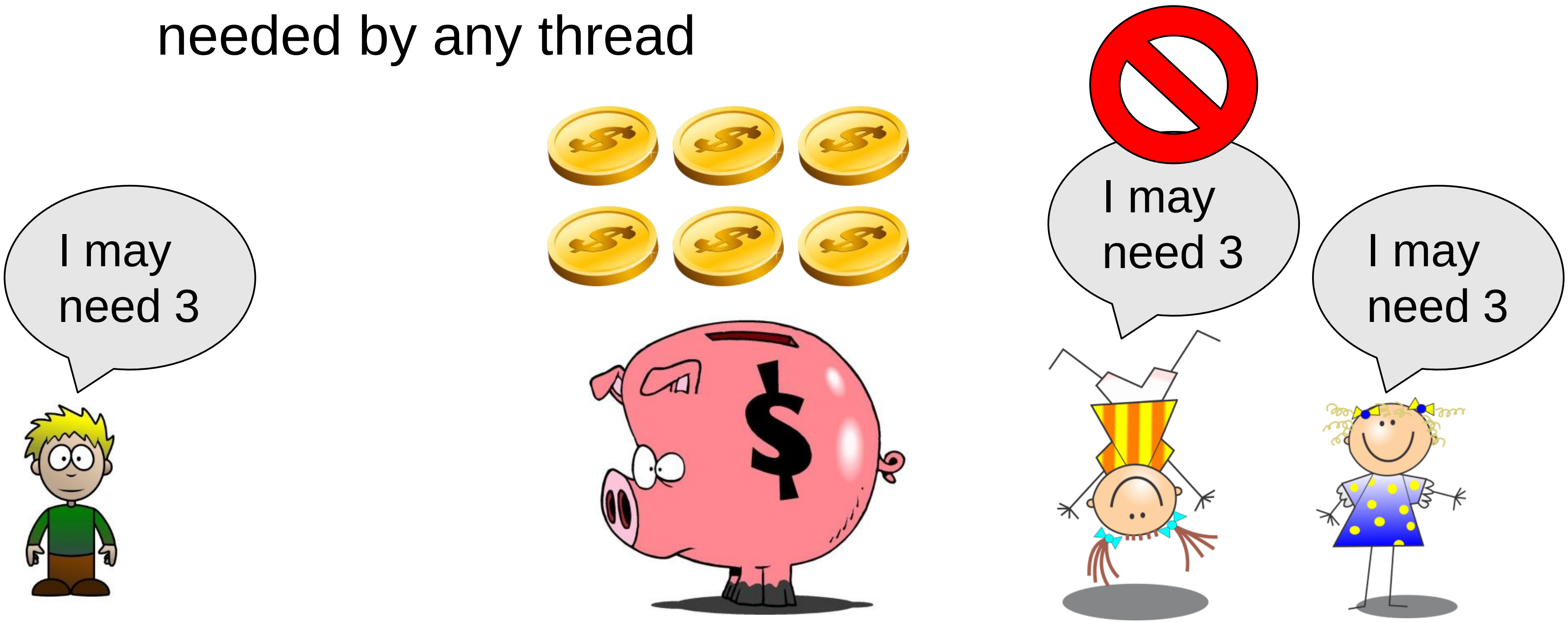
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  - "dimension ordering" (X then Y)

Dimension Ordering



Another Idea: Accounting

- First Idea:
  - Thread t states maximum resource needs in advance
  - System allows thread t to proceed if: $([\mathrm{Avail}] - [Request_t] \geq Max)$ remaining that might be needed by any thread

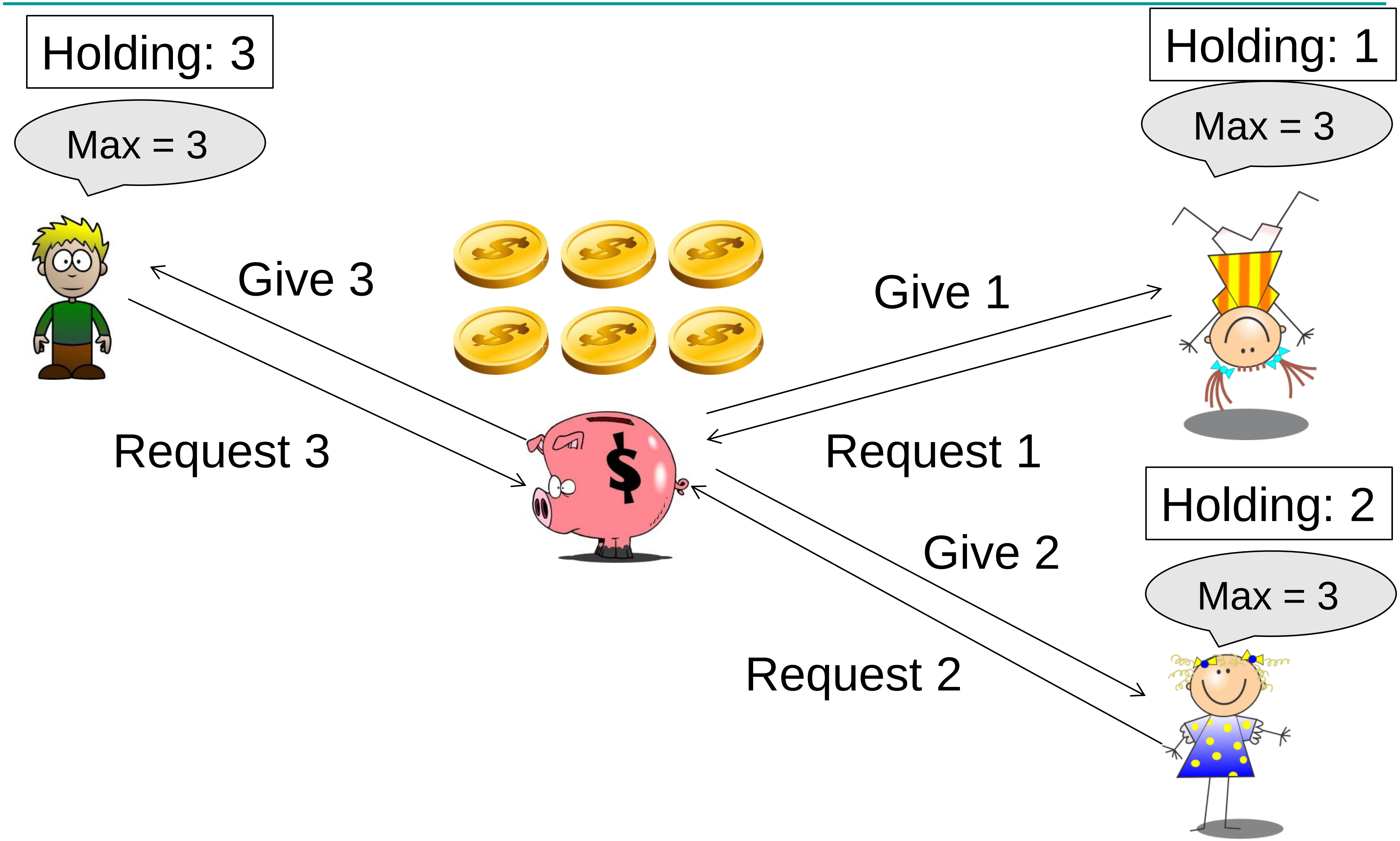


Banker's algorithm (less conservative):

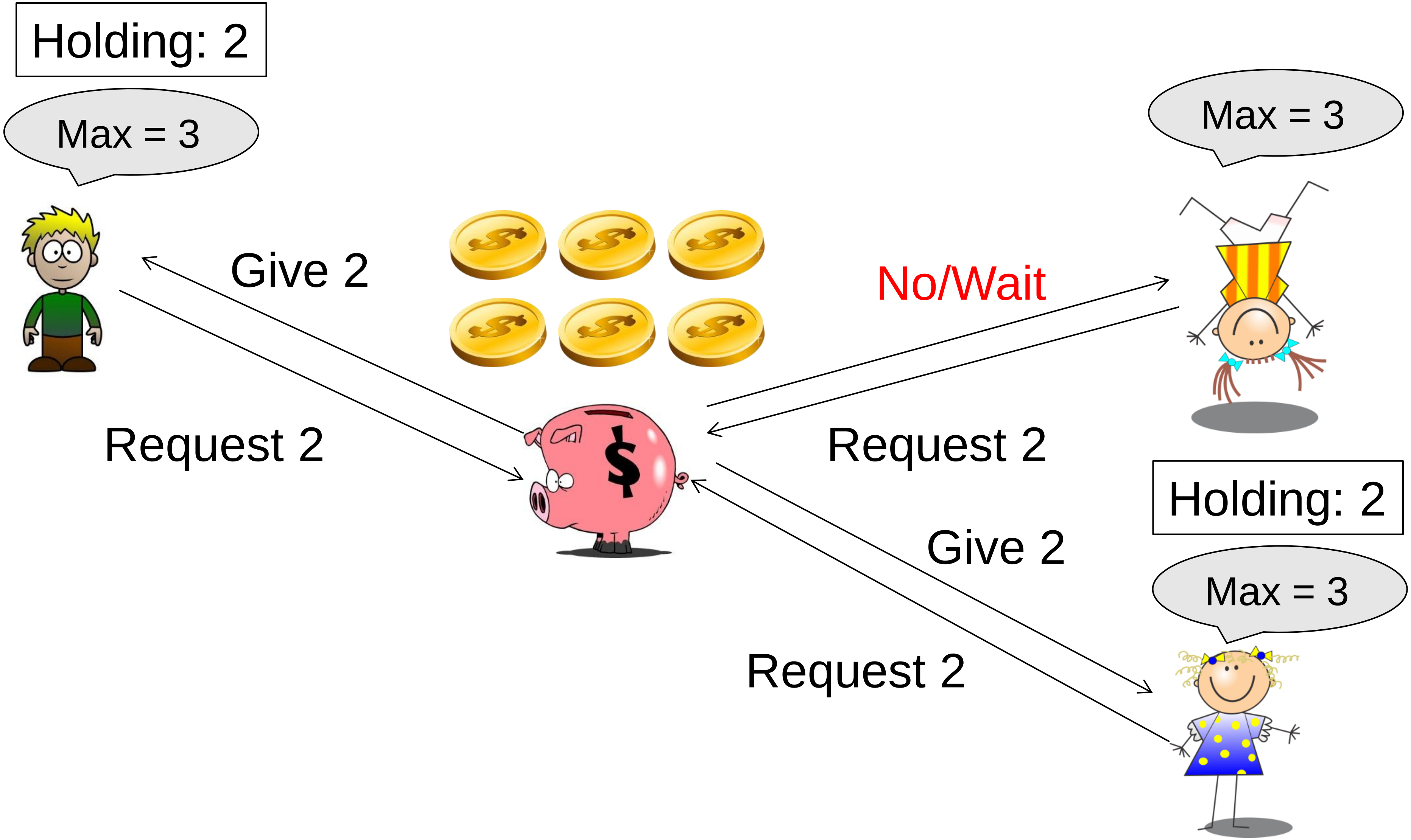
- Allocate resources dynamically
  - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
  - Technique: Pretend each request is granted, then run deadlock detection algorithm, substituting
    $([Max_{node}] - [Alloc_{node}] \leq [Avail])$ for $([Request_{node}] \leq [Avail])$
    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

Banking Example 1



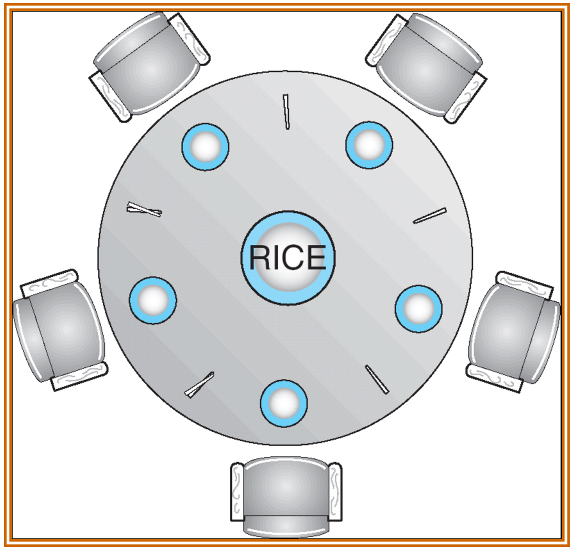
Banking Example 2



Banker's Algorithm Example

Banker's algorithm with dining students

- Safe (won't cause deadlock) if when you try to grab chopstick either:
  1. Not the last chopstick
  2. Is the last chopstick but someone will have two afterwards



- What if k-handed students? 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 $k - 2$
  – etc…

Image Source: http://gmythology.blogspot.com/2008/12/hundred-handed-ones-hecatoncheires.html

Conclusion

- Starvation and Deadlock