CSC 112: Computer Operating Systems
Lecture 4

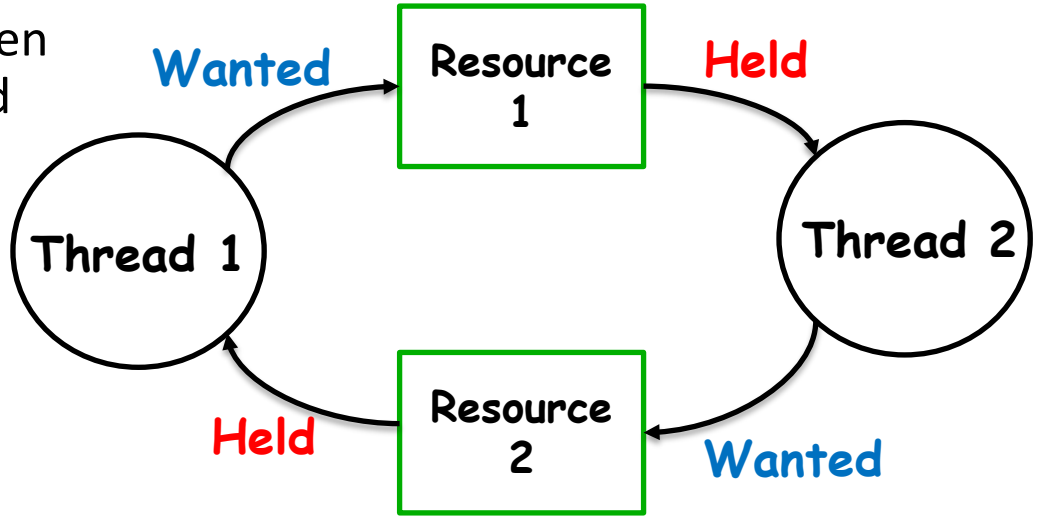
Deadlocks

Department of Computer Science,
Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

Deadlock

- Definition: A set of threads are said to be in a deadlock state when every thread in the set is waiting for an event that can be caused only by another thread in the set
- Conditions for Deadlock
- Mutual exclusion
  - Only one thread at a time can use a given resource
- Hold-and-wait
  - Threads hold resources allocated to them while waiting for additional resources
- No preemption
  - Resources cannot be forcibly removed from threads that are holding them; can be released only voluntarily by each holder
- Circular wait
  - There exists a circle of threads such that each holds one or more resources that are being requested by next thread in the circle



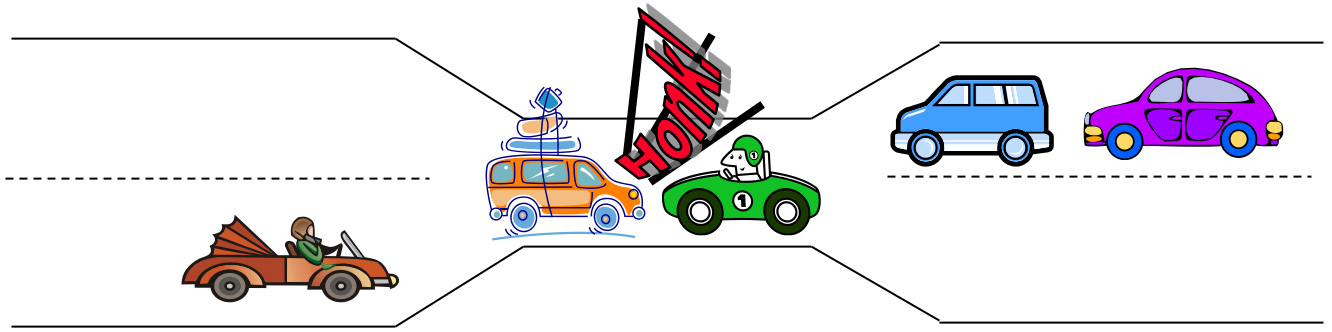
Not a perfect analogy, just a fun image!

Starvation vs Deadlock

- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority thread
- Deadlock: circular dependency waiting for resources
- Deadlock $\Rightarrow$ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Bridge Crossing Analogy

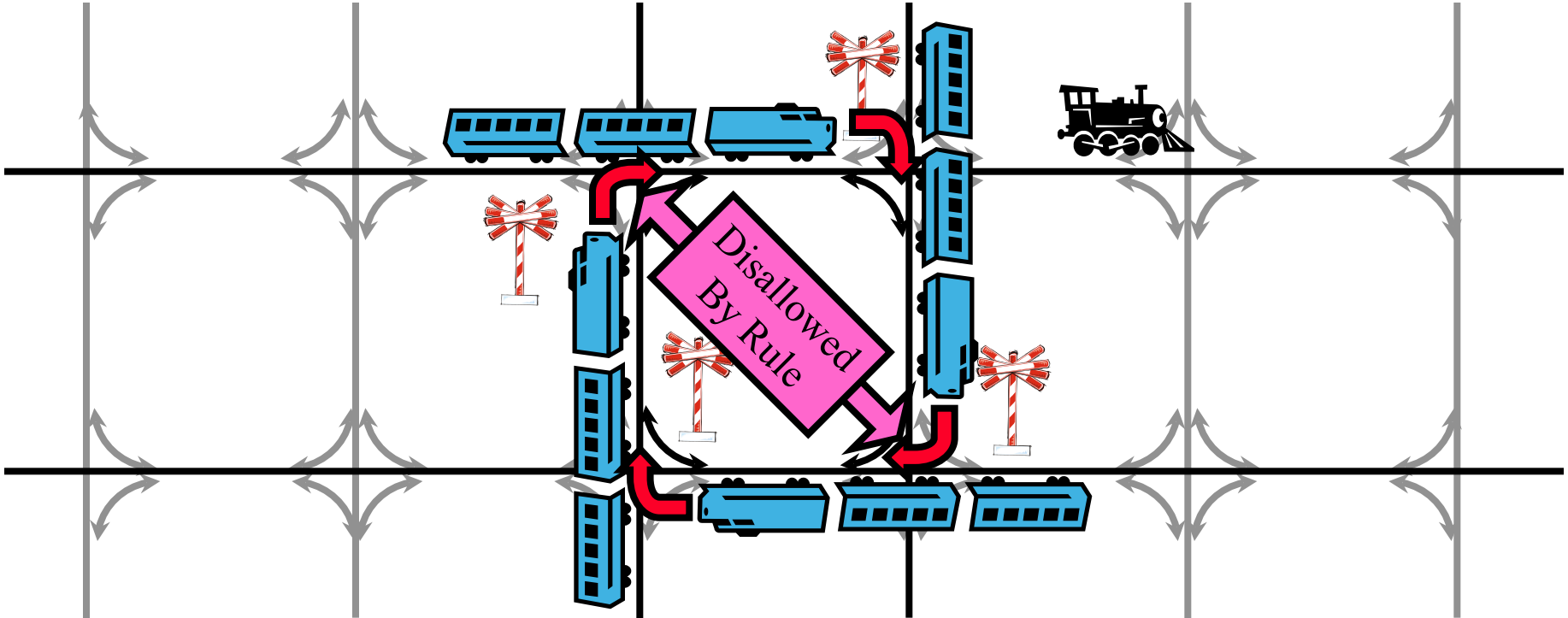
- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires 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
  - Heavy traffic going east ⇒ no car can go west



4

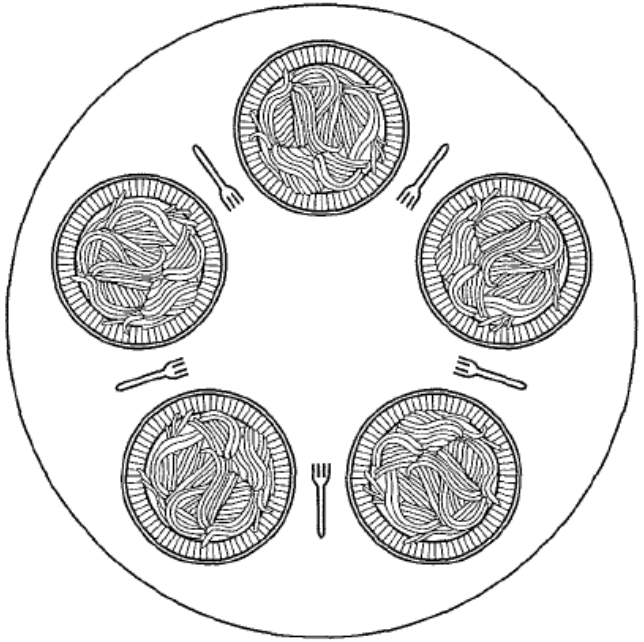
Train Example (Wormhole-Routing for Network-on-Chip)

- Circular dependency (Deadlock!)
  - Each train wants to turn right, but blocked by other trains
  - Similar problems occur for Network-on-Chip
- One solution:
  - Force ordering of channels (fixed global order on resource requests)
    - » Protocol: Always go horizontal (east-west) first, then vertical (north-south)
  - Called “dimension ordering” (X then Y)



Dining philosophers

- 5 forks/5 philosophers
  - Need two forks to eat
- What if all grab left fork at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a fork
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let philosopher take last fork if no hungry philosopher has two forks afterwards



DINING-PHILOSOPHERS PROBLEM: SIMPLIFIED
https://www.youtube.com/watch?v=VSkvwzqo-Pk

Handling Deadlocks

- Deadlock prevention
  - Make sure one of the conditions necessary to create a deadlock cannot be present in the system
- Deadlock detection
  - Resource Allocation Graph (cannot handle multi-instance resources well)
  - Banker's algorithm for detecting (potential) deadlocks
- Deadlock recovery
  - Let deadlock happen
  - Monitor the system state periodically to detect when deadlock occurs
  - Take action to break the deadlock

7

Deadlock Prevention Techniques

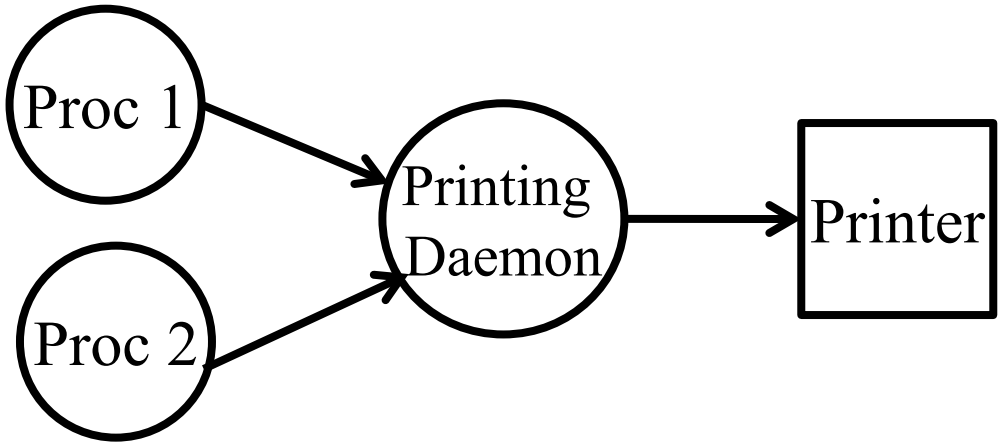
- Break “mutual exclusion” by spooling resources
- Break “hold and wait” condition: Make all threades request everything they’ll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 forks, request both at same time (our solution to dining philosopher problem)
- Break “circular wait” condition:
  - Force all threades to request resources in a particular order to prevent cyclic use of resources. Example:
    - » Request disk, then memory, then…
    - » Recall: simple solution to dining philosopher problem
    - » May not be practical, since runtime resource usage pattern is generally unknown
  - Banker’s algorithm can prevent future “circular wait” conditions by detecting *potential* deadlocks
- Break “no preemption” condition:
  - Forcibly remove resources from threads

Condition	Approach
Mutual exclusion	Spooling
Hold and wait	request all resources initially
Circular wait	Request resources in a particular order
No preemption	Take resources away

8

Spooling

- A single daemon thread directly uses the resource; other threades send their requests to the daemon, e.g.:
- The resource is no longer directly shared by multiple threades



9

Request all resources initially

- Disallow hold-and-wait
  - Make each thread request all resources at the same time and block until all resources are available to be granted simultaneously
    - » One solution to Dining Philosopher problem
  - May be inefficient
    - » thread may have to wait a long time to get all its resources when it could have proceeded and completed a significant portion of it work with the resouces that were already available
    - » Resources allocated to a thread may remain unused for long periods of time blocking other threades
    - » threades may not know all resources they will require in advance.

10

Order resources numerically

- Prevent circular wait
  - Define a total order of resources; If a thread holds certain resources, it can subsequently request only resources that follow the types of held resources in the total order.
  - This prevents a thread from requesting a resource that might cause a circular wait.
    - » Example; all threads requests sem1 before sem2
    - » Another solution to Dining Philosopher's problem
  - Introduces inefficiencies and can deny resources unnecessarily.

```
semaphore sem1, sem2;
void t1( ) {
  sem1.wait();
  sem2.wait();
  //Critical Section
  sem2.post();
  sem1.post();
}
void t2( ) {
  sem2.wait();
  sem1.wait();
  //Critical Section
  sem1.post();
  sem2.post();
}
```

Possible deadlock

```
semaphore sem1, sem2;
void t1( ) {
  sem1.wait();
  sem2.wait();
  //Critical Section
  sem2.post();
  sem1.post();
}
void t2( ) {
  sem1.wait();
  sem2.wait();
  //Critical Section
  sem2.post();
  sem1.post();
}
```

No deadlock

Take resources away

- Allow preemption. Can be implemented different ways
  - If a thread holding a resource is denied another resource and forced to wait, it must relinquish the resource it is holding and request it again (if needed) when the blocked resource is available
  - If a thread requests a resource that is in use (usually by a lower priority thread), the thread using the resource will be preempted and the resource will be supplied to the requesting thread.
  - Requires additional OS complexity
- Mainly used for deadlock recovery, not prevention

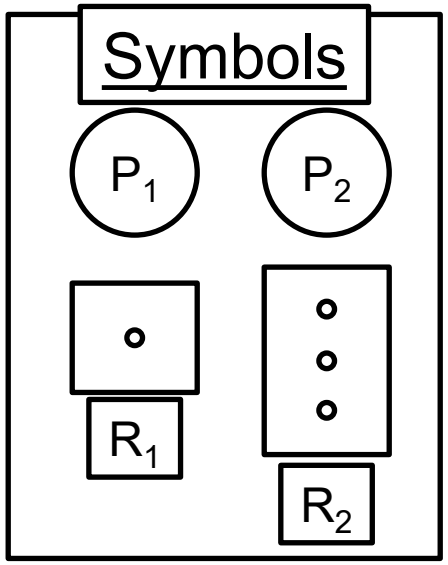
Ostrich algorithm

- Ignore the possibility of deadlock, maybe it won't happen
  - In some situations this may even be reasonable, but not in all
  - If a deadlock in a thread will happen only once in 100 years of continuous operation we may not want to make changes that will likely decrease efficiency to avoid that rare event.
  - Events will occur randomly, we don't know that the 1 in 100 years will not occur in 1 second. If a deadlock in a thread will happen on average once per minute, we probably want to do something other than implement the ostrich algorithm and ignore it
- In mission critical applications, the ostrich algorithm approach is inappropriate if a catastrophic failure may result from a deadlock

Resource-allocation graph (RAG)

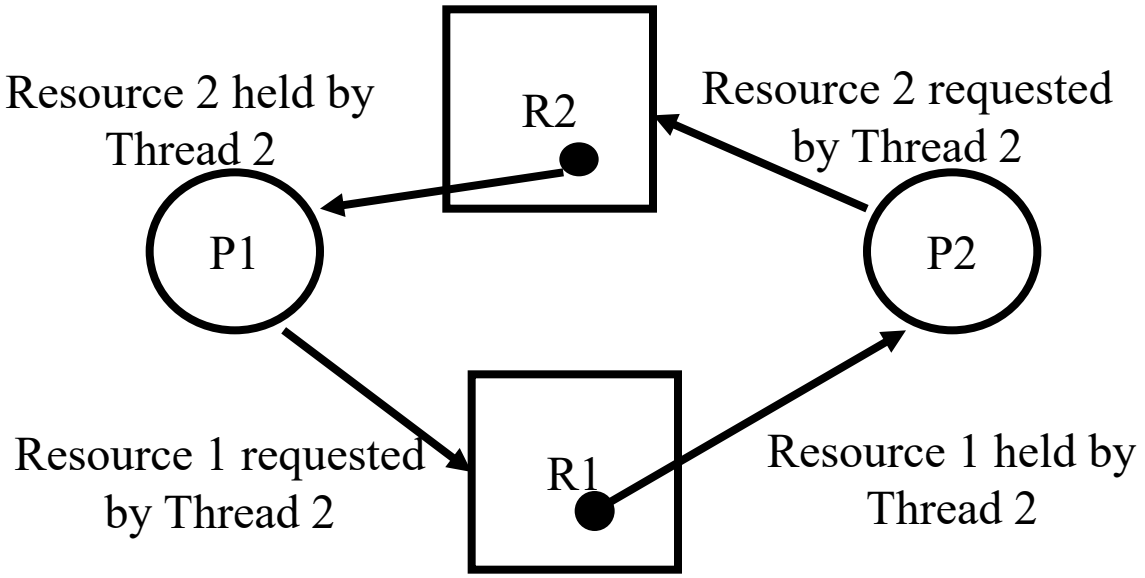
- System Model
  - A set of threades P_1, P_2, . . ., P_n
  - Resource types R_1, R_2, . . ., R_m

    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 (RAG):
  - V is partitioned into two types:
    - » $P = \{P_1, P_2, \ldots, P_n\}$, set of threades in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, set of resource types in system
  - request edge – directed edge $P_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$



RAG for deadlock detection

- For any given sequence of requests for and releases of resources a RAG can be constructed
- We check the graph
  - no cycle → no deadlock
  - Each resource has a single instance AND cycle → deadlock (necessary and sufficient)
  - Each resource has multiple instances AND cycle → maybe deadlock (but not sufficient condition)
    - » Need Banker's algorithm to detect deadlocks



A RAG with a deadlock

A deadlock example

A
Request R
Request S
Release R
Release S

(a)

B
Request S
Request T
Release S
Release T

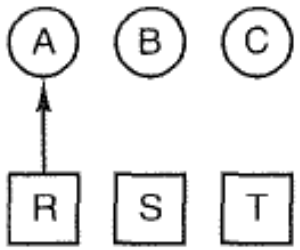
(b)

C
Request T
Request R
Release T
Release R

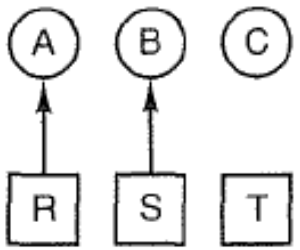
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

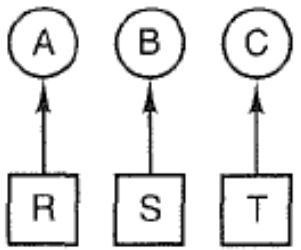
(d)



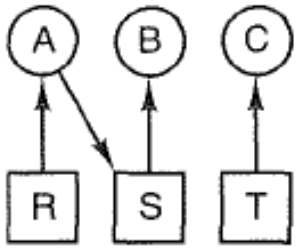
(e)



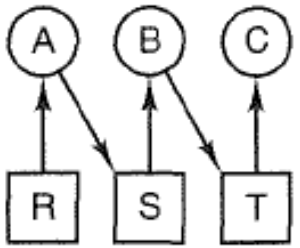
(f)



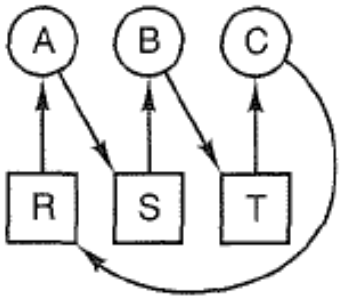
(g)



(h)



(i)



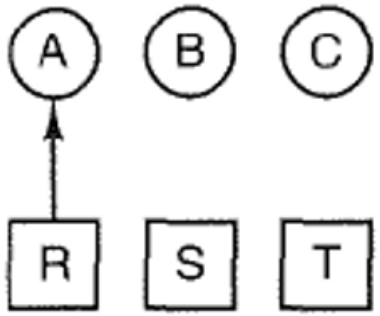
(j)

16

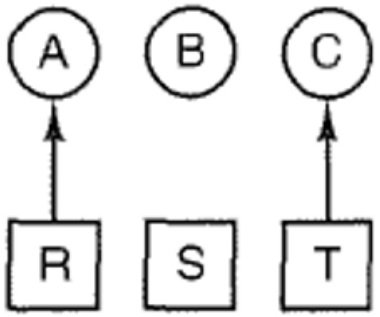
Deadlock is avoided by delaying B's request

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

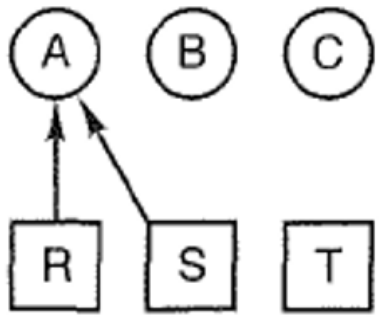
(k)



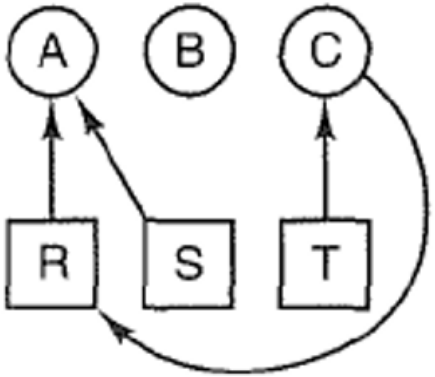
(l)



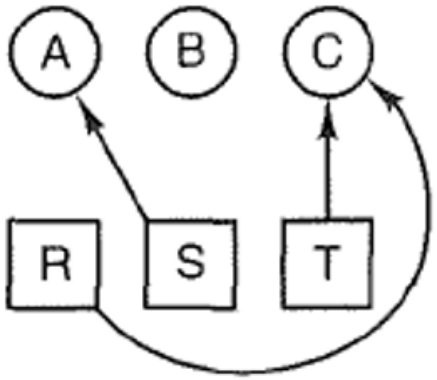
(m)



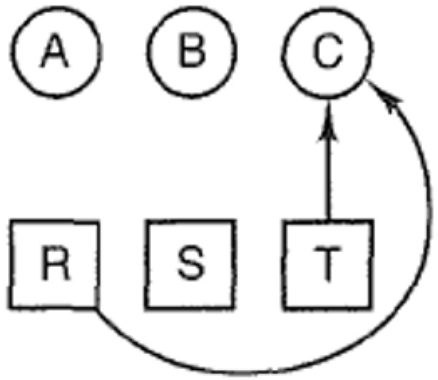
(n)



(o)

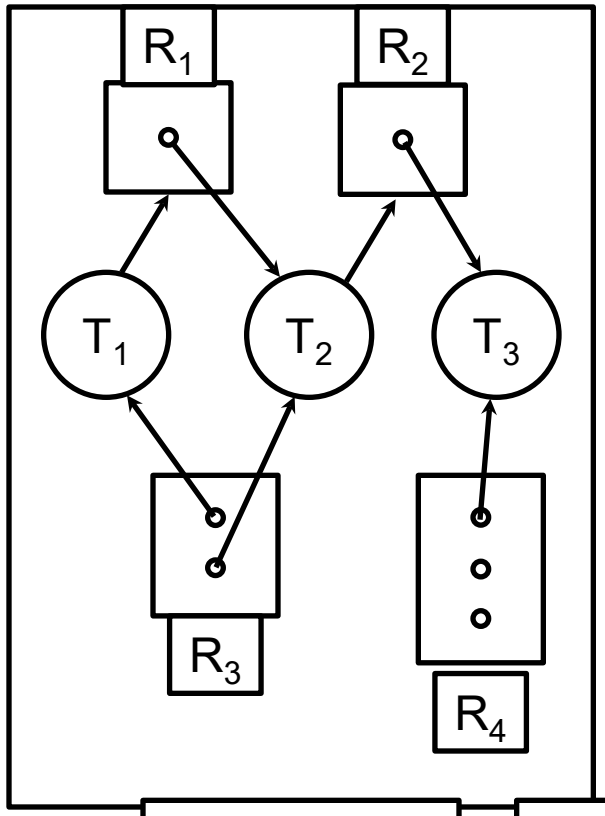


(p)

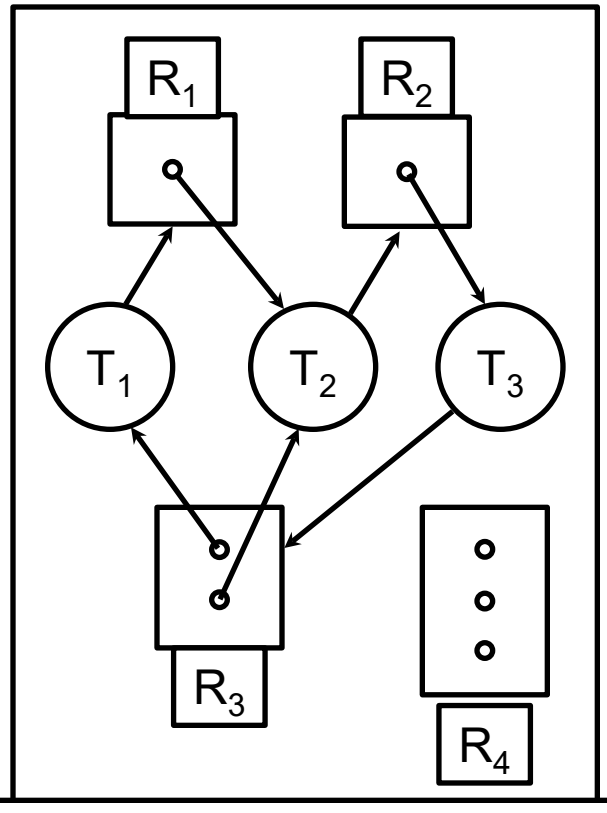


(q)

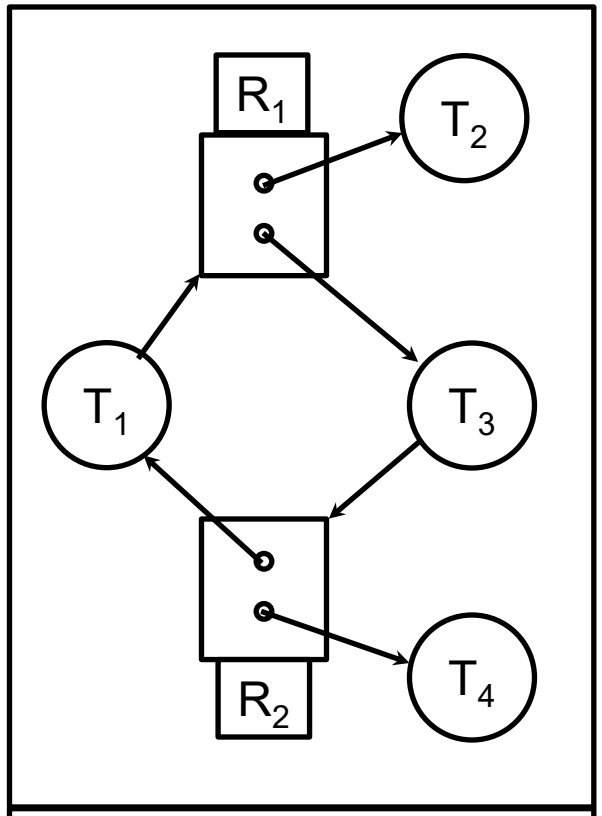
Resource Allocation Graph Examples



No Cycle;
No Deadlock



Deadlock (cycle R_3->T_2->R_2->T_3
And cycle R_3->T_1->R_1->T_2->R_2->T_3



Cycle,
but No Deadlock
(T_2 or T_4 may release
res later)

Banker's algorithm for deadlock detection

- To avoid deadlocks we need to be able to detect them, preferably before they occur.
- RAG can only detect deadlocks reliably for the case of single-instance resources.
- Banker's algorithm is more general and can deal with multiple-instance resources. It is used to recognize when it is safe to allocate resources
  - Analyze the state of the system; If the state is unsafe, take actions to break actual or potential deadlocks and bring the system back to a safe state
  - Do not grant additional resources to a thread if this allocation *might* lead to a deadlock

Problem Definition

- Consider a system with *n* threades and *m* different types of resources. $E = (E_1, E_2, \dots, E_m)$: *Existing resource vector*
  - We can have multiple instances of a resource type, so the value of E_i is the number of resources of type i that are exist in the system
- We keep track of how many instances of each resource type are currently available (not in-use) with $A = (A, A_2, \dots, A_m)$: Available resource vector
- C: *Current allocation matrix* denotes which threades are using which of the resources that are in use.
  - e.g., if thread *i* is using 2 resources of type *j* then $C_{ij} = 2$. thread *i has claimed* these 2 resources already.
- *R: Total request matrix* denotes which threades will need which of the resources during their execution
  - e.g., if thread *i* need 4 resources of type *j* then $R_{ij} = 4$. thread *i will need a total of* 4 resources during its execution.
- For each thread *i* and resource *j*: $C_{ij} \leq R_{ij} \leq E_j, \forall i, j$

Four data structures encode current state of the system

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Total Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Safe states and unsafe states

- The state of the system can be either safe or unsafe
  - A safe state is a state in which there exists at least one sequence of resource allocations that will allow all threades in the system to complete without deadlock
  - An unsafe state is a state in which there exists no sequence of resource allocation that will complete all threades in the system without deadlock
- Each time a resource is requested, determine the state of the system after the resource is allocated, and if that state is safe or unsafe (will potentially deadlock). If the state will be unsafe we block the thread and do not allocate the requested resources (to avoid potential deadlock)

Bankers algorithm: preliminaries

- Compute *R-C:* Resources needed matrix
- To determine if a thread *i* can run to completion, we need to compare two vectors:
  - $(R\text{-}C)_i$: row *i* in the matrix *R-C* of unmet resource needs
  - A: available resources vector *A*
  - $(R\text{-}C)_i <= A$ if $(R_{ij} - C_{ij}) <= A_j$ for all resource type *j*

Banker's algorithm

Algorithm CheckSafety() for checking to see if a state is safe:

1. Compute resources needed *R-C*
2. ook for a thread *i* that can run to completion by finding an unmarked row *i* with $(R\text{-}C)_i \leq A$. If no such row exists, system will eventually deadlock since no thread can run to completion
3. Assume thread of row *i* requests all resources it needs and finishes. Mark thread *i* as completed, free all its resources and add the *i-th* row of *C* to the *A* vector
4. Repeat steps 1 and 2 until either all threades marked terminated (initial state is safe) or no thread is left whose resource needs can be met (there is a deadlock, so initial state is unsafe).

Banker's algorithm cont'

- When a thread makes a request for one or more resources:
  - Update the state of the system assuming the requests are granted.
  - Determine if the resulting state is a safe state by calling CheckSafety()
    - » If so grant the request for resources.
    - » Otherwise, block the thread request until it is safe to grant it.

An example system: starting state

- 4 threads P1 through P4; 4 resource types with 10, 5, 6, 5 instances each.
- Current system state encoded in matrices *R*, *C* and vectors *E*, *A*.

Total request matrix

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix}$$

Current allocation matrix

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

Resources in existence

$$E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix}$$

Resources available

$$A = \begin{bmatrix} 2 & 3 & 2 & 4 \end{bmatrix}$$

A is obtained by subtracting each column sum of *C* from E
10-(1+5+2+0)=2, 5-(0+1+1+0)=3
6-(0+1+1+2)=2, 5-(0+1+0+0)=4

Request to check for safety

- Assume the starting state is a safe state (can be checked with Banker's algorithm)
- P2 is now requesting 2 more of Resource 1 and 1 more of Resource 3
- Do we grant this request? Might this request cause deadlock?
  - Step 1: Calculate the state of the system if this request is filled
  - Step 2: determine if the new state is a safe state with Bankers algorithm

An example system: new state

Total request matrix

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix}$$

Current allocation matrix

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

Resources needed matrix

$$R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

Resources in existence

$$E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix}$$

Resources available

$$A = \begin{bmatrix} 0 & 3 & 1 & 4 \end{bmatrix}$$

A is obtained by subtracting each column sum of C from E
10-(1+7+2+0)=0, 5-(0+1+1+0)=3
6-(0+2+1+2)=1, 5-(0+1+0+0)=4

An example: is new state safe

- Check row 1 of matrix *R-C*

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

$$E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 3 & 1 & 4 \end{bmatrix}$$

- $(R\text{-}C)_1$ = [2, 2, 2, 1] > A
- P1 cannot run to completion

An example: is new state safe

- Check row 2 of matrix *R-C*

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

$$E = [10 \quad 5 \quad 6 \quad 5] \qquad A = [0 \quad 3 \quad 1 \quad 4]$$

- $(R\text{-}C)_2$ = [0, 0, 1, 0] <= A
- Allocate resources and run P2 to completion;
- Free all its resources and add them to A:
- A=[0 3 1 4]+[7 1 2 1] = [7 4 3 5]

An example: is new state safe

- Check row 1 of matrix *R-C* again

Row 2 thread marked as completed

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

$$E = [10 \quad 5 \quad 6 \quad 5] \qquad A = [7 \quad 4 \quad 3 \quad 5]$$

- $(R\text{-}C)_1$ = [2, 2, 2, 1] <= A
- Allocate resources and run P1 to completion;
- Free all its resources and add them to A:
- A = [7 4 3 5] + [1 0 0 0]=[8 4 3 5]

An example: is new state safe

- Check row 3 of matrix *R-C*

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

$$E = [10 \quad 5 \quad 6 \quad 5] \qquad A = [8 \quad 4 \quad 3 \quad 5]$$

- $(R\text{-}C)_3$ = [1, 0, 3, 0] <= A
- Allocate resources and run P3 to completion;
- Free all its resources and add them to A:
- A = [8 4 3 5] + [2 1 1 0] = [10 5 4 5]

An example: is new state safe

- Check row 4 of matrix *R-C*

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

$$E = [10 \quad 5 \quad 6 \quad 5] \qquad A = [10 \quad 5 \quad 4 \quad 5]$$

- $(R\text{-}C)_4$ = [4, 2, 0, 1] <= A
- Allocate resources and run P4 to completion
- Free all its resources and add them to A:
- A = [10 5 4 5] + [0 0 2 0] = [10 5 6 5]

An example: is new state safe

- Now:

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

$$E = [10 \quad 5 \quad 6 \quad 5] \qquad A = [10 \quad 5 \quad 6 \quad 5]$$

- All thread can complete successfully. Therefore, the starting state is a safe state
- Allocate the requested resources (2 more of resource 1 and 1 more of resource 3) to P2, and proceed with execution of all threads

Next Request to Check for Safety

- Now start from this new safe state, and consider the next request for resources
- Thread 1 is now requesting 1 more of resource 3
- Do we grant this request? Might this request cause deadlock?
  - Step 1: Calculate the state of the system if this request is filled
  - Step 2: Determine if the new state is a safe state, use the bankers algorithm

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

$$E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 3 & 1 & 4 \end{bmatrix}$$

New starting state: next request, is this state safe?

- Check all rows

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \quad R - C = \begin{bmatrix} 2 & 2 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}$$

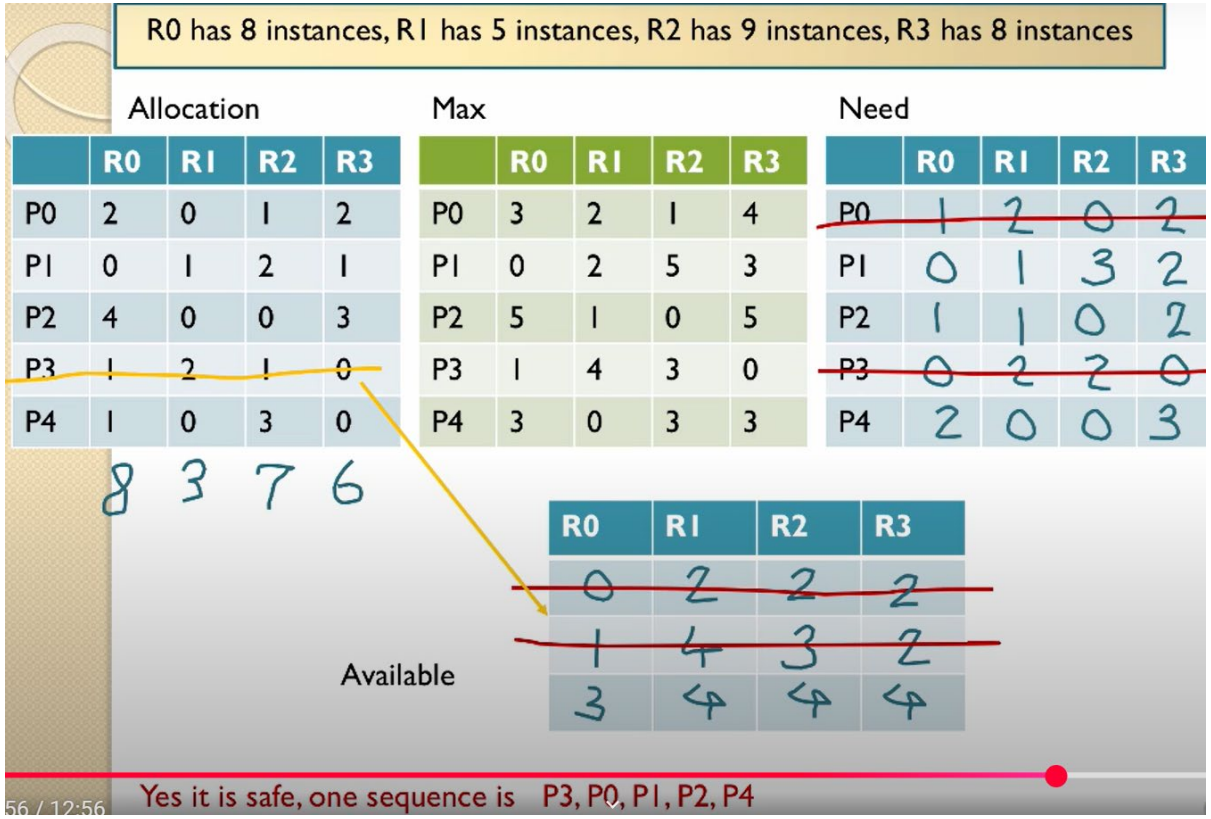
$$E = [10 \quad 5 \quad 6 \quad 5] \qquad A = [0 \quad 3 \quad 0 \quad 4]$$

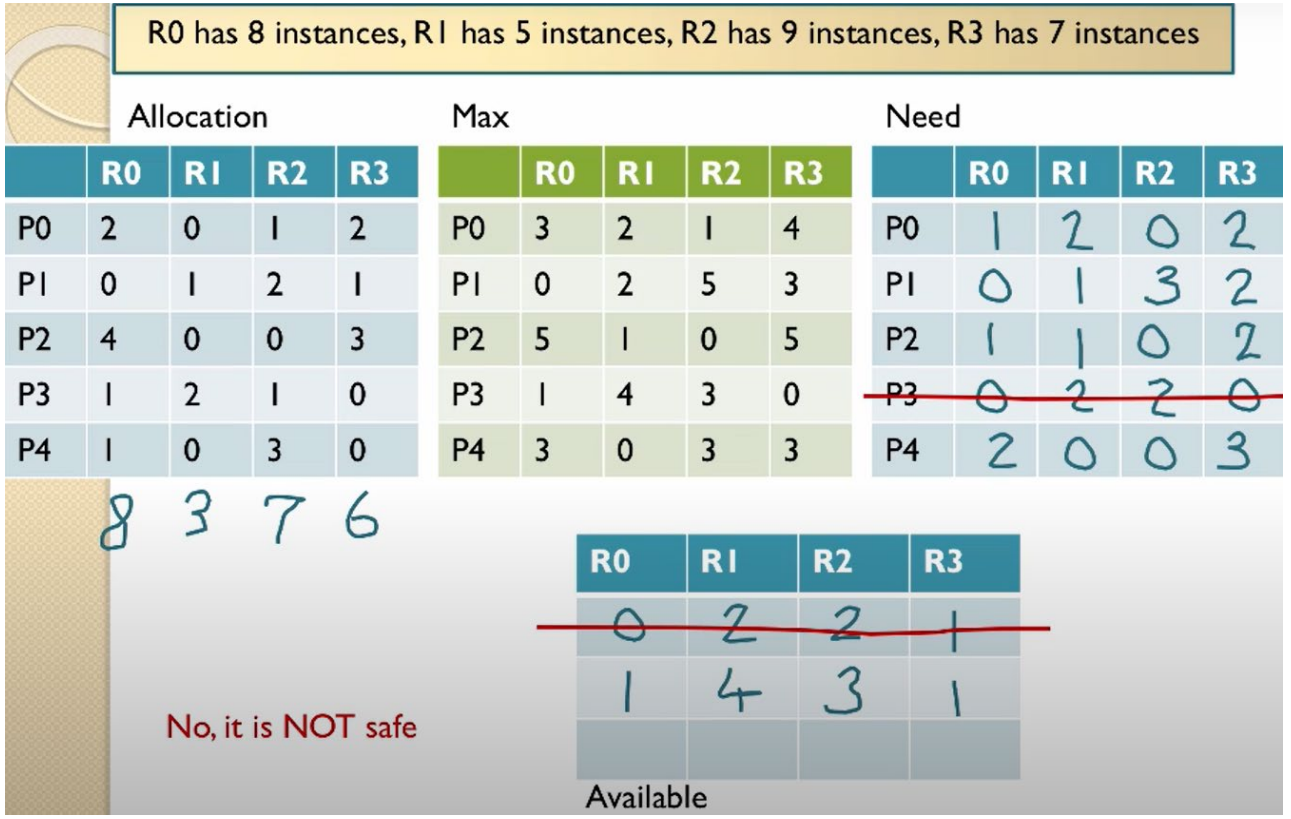
- $(R\text{-}C)_1 = [2, 2, 1, 1]$ $(R\text{-}C)_1 > A$
- $(R\text{-}C)_2 = [0, 0, 1, 0]$ $(R\text{-}C)_2 > A$
- $(R\text{-}C)_3 = [1, 0, 3, 0]$ $(R\text{-}C)_3 > A$
- $(R\text{-}C)_4 = [4, 2, 1, 0]$ $(R\text{-}C)_4 > A$

No thread can run to completion. The state is unsafe, as threades may be deadlocked

Video tutorial of Bankers algorithm

- Deadlock avoidance: https://www.youtube.com/watch?v=AvPjOyeJbBM





Unsafe state vs. deadlock

- The unsafe state indicates not that the system is deadlocked or will become deadlocked, but that there is potential for deadlock if the system operates in that state. Thus, to avoid deadlock, we do not allow the system to allocate resources that would put it into an unsafe state.
- This is a conservative strategy. It is possible that threads blocked because of a risk of deadlock would not in fact cause a deadlock during execution
- We are basing deadlock detection on worst case assumptions
  - The thread may use ALL the resources it needs at any time

Banker's algo applied to Dinning Philosophers

- State is safe if when a philosopher tries to take a fork, either
  - It is not the last fork,
  - Or it is the last fork, but someone will have 2 forks afterwards.
  - Equivalently: do not let a philosopher take the last fork if no one will have 2 forks afterwards.
- Consider N philosophers
  - If each of the N-1 philosophers holds his left fork, then the N^{th} philosopher will be prevented from taking the last fork.
  - If a philosopher is holding his left fork, he can safely pick up his right fork if available, and vice versa.
  - If one or more philosophers are holding 2 forks and eating, then any remaining forks can be picked up safely by any other philosopher.
- Banker's algorithm can be used to verify each of these scenarios

Banker's algo applied to Dinning Philosophers cont'

- Model each fork as a separate resource, since each philosopher can only pick up his left and right forks.
- Suppose we have 5 philosophers numbered 1-5, and 5 forks numbered 1-5; philosopher i has left fork numbered i, and right fork (i+1)%5.

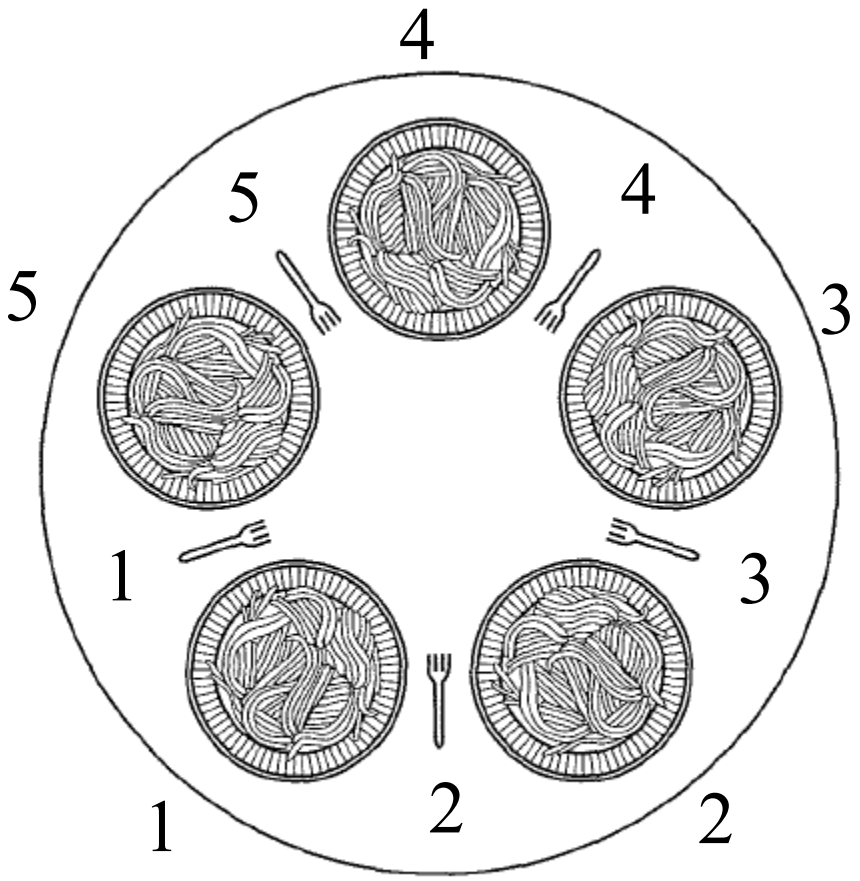


Figure 2-44. Lunch time in the Philosophy Department.

When 4 philosophers each holds his left fork

Total Request matrix

$$R = \begin{vmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Current allocation matrix

$$C = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Resources in existence

$$E = \begin{vmatrix} 1 & 1 & 1 & 1 & 1 \end{vmatrix}$$

Resources available

$$A = \begin{vmatrix} 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Philosophers 1-4 each is holding his left fork. If the 5th philosopher makes a request for his left fork, should we grant it?

The deadlocked state when each holds his left fork

Total Request matrix

$$R = \begin{vmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Current allocation matrix

$$C = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Resources in existence

$$E = \begin{vmatrix} 1 & 1 & 1 & 1 & 1 \end{vmatrix}$$

Resources available

$$A = \begin{vmatrix} 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

No. Here is the deadlock state reached if the request is granted.

Multi-Armed Lawyers

- Consider a large table with IDENTICAL multi-armed alien lawyers. In the center is a pile of chopsticks. In order to eat, a lawyer must have one chopstick in each hand. The lawyers are so busy talking that they can only grab one chopstick at a time. Design a deadlock-free algorithm using monitors and Bankers algorithm. Assume total number of chopsticks >= number of hands of each lawyer, so at least one lawyer can eat.
- It is not a generalization of the 2-armed Dining Philosophers problem. Since the chopsticks are in a pile at center of the table, we should model them as a single resource with multiple instances, instead of multiple resources for the Dining Philosophers, where each fork (chopstick) has a fixed position in-between two philosophers. Hence the R and C matrices have a single column.

Ack: this example is taken from UC Berkeley CS162 course.

Example: 5 Lawyers, each with 2 arms, 5 chopsticks

Total Request matrix R (NumArms=2)
Current allocation matrix C
Resources in existence E
Resources available A

$$R = \begin{vmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{vmatrix}, C = \begin{vmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{vmatrix}, E = |5|, A = |5| \qquad R = \begin{vmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{vmatrix}, C = \begin{vmatrix} 2 \\ 2 \\ 0 \\ 0 \\ 0 \end{vmatrix}, E = |5|, A = |1|$$

- Initially, all chopsticks are free.
- Two lawyers grab two chopsticks each and start eating. No other lawyers can eat.

Multi-Armed Lawyers Solution w/ Monitors

```
public class BankerTable {
    Lock lock = new Lock();
    Condition CV = new Condition(lock);

    int[] AllocTable;                          /* Keep track of allocations */
    int NumArms, NumChopsticks;                /* Keep other info */

    public BankerTable(int NumLawyers, int NumArms, int NumChopsticks) {

        AllocTable = new int[NumLawyers];      /* Array of allocations (init 0)*/
        this.NumArms = NumArms;                /* Remember # arms/lawyer */
        this.NumChopsticks = NumChopsticks;    /* Remember # chopstocks/table */

    }
    public void GrabOne(int Lawyer) {

        lock.Acquire();

        /* Use banker algorithm to check whether allocation is ok */
        while (!BankerCheck(Lawyer))
            CV.wait();

        /* Move one chopstick from table to lawyer */
        NumChopsticks--;
        AllocTable[Lawyer]++;

        Lock.Release();

    }
    public void ReleaseAll(int Lawyer) {

        lock.Acquire();
        NumChopsticks += AllocTable[Lawyer]; /* Put chopsticks back on table*/
        AllocTable[Lawyer] = 0;                /* Lawyer has none left */
        lock.Release();
    }
```

- GrabOne() allows a lawyer to grab one chopstick. It puts a lawyer to sleep if he cannot be granted a chopstick without potentially deadlocking the system.
- ReleaseAll() allows a lawyer to release all chopsticks that he is holding. It wakes up any other lawyers that can proceed.
- BankerCheck() method takes a Lawyer number, checks resources, and returns true if a given lawyer can be granted one new chopstick.

BankerCheck()

```
boolean BankerCheck(int Lawyer) {
/*
 * This method implements the bankers algorithm for the
 * multi-armed laywer problem. It should return true if the
 * given lawyer can be granted one chopstick.
 */

    int i;
    AllocTable[Lawyer]++;          /* Hypothetically give chopstick to lawyer */
    for (i = 0; i < AllocTable.length; i++)
        /* Is current number on table (NumChopsticks-1) enough to let Lawyer i eat? */
        if ((AllocTable[i] + (NumChopsticks - 1)) >= NumArms)
            break;                 /* Yes! Break early */
    AllocTable[Lawyer]--;          /* Take away hypothetical */
    Return (i < AllocTable.length);  /* Returns true if we broke at any time */
}
}
```

- In its general form, the Banker's algorithm makes multiple passes through the set of resource takers for deadlock detection. But this particular application allows the BankerCheck() method to take a single pass though the Lawyers' allocations. This is because every resource taker has identical properties: every Lawyer has the same NumArms (maximum allocation). As a result, if we can find a single Lawyer that can finish, given the remaining resources, we know that all Lawyers can finish. Reason: once that Lawyer finishes and returns their resources we know that there will be at least NumArms chopsticks on the table – hence everyone else can potentially finish. Thus, we don't have to go through the exercise of returning resources and reexamining the remaining Lawyers (as in the general Banker's algorithm).

- State is safe if when a lawyer tries to take a chopstick, either
- It is the last chopstick, but someone (lawyer i) will have NumArms chopsticks afterwards
- Or it is the 2nd to last chopstick, but someone (lawyer i) will have NumArms-1 chopsticks afterwards
- Or it is the 3rd to last chopstick, but someone (lawyer i) will have NumArms-2 chopsticks afterwards
- …
- And so on…
- Q: Why didn't we check for the case of NumChopsticks == 0?
- A: In this case, (NumChopsticks-1) == -1, hence the if statement would always fail – exactly what we would want to do when NumChopsticks == 0.

Minimum Resource Constraint

- In all our problem formulations, we have assumed there are a minimum number of resources to allow at least one thread to finish. Without this constraint, the system cannot even start execution, hence the problem is ill-defined.
  - Consider the dining philosophers problem with a single fork, or no fork available.

When to run Banker's algorithm?

- Run it each time a resource allocation request is made. This can be expensive.
- Run it periodically driven by a timer interrupt. A longer period between checks gives
  - Higher efficiency due to less calculation involved in the checking
  - Undetected deadlocks can persist for longer times
- What to do if an actual deadlock is detected?

Deadlock recovery

1. Abort all deadlocked threades:
   - most common solution implemented in OSs.
2. Rollback:
   - Back up each thread periodically.
   - in case of deadlock roll back to the previous backup (checkpoint).
   - It is possible the deadlock may reoccur.
   - Usually the deadlock will not reoccur due to the nondeterministic nature of the execution of concurrent threades (there may be a different interleaving of instruction executions the next time).

Deadlock recovery: 2

3. Successively abort deadlocked threads.
   - Abort 1 deadlocked thread at a time.
   - Then check if the deadlock still occurs
     - » If it does abort the next thread.
     - » If it does not, continue execution of the remaining threades without aborting any more threades.
4. Successively preempt resources from blocked threads
   - Preempt 1 deadlocked resource in 1 thread.
   - Roll back that thread to the point where the preempted resource was allocated.
   - Check if deadlock still occurs
     - » If it does preempt resource from the next thread.
     - » If it does not, continue execution of the remaining threades without preempting any more resources.

Choosing threads/resources

- For options 3 and 4 it is necessary to choose which of the possibly deadlocked threads to abort or which resource to preempt (and the possibly deadlocked thread to preempt it from)
- Can base this decision on different criteria
  - Lowest priority
  - Most estimated run time remaining
  - Least number of total resources allocated
  - Smallest amount of CPU consumed so for

Issues

- None of these approaches is appropriate for all types of resources
  - Some threads (like updating a database) cannot be killed and rerun safely.
  - Some resources cannot be safely preempted (some of these like printers can be preempted if spooling is used).
  - Some threads cannot be rolled back.
    - » How do you roll back shared variables that have been successively updated by multiple threads.
  - Thread rollback is expensive.
    - » Successive checkpoints must save both image and state.
    - » Multiple checkpoints need to be saved for a thread.

Communication deadlocks

- Thread A sends a request message to thread B, and then blocks until B sends back a reply message.
- Suppose that the request message gets lost. A is blocked waiting for the reply. B is blocked waiting for a request asking it to do something. Deadlocked.
- Deadlock not due to shared resources but due to message communication.

Summary of Banker's algorithm

- Look one step ahead: upon receiving a request from a thread, assume the request is granted hypothetically, run deadlock detection algorithm to evaluate if the system is in a safe state.
  - A state is safe if from this state, there exists a sequence of thread executions {P_1, P_2, … P_n} with P_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..that can finish successfully.
- Grant the request if next state is safe.
- Algorithm allocates resources dynamically, and allows the sum of maximum resource needs of all current threades to be greater than total resources
- It is a conservative algorithm, since each thread must declare the maximum resource requests, which may be a pessimistic estimate of the actual resource requests at runtime.