CSC 112: Computer Operating Systems
# Lecture 4

Deadlocks

Department of Computer Science,
Hofstra University

Deadlock

- Definition: A set of processes are said to be in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set

- Conditions for Deadlock

- Mutual exclusion
  - Only one process at a time can use a given resource

- Hold-and-wait
  - processes hold resources allocated to them while waiting for additional resources

- No preemption
  - Resources cannot be forcibly removed from processes that are holding them; can be released only voluntarily by each holder

- Circular wait
  - There exists a circle of processes such that each holds one or more resources that are being requested by next process in the circle
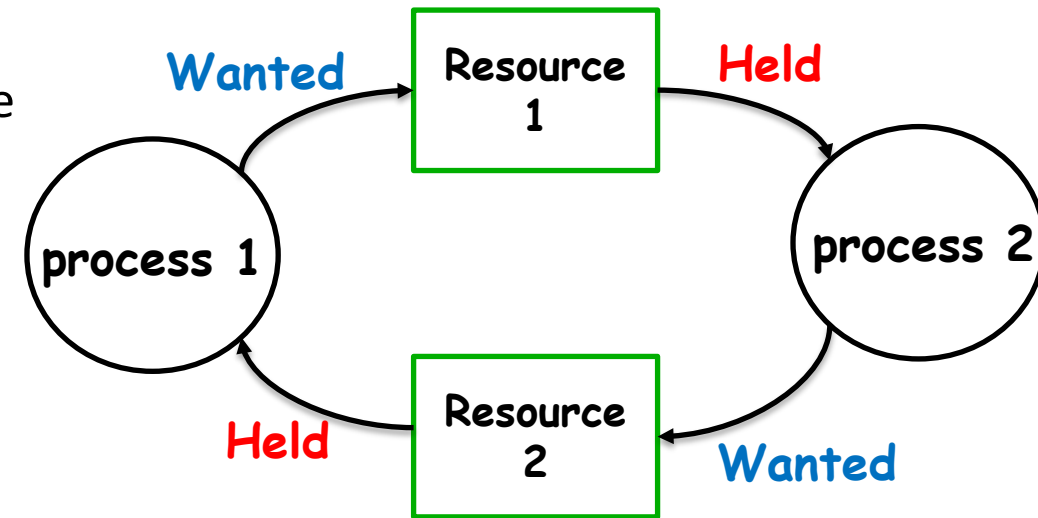

Not a perfect analogy, just a fun image!

Starvation vs Deadlock

- Starvation: process waits indefinitely
  - Example, low-priority process waiting for resources constantly in use by high-priority process
- Deadlock: circular dependency waiting for resources
  - Starvation can end (but doesn't have to)
 - Deadlock cannot 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 $\Rightarrow$ no car can go west
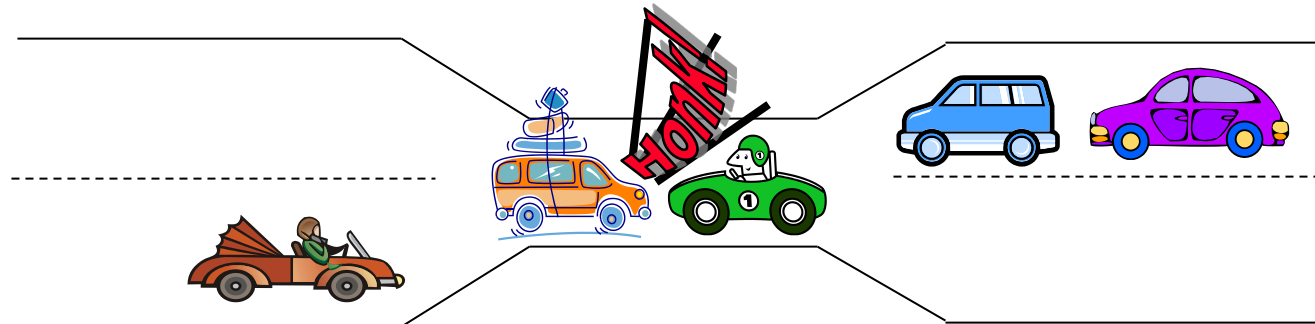
Train Example (Wormhole-Routing for Network-on-Chip)

- 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)
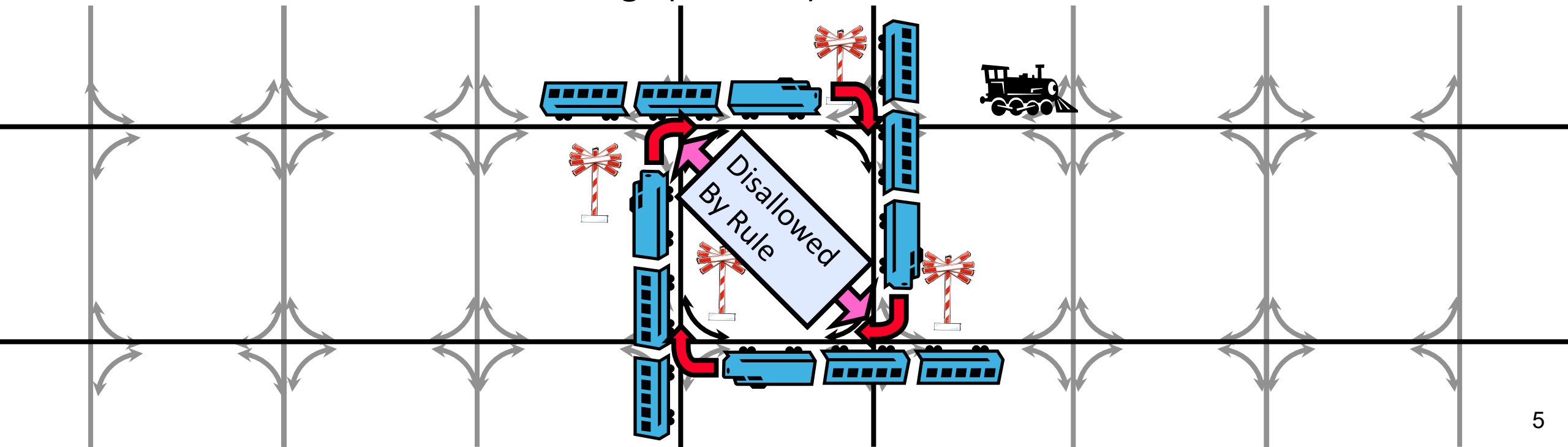
Handling Deadlocks

- ## 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

Deadlock Prevention/Recovery Techniques

1) Break "mutual exclusion" by spooling resources
2) Break "hold and wait" condition: Make all processes request everything they'll need at the beginning.
    1) Problem: Predicting future is hard, tend to over-estimate resources
    2) Let each philosopher pick up both left and right forks atomically within a critical section (L3, "Semaphore-based Solution I")
3) Break "circular wait" condition:
    1) Force all processes to request resources in a particular order.
        1) May not be practical, since runtime resource usage pattern is generally unknown
    2) Let each philosopher pick up lower-numbered fork before higher-numbered fork (modulo N) (L3, "Semaphore-based Solution III")
    3) Banker's algorithm can prevent future "circular wait" conditions by detecting *potential* deadlocks
4) Break "no preemption" condition:
    1) Forcibly remove resources from process

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

(1) Spooling

- A single daemon process directly uses the resource; other processes send their requests to the daemon, e.g.:
- The resource is no longer directly shared by multiple processes
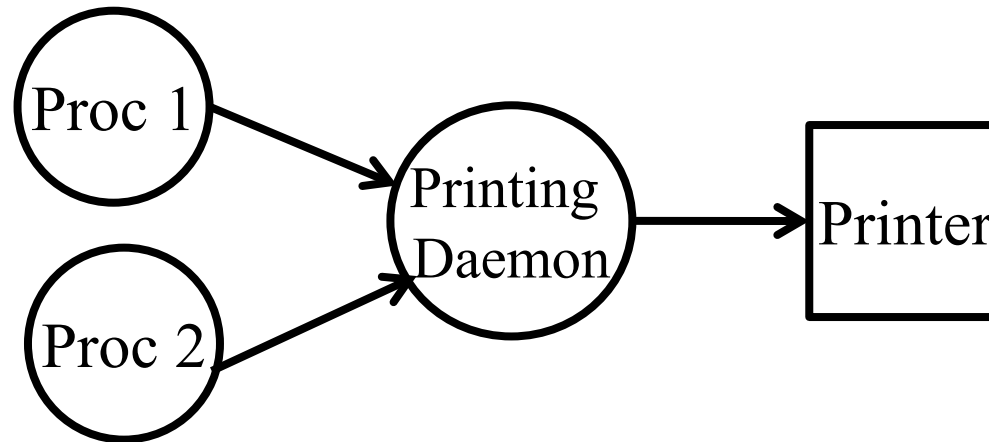
(2) Request all resources initially

- **Disallow hold-and-wait**
  - Make each process request all resources at the same time, and block until all resources are available to be granted simultaneously
  - May be inefficient
    - » Process 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 currently granted resources
    - » Resources allocated to a process may remain unused for long periods of time, blocking other processes
    - » processes may not know all resources they will require in advance.

(3) Order resources numerically

- **Prevent circular wait**

  - Define a total order of resources; If a process holds certain resources, it can subsequently request only resources that follow the types of held resources in the total order.

  - This prevents a process from requesting a resource that might cause a circular wait.
    - » Example; all processes 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

(4) Take resources away

- Allow preemption. Can be implemented in different ways
  - Abort all deadlocked processes: most common solution implemented in OSs.
  - If a process 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 process requests a resource that is in use (usually by a lower priority process), the process using the resource will be preempted and the resource will be supplied to the requesting process.
  - Requires additional OS complexity
- 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 process 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.
- 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 processes $P_1, P_2, \ldots, P_n$
  - 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 process utilizes a resource as follows:
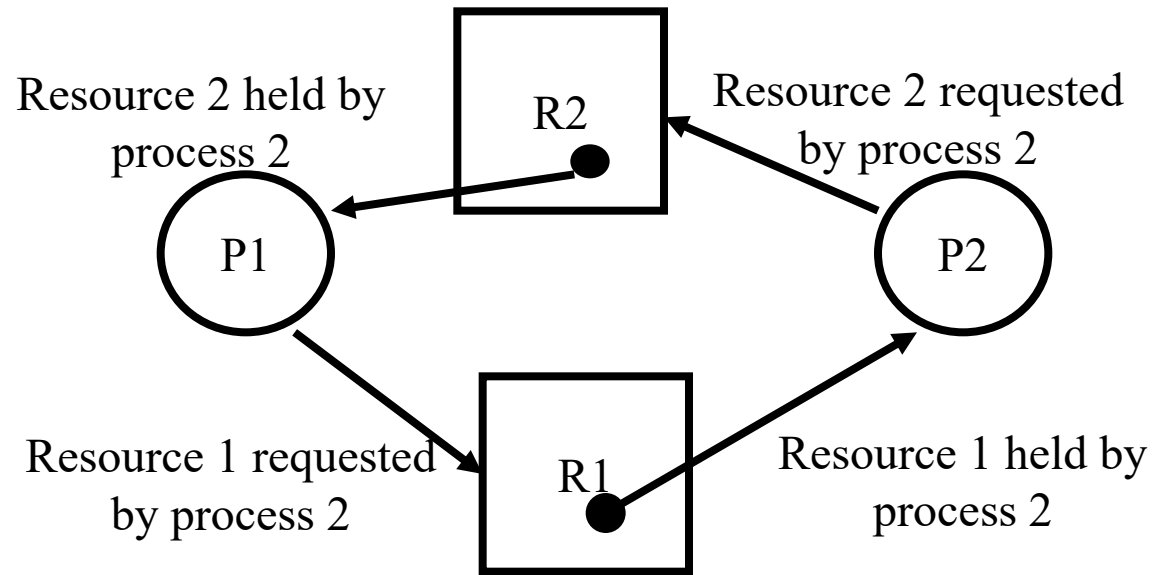   - » `Request() / Use() / Release()`
- **Resource-Allocation Graph (RAG):**
  - V is partitioned into two types:
    - » $P = \{P_1, P_2, \ldots, P_n\}$, set of processes in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, set of resource types in system
  - request edge – directed edge $P_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$

Symbols
P_1 P_2
R_1 ∘ R_2 ∘∘∘

13
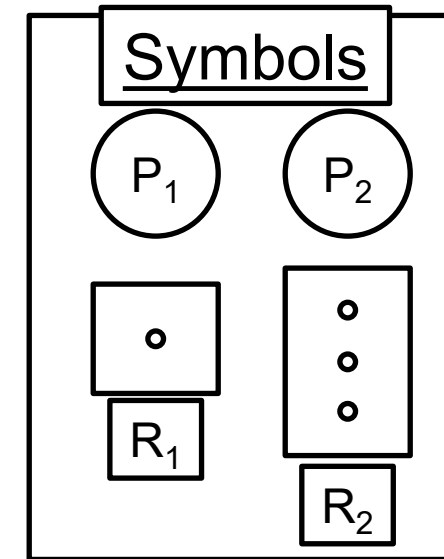
RAG for deadlock detection

- 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

Resource 2 held by process 2

R2

Resource 2 requested by process 2

P1

P2

Resource 1 requested by process 2

R1

Resource 1 held by process 2

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
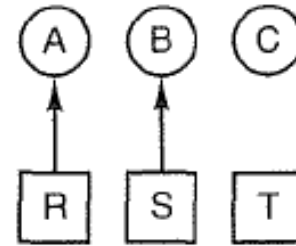Request R
Release T
Release R
(c)

1. A requests R
2. B requests S
3. C requests T
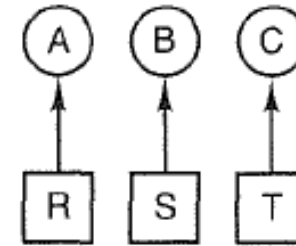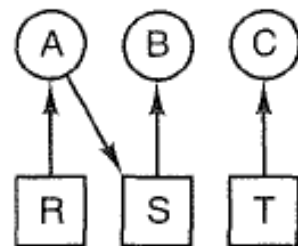4. A requests S
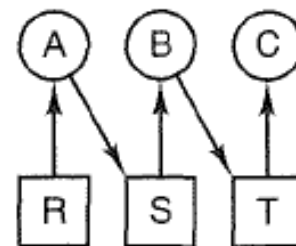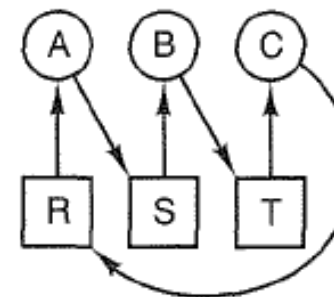5. B requests T
6. C requests R
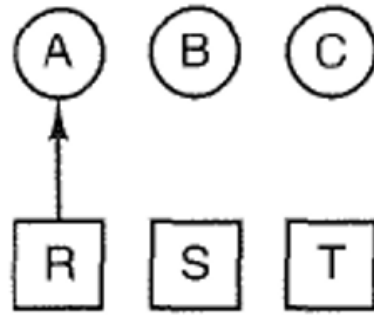   deadlock
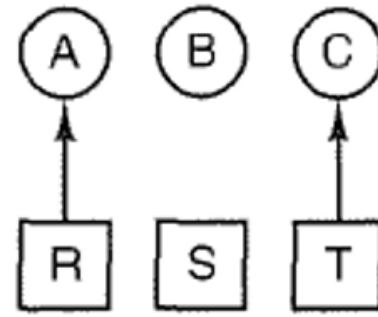
(d)
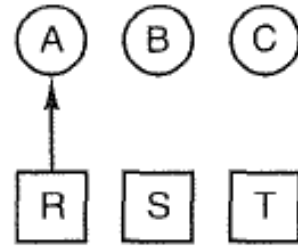
(e)

(f)

(g)

(h)

(i)

(j)

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
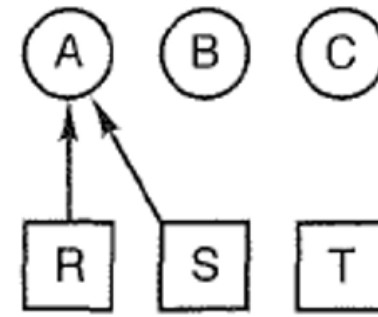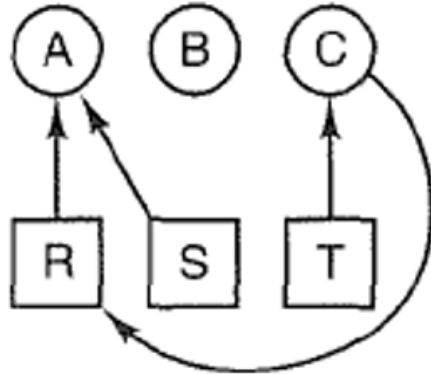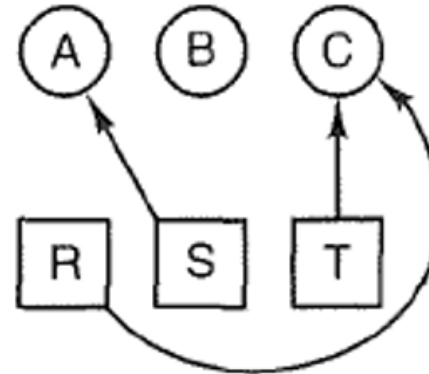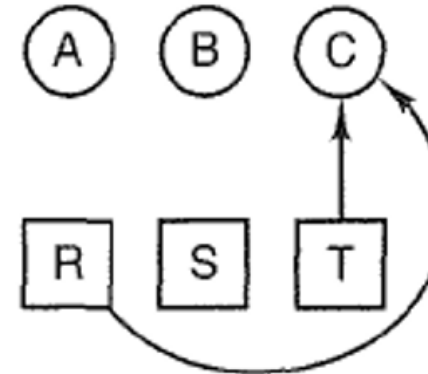6. A releases S
   no deadlock

(k)

(l)

(m)

(n)

(o)

(p)

(q)

No Cycle;
No Deadlock

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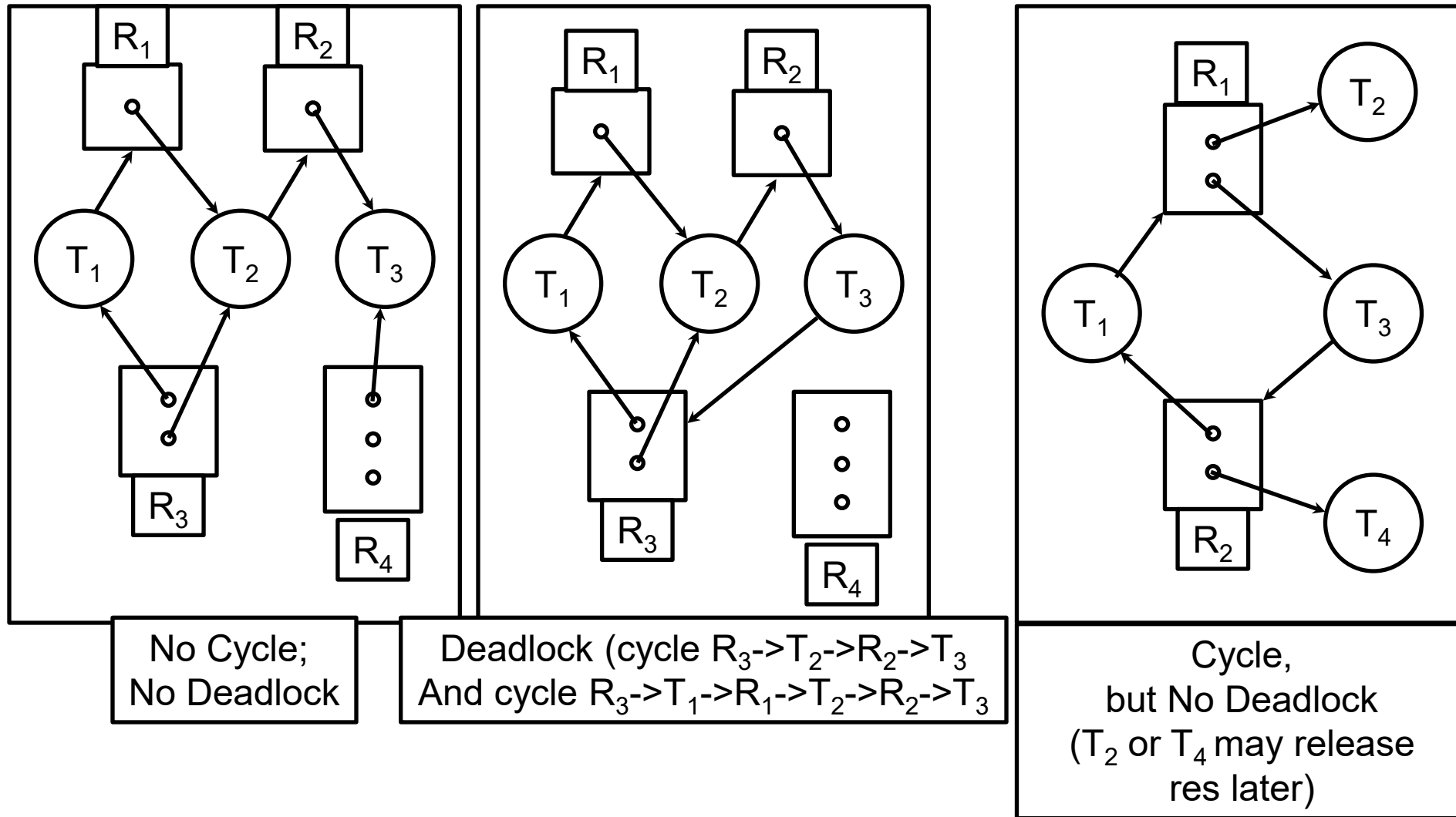And cycle R_3->T_1->R_1->T_2->R_2->T_3

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 process if this allocation *might* lead to a deadlock
- Banker's Algorithm was developed by Edsger Dijkstra, inspired by the way banks manage loans to ensure they do not run out of resources.
  - It ensures that loans are only granted if the bank can still meet the withdrawal needs of all its account holders, even in the worst-case scenario where everyone withdraws their funds simultaneously. Similarly, in computing, the algorithm ensures that resources are allocated to processes in a way that avoids unsafe states or deadlocks

Problem Definition

- Consider a system with *n* processes and *m* different types of resources.

- *Total resource vector $E = (E_1, E_2, \ldots, E_m)$*
  - Each resource type may have multiple instances, so the value of E_i is the number of instances of resource type i.

- *Available resource vector $A = (A_1, A_2, \ldots, A_m)$*
  - It keeps track of how many instances of each resource type are currently available (not in-use).

- *Allocation matrix C* denotes which processes are using which resources.
  - e.g., if process *i* is using 2 resources of type *j* then C_{ij} = 2.

- *Max matrix R* denotes the maximum number of instances of each resource that each process needs during its execution.
  - e.g., if process *i* needs maximum 4 instances of resource type *j* during its execution, then R_{ij} = 4.

- *Need = Max − Allocation:* denotes the additional number of instances of each resource that each process needs to finish its execution.

- For each process *i* and resource *j*: $C_{ij} \leq R_{ij} \leq E_j, \forall i, j$

Four data structures encode current state of the system

Total

$(E_1, E_2, E_3, \ldots, E_m)$

Available

$(A_1, A_2, A_3, \ldots, A_m)$
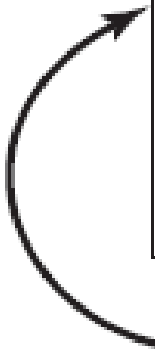
Allocation

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

Row n is current allocation to process n

Max

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

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 processes in the system to complete without deadlock, i.e., there exists a sequence of process executions {Pi, Pj, ... Pk} with Pi requesting all remaining resources, finishing, then Pj requesting all remaining resources, ..., until all processes complete successfully.
  - An unsafe state is a state in which there exists no sequence of resource allocations that will allow all processes in the system to complete without deadlock.

Banker's algorithm

- Look one step ahead: upon receiving a request from a process, assume the request is granted hypothetically, run deadlock detection algorithm to evaluate if the system is in a safe state.

- Grant the request if next state is safe.

- Algorithm allocates resources dynamically, and allows the sum of maximum resource needs of all current processes to be greater than total resources

- It is a conservative algorithm, since each process must declare the maximum resource requests, which may be a pessimistic estimate of the actual resource requests at runtime.

Banker's algorithm: preliminaries

- Compute *Need = Max – Allocation*

- To determine if a process *i* can run to completion, compare two vectors:
  - *(Need)$_i$*: row *i* in the Need Matrix of unmet resource needs
  - A: available resources vector *A*
  - *(Need)$_i$ <= A* if *Need$_{ij}$ <= A$_j$* for all resource types *j*

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

1. Compute *Need = Max – Allocation*

2. Look for a process *i* that can run to completion by finding an unmarked row *i* with *(Need)$_i$ ≤ A*. If no such row exists, system will eventually deadlock since no process can run to completion

3. Assume process *i* requests all resources it needs and finishes. Mark process *i* as completed, free all its resources and add the *i-th* row of *Allocation* to the *Available* vector

4. Repeat steps 1 and 2 until either all processes are marked as completed (initial state is safe), or no process is left whose resource needs can be met (there is a deadlock, so initial state is unsafe).

Banker's algorithm cont'

- When a process 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, deny the process request until it is safe to grant it.

- 4 processes 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.

Max

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

Allocation

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

Total

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

Available

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

Available 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 instances of Resource 1 and 1 more instance 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 fulfilled
  - Step 2: Determine if the new state is a safe state with Banker's algorithm

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

Max

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

Allocation

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

Need = Max – Allocation

Total

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

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 Need matrix

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

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

- $(\text{Need})_1$ = [2, 2, 2, 1] not <= A
- P1 cannot run to completion

An example: is new state safe

- Check row 2 of Need matrix

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

Max   Allocation   Need

Total   Available

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

- $(\text{Need})_2$ = [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 Need matrix again

Process 2 marked as completed

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

Max   Allocation   Need

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

Available
$$A = \begin{bmatrix} 7 & 4 & 3 & 5 \end{bmatrix}$$

- $(Need)_1$ = [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 Need Matrix

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

$$
E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix} \qquad A = \begin{bmatrix} 8 & 4 & 3 & 5 \end{bmatrix}
$$

- $(\text{Need})_3$ = [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]

- Check row 4 of Need Matrix

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

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

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

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

Available
$$ A = \begin{bmatrix} 10 & 5 & 4 & 5 \end{bmatrix} $$

- $(Need)_4$ = [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]

- Now:

$$
\begin{array}{c}
\text{Max} \\
R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix}
\end{array}
\quad
\begin{array}{c}
\text{Allocation} \\
C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 1 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}
\end{array}
\quad
\begin{array}{c}
\text{Need} \\
R - C = \begin{bmatrix} 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 \\ 4 & 2 & 0 & 1 \end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
\text{Total} \\
E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix}
\end{array}
\qquad
\begin{array}{c}
\text{Available} \\
A = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix}
\end{array}
$$

- All process can complete successfully. Therefore, the starting state is a safe state

- Allocate the requested resources (2 more instances of resource 1 and 1 more instance of resource 3) to P2, and proceed with execution of all processes

- Now start from this new safe state, and consider the next request for resources: Process 1 is now requesting 1 more instance 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 Banker's algorithm

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

Max   Allocation   Need

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

Total   Available

- Check all rows

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

Max — Allocation — Need

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

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

- $(Need)_1 = [2, 2, 1, 1]$ not $<= A$
- $(Need)_2 = [0, 0, 1, 0]$ not $<= A$
- $(Need)_3 = [1, 0, 3, 0]$ not $<= A$
- $(Need)_4 = [4, 2, 1, 0]$ not $<= A$

No process can run to completion. The state is unsafe, so we deny Process 1's request for 1 more instance of resource 3.

Video tutorial of Banker's algorithm I

- Deadlock avoidance https://www.youtube.com/watch?v=AvPjOyeJbBM
- Total resources: [8, 5, 9, 8]

- Banker's Algorithm explained https://www.youtube.com/watch?v=T0FXvTHcYi4
- Total resources: [3, 14, 12, 12]

Available + Allocation = New Available

	Allocation A B C D	Max A B C D	Available A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

	Need A B C D	
P0	0 0 0 0	<= <1 5 2 0> = T
P1	0 7 5 0	<= <2 14 12 12> = T
P2	1 0 0 2	<= <1 5 3 2> = T
P3	0 0 2 0	<= <2 8 8 6> = T
P4	0 6 4 2	<= <2 14 11 8> = T

$$<1\ 5\ 2\ 0> + <0\ 0\ 1\ 2> = <1\ 5\ 3\ 2>$$
$$<1\ 5\ 3\ 2> + <1\ 3\ 5\ 4> = <2\ 8\ 8\ 6>$$
$$<2\ 8\ 8\ 6> + <0\ 6\ 3\ 2> = <2\ 14\ 11\ 8>$$
$$<2\ 14\ 11\ 8> + <0\ 0\ 1\ 4> = <2\ 14\ 12\ 12>\ N.Available$$

The Safe Sequence:

P0 P2 P3 P4 P1

38

- Consider N philosophers and N forks.
  - (1) If each of the N-1 philosophers holds his left fork, then the Nth philosopher will be prevented from taking the last fork.
  - (2) If a philosopher is holding one fork, he can safely pick up the other fork.
  - (3) 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. Let's focus on scenario (1) next.

- 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. (Here indices start from 1 instead of 0 in Lecture 3.)
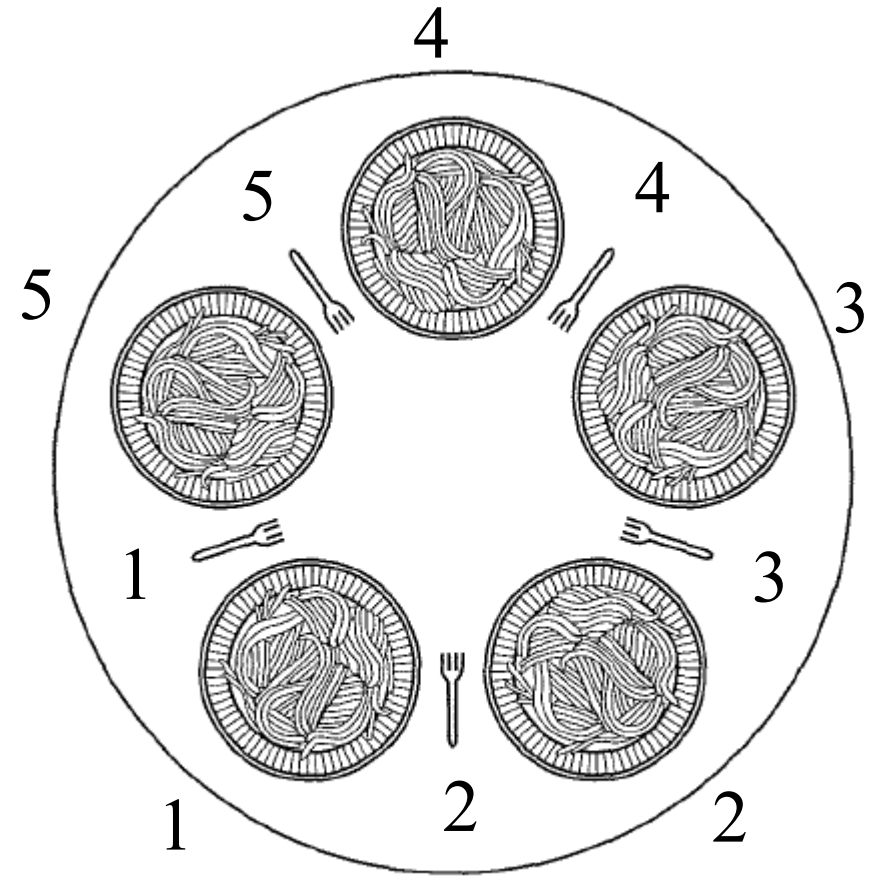

Figure 2-44. Lunch time in the Philosophy Department.

Max

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

Allocation

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

Need

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

Total

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

Available

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

Philosophers 1-4 each is holding his left fork. We can use Banker's algorithm to check that the current state is safe, e.g., with execution sequence of P4, P3, P2, P1, P5.

Now, if philosopher 5 makes a request for his left fork, should we grant it?

Five philosophers each holding his left fork

Max

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

Allocation

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

Need

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

Total

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

Available

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

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

Minimum Resource Constraint

- In all our problem formulations, we have assumed there are a minimum number of resources to allow at least one process 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.

Communication deadlocks

- process A sends a request message to process 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.